FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Architecture-driven Design and Configuration of Messaging Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Philipp Waibel
Matrikelnummer 0716754

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: o.Univ.Prof. Mag. Dr. Schahram Dustdar
Mitwirkung: Christoph Dorn, Ph.D.

Wien, 19.02.2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Architecture-driven Design and Configuration of Messaging Systems

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Philipp Waibel
Registration Number 0716754

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: o.Univ.Prof. Dr. Schahram Dustdar
Assistance: Christoph Dorn, Ph.D.

Vienna, 19.02.2015

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Philipp Waibel
Aichhorngasse 3, 1120 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

First of all I want to give my sincere thanks to my advisor Prof. Schahram Dustdar and to my co-advisor Christoph Dorn who has always taken the time to provide me with feedback and motivated me to improve myself from the first day on. Additionally I want to thank the people at ilogs information logistics GmbH for their support and the opportunity to work with them, which guided us to the fundamental idea of this thesis.

Furthermore I want to thank my fellow students for the interesting discussions during the years of study and my girlfriend for her ongoing support. Finally I would like to give my sincere gratitude to my family for their support.

Abstract

Nowadays, the use of services as the foundation stone, to develop a complex and reliable system, is an established method. These systems are composed of several services, where each of them fulfills a special (sub-)task in the overall service-oriented system. Such a system has the characteristics of extremely loose coupled services, unpredictable service availability, dynamically changing numbers of service instances and discourages a central execution control system. Among other methods, a message-based communication is an approved and natural way for the services to communicate. Nevertheless the design and development of a consistent message-based service system isn't trivial at all. To be more specific, it is a major problem to achieve an overall consistent configuration, where the messages get routed as it is provided in the design of the system. Whereas orchestration and choreography-based approaches have proved to be successful in designing composite services in workflow-centric styled systems, they aren't quite as useful in systems with a message-centric architecture. Though, in different scenarios, a message-centric architecture can be a better match for complex service systems that have the characteristics outlined above.

The goal of this work is to investigate how a software architecture-centric approach can be used to design and configure message-based service systems. More specifically, the approach uses an architecture description language (ADL) to model the high-level architecture of a message-based service system, and to configure the message relevant aspects of the services, respectively system. Furthermore, consistency checking is performed on the ADL document to ensure the consistency of the system. Finally, the package is completed by an architecture-to-configuration transformation, which ensures the correct implementation of the planned system. Furthermore changes of the system, which may be done at a later point in time, are propagated to the already transformed system without losing any information.

To prove the utility of the approach, a prototype, on basis of xADL and ArchStudio 4, is developed. The prototypical application is designed to transform the modeled and configured architecture into Mule ESB workflows and Apache ActiveMQ configuration files. Furthermore this prototype is used to implement a real world service system.

Kurzfassung

Heutzutage ist die Nutzung von Services als Grundbausteine für die Entwicklung von komplexen und zuverlässigen Systemen ein etabliertes Verfahren. Diese Systeme werden aus mehreren Services zusammengestellt, wobei jeder von ihnen eine spezielle (Teil-) Aufgabe in dem Gesamtsystem übernimmt. Ein solches System hat die Eigenschaften von locker gekoppelten Services, unvorhergesehenen Service Verfügbarkeiten, dynamisch verändernder Anzahl an Service-Instanzen und sollte falls möglich kein zentrales Steuerungssystem haben. Neben anderen Methoden ist die nachrichtenbasierte Kommunikation eine gute und natürliche Weise für die Kommunikation zwischen den Services. Dennoch ist die Konstruktion und Entwicklung eines konsistenten, nachrichtenbasierten Service-Systems nicht trivial. Speziell ist es ein großes Problem, ein System zu erhalten, in dem die Nachrichten so weitergeleitet werden, wie es in der Planung des Systems vorgesehen ist. Während sich bei Orchestration- und Choreography-basierten Ansätzen die Entwicklung von Systemen auf Workflow-basierten Architekturen als erfolgreich erwies, sind sie nicht sehr nützlich für Systeme mit einer nachrichtenbasierten Architektur. Jedoch kann, in verschiedenen Szenarien, eine nachrichtenbasierte Architektur besser geeignet sein, um komplexe Service-Systeme zu entwickeln, die die oben beschriebenen Eigenschaften haben.

Das Ziel dieser Arbeit ist es zu untersuchen, wie ein Software Architektur-zentrierter Ansatz verwendet werden kann, um nachrichtenbasierte Service-Systeme entwerfen und konfigurieren zu können. Dafür verwenden wir eine Architecture Description Language (ADL), um die Architektur des Systems modellieren zu können, und um die Nachricht relevanten Aspekte konfigurieren zu können. Weiters wird eine Konsistenzprüfung auf dem ADL Dokument durchgeführt, um die Konsistenz des modellierten Systems zu gewährleisten. Das Paket wird durch eine Architektur-zu-Konfigurations Transformation abgeschlossen, die sicherstellt, dass das geplante System ordnungsgemäß umgesetzt wird. Des Weiteren können Veränderungen die zu einem späteren Zeiptunkt getätigt werden, in das bereits transformierte System integriert werden, ohne dass Informationen verloren gehen.

Um die Nützlichkeit dieses Ansatzes zu beweisen wird ein Prototyp, auf der Grundlage von xADL und Archstudio 4, entwickelt. Dieser Prototyp transformiert die geplante und konfigurierte Architektur in Mule ESB Workflows und in Apache ActiveMQ Konfigurationsdateien. Des Weiteren wird dieser Prototyp verwendet, um ein reales Service-System zu entwickeln.

Contents

List of Figures

CHAPTER 1

Introduction

The following will give an introduction into the topic of this thesis. First we will discuss the general problem and motivating scenario which sets the basis of our work. Afterwards, the aim of this work is defined and an overview of the thesis structure is given.

1.1 Problem Statement

Nowadays, the use of distributed services is a common way to build complex reliable systems. This kind of distribution has the advantages of scalability, robustness and easy adaptation, but it also has to deal with different operating systems, data formats and languages [47]. Furthermore, a complex service-centric system may has to struggle with (i) unpredictable service availability, (ii) dynamically changing number of service instances and (iii) no central execution control. To encourage this new service-centric system trend, several languages, which support the composition of different services to one service-centric system, appeared in the last two decades. Prominent examples are orchestration languages, like BPEL [1] or JOpera [54], and choreography languages, like WS-CDL [37] or BPEL4Chor [15]. Nevertheless, those languages assume a workflow-centric system architecture, which doesn't fit all application scenarios, especially if they have to deal with the three problems outlined above. In contrast to this, a publish-subscriber based architecture is capable of handling unpredictable service availability, dynamic changing number of service instances and doesn't require a central execution control. Thus, this architecture approach is a better match for complex service-centric systems that have to deal with those circumstances.

Nevertheless, the design and configuration of such service-centric systems, which consist of several decentralized and loosely coupled services, are by no means trivial and raise several challenging questions. One question is: "Does the implemented system reflect the planned system?" The optimistic approach, to build a complex service system, consists of two steps: (1) design the system by specifying the individual services and the connection between them and then (2) implement the planned system. The output of this approach is on the one hand

an informal design document and on the other hand the implemented system. But it doesn't guarantee that the implemented system really reflects the planned system. This leads to the second question: "Will changes in the design or in the implementation be propagated in the correct way and will the system, after the update, be in a coherent state?" As already seen in the first question, it can't be guaranteed that the implemented system reflects the planned system. It is even harder to guarantee that updates of the design or configuration will be properly committed to the implementation. All in all it is only a matter of time before the design and the implementation aren't consistent anymore. This question also engages a problem that can occur if an engineer changes the implementation directly. Again this leads to an inconsistency between the implementation and the planned system and a coherent system can't be guaranteed. Another more project management specific question is: "Who is responsible for individual system parts and who for the overall system design and configuration?" This question engages the problem of an unclear separation of development and planning concerns.

1.2 Motivating Scenario

Different scenarios where a public-subscriber architecture is used for the underlying messaging system exist. One use case for such an architecture is a modern parking management system.

1.2.1 Parking Management System

A modern parking management system consists of a high number of distributed services. In a simplified way a typical system configuration contains four component types:

Parking Sites are the car parks, or more exactly the terminals (e.g. Ticket Terminals at entrance and exit, Payment Terminals) and servers of a car park. Each modern car park server can offer different services. This service provides on the one hand static information about the parking site's structure (e.g. location, number of parking slots, opening times, rates) and on the other hand highly dynamic information like current capacity or reserved spots.

Message Filtering & Enhancing are used to filter the parking sites to get only the required ones, including the required details, and to bring them to a uniform format. This step is mandatory because of the different car park systems.

Location or Type-specific Aggregators are services for collecting property-specific static data from the parking sites. Those services are mainly used for administrative work (e.g. carrier specific parking sites management and customer account management). Different types of parking sites can be combined by the same aggregator service. For example, one aggregator could be used to serve all parking sites in a city, operated by the same carrier, and another could be used to serve all caravan parking sites.

Parking Point-Of-Sale (POS) are the direct sales services. There are several business cases like parking slot prebooking at airports, hotels, train stations or car park searching apps and websites. These services receive the filtered and customized static data from their

corresponding aggregator service and the dynamic data directly from the filtered parking sites.

Figure 1.1 shows a simplified but typical combination of those four component types.

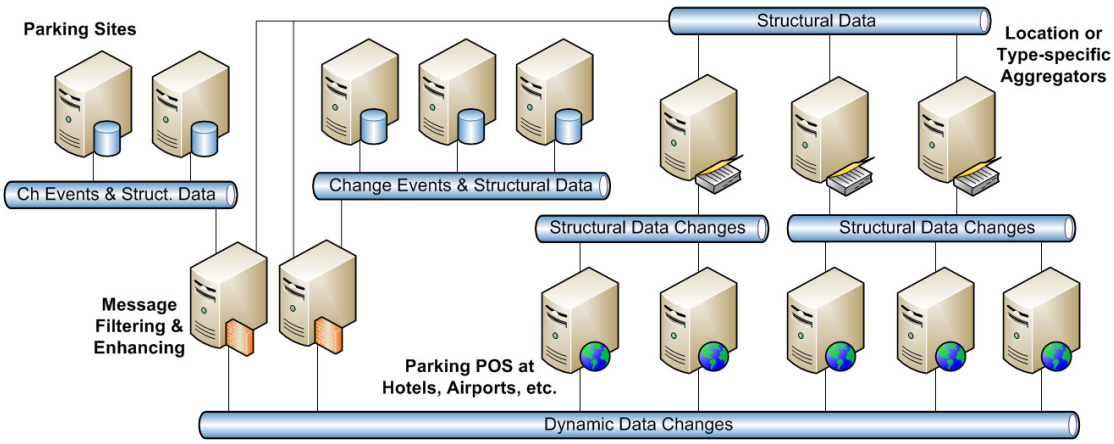


Figure 1.1: A Modern Parking Management System composed of Parking Sites, Message Filtering & Enhancing, Location or Type-specific Aggregators and Parking Point-Of-Sale (POS). *Note:* The icons depict services and not servers. [16]

In addition to the functional requirements, the non-functional requirements also play an important role. Among other non-functional requirements, the system has to be highly available and has to operate with a minimum of response time.

As can be seen such a scenario has to deal with the challenges from the introduction. First of all the system doesn't rely on a central execution controller which coordinates the information exchange between the services. Instead the architecture of the message system defines the structure and information exchange between them. Furthermore the system has to deal with an unpredictable service availability, e.g. services can be disconnected (perhaps through a network failure) or temporarily overloaded. The system must also be able to handle new introduced services, e.g. a new car park is added or to enable load balancing by adding an already existing service multiple times, without disturbing the normal operation of the system nor requiring complicated reconfigurations.

Beside those technical requirements the challenges introduced regarding the design and configuration of such a system must also be considered. It has to be guaranteed that the services consume and provide their data to the right messaging channels. Even if a service is changed, a messaging channel name changes, network addresses change, or other changes to the system occur. Incorrectly performed changes can easily result in situations where Filter services publish to the wrong topic, a POS service subscribes a false topic, multiple POS services use the same reply channel for requests or many other similar inconsistencies may occur.

1.3 Aim of the Work

The goal of this thesis is an approach which supports the architecture and development of distributed message-centric service systems. Specifically, our approach will offer a way to model and describe the high-level architecture of the system by the use of components (representing services) and connectors (representing the message channels). This architecture model can then be enriched with specific configuration details for the message-oriented middleware (MOM) and the service endpoints. A constraint check ensures that the modeled architecture including the provided configuration is consistent and finally an architecture-to-configuration transformation generates the planned connection components for the individual services and the configurations for the message broker. Furthermore the approach will support the propagation of changes in the architecture and configuration to already transformed configuration without altering user-defined code.

This approach, especially the combination of consistency check, architecture-to-configuration transformation and change propagation, will ensure that the planned message-centric service system and finally transformed system will perform as desired. Furthermore the consistency check can help system architects to find architectural flaws as early as possible.

To prove the practical utility of the approach we will present an implementation thereof, which will be finally used to develop a real world service system, to be specific a parking management system as discussed in Section 1.2.1.

Parts of this thesis are published as a full paper at the ICSOC 2014 conference, with the title "Architecture-Centric Design of Complex Message-Based Service Systems" [16]. In the paper we discuss briefly the general problem statement, our approach and the result of the work. The acceptance of the paper strengthens the scientific relevance of this thesis topic.

1.4 Methodological Approach

The methodological approach consists of the following steps:

- First, related work with focus on the design of distributed service-centric systems and architecture-centric software development will be analyzed. We will analyze how a system can be developed by the composition of different services, which methods/technologies exist and what their advantages and disadvantages are. The second part focuses on architecture-centric software development, which is the main concept of our approach. We will discuss what its nature is and what its benefits are. This will then lead us to architecture description languages (ADL), architecture-implementation mapping and message-system consistency.

- Second, we will discuss the design of the approach. For this we will separate the approach into four parts. The first part is concerned with the general consistency of message-based service systems. We will discuss which kind of inconsistencies can occur, how our approach handles them and prevents them. The second part is concerned with the allocation of the message-based system aspects (e.g. message broker configuration, service endpoint configuration, channel names) to the architecture description language (ADL) elements

(e.g. Components, Connectors, Interfaces). After this the bi-directional association of the message-based system aspects and the ADL elements will be discussed and finally how changes can be propagated by using the association discussed above.

- Third, we will introduce the programmed tool that implements our approach. In this part we will first discuss the overall workflow of our system and the extension of ArchStudio 4 that launches the workflow. Then we will specify what information is required for the ESB system and the MOM to generate a messaging system. An allocation of this required information to the xADL elements, on the base of the allocation discussed in the approach, will then make it possible to use xADL to model and configure a message-based system. After the allocation, the consistency checks will be discussed and finally the architecture-to-configuration transformation, including the change propagation which uses the association of the message system aspects and ADL, as defined in our approach.

- Finally, the motivation scenario "Parking Management System", presented in Section 1.2.1, is consolidated to discuss the utility of our approach. To do this we will use the implemented tool support to generate the scenario and present the benefit for a development team.

1.5 Organization

The remainder of this thesis is structured as following:

- Chapter 2 explains important terms and concepts used for the design of distributed service-centric systems and in architecture-centric software development. Additionally, the chapter provides an overview on the current related work of these topics.

- Chapter 3 will stepwise introduce the concept and design of our approach by first discussing general message-based service system consistency challenges. After that the allocation and association of message system aspects to ADL elements will be discussed and finally the change propagation mechanism.

- In Chapter 4 the implementation of the prototype will be discussed. This implementation will use Mule ESB and Apache ActiveMQ to prove the concepts of our approach.

- Chapter 5 evaluates the concepts of our approach and the implemented prototype by using the "Parking Management System" as evaluation scenario.

- Chapter 6 recapitulates the approach and the prototype with a short summary and gives an outlook of our future work.

CHAPTER 2

Related Work

This chapter gives an overview of the terms and concepts that are important for the topic of this thesis. Additionally, important related work to these terms and concepts are presented and analyzed. First of all, we will discuss the design of distributed service-centric systems. We will define what *Service-Oriented Computing* is and what the benefits are, following the concept of *Service Composition* and *Enterprise Application Integration*. Second we will discuss *Architecture-centric Software Development* in a greater detail. We will first analyze the concept of *Architecture-centric Software Development* and then proceed with *Architecture Description Languages*. Finally we will discuss *Architecture-implementation mapping* and *Message-system Consistency Analysis*.

2.1 Design of Distributed Service-Centric Systems

The size and complexity of software programmes increases steadily. Consequently also the development time and costs increase. The concept of Service-Oriented Computing (SOC) [33, 53] offers a remedy, because in it, reusable software elements, called services, are used as fundamental constructs for rapid and low-cost development of distributed service-centric applications [53]. Dustdar and Krämer described services as "autonomous, platform-independent computational elements that can be described, published, discovered, orchestrated, and programmed using standard protocols to build networks of collaborating applications distributed within and across organizational boundaries" [17]. A service can perform easy functions, like a simple addition of two integer values, as well as more complex functions. Furthermore a composition of services can be exposed as a service or may be offered as a new service-centric system [52].

2.1.1 Service Composition

Service composition is the process of solving complex problems by combining and ordering already existing services in a new way [46]. There are two main service composition paradigms for designing and configuring a service-centric system [52, 55, 57]:

Service Orchestration Orchestration is the concept of reusing and composing existing services to build up an executable business process. It uses internal as well as external services to define long-lived and transactional processes [6]. The interaction between the services is coordinated by a central controlling entity which is responsible for the business logic and the task execution order at runtime. Only this entity is aware of the other involved services. To define the order of the service execution and possible conditions under which they are executed, several process-modelling languages are available [18] such as WS-BPEL [1], YAWL [60], or JOpera [54]. In contrast to these languages, which all require a central orchestration service, decentralized orchestration approaches also exist. Those approaches distribute control flow tasks of the central controlling entity among different entities. Thus the central controlling entity, as the single point of failure, is eliminated. Examples of this approach are the works of Nanda et al. [50] and Yildiz et al. [61].

Service Choreography Choreography composes existing services for business collaboration. It describes the peer-to-peer observable interaction between the services and doesn't rely on a central controlling entity [6,57]. Therefore the participating services have to know of each other and their own role in the system. This characteristic includes that the services may have to be altered to fulfil this composition. Examples for choreography definition languages are WS-CDL [37], BPEL4Chor [15], MAP [6], or Let's Dance [62].

The two composition paradigms overlap somewhat, because choreography can be used to exchange messages between different systems that are composed of orchestrated services.

However, both composition approaches use a workflow-like composition where services play fixed roles and have to be highly available. Our work focuses more on composed service systems which rely primarily on more loosely coupled one-way events and less on tightly coupled request/reply style data exchange.

2.1.2 Enterprise Application Integration

Modern distributed service-centric systems consist of many different services and applications. According to Hohpe and Woolf "Enterprises are typically comprised of hundreds, if not thousands, of applications that are custom built, acquired from a third party, part of a legacy system, or a combination thereof, operating in multiple tiers of different operating system platforms" [32]. To handle this challenge, Enterprise Application Integration (EAI) and later Enterprise Service Bus (ESB) were introduced. The idea behind EAI is to provide an integration framework to combine the different, already available, services without requiring significant changes. Linthicum described EAI as "the unrestricted sharing of data and business processes among any connected applications and data sources in the enterprise" [39]. To achieve this, EAI uses adapters and standardized middleware to connect and integrate the services [36]. In general the communication between the integrated services is handled through *Messaging* where each participating service is connected to a common messaging system, normally a message-oriented middleware (MOM), to exchange data. Scheibler and Leymann provided in their work [58] a framework for configuration and execution of EAI patterns, which transform the patterns to platform specific code, namely BPEL.

2.1.2.1 Enterprise Service Bus

Enterprise Service Bus (ESB) is a subcategory of EAI, where the communication between the services is handled over a bus. Clements et al. pointed out in their book [10] that without an ESB the communication between service provider and service consumer is a direct point-to-point communication. Though by integrating an ESB the architecture follows a hub-and-spoke design, where the communication is handled through the ESB system. Therefore an ESB is operating in an intermediary layer between the service consumer and service provider helping routing message between them. To support this it provides utilities for message routing, message transformation, security checks, transaction management and application adapters [9, 10, 44]. In summary, the bus functions as transport facilitator as well as transformation facilitator in a service-centric system that is composed of several disparate services and computing environments [52].

2.1.3 Message Exchange Pattern

At this stage we also have to discuss the common patterns for message exchange between composed services. *Message Exchange patterns* (MEP) are patterns that describe the communication between two or more services. We will only discuss patterns that are relevant for this thesis. The interested reader is referred to the book of Josuttis [34] and to the book of Hohpe and Woolf [32].

One-Way As the name suggests, in this pattern a sending service sends a message to a receiving service without expecting a response message.

Request-Response In contrast to the *One-Way* pattern this one includes a response message. The sending (requesting) service sends a message to the receiving (responding) service, which then sends a response back to the requesting service. Another name for this pattern is *Request-Reply*. A special version of this pattern is the asynchronous non-blocking request-response pattern where the sending service doesn't wait for a response.

Publish-Subscriber The Publish-Subscriber messaging pattern is a pattern where a sender service, called publisher, generates new messages/events and publishes them to the communication medium. The receiving services, called subscribers, are services that listen to that communication medium and if there is a new message, they read it. The difference between a publish-subscriber pattern and a point-to-point connection is that in publish-subscriber the sender (publisher) doesn't directly send the message to a specific receiver (subscriber), instead the subscriber registers his interest in a specific topic and the publisher publishes the message to that topic. For complex service systems this has a significant advantage over a message queue architecture, namely that the services are loosely coupled. The publisher doesn't have to know something about the subscriber, he only has to know anything about the topic that a subscriber listens to [19]. This characteristic can help to build complex systems that are not in need of a centralized execution control, but are still capable of handling unpredictable service availabilities and dynamically fluctuating service instances (e.g. adding or removing services).

2.1.4 Tool Support

Several tools that help to design distributed service-centric systems exist. In this section we will focus on the tools that are important for this thesis. There are two subsections of tools/technologies: (i) tools/technologies for the communication between the services and (ii) tools/technologies for the general service composition.

2.1.4.1 JMS

Java Messaging Service (JMS) API is a message-oriented middleware API based on Java. It is a messaging standard that allows applications to create, send and receive messages using reliable, asynchronous and loosely coupled communication [51]. Therefore most modern ESB systems use JMS-based middlewares as their messaging infrastructure [44]. It supports two types of communication channels, point-to-point and publish-subscriber. Point-to-point is a message queue which connects two participants together. Each receiving participant has an incoming queue which retains received messages until they are consumed or the message time expired. The publish-subscriber channel uses the already discussed *Publish-Subscriber* message exchange pattern where the message gets published to a particular message topic.

Several implementations exist like Apache ActiveMQ [3], OpenJMS [4], HornetQ [30] and many more. For this project we use the popular **Apache ActiveMQ** message broker.

2.1.4.2 ESB

Over the last few years, ESB has become more and more popular, so there are several different platforms [26, 35] like Mule ESB [48], OpenESB [45] or Fuse ESB [31].

We decided to use **Mule ESB** because it is a modern, often used and reliable ESB system. It is open-source with an optional commercial support and additional commercial tools. It defines process workflows that can be enriched with different message processing components and message endpoints. Endpoints at the beginning and end of the workflow define the message interfaces of the workflow. It uses XML based configuration files to describe the internal structure of a a workflow and how the workflow components are configured and connected together. For the underlying messaging infrastructure different JMS implementation, including Apache ActiveMQ, are supported.

2.2 Architecture-Centric Software Development

To define what *architecture-centric software development* is, we first have to define what *software architecture* is. Software architecture is concerned with the high-level architectural view of a system. To define the architecture of a system, coarser-grained architectural elements rather than lines-of-code are used [43]. Specifically, an architecture describes a system by a set of components and connectors, their configuration and the interaction among them [59]. Software architecture represents a method to document the system structure and is therefore an important planning facility, especially in large complex systems. It helps the development team to focus on the "big-picture" of the system, document principal design decisions, provide a common ground

for discussion, support the reuse of components between different projects and reduce development costs [7, 10, 42, 59]. In addition, software architecture provides different analysis methods to build more reliable, scalable and portable systems [27].

Architecture-centric software development is a development method where the software architecture is a key element in the software development process. "Architecture-centric development emphasizes that software architecture, instead of being a documentation artifact that is peripheral to code development, should play an essential role throughout the software development lifecycle" [63]. The goal of architecture-centric software development is to use the software architecture as the blueprint of the software system and use this blueprint to generate code fragments out of the architecture. This is, in fact, the general idea of our approach. The generation/transformation of the code fragments and also the propagation of changes is often supported by tools. In their paper [64] Zheng and Taylor illustrate that architecture-centric software development can be separated into four variants. Those variants are *Architecture refinement*, *Framework and middleware-based development*, *Architecture description language tool support* and *Domain-specific software architectures (DSSA)*. *Architecture refinement* is an approach where an abstract higher-level architecture is mapped to a concrete and finer architecture. *Framework and middleware-based development* provides programming constructs and architecture concepts. In the variant *architecture description language tool support*, architecture description languages (ADL) are used to describe, model and configure a system. This modelled architecture then can be used to generate code fragments. *Domain-specific software architectures (DSSA)* uses already available domain knowledge and promotes it to a high-level of abstraction. For our approach we use *Architecture description language tool support*, because it allows us to describe the architecture of a message-based service system in a way that it can be analyzed and transformed to code/configuration fragments.

2.2.1 Architecture Description Languages

Architecture description languages (ADLs) support architecture-centric software development by providing notations and tools with which the architecture of a system can be modeled, configured and analysed. By now, a wide range of different ADLs exists. Prominent ADL examples are Darwin [41], Rapide [40], Wright [2], Acme [29] and xADL [12, 14]. All of them have different varying abilities, nevertheless most of them use Components, Connectors, Interfaces, Links and Configurations to model an architecture. Both the paper of Medvidovic and Taylor [43] and the book "Software Architecture: Foundations, Theory, and Practice" [59] from Taylor, Medvidovic and Dashofy present a classification and comparison of different ADLs. In this research, we will focus on the language xADL 2.0 [1]. We chose this language because of its high extensibility and available tool support. In the following we will discuss xADL in detail, for a description of other ADLs the interested reader is referred to corresponding papers.

2.2.1.1 xADL

xADL is an architecture description language that uses XML and XML-schemas to model and configure system architectures. An xADL document has to be a well-formed and valid

[1] Throughout this thesis any reference to xADL, is referring to the version 2.0.

XML-document. To guarantee that an xADL document is valid, a set of XML-schemas was designed [12]. Specifically, the XML-schemas for xADL are defined in a modular design approach where each logically separable part is a separate module and adds a set of attributes to the language. The modules combined represent the language xADL 2.0. In addition to the existing ones, new modules can be added by extending already existing modules or define new ones. Figure 2.1 shows the current set of xADL schemas and their relationship to each other. The figure is excerpted from [14]. The modules *Structure & Types* and *Abstract Implementation* are directly related to this thesis and will therefore be discussed in detail. The interested reader will find a detailed description of the the additional modules in the work of Dashofy, Hoek and Taylor [12].

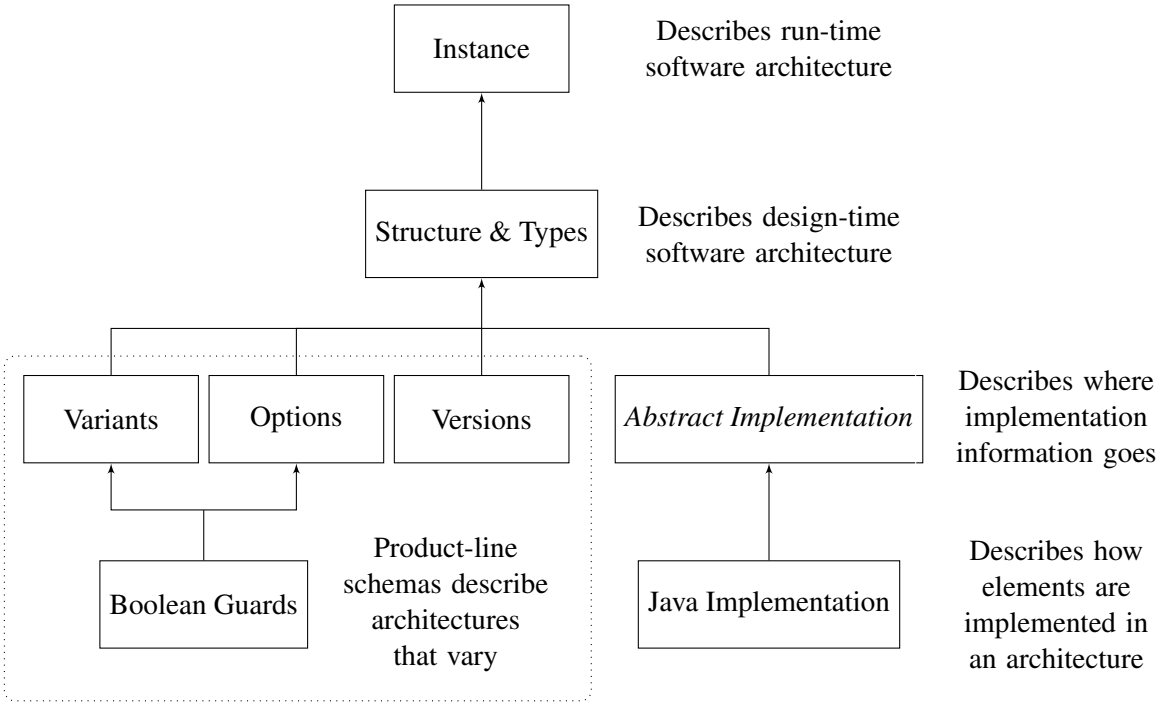


Figure 2.1: xADL XML schemas and the relation between them [14].

To model the system architecture, xADL uses a *Structure & Type* schema [12]. This schema provides the elements *Component*, *Connector*, *Interface*, *Links*, *ComponentType*, *ConnectorType* and *InterfaceType*. The elements *Component*, the loci of computation, and *Connector*, the loci of communication, are used to represent the instances of the system at the design-time and the element *Interface* is used to set interfaces on the components and connectors. *Links* are the wiring between the components and connectors, respectively their interfaces. *Component* and *Connector* together with the wiring between them, represent the architectural topology of the system. *ComponentType*, *ConnectorType*, *InterfaceType* are used to assign types to the structure elements. Structure elements, of the same kind (*Component*, *Connector*, *Interface*), with the

same behavior or implementation can share a type element to reason about the similarity among those elements. The component types and connector types expose *Signatures* which also refer to an *InterfaceType*. Each interface of a component or connector refers to a signature of the corresponding type. To enrich the architecture with implementation details, they can be mapped to the individual architecture elements. For this, xADL uses an *abstract implementation* schema as an abstract entry point to include concrete implementation to the elements. This abstract entry point can be extended with a concrete implementation technology. Natively, xADL has the ability to define a Java-based implementation for the elements, which can be deployed on a single JVM.

2.2.2 Architecture-Implementation Mapping

The term "architecture-implementation mapping" describes the process of mapping the defined architecture to implemented code and vice versa. This can be divided into two separate parts: *one-way mapping* and *two-way mapping*. One-way mapping is used when the change propagation is uni-directional, namely from architecture to implementation. This implies that the complete implementation of the program is generated from the architecture and all changes have to be done in the architecture and then propagated to the implementation. Two-way mapping, on the other hand, is bi-directional. This means that changes are propagated from architecture to implementation and vice versa.

In the work of Zheng and Taylor they present a third mapping approach called *1.x-way mapping* [63]. Their approach only allows the propagation of changes from architecture to implementation. But in difference to one-way and two-way mapping, *1.x-way mapping* separates the architecture-prescribed code from user-defined code. This means that each architectural component is divided into architecture-prescribed code, code that is generated and altered by supporting tools, and user-defined code, code that is manually programmed by a developer and therefore can only be changed by him. To achieve this behavior, the proposed system uses *deep separation* where generated and non-generated code is divided in different functions or classes. This has the benefit that it prevents mistaken changes, both structural and behavioral, of architecture-prescribed code by a programmer. Programmers are only allowed to do changes in the user-defined section of each architecture component. By using this method most of the architectural changes, like configuration and behavioral changes, can be mapped automatically to the code, respectively the architecture-prescribed code, by an architecture-based code regeneration mechanism. Additionally all architectural changes are recorded and classified into an architectural changes model. Nevertheless some changes also need modifications in the user-defined code. To address this concern architecture change notifications are generated and sent to the responsible user-defined code programmer if modifications of this code are necessary. These notifications contain detailed information about the changed architecture elements so the programmer can adapt the user-defined code. They proved their approach by providing a tool for mapping between an architecture and its underlying Java implementation.

2.2.3 Message-System Consistency Analysis

Consistency analysis intends to ensure that the planned architecture doesn't violate predefined constraints. Therefore, consistency checking plays an important role in architecture-centric software development and gets even more important if the system is composed of highly decoupled components. Even if the verification of each component in an isolated test system is easy, the verification of the overall system can be a challenging task. The high decoupling of the components also furthers the individual development of the component. This will lead to an increasing complexity of the verification process.

One way to do consistency analysis is a bottom-up approach where the structure of a message based system is created from the implemented systems. With this approach, a higher level view of the message system, contrary to the low level code view, can be generated. This higher level view contains only the necessary information of the message system and therefore it is easier to find inconsistencies in the implemented message system. Two popular examples are presented in the papers [25] and [38]. Garcia et al. present in their paper [25] a static analysis tool, called Eos, that can identify message types and message flows within an already existing distributed event-based system (DEB). Their approach is able to analyse Java and Scala code to gather message information. The aim of the tool is to support the development team in maintaining a DEB system by identifying the scope and impact of required changes. The work of Lee et al. [38] introduces a visualization and analysis tool for DEB system, called ViVA (Visualizer for eVent-based Architectures). The tool can visualize the message exchange between the components of a DEB system. This visualisation can help engineers understand the message flow in a clearer way. The tool uses the already discussed static analysing tool Eos from Garcia [25] to analyse the message information.

Another approach for consistency analysis of message based systems is the use of a model checker. Garlan et al. described in their paper [28] a generic, parametric publish-subscriber model checking framework. Their approach captures publish-subscriber run-time event management and dispatch policy and models them in a reusable, parameterized state machine model. This state model is then checked by the system. Caporuscio et al. present in their paper [8] a way to compositionally model checking middleware-based software architecture descriptions. In contrast to the discussed work of Garlan et al. [28], where the verification requires the whole composite system to be modelled, the focus of Caporuscio is on compositional verification. The approach uses an assume-guarantee methodology to reduce properties verification from global to local. In the paper [5], Baresi et al. present an approach that uses a model checker for the fine-grained verification of systems with a publish-subscriber architecture. For the automatic verification they extended the input language and the verification engine for the model checker Bogor [56]. Their approach offers the possibility to evaluate different publish-subscriber guarantees, like message reliability, message ordering, message filtering, message priorities and message delays. All presented checks require that detailed information about the internal behavior of the components are known and modelled in the model checker.

Those approaches are not suitable for our use, because (i) our approach only considers the structure of the message-system, we don't assume knowledge of the service internal behavior, and (ii) the listed analysis methods are significantly more fine-grained than our purpose requires.

2.2.4 Tool Support

Tool support is an important decision factor for the choice of an ADL. Tool support can be divided into two parts: (i) support that helps during the development of new tools, those are tools that are used by tool builders, and (ii) tools that use the language, those are tools that are used by software architects. In the following we will discuss the available tools for the language xADL 2.0.

2.2.4.1 xADL Infrastructure Tool Support

As discussed before, the architecture description language xADL offers different kinds of supporting tools [13, 14]. We will now discuss two tools that support the development of new tools for xADL.

Apigen This is a tool which automatically generates new data binding libraries out of given xADL schemas (for detailed information about the xADL schemas see Section 2.2.1.1). Data binding libraries map XML elements and attributes to code, in the case of Apigen into Java objects. This tool has the benefit that there is no need to manually rewrite the data binding libraries each time a schema is added, changed or removed. If changes of the schemas are necessary the changes are done in the corresponding schema documents and passed to the Apigen tool which generates or, if they already exist, modifies the data binding libraries. Those new data binding libraries then offer the new object-oriented interfaces to work with the xADL documents.

xArchADT The previously discussed data binding library provides object-oriented interfaces to edit xADL documents. This assumes that the tools that are used to edit an xADL document share an address space. But generally in distributed and event-based systems this is not the case and therefore the use of such libraries is difficult. To address this concern xArchADT was developed. xArchADT is a wrapper around the data binding library that provides an event-based interface. It provides a "flatten" interface where each xADL element gets an identifier that is used to refer to the element. If the architecture is changed xArchADT emits an event to inform all tools that are listening for changes.

These tools help to build new tools for working with xADL. Because of these tools and the extensibility of xADL, it is a perfect choice for investigating new architectural approaches, as we do here.

2.2.4.2 ArchStudio 4

ArchStudio 4 is a development environment for software architecture modelling and metamodelling that uses xADL 2.0 as the primary modelling notation [11]. It is integrated within the Eclipse platform as a plug-in extension. The ArchStudio environment is composed of different tools that help architects to visualize and analyze architecture models. Among others, these tools are *Archipelago*, *ArchEdit* and *TypeWrangler*.

Archipelago This is the graphical editor of ArchStudio, which enables the display of the structure of an xADL 2.0 architecture in a boxes-and-arrows format. Archipelago provides a user-friendly point-and-click interface to edit the architectural structure. This interface allows addition or removal of components, interfaces and links to and from the model. Archipelago accesses the architecture description through the already discussed xArchADT. Consequently all changes that are done in the editor are immediately reflected in the underlying xADL document and all tools that are listening for changes are informed about the change. In addition to the structural modelling, it also offers a basic statechart and interaction editor for the behavioral modelling of the architecture.

TypeWrangler As discussed in section 2.2.1.1 xADL uses a structure & type schema to model and configure the elements of the architecture. The tool TypeWrangler can be used to do the mapping between the structure element (e.g. Component) and the corresponding type (e.g. ComponentType). For this it offers an easy to use graphical interface. Again, TypeWrangler is an event-based software component that uses xArchADT to access the architecture description.

ArchEdit This is a tool which represents the architecture description in a clear tree format where each node can be expanded, collapsed or edited [14]. In other words, it displays the tree structure of the xADL document or, more specifically, the XML schema of it, in a graphical tree structure. This gives the architect of the system direct access to the architecture description. If the xADL schema is altered (e.g. new features are added to xADL) the tool doesn't have to be altered, because ArchEdit builds its view and interfaces directly from the XML schemas. Also, this tool uses xArchADT to access the architecture description.

CHAPTER 3

Approach

In this Chapter, we present the approach of our work. At the beginning we describe the general design principles of our approach. Afterwards we give an overview of the idea behind the approach, including the workflow and the overall structure, which is split into three main components: *Architecture-level Editor*, *Consistency Checks* and *Architecture-to-Configuration transformation*. Then we will introduce a small example which will be used to discuss and explain the concepts of our approach step by step. After this we will analyze general consistency problems which may occur during the design of message-based service systems and how they can be prevented. The next section will first analyze how the message system aspects can be allocated on different xADL elements, finishing with the final decision on how our approach distributes them among the xADL elements. Furthermore, this chapter will analyze how the mapping between the message system aspects and the xADL elements can be achieved. The last section will discuss the problems that may occur if changes have to be propagated and how our approach deals with them.

3.1 Design Principles

As stated in Section 1.3, the aim of this work is to support the architecture-centric development of distributed message-centric service systems. To achieve this goal, the core design principles of our approach are:

- For architecture-centric development, the support should start at the beginning of the development process and should be present during the whole development lifecycle. The first steps in an architecture-centric development are concerned with the planning of the system or at least parts of the system. A supporting system has to support all development phases and therefore the support has to start during the first planning phases.
- The consistency of the messaging system should be guaranteed at all times. It can be relatively simple to guarantee the consistency of a simple service oriented system, however

with increasing complexity of the system, the complexity of the consistency checks also increase. In addition to the checks that will be done during the first planning phase, the consistency also has to be guaranteed if the architecture of the system changes during the development. Those changes may work for a particular part of the system but can leave the overall system in an inconsistent state. This, for example, can happen when an engineer who is concerned with the change may not have enough information about the overall system, especially in a bigger development team.

- The message routing aspects of the system should be separated from messaging processing aspects. For a tool that does architecture-to-configuration transformation and change propagation, a clear separation of the generated code and the user-defined code is mandatory. This separation should reflect the responsibilities of a software architect and a software developer. The architecture team members of the system are responsible for message routing code and the development team members for the message processing code.

- Messaging interfaces that are designed and configured by the system architecture should be automatically generated as a messaging routing skeleton for the implementation. To support an architecture-centric development, the tool should be able to transform the planned messaging system architecture to a messaging routing skeleton. This skeleton has to include all information that is provided through the architecture document. During the development this generated system will then be enriched with invocation-centric message processing.

- Architectural changes should be propagated completely to the implementation. Changes, including altering already planned and implemented components as well as adding new components, can happen at every stage of the development process. These modifications have to be propagated to the code without altering the other parts of the implementation.

3.2 Overview

Our approach is an architecture planning and configuration tool for complex service systems. Furthermore constraint checks ensure that the architecture of the system is consistent before the messaging interface implementations, including the MOM configuration, will be automatically generated. Figure 3.1 represents the workflow of our approach.

In the first step, the architecture team plans the high-level architecture of the message-based system by using a graphical boxes-and-arrow based editor, in which the services are represented by architecture-level components and the message channels, between the services, by architecture-level connectors. Each architecture-level component and connector combines several architecture-level interfaces, which represent the interaction points among the services via messages. Links between interfaces of different components and connectors represent the message path. The second step is concerned with the configuration of the architecture elements. Each component, connector and interface can be configured with specific properties like connection addresses, communication directions (in, out, in-out), service specific details, etc. In the third step, a consistency check algorithm will search for inconsistencies in the architecture and

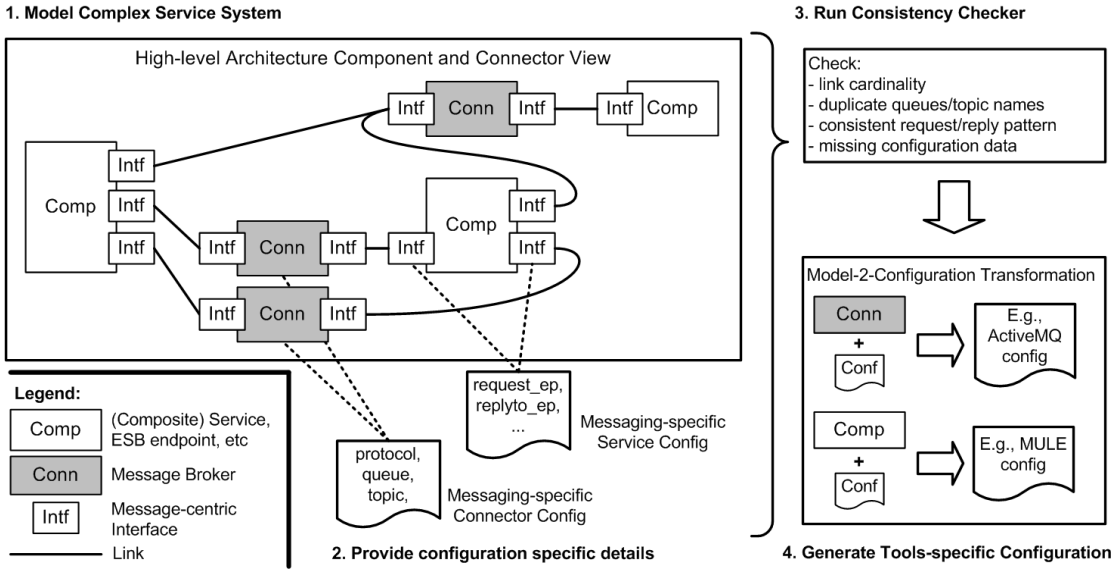


Figure 3.1: Workflow of our suggested approach [16].

in the configuration. It will iterate through all message system relevant components, connectors, interfaces and links and will check the overall topology for architectural flaws as well as configuration flaws of each element. In the last step, the consistent architecture together with the configuration details will be used for a model-to-configuration transformation where a messaging skeleton of the planned system will be generated. The transformation will generate on the one hand the configuration for the MOM, by transforming the connectors including the message-centric connectors configuration, and on the other hand the service interface implementations, by transforming the components including the message-centric components configuration. To provide a stable basis for further development steps, especially the development of the message processing aspects of the services, our approach will generate service interfaces for an Enterprise Service Bus (ESB).

To support the development process during all phases, the steps shown in the figure can be used in an iterative fashion. In other words, any step can be done and repeated at any state of the development process. Changes in the architecture will be propagated, after the complete system passes the consistency checks, to the already existing message-based system without changing user-defined code.

As depicted in Figure 3.2, our approach is composed of three main components, represented as ovals. The first part is the *Architecture-Level Editor*, with which the high-level architecture of the system can be modelled and configured. For expressing the architecture we will use the highly-extendable Architecture Description Language xADL 2.0. The output of the architecture-level editor, the architecture document in xADL, will then be used by the *Consistency Check* to check all message system relevant elements for consistency. If the provided architecture is consistent, then the architecture document will be used for the *Architecture-to-Configuration transformation* where the implementation or, more specifically, configuration, (represented by a

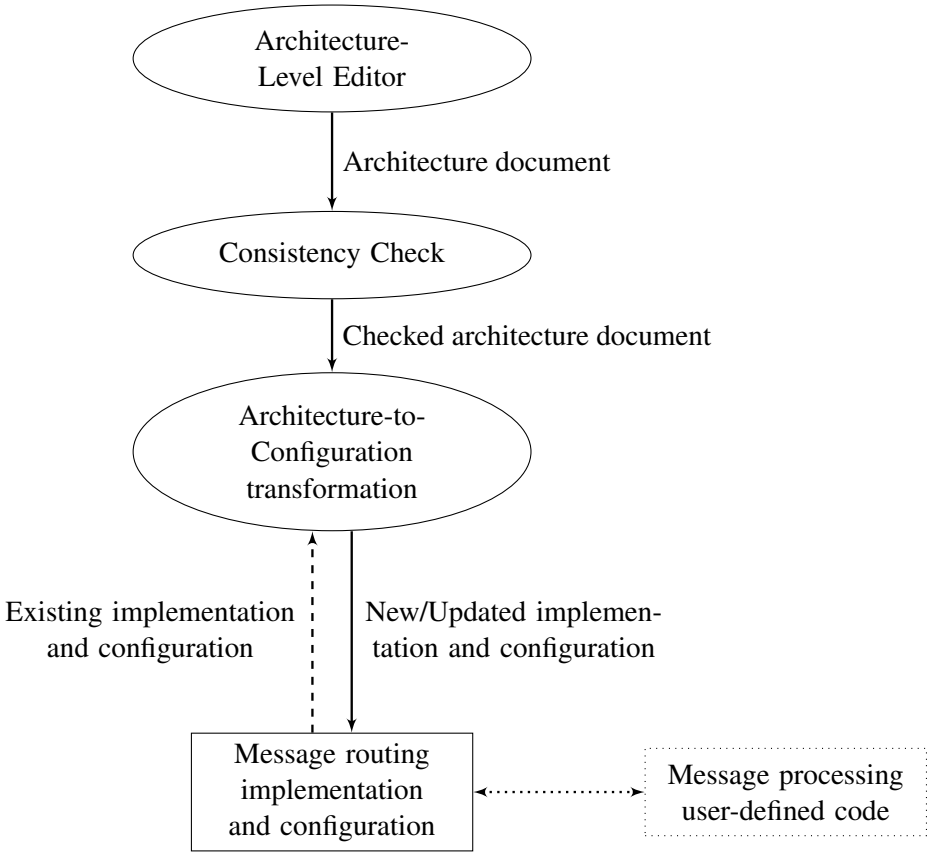


Figure 3.2: Structural overview of the approach

rectangle) of the message system skeleton will be generated. If the implementation/configuration already exists the transformation will read the available parts and change them. An important aspect of our approach is the separation of the architecture-prescribed implementation and configuration message routing code and the user-defined message processing code. Our system will only provide the ability to generate the message routing code. The message processing implementation is then the task of the development team, which is why it is depicted as a dotted box and connection.

One benefit of publish-subscriber is that it is able to handle a dynamically changing number of service instances, nevertheless our approach will target event-based systems with a rather stable set of publishers and subscribers. This won't be a restriction for most message-centric systems because most of them don't have an unpredictable number of services. Of course, our system will be able to handle unpredictable service availability, especially because of the use of proven technologies like ESB and MOM. But at least at the development stage, the number of publishers and subscribers and their type has to be known.

3.3 Ongoing Example

In the remainder of this thesis, we will use this example to help the reader understand the discussed methods and technologies in a clearer way. The examples will be presented in grey boxes.

This example represents a service system containing three services. The purpose of the system is that a user can submit and start time-consuming tasks. These tasks will then be executed through another service and the results will be sent back to the requesting service. In addition to the *execution service* a third service will log all submitted tasks. Figure 3.3 shows the interaction between the three services.

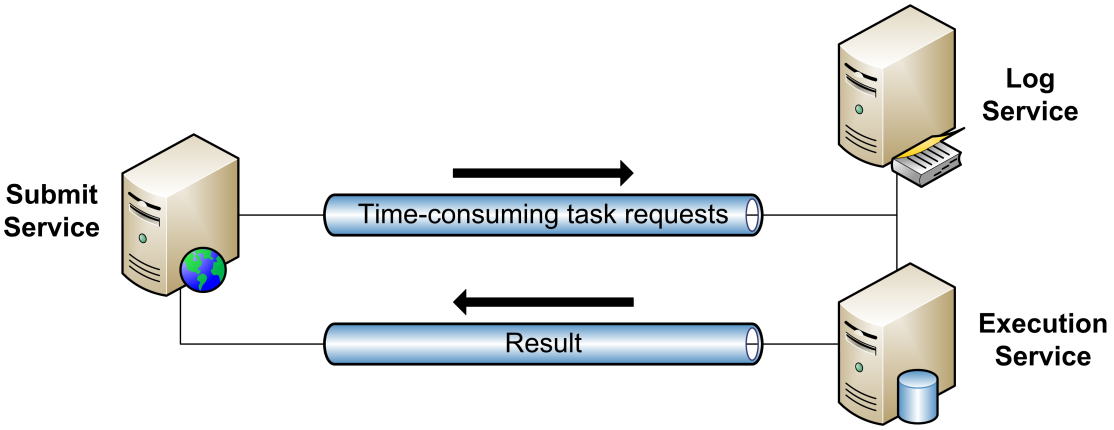


Figure 3.3: Three service system with a publish-subscriber and one point-to-point connection.

Submit Service This is the publisher service and is responsible for the communication with the user. The service is a web-service where the user can submit and start a new task. The new task will then be published to a publish-subscriber channel. After sending the task to the publish-subscriber channel, it will wait for the response on the point-to-point connection and will display it.

Log Service This service subscribes the publish-subscriber channel and gets all the published messages on the channel. It is responsible for logging all submitted tasks.

Execution Service This service also subscribes to the publish-subscriber channel, but in contrast to the *Log Service*, it is responsible for executing the tasks. When a task is accomplished, the service will send the result over a point-to-point channel back to the *Submit Service*.

Time-consuming task requests This is the publish-subscriber channel that is used to send tasks from the *Submit Service* to the *Log*- and *Execution Service*.

Result is the point-to-point channel that is used to send the task results from *Execution Service* back to *Submit Service*.

For the message transport, one message broker should be used and for a high reliability of the system, regardless of service or server losses, durable channels should be used. This implies that the message broker has to persist the messages.

3.4 Message-Based Service System Consistency

During the planning and development of a complex message-centric service system, it can be difficult to ensure the overall consistency of the system and the message broker configuration. Especially if the structure and configuration of the system gets modified during the development, e.g. changes in the message flow or additional services could be integrated. As discussed previously, the changes could be proper for a particular part of the system, but could leave the overall system in an inconsistent state. In the following we will discuss the possible inconsistencies of a message-based service system.

The inconsistencies can be separated into Architectural level and Component level.

3.4.1 Architectural Level Inconsistencies

Architectural level inconsistencies are flaws in the architecture of the messaging system. In the following, we will discuss some common mistakes that can happen during the message-centric system development. Most of them have their origin in the configuration of the channel names, which has to be defined in each endpoint separately. As we will see this can be very error-prone, e.g. a channel name can be written incorrectly or forgotten to defined.

1. **Inconsistency:** Two different channels have the same channel name.
   Description: This inconsistency has the result that it is unclear which channel will be used for sending a message and therefore it is uncertain which service will receive it.

   Ongoing Example 1: Two different connectors have the same channel name.

   For our ongoing example this would be the case if both channels, “Time-consuming task requests” and “Result”, have the same name. For the functionality it won’t be a problem, because the type of channel (publish-subscriber and point-to-point) is different, but the architecture will become unclear.

2. **Inconsistency:** Two interfaces that are connected over a message channel, have the same direction.
   Description: If two interfaces are connected and are not bidirectional, they must have different directions, one has to be set to sending and the other one to receiving a message.

   Ongoing Example 2: Two connected interfaces have the same direction.

   In our ongoing example the interface of the *Submit Service*, which is listening to the point-to-point channel “Result”, can be set to a sending interface as opposed to a receiving interface. If this is the case, the *Submit Service* wouldn’t be able to receive

a result from the *Execution Service*.

3. **Inconsistency:** A messaging service endpoint isn't connected to a channel, i.e. the channel name isn't set in the endpoint.
   Description: Each messaging service endpoint needs the name of the connected channel. If the channel name is not set, the service won't be able to receive a message. This could be intentional, because for example it will be developed at a future date, but it is also possible that it has been forgotten to add the channel name.

   Ongoing Example 3: A messaging service endpoint isn't connected to a channel.

   For example, the *Log Service* doesn't subscribe the channel "Time-consuming task requests" because the channel name has not been added to its endpoint. If this is the case the *Log Service* wouldn't receive any messages over this channel.

4. **Inconsistency:** Misspelling in the channel name of a messaging service endpoint.
   Description: If the channel name is misspelled, the service won't be able to receive a message because it will listen to the wrong or not existing channel.

   Ongoing Example 4: Misspelling in the channel name of a messaging service endpoint.

   For example, the *Execution Service* could listen to the publish-subscriber channel "Time-consuming" instead of listening to "Time-consuming task requests". If this is the case, the *Execution Service* would never receive a message.

5. **Inconsistency:** A publish-subscriber channel is used for publishing messages within the scope of a request-reply pattern.
   Description: This could lead to several response messages, because each subscriber could reply to the request. This could be intentional but then the receiving service has to handle these multiple replies.

   Ongoing Example 5: A publish-subscriber channel is used for publishing messages within the scope of a request-reply pattern.

   In our ongoing example the publish-subscriber channel "Time-consuming task requests" is used for a request-reply pattern, but in the example only the *Execution Service* is responding to the *Submit Service* and so it only receives one result. If the *Execution Service* also sent results back to the *Submit Service* via the channel "Result", then this has to be considered in the development of the *Submit Service*.

6. **Inconsistency:** Several services are listening to a point-to-point connection.
   Description: As the name suggests a point-to-point connection is a connection between only two participants. If there are several services listening to a new message on the same point-to-point it can't be guaranteed who will receive the message. In the end, the

behavior of the point-to-point depends on the MOM implementation, but in general only one participant will receive the message. Again this can be intentional, e.g. for load balancing systems.

> **Ongoing Example 6: Several services are listening to a point-to-point connection.**
>
> If the channel "Time-consuming task requests" is configured as a point-to-point connection only the *Execution Service* or the *Log Service* will receive the message from Service the *Submit Service*.

3.4.2 Component Level Inconsistencies

Component level inconsistencies are flaws in the configuration of each message component. Once again, the following examples will discuss some common component level inconsistencies.

7. **Inconsistency:** Two endpoints, connected to the same channel, are configured for different channel types.
   Description: E.g. the sending endpoint is configured for point-to-point and the listen endpoint is configured for publish-subscriber. With this inconsistency the listening endpoint won't receive any messages.

   > **Ongoing Example 7: Two endpoints connected to the same channel are configured for different channel types.**
   >
   > In our ongoing example this can happen to the "Time-consuming task requests" channel, when the sending endpoint, of the *Submit Service*, uses the channel for a point-to-point connection and the listening endpoint, of the *Log Service*, is configured for publish-subscriber.

8. **Inconsistency:** The connection configuration of the interfaces is inconsistent.
   Description: For example this can be an incorrectly configured IP-Address of the MOM or a false port.

   > **Ongoing Example 8: The connection configuration of the interfaces is inconsistent.**
   >
   > If for example, the sending endpoint of the *Submit Service* has a wrong IP-Address of the message broker, it won't be able to send a message to the channel or, more specifically, to the *Log-* or *Execution Service*.

9. **Inconsistency:** The request-reply service endpoints are not defined correctly.
   Description: For the request-reply pattern the replying service has to know where the answer should be sent to. This can be defined static or dynamic. In the static way the answer is sent to a predefined channel. For the dynamic way, the message has to be enriched with the name of the reply channel. This is the responsibility of the requesting

endpoint. On side of the requesting service the receiving endpoint has to listen to this channel for reply messages.

Ongoing Example 9: The request-reply service endpoints are not defined correctly.

For the reply message a dynamic allocated channel should be used. The requesting endpoint on the *Submit Service* will enrich the message with the reply point-to-point channel "Result". After the calculations on the *Execution Service* are done, it will send the results back to the channel that is defined in the message from the *Submit Service*. All four involved service endpoints (requesting endpoint on the *Submit Service*, receiving endpoint on the *Execution Service*, replying endpoint on the *Execution Service* and receiving endpoint on the *Submit Service*) have to be configured specifically for the request-reply pattern.

3.4.3 Discussion: Message-Based Service System Consistency Checking

During the development of a system, inconsistencies in different parts of the system cannot be avoided. Those inconsistencies can happen at the first planning stages as well as in later implementation phases. While during the initial planning phases most inconsistencies can be found easily and repaired fast, in later phases it can be difficult to find them and even harder and more expensive to fix them. Consequently finding and resolving inconsistencies as soon as possible is an important factor for saving both time and money.

Our approach will use *consistency checking* to help the architecture and development team preventing inconsistencies as early as possible. As mentioned previously, the consistency checks can be initialised at every development stage for the best possible inconsistency prevention. Our algorithm will use the provided xADL architecture document and will iterate through all message-centric system relevant xADL elements to find flaws at the architecture level as well as at component level. The algorithm will then declare several errors and warnings, including recommendations for how to solve the inconsistencies. Table 3.1 discuss how our approach supports the architecture and development team by detecting and preventing the inconsistencies discussed in Section 3.4.1 and 3.4.2. The table shows how the consistency is guaranteed and if our algorithm outputs an error or a warning for that inconsistency.

3.5 Allocation of Message System Aspects to ADL Elements

As already discussed our approach uses the architecture to define the message routing aspects of the system. This includes the definition of the architecture of the high-level overall complex service system as well as the configuration of the message-oriented-middleware (MOM). To achieve this, our approach uses the highly-extensible architecture description language xADL 2.0. It is an architecture description language that uses XML and XML-Schemas to model and configure system architectures, as discussed in Section 2.2.1. To model the system architecture, xADL uses a *Structure & Type* schema containing the elements *Component*, *Connector*, *Interface*, *Links*, *ComponentType*, *ConnectorType* and *InterfaceType*. The Components are the

No.	Consistency Check	Error / Warning
1.	Compare all channel names for multiple channel name use.	Error
2.	Check the direction of all connected endpoints.	Error
3.	As already said we only analyse message-centric system relevant parts. If an endpoint isn't connected to another endpoint it isn't message-centric system relevant. This has the benefit that an architect can add elements that will be developed at a future date.	-
4.	Can't happen, because the architecture-to-configuration transformation will generate the service endpoints with the configuration of the MOM.	-
5.	Check if a publish-subscriber channel is used with the connection configuration of a request-reply pattern.	Warning
6.	Check all point-to-point connections for several connected service endpoints.	Warning
7.	Can't happen, because the architecture-to-configuration transformation will generate the service endpoints with the configuration of the MOM.	-
8.	Can't happen, because the architecture-to-configuration transformation will generate the service endpoints with the configuration of the MOM.	-
9.	Check if all including endpoints are set correctly. We can only check if they are set correctly if at least one endpoint is configured for the request-reply pattern.	Error

Table 3.1: Checks to support the architecture and development team, by finding inconsistencies. (Numbers refer to inconsistencies discussed in Section 3.4.1 and 3.4.2)

locality of computation and the Connectors are the locality of communication. Each Component and Connector can have several Interfaces with which the elements can be connected over Links. The Component and Connector, together with the wiring between them, define the structure of the system, the elements *ComponentType*, *ConnectorType* and *InterfaceType* are used to assign types to the structure elements. Each element can specify concrete implementation details. To accomplish that, xADL uses an *abstract implementation* schema as an abstract entry point. This abstract entry point can be extended with a concrete implementation technology. Figure 3.4 presents the relation between structure, type and implementation.

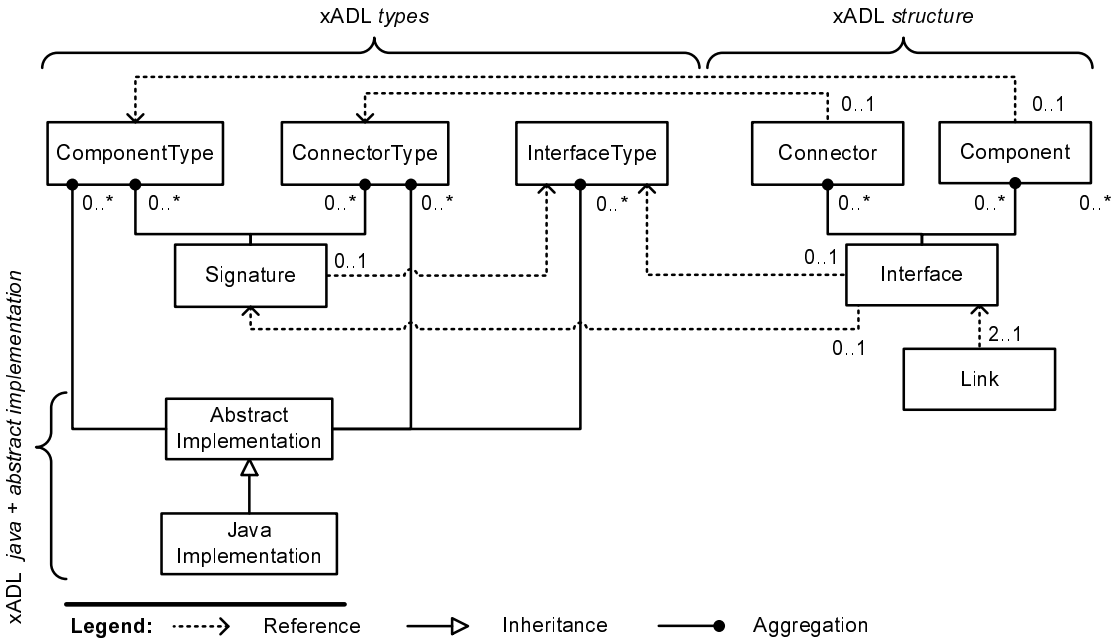


Figure 3.4: Relation between the structure, type and implementation elements [16].

In the following section we discuss the extension of xADL that has to be done to store the configuration of the message-centric system in xADL.

3.5.1 Message-Centric ADL Extension

To configure a message-centric system we had to extend xADL in a way that it can support an architect by planning the four core architecture elements of a message-centric complex service system. Those are (i) the *MOM*, (ii) the message *channels*, (iii) the *service endpoints* and (iv) the *services* that combines related service endpoints. To reach this goal we will extend xADL so the four core architecture elements can be configured within it. We defined three different variants to distribute those elements among the xADL elements, depicted in table 3.2. In the following we will discuss those variants and will choose the one that will be used for our approach. In addition to the configuration of the four architecture elements our approach also requires the ids (see Section 3.6) and the paths to the output files. We divided each of the four core architecture elements into three parts (Section 3.5.1.5 will discuss the elements in greater detail):

[Service | Service endpoint | MOM configuration] id Defines the id that is used for the mapping between the architecture element and the implementation/configuration element (see Section 3.6 for details).

[Service | MOM configuration] file path Defines the path of the implementation/configuration file. Each implementation/configuration of an architecture element can be held in a separate file, defined by this attribute.

[Service | Service endpoint | Channel | MOM] configuration Contains the specific configuration attributes of the elements.

	Variant 1	**Variant 2**	**Variant 3**
Component	Service id	Service id Service file path Service configuration	Service id Service file path Service configuration
ComponentType	Service file path Service configuration		
Signature	Service endpoint id Service endpoint configuration		
Interface		Service endpoint id Service endpoint configuration	Service endpoint id Service endpoint configuration
InterfaceType			
Connector	Channel configuration	Channel configuration MOM configuration file path	Channel configuration
ConnectorType	MOM configuration id MOM configuration file path MOM configuration	MOM configuration id MOM configuration	MOM configuration id MOM configuration file path MOM configuration
Note: [Service | Service endpoint | MOM configuration] id: Id for change propagation. [Service | MOM configuration] file path: Path to the Service and MOM configuration file. [Service | Service endpoint | Channel | MOM] configuration: Special configuration values.			

Table 3.2: Different variants to distribute the configuration information in the xADL Structure & Type schema.

3.5.1.1 Variant 1

Service
This variant uses the ComponentType to configure the services. The Components, which have the same ComponentType, are instances of the same service. Because they receive their configuration from the ComponentType, each service gets a unique id, the id of the assigned Component, to separate it from the other services.

Pros:

+ If many identical Components are needed, only a few ComponentTypes have to be created.

Cons:

- If many different Components are needed, many ComponentTypes have to be created.
- Also for small differences in the services a new ComponentType has to be created.
- Many different Components and ComponentTypes can lead to an unclear architecture.

Service Endpoint
Due to the fact that the configuration of the services take place in the ComponentTypes, the configuration of the service endpoints is done in the Signatures of the ComponentTypes. To propagate possible changes each endpoint gets the id from the Signature.

Pros:

+ The endpoints have to be configured in the ComponentType once and will then be used for all Components with the same ComponentType.

Cons:

- Also for small differences in an endpoint a new ComponentType has to be created.

MOM
In this variant the configuration of the MOM is done in the ConnectorType and the Connector defines if the communication channel behaves as a publish-subscriber channel or point-to-point channel and sets the name of the channel.

Pros:

+ Clear separation of the server configuration (e.g. ports, persistence adapter) and the channel configuration.
+ Each ConnectorType can be saved in a separate configuration file. This can be used to create different server configurations.

Conclusion
This variant fits well if there are several identical Components. In such a scenario only a few, or even only one, ComponentType has to be defined, this ComponentType can then be used for all identical Components. But if the services are different, each Component needs his own ComponentType. This can lead to an unclear and complex architecture and with a growing complexity there is also a growing risk for an incorrect mapping between a Component and a ComponentType. The separation of the Connector and ConnectorType, respectively the separation of the channel configuration and the server configuration, is logical and unambiguous.

Ongoing Example 10: Variant 1

If we applied variant 1 to our ongoing example we would have to create for each service (Submit, Log, Execution) a Component and a ComponentType. This has to be done because all services are different from each other: (i) the *Log Service* has only one interface while the *Submit-* and the *Execution Service* have two, (ii) the *Submit Service* and also the *Log Service* have two interfaces but the interfaces from the *Submit Service* have to be configured for *request-reply* (see Section 3.5.1.5) and the interfaces of the *Log Service* does not. For the connection, this variant needs one ConnectorType to define all MOM specific configurations and two Connectors to define the publish-subscriber and the point-to-point connection for the reply.

Type	Amount
Component	3
ComponentType	3
Connector	2
ConnectorType	1

Table 3.3: Needed amount of Components, ComponentTypes, Connectors and ConnectorTypes.

3.5.1.2 Variant 2

Service
In this variant, all configuration information of a service is stored in the Component, therefore the ComponentType holds no relevant information. This implies that each service needs its own Component, but they can use the same ComponentType as long as they have the same amount and type of endpoints. This fits for most scenarios, due to the fact that most of the services are different and need a specific configuration.

Pros:

+ A ComponentType can be used for different Components, as long as the amount and type of endpoints are the same.

Cons:

- If the services are the same and need the same configuration, it has to be configured for each Component.

Service Endpoint
Due to the fact that all service information is stored in the Components, the endpoint information is stored in the Interfaces. This is meaningful, because often the only difference between two services are the endpoints, e.g. both services have two endpoints, one *in* and one *out*, but in the first service they are configured for the *request-reply* pattern and in the second they are configured as normal *in* and normal *out* endpoint. If the endpoint configuration would be in the Signature both services would require different ComponentTypes.

Pros:

+ Often the only difference between two services are the endpoints, and therefore there is no need for different ComponentTypes as long as the amount and direction of the interfaces stay the same.

Cons:

- If the endpoints of several services are the same, they have to be configured for each Component again.

MOM
In contrary to Variant 1 this approach stores the file path, for the configuration file of the MOM, in the Connector. This enables the possibility to generate a new configuration file for each channel (Connector) without creating different ConnectorTypes. This can be useful if the configuration of the MOM server should be generated for several servers.

Pros:

+ Many different MOM configuration files can be generated easily.

Cons:

- Configuration effort is higher, because for each Connector the correct file path has to be entered.

- Different configuration files could also be generated with different ConnectorTypes.

Conclusion
This Variant fits well for a complex messaging system where most of the services are different to each other, which is often the case. In such a scenario it is mandatory to configure each service

(Component) for itself and therefore this approach can model the structure in a clear way. As discussed above, this version stores the file path, for the MOM configuration, in the Connector. This could be useful if different channels need to be routed over different MOM servers, but if this is not the case it can be rather complicated to set the right file path to all Connectors.

Ongoing Example 11: Variant 2

For our ongoing example, we would again need three Components and two Connectors. But in contrast to Variant 1, we only need two ComponentTypes, one for the *Log Service* (with one *in* interface) and one for the the *Submit-* and the *Execution Service* (with one *in* endpoint and one *out* endpoint). The configuration of the service endpoints (e.g. the endpoints of the *Submit Service* have to be configured for the *request-reply* pattern) is done at the interfaces of the Components. As discussed in the specification of this example we only want one MOM server with an active persistence adapter. To accomplish that, we need one ConnectorType to configure the persistence adapter. The file path has to be added to each Connector.

Type	Amount
Component	3
ComponentType	2
Connector	2
ConnectorType	1

Table 3.4: Needed amount of Components, ComponentTypes, Connectors and ConnectorTypes.

3.5.1.3 Variant 3

This last variant is a combination of Variant 1 and 2.

Service
For the services the Variant 2 approach is used (discussed in section 3.5.1.2). In this approach all service configuration information is stored in the Component. The ComponentType will hold no relevant information for our purpose.

Service Endpoint
For the endpoint, the Variant 2 approach is used as well (discussed in section 3.5.1.2). In Variant 2 all endpoint information is stored in the Interfaces.

MOM
In this variant we use the approach from Variant 1 (discussed in section 3.5.1.1) where the configuration of the MOM server is done in the ConnectorType. This variant uses the Connector to define a publish-subscriber channel or point-to-point channel and to set the name of the channel.

Conclusion
This version takes the service and service endpoint from Variant 2 and the MOM configuration from Variant 1. With the service and service endpoint definition in the Components and Interfaces it fits well for scenarios where most of the services are different from each other. The separation of the MOM server configuration and channel name configuration has the same separation as many available MOM servers, where the channel name is defined in the service endpoints and not in the MOM.

Ongoing Example 12: Variant 3

For the services we need, same as in Variant 2, three Components, i.e. one for each service and two ComponentTypes, one for the *Log Service* and one for the *Submit-* and the *Execution Service*. For the connection, this variant also needs one ConnectorType to define the JMS server and two Connectors to define the channels.

Type	Amount
Component	3
ComponentType	2
Connector	2
ConnectorType	1

Table 3.5: Needed amount of Components, ComponentTypes, Connectors and ConnectorTypes.

3.5.1.4 Summary & Decision

To sum up the three variants:

Variant 1 fits well if there are several identical services. But commonly the services are different, at least in a small way.

Variant 2 can handle different services in a more elegant way, because the service and service endpoint information is stored in Component and Interface. It also supports the developers if they need several different MOM server configurations.

Variant 3 places the services and service endpoints into Components and Interfaces. Same as in Variant 2, this combination fits well if there are different services. The MOM server configuration is set into the ConnectorType and the MOM channel configuration is set into the Connector.

In our work we decided to implement Variant 3, which combines the best characteristics of Variant 1 & 2. This is because (i) almost all services in a modern service system are different to each other and (ii) the separation of MOM server configuration and channel name configuration is the same separation as reached with many different available MOM implementations.

3.5.1.5 Message-Centric Properties

In the following, the purpose and the configuration properties of the previous allocated elements *id*, *file path* and *configuration* will be discussed in detail.

Id
The *id* element is used for the association of the message system aspects to the xADL elements and will be a property in the xADL extensions *Service*, *Service endpoint* and *MOM*. This property will be discussed in more detail in Section 3.6.

File path
An important aspect of our approach is the separation of the implementation/configuration into different files. For this, we included the property *file path*. The property can be used to define individual files for each Service implementation or MOM configuration. This supports a clear separation of development tasks and responsibility.

Configuration
As already defined, the element *Configuration* contains message system specific configuration properties of the xADL extensions *Service*, *Service endpoint*, *Channel* and *MOM*. In the following we will specify those properties.

Service configuration applies to an xADL Component. As already mentioned, our approach doesn't require knowledge of the internal service behavior and will only use messaging system relevant parts of the system. Therefore this extension is used to mark the architecture-level component as a service.

Service endpoint configuration applies to an xADL Interface element associated with a component and defines a service endpoint in the messaging system. For our approach, these extensions need three properties. The first one sets the durable name of the endpoint. The name is required if the endpoint is used as a durable subscriber. The last two properties are used to specify the participating interfaces in a request-reply pattern, where the reply interface is set dynamically[1]. In fact, this pattern comprises four interfaces: on the requesting service one sending (sends the request message) and one receiving (receives the reply message) interface and on the replying service one receiving (receives the request message) and one sending (sends the reply message) interface. These interfaces have to be marked as such. We suggest to use two properties to define the special roles, specifically *ToReplyInterface* and *ToRequestInterface*. On the requesting service, the sending interface has to enrich the message header with connection information of the replying point-to-point channel where the reply should be sent to. This is done via the *ToReplyInterface* property that adds a reference from the sending interface to the receiving interface. To complete the requesting service the receiving interface needs a reference to the sending interface via the property *ToRequestInterface*. On the replying service, the receiving interface gets a reference to the sending interface with the *ToReplyInterface* property. To

[1] For the rest of this work we will assume that the reply channel, of a request-reply pattern, is always set dynamic.

fulfill the request-reply pattern the sending interface is referred to the receiving interface via the *ToRequestInterface* property. The example box 13 will configure the interfaces of the *Submit Service* and the *Log Service* of the ongoing example for the requesting-reply pattern.

Channel configuration applies to an xADL Connector element and adds the ability to use it as a messaging channel. Two properties are needed to define the channel type, publish-subscriber (topic) or point-to-point (queue), and the name of the channel.

MOM configuration applies to an xADL ConnectorType element and contains the configuration for the MOM. In our suggested approach it will contain the information that is needed for the connection, like the URL of the message broker.

Ongoing Example 13: Request-Reply pattern configuration

In the ongoing example, the *Submit-* and the *Execution Service* use a request-reply pattern to send the time-consuming task and receive the result of the task. In the example, the *Submit Service* is the requesting service and the *Execution Service* the replying one. Figure 3.5 depicts both Services (colored white) and the communication channels (colored grey). Note that for a clearer figure the *Log Service* has been omitted and only the pattern relevant *Submit-* and *Execution Service* are shown.

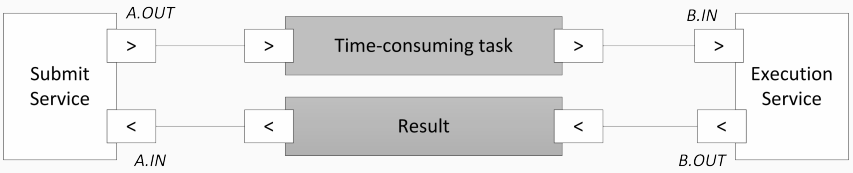
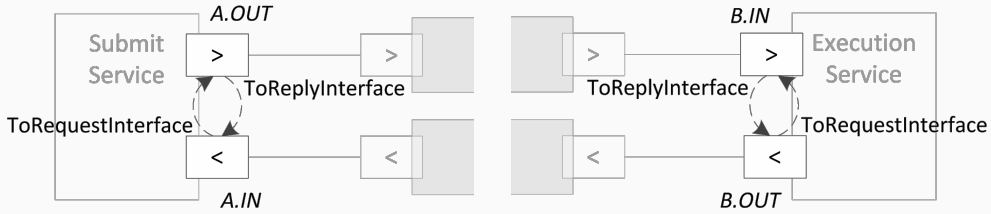


Figure 3.5: Request-Reply pattern of the *Submit-* and the *Execution Service*. *Note:* For a clearer figure only the request-reply participants (*Submit Service* and *Execution Service*) are shown.

Each service has two interfaces, the arrows in the rectangles represent the communication direction. As depicted in the figure the interfaces were labeled *A.IN*, *A.OUT*, *B.IN* and *B.OUT* to use them for the following description. The request-reply flow is as followed: (1) *A.OUT* enriches the message header with the connection information of the reply channel *Result*, (2) *A.OUT* sends the request message over the *Time-consuming task requests* channel, (3) *B.IN* receives the request message and the *Execution Service* runs the task, (4) *B.OUT* sends the result of the task as a reply message over the channel that is defined in the message header (in this example the *Result* channel), and finally (5) *A.IN* receives the reply message.
To identify the role of the interfaces our approach uses two properties called *ToReplyInterface* and *ToRequestInterface*. Both of them are properties for component interfaces and refer to a corresponding interface at the same component. Figure 3.6 and table 3.7 shows on which interfaces the properties have to be defined and to which

interfaces the properties refer.



(a) Request-Reply pattern of the *Submit-* and the *Execution Service*.

(b) Request-Reply pattern of the *Submit-* and the *Execution Service*.

Figure 3.6: Interface configuration for the request-reply pattern of the *Submit-* and the *Execution Service*the *Submit-* and the *Execution Service*

Interface	Property	Refer to interface
A.OUT	ToReplyInterface	*A.IN*
A.IN	ToRequestInterface	*A.OUT*
B.IN	ToReplyInterface	*B.OUT*
B.OUT	ToRequestInterface	*B.IN*

Figure 3.7: Configuration of the participating request-reply interfaces.

3.6 Association of Message System Aspects and ADL Elements

For a tool that supports architecture-centric software development, an explicit mapping between the architecture elements and the implementation/configuration elements is required.

Among others, the two most used mapping variants are:

Bi-directional Both associated elements get a unique id and each gets a reference to the unique id of the corresponding element. To find an element one can follow the reference id, stored in the first element, to the second element. To search in the other direction it is the same procedure. With this variant it is computationally cheap to find the associated element because it is not necessary to search through all elements. If the mapping is used for a 1:1 association each element has only one reference to another element. If it is used for a 1:N, M:1 or M:N mapping, an element has to hold a list of references to the associated elements.
Pro: It is easy and computationally cheap to find the corresponding element.
Con: Can be complicated to maintain all references correctly and can lead to an inconsistent mapping, especially in more complex systems.

Uni-directional One element gets a unique id and the other element the reference to it. With this mapping the search for the element with the unique id by the reference stored in the second element is cheap. But the other way around is computationally expensive, because all elements have to be checked if they refer to the element with the unique id. If two elements are connected with a uni-directional mapping, only one element has to hold the

reference to the associated element. If more than two elements are connected, one element has to hold a list of references to the associated elements.
Pro: Only one unique id and a reference to it is needed, therefore it is clearer and easier to maintain.
Con: The search direction from the second element (the element with the reference) to the first element (the element with the id) is computationally cheap, the other way is computational expensive.

For the mapping between the architecture and the implementation/configuration, a 1:N mapping is required, because different implementation and configuration elements are connected to the same architecture elements. In order to achieve a clear structure we suggested to use a uni-directional mapping where the architecture element has a unique id and the implementation that is, the configuration, refer to this id. As referring id, we suggest to use the unique xADL id that each xADL element has. They are used by xADL for the internal linking between different elements. For more information about the id, the interested reader is referred to the xADL documentation [21] and the paper of Dashofy, Hoek and Taylor [13]. These ids will stay the same, even if parts of the elements get changed, and normally the ids don't have to be changed by a user. Another benefit of the uni-directional mapping is that it is not necessary to add an additional attribute to the xADL element to store the reference to the implementation or configuration elements.

In the following we will discuss the process of finding implementation/configuration element associated to an xADL element and the other way around:

xADL element → implementation/configuration element Due to the fact that each implementation/configuration element holds the id of the xADL element it is necessary to search through all implementation/configuration elements and check if the stored reference id is the searched xADL id.

Implementation/configuration element → xADL element For this direction, the reference id must be used, held by the implementation/configuration element, and then searched for that id in the corresponding xADL architecture document. The search for ids in an xADL document is supported by tools and computationally cheap.

3.7 Change Propagation

As Zheng and Taylor pointed out in [63], the use of architecture modelling doesn't stop after the initial planning phase. It is quite usual that during the development phase the architecture of the system, or at least the configuration of some parts, will change. For an architecture-driven design and configuration tool, it is therefore necessary to support the propagation of changes in the architecture to the already transformed system and thereby retain the consistency between an architecture specification and the implementation. For the propagation process it must be considered that user-defined code could also exist in the transformed implementation. This code has to be preserved during the propagation and only the code that was transformed by the tool should be changed.

For change propagation, four questions have to be considered: (1) Which element has to be changed, (2) how can the existing code, that needs to be changed, be found, (3) how can the element be changed without destroying the user-defined code and (4) what can be done if there is a conflict?

3.7.1 Change Management Strategy

The easiest way to propagate changes is to do a full regeneration of the code. But this would ignore all user-defined code and would delete those code segments. Therefore, our approach will only change the architecture-prescribed implementation. To do this we suggest to solve the four questions mentioned above in the following way:

1. *"Which element has to be changed?"*: Our approach suggests a complete code regeneration of the architecture-prescribed code parts. This means that not only the changed elements or attributes of them are included but rather all elements that are relevant for the message system. We decided to use this radical approach because it means we can guaranteed that the generate/updated system is the planned system, even if parts of an architecture-prescribed element got changed accidentally in the implementation. The approach will only incorporate elements that are relevant for the message system. More precisely it will only include elements that are connected to other elements. This means that if a service isn't connected with another service, it won't be included in the change propagation, even if the service has a representing component in the architecture document.

2. *"How can the existing code, that needs to be changed, be found?"*: As discussed in Section 3.6 we will use the ids provided by xADL to do the mapping between the architecture elements and the implementation elements. For more details see the corresponding Section.

3. *"How can the element be changed without destroying the user-defined code?"*: Our approach separates the architecture-prescribed code (message routing) from the user-defined code (message processing), therefore most changes can be done without touching the user-defined code. Nevertheless some changes need modifications of the user-defined code, as discussed in the work of Zheng and Taylor [63]. In their work, they solved the problem by introducing change notifications that notify the developers if modifications of the user-defined code are necessary. In the current development stage, our approach doesn't provide such a mechanism, it only changes the message routing part and leaves the message processing part without comment. This is considered as part of the future work.

4. *"What can be done if there is a conflict?"*: This can be divided into two subparts.

    a) *Update conflicts*: Those are conflicts that happen if both related attributes have changed and it can't be decided which one is the new one. As already discussed our approach always uses the attributes of the architecture elements and overrides the attributes in the implementation.

b) *Remove conflicts*: Those are conflicts that appear if an element is removed in the architecture. If this is the case, our suggested solution lets the user decide if the corresponding implementation element should be removed, including the message processing code, or only the mapping id to the architecture element. By deleting the id, the reference to the architecture element gets lost and will then be considered as user-defined code. This may be desired if the message processing code should not be deleted.

Furthermore a change propagation algorithm has to handle three different types of changes: (1) a new architecture element is added, (2) an existing architecture element is updated and (3) an architecture element is deleted.

The four answers and the three types of changes in mind, the algorithm has the following steps:

1. Search for all existing architecture-prescribed code elements, which have to be updated, by searching for the xADL id.

2. Regenerate the existing architecture-prescribed code elements with the new information.

3. Add all new elements, i.e. all elements that are new but for that no architecture-prescribed code exists.

4. For all architecture-prescribed code elements which already exist but aren't used anymore, i.e. elements that were deleted in the architecture. Ask the user if the implementation element should be removed or only the mapping xADL id to the architecture element.

    a) If the user decides to remove the existing architecture-prescribed code element, remove it.

    b) If the user decides to remove the mapping to the existing architecture-prescribed code element, remove the xADL id.

CHAPTER 4

Realization

In this chapter, we will present the prototype which is implemented based on the approach. The first section will present the "big picture" of the implementation. Specifically, we will discuss the four main parts of the system (*xADL extension*, *Architecture-level Editor*, *Consistency Checks* and *Architecture-to-Configuration transformation*) and the collaboration between them. Subsequently, each part will be discussed in detail. This chapter will also analyze the data model of a Mule ESB workflow configuration file, an Apache ActiveMQ configuration file and an xADL document, including the description how xADL was extended by us. Conclusive we will discribe the installation and usage of our extension. For the implementation we used xADL 2.0 [21] and ArchStudio 4 [11, 20]. As output for the transformation we decided to produce Mule ESB [48] workflow and Apache ActiveMQ [3] configuration files. The system was developed and tested in a Mac OSX and Windows environment, with an ArchStudio 4 version 4.1.50, Mule ESB version 3.5 and Apache ActiveMQ version 5.9.

4.1 Big Picture

As depicted in Figure 4.1, our implementation has four top level entities:

- **xArchADT / xADL 2.0 Document:** This is the interface to the xADL architecture document that holds all architectural information. It is the central point of our implementation, because all other entities depend on this interface. To access the xADL elements, we will use the xADL infrastructure tool *xArchADT*, which was discussed in Section 2.2.4.1. As already discussed the provided standard xADL 2.0 Schema will be extended to fit our special use case.

- **Architecture-Level Editors:** This entity comprises the ArchStudio 4 editors *Archipelago*, *ArchEdit*, *TypeWrangler* and *MSG Launcher*. Note that ArchStudio 4 has more editors, but for our purpose we will only use these four editors. The first three editors are already available in ArchStudio 4 and can be used to design and configure the architecture. The

last one, *MSG Launcher*, is implemented by us and is used as a launch point for the messaging system consistency check and the architecture-to-configuration transformation.

- **Consistency Check:** This is the module that is concerned with the consistency of the architecture. Before the transformation process launches the *Consistency Check* entity searches for inconsistencies in the architecture.

- **Architecture-to-Configuration Transformation:** This entity is concerned with the actual transformation of the architecture to the Mule ESB workflow and ActiveMQ configuration. It gets started after the architecture is checked for inconsistencies. It uses the *xADL Model* and translates it into an internal model for the Mule ESB workflow configuration and for the ActiveMQ configuration. These models will then be used to generate the actual configuration files.

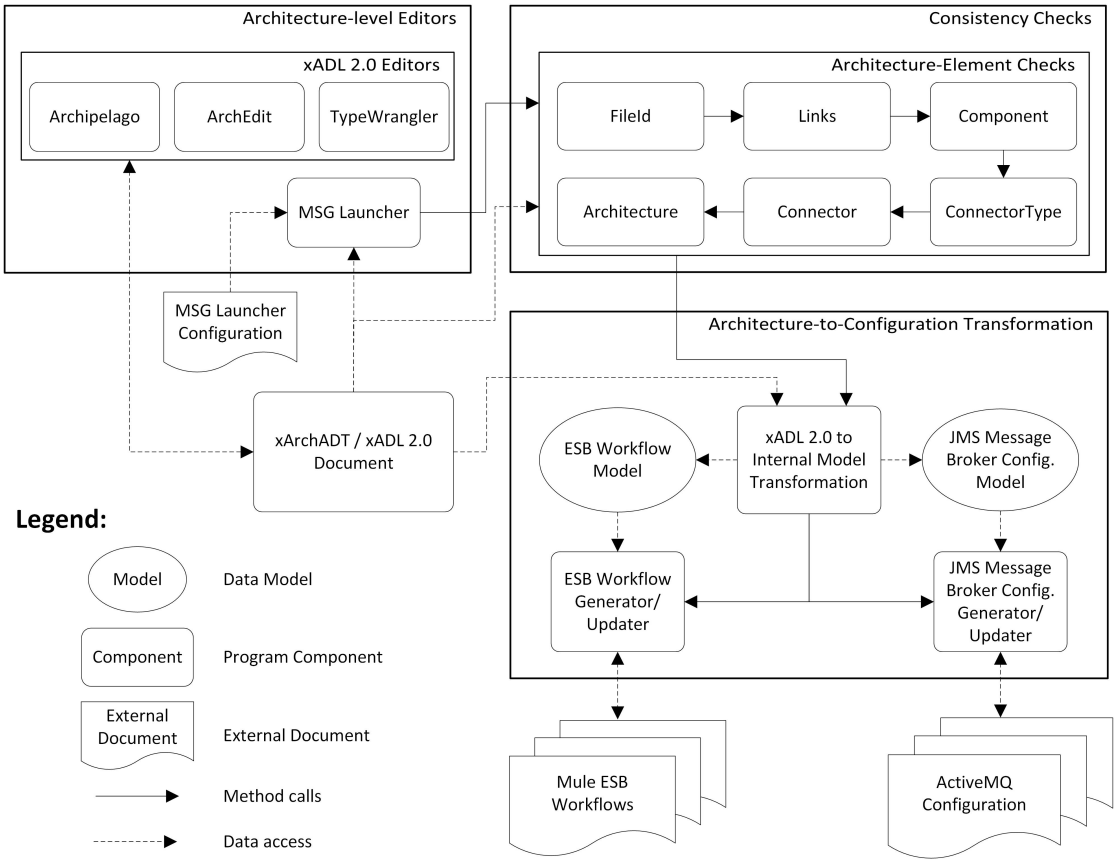


Figure 4.1: Overview of our implementation.

In the following sections the four top level entities will be discussed in detail.

4.2 Architecture-Level Editors

For an easier usage of xADL, we used the development environment *ArchStudio 4*. ArchStudio 4 is the official tool support for the language and includes different tools to work with xADL. As depicted in Figure 4.2, our implementation uses already available editors as well as a new tool that extends ArchStudio 4 with the ability to launch the Consistency Check and the Architecture-To-Configuration transformation. The tool is called *MSG Launcher* (Messaging-system generator Launcher) and will be discussed in the following section.

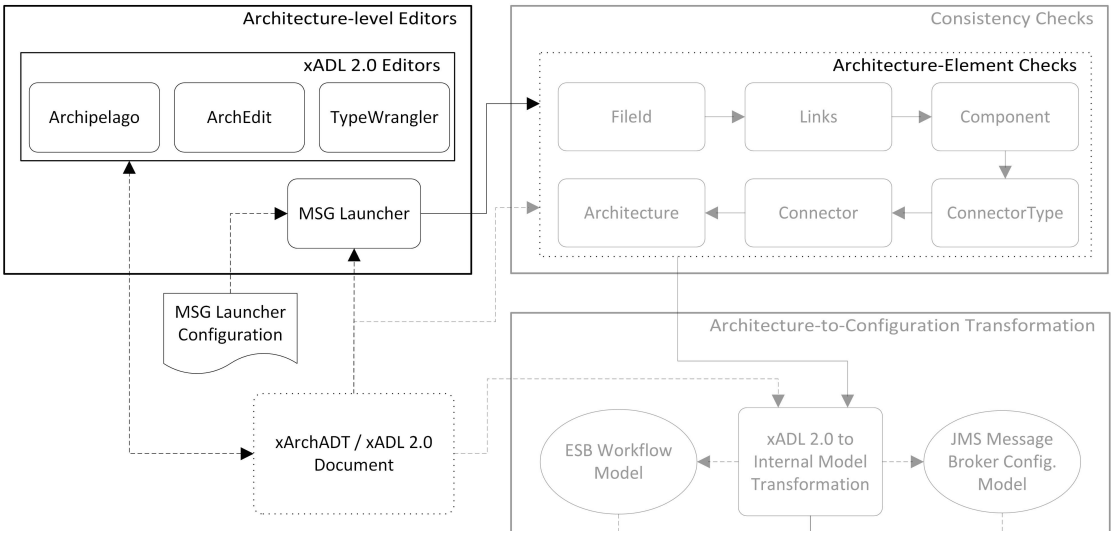


Figure 4.2: Architecture-level Editors Module and associated components

4.2.1 xADL 2.0 Editors

The included tools *Archipelago*, *ArchEdit* and *TypeWrangler* are the three already available ArchStudio 4 tools that are primarily used for our suggested solution. All three tools are editors for xADL 2.0 and are discussed in more detail in section 2.2.4.2 and in [13, 14]. The example box 14 shows the architecture of the ongoing example, drawn with the tool *Archipelago*.

Ongoing Example 14: Archipelago Structure

Figure 4.3 shows the architecture of our ongoing example drawn with the ArchStudio 4 tool Archipelago. The dark blue boxes are components and represent services, the light beige boxes are connectors and represent the message transport channels.

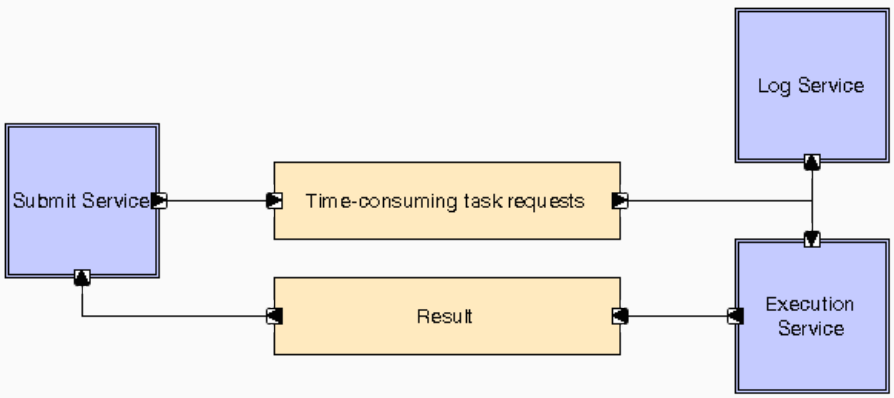


Figure 4.3: Structure of the ongoing example in Archipelago.

4.2.2 MSG Launcher

We extended ArchStudio by the tool *MSG Launcher*, which is the starting point of the transformation workflow, including Consistency Check and the Architecture-to-Configuration transformation. As depicted in Figure 4.2, the tool uses the xADL model and an additional configuration file to start the process. The xADL model is necessary for defining which structure of the architecture should be generated. As defined in Section 3.5.1.5 our approach provides the possibility to separate the transformed ESB workflow implementation and JMS Message Broker configurations in different files. Those different files are defined in the additional configuration file that is loaded by *MSG Launcher*. It holds the paths to the ESB workflows and the JMS Message Broker configuration files, including an id for each path, in an XML file (see example box 15 for the configuration file of the ongoing example). *MSG Launcher* can be used to read the configuration file and edit the file paths and ids. The ids are then used in the xADL Components and Connectors to define where the transformed configuration should be saved. We decided to hold the file paths in a separate file and map the xADL elements by an id to the file path and not to save the path directly in the xADL document, to offer the user an easy way to change the paths and to separate this non architectural information from the architectural ones. After reading this configuration file and defining the architecture structure file, the *MSG Launcher* extension starts the *Consistency Check*. Figure 4.4 depicts our ArchStudio extension including schema extension, the transformation file mapping and exemplary inconsistency alerts.

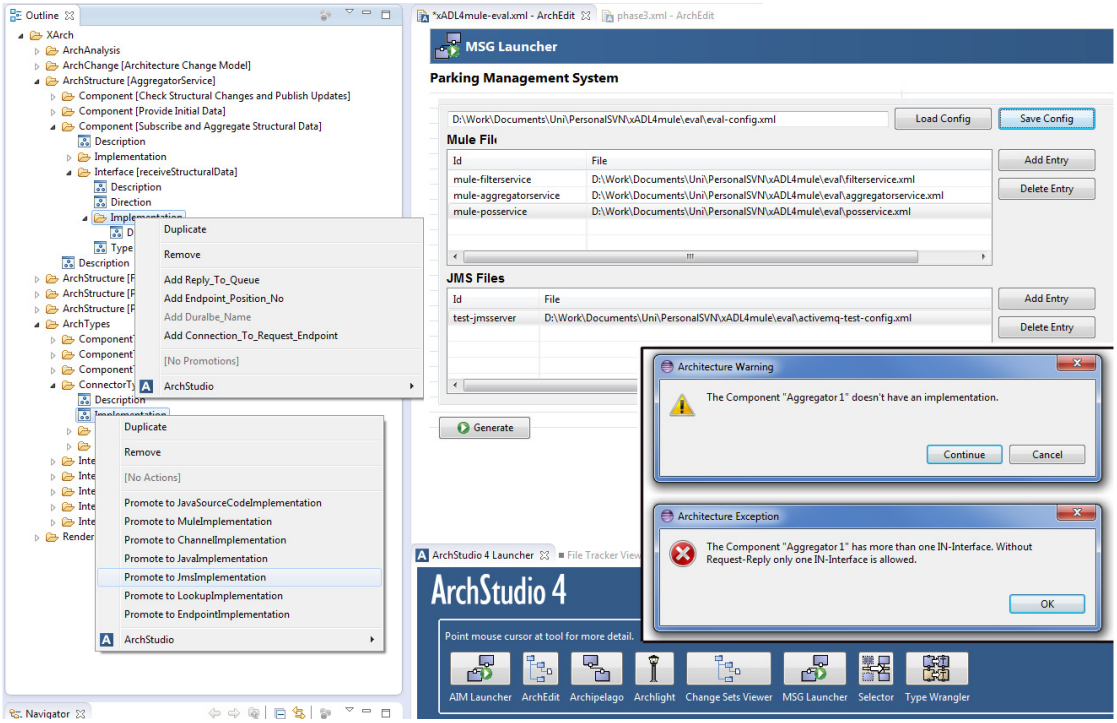


Figure 4.4: Screenshot of our ArchStudio extension, including schema extension (left), the transformation file mapping (top) and exemplary inconsistency alerts (inset) [16].

Ongoing Example 15: MSG Launcher Configuration

Listing 4.1 defines a configuration file for the ongoing example where one ActiveMQ message broker configuration file and two Mule ESB workflow files will be generated. The ActiveMQ message broker configuration will be located at the absolute path "/absolute/-path/to/activemq.xml", the Mule ESB workflow implementation file for the *Submit Service* at "/absolute/path/to/muleSubmit.xml" and for the *Log-* & the *Execution Service* at "/absolute/path/to/muleLogExecution.xml". The corresponding ids *Jms_0*, *Submit_Service* and *Service_Log_Execution* have to be used in the corresponding xADL Components and ConnectorType to map them to the file paths.

Listing 4.1: MSG Launcher Configuration for one JMS Message Broker configuration file and two Mule ESB workflow files.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <output_file_config>
3    <jms_list>
4      <output_file>
5        <id>Jms_0</id>
6        <filepath>/absolute/path/to/activemq.xml</filepath>
7      </output_file>
8    </jms_list>
9    <mule_list>
10     <output_file>
11       <id>Service_Submit</id>
12       <filepath>/absolute/path/to/muleSubmit.xml</filepath>
13     </output_file>
14     <output_file>
15       <id>Service_Log_Execution</id>
16       <filepath>/absolute/path/to/muleLogExecution.xml<
             /filepath>
17     </output_file>
18   </mule_list>
19 </output_file_config>
```

4.3 Basic Data Model

Before we discuss the xADL extensions we have to discuss the underlying data model of a Mule ESB workflow, an ActiveMQ configuration file and xADL. For Mule ESB and Apache ActiveMQ we will also discuss what information and configuration details are mandatory to build up a messaging system.

4.3.1 Mule ESB Data Model

As discussed in Section 2.1.4.2 Mule ESB uses XML based configuration files to construct the workflows. These files describe the internal structure of a Mule ESB workflow[1] and how the workflow components are configured and connected together. During runtime, these workflow components are processed in a sequential order, whenever the workflow is triggered by an event (e.g. a received message). The example box 16 shows the Mule ESB XML configuration of the *Log Service's* workflow from our ongoing example. The example box will briefly describe the main elements of the XML-File that are relevant for our purpose. The interested reader will find a more detailed description on the Mule ESB documentation website [49].

[1] Throughout this thesis any reference to workflow is referring to a Mule ESB workflow.

Ongoing Example 16: Mule ESB workflow XML configuration

Listing 4.2 shows the workflow of the *Log Service* of the ongoing example. The workflow listens to a JMS queue and prints out the received message to the console output. The Mule Components are processed in a sequential order.

Listing 4.2: Minimum Mule ESB Workflow Configuration to listen to a JMS queue

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <mule xmlns:jms="http://www.mulesoft.org/schema/mule/jms"
      xmlns="http://www.mulesoft.org/schema/mule/core"
      xmlns:spring="http://www.springframework.org/schema/beans"
      version="EE-3.4.0" xmlns:xsi="http://www.w3.
      org/2001/XMLSchema-instance" xsi:schemaLocation="...">
3   <jms:activemq-connector name="ActiveMQ" brokerURL="
      tcp://localhost:61616"/>
4   <flow name="logService" doc:name="Log Service">
5     <jms:inbound-endpoint topic="request.topic" doc:name="JMS"
        connector-ref="ActiveMQ"/>
6     <logger message="#[payload]" level="INFO" doc:name="Logger
      "/>
7   </flow>
8 </mule>
```

mule is the root element and as such, it encloses all workflows in this file.

jms:activemq-connector defines the JMS broker configuration. This example uses Apache ActiveMQ as the JMS broker. Apache ActiveMQ is a widely used message server and is supported by Mule ESB with a specialised connector. In addition to the ActiveMQ connector, Mule ESB also offers connectors for *Web logic JMS* and *Mule MQ*. Beside the predefined connectors it is also possible to connect to other JMS servers by defining a ConnectionFactory object. The broker will use the address *tcp://localhost:61616* to send and receive messages, this is defined by the attribute *brokerURL*. The attribute *name* sets the name of the connector, this name has to be unique and will be used as reference to the connector.

flow is the element which defines the actual workflow. This element encloses all workflow specific components. Each workflow requires its own *flow* element as a child of the *mule* element. One *mule* element could have several *flow* elements.

jms:inbound-endpoint defines the workflow component that is listening to the JMS queue. If the channel contains a new message, this component will read the message and pass it to the next component in the workflow. The attribute *topic* configures the type and name of the channel, in this example the type is a publish-subscriber channel (for a point-to-point connection the attribute *queue*, instead of *topic*, has to be used) and the

name of it is *request.topic*. The attribute *connector-ref* is the reference to the corresponding connector. The opposite of an *inbound-endpoint* is an *outbound-endpoint*. The *outbound-endpoint* is responsible for the output of a message to a channel or to another communication medium.

logger is the component that will log the received message. This component isn't part of the messaging system and therefore we won't discuss the usage of this component in detail.

4.3.1.1 Request-Reply

To implement a *request-reply* pattern, Mule ESB offers a *request-reply* element. This element encloses the designated outbound-endpoint (the requesting interface) and inbound-endpoint (the reply interface). To specify the responding channel the JMS message property *MULE_REPLYTO* has to be added before the message will be sent. This has to be done by adding the element *message-properties-transformer* and the enclosed *add-message-property*. In the example box 17, we will show the usage of the *request-reply* pattern on the basis of our ongoing example.

Ongoing Example 17: Request-reply

In our ongoing example the *Submit Service* implements the *request-reply* pattern, it sends the request over the publish-subscriber channel and receives the message by the point-to-point queue. Listing 4.3 shows the workflow of the *Submit Service*. For a clear representation the components that aren't important for the message transport (the generation of the message and the print out of the received reply) are commented out. The request will be send to the publish-subscriber channel (topic), with the name *request.topic*, and the reply will be received on the point-to-point channel (queue), with the name *reply.queue*.

Listing 4.3: Mule ESB Workflow Configuration for a request on topic: request.topic and reply on queue: reply.queue.

```
1  <flow name="submitService" doc:name="Submit␣Service">
2    <!-- Generate message -->
3    <message-properties-transformer>
4      <add-message-property key="MULE_REPLYTO" value="
           jms://reply.queue"/>
5    </message-properties-transformer>
6    <request-reply>
7      <jms:outbound-endpoint connector-ref="Active_MQ" topic="
           request.topic" />
8      <jms:inbound-endpoint connector-ref="Active_MQ" queue="
           reply.queue" />
9    </request-reply>
10   <!-- Print received reply -->
11 </flow>
```

message-properties-transformer is the element to add message properties to the message.

add-message-property will add one message property to the current message header. This element takes a key-value pair. In our scenario it takes the key *MULE_REPLYTO*, to specify the reply channel, and the value *jms://reply.queue*, to specify the name of the reply channel.

request-reply encloses the corresponding outbound-endpoint (to send the request) and the inbound-endpoint (to listen to the reply).

4.3.1.2 Mandatory information

To set up a working messaging skeleton it is mandatory to know which information is required. As we can see above the elements *mule*, *connector* (e.g. *activemq-connector*), *flow*, *[inbound|outbound]-endpoint* are mandatory elements to build up a system and for a request-reply pattern the elements *message-properties-transformer*, *add-message-property* and *request-reply* are also needed. Table 4.1 shows mandatory attributes of the elements and table 4.2 the elements that are needed for a request-reply pattern.

Element	Attribute name	Description
flow	name	Name of the flow. This name can be used to make a reference to the flow.
[inbound | outbound] -endpoint	durableName	If durable connection is enabled, this represents the durable name of the endpoint.
	exchange-pattern	Defines if this workflow should send a response or not (allowed values: *one-way*, *request-response*).
	[queue | topic]	Name of the queue or topic. By using the attribute queue the endpoint is reading/writing a queue and by using topic the endpoint is publishing/subscribing a topic.
	connector-ref	Reference to the corresponding connector.
activemq-connector	brokerURL	URL to the JMS server
	name	Name of the connector. This name can be used to make a reference from the JMS endpoint to the connector.
	durable	Defines if the topic subscribers are durable.
	specification	Sets the JMS specification (allowed: 1.0.2b, 1.1).

Table 4.1: Attributes of Mule ESB component flow.

Element	Attribute name	Description
message-properties-transformer	*No mandatory attributes*	
add-message-property	key	Name of the JMS message property (e.g. for request-reply: MULE_REPLYTO).
	value	Name of the reply queue.
request-reply	*No mandatory attributes*	

Table 4.2: Attributes of Mule ESB component add-message-property.

4.3.2 Apache ActiveMQ Data Model

As mentioned earlier, our prototype will use Apache ActiveMQ as the JMS message server. We decided to use ActiveMQ because it is a widely used, state-of-the-art messaging server and has good support by Mule ESB. The server configuration of ActiveMQ is also done by an XML-formatted configuration file. The example box 18 shows the ActiveMQ configuration file for the ongoing example. Again we will only discuss the parts that are relevant for our purpose. The interested reader will find a more detailed documentation at the Apache ActiveMQ Website [23].

Ongoing Example 18: ActiveMQ Configuration

Listing 4.4 shows the ActiveMQ message broker configuration file of our ongoing example. The server will run on the local machine (IP 127.0.0.1) and will listen on port 61616. Additionally it will persist the messages by using a KahaDB database at the directory connect/kahadb.

Listing 4.4: A typical configuration of an ActiveMQ messaging server.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:amq="http://activemq.apache.org/schema/core"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="...">
3    <broker xmlns="http://activemq.apache.org/schema/core">
4      <persistenceAdapter>
5        <kahaDB directory="connect/kahadb"/>
6      </persistenceAdapter>
7      <transportConnectors>
8        <transportConnector uri="tcp://127.0.0.1:61616"/>
9      </transportConnectors>
10   </broker>
11 </beans>
```

beans is the root element.

broker is the actual broker element which configures an ActiveMQ broker.

persistenceAdapter encloses all persistence specific elements. This element is only required if the database path is different to the default one or another database than *KahaDB* is used.

kahaDB configures message persistence for the message server. In this example we use KahaDB to persist the messages. KahaDB is a database specialised in message persistence and is part of the Apache ActiveMQ project [24]. It is the default persistence solution of ActiveMQ, but ActiveMQ also supports other solutions like LevelDB [a] and all the major SQL databases [b].

transportConnectors define the protocols with which the clients can connect to a broker. A single broker can have several open connectors with different protocols and listen ports [c].

transportConnector configures one connector. In this example it is a TCP connection on the port 61616.

[a] http://activemq.apache.org/leveldb-store.html
[b] http://activemq.apache.org/jdbc-support.html
[c] http://activemq.apache.org/configuring-transports.html

4.3.2.1 Mandatory information

The mandatory information for a ActiveMQ server configuration are *transportConnectors* and the enclosed *transportConnector*. Table 4.3 shows the minimum attributes for these elements. To configure the persistence adapter the additional elements *persistenceAdapter* and the corresponding database element, in our case *kahaDB*, are required. Table 4.4 shows those attributes.

Element	Attribute name	Description
transportConnectors	*No mandatory attributes*	
transportConnector	uri	Sets the address of the channel.

Table 4.3: Attributes of an ActiveMQ transport configuration.

Element	Attribute name	Description
persistenceAdapter	*No mandatory attributes*	
kahaDB	directory	Sets the directory where the persistent messages should be stored.

Note: For other database types, the element tag name kahaDB has to be changed to the corresponding tag name, e.g. levelDB for LevelDB Store.

Table 4.4: Attributes of an ActiveMQ persistency configuration.

4.3.3 xADL Data Model

As already discussed in Section 2.2.1.1, xADL 2.0 consists of different XML-Schemas and extensions of them where each adds a set of attributes to the language. Our approach provides that xADL will be extended by adding different implementation schemas. Representatively for the schemas, which were extended by us, the following will discuss the structure of the XML-Schema *InterfaceType*. The description starts with the definition of the type, following the definition of the *abstract Implementation* extension and at the end the definition of the concrete

Java Implementation extension. We will only discuss the extensions that are important for our research, the interested reader is referred to the detailed extension overview at [22].

Listing 4.5 presents the XML-Schema definition of *InterfaceType*. For a clearer reading all header information and comments were removed. As the listing shows the type *InterfaceType* has one description element and the already discussed xADL id (see Section 3.6). The description element can be used to add additional non-functional information, it has the XML-Schema type *archinstance:Description* which extends *string*[2]. Id has the XML-Schema type *archinstance:Identifier* which extends *ID*[2].

Listing 4.5: XML-Schema definition of the InterfaceType

```
1 <xsd:complexType name="InterfaceType">
2   <xsd:sequence>
3     <xsd:element name="description" type="archinstance:Description"/>
4   </xsd:sequence>
5   <xsd:attribute name="id"          type="archinstance:Identifier"/>
6 </xsd:complexType>
```

Listing 4.6 adds the possibility to define implementations for *InterfaceType*. Line 1 defines a new XML abstract type, called *Implementation*, which is the abstract hook-up point that can be extended by other XML elements. Lines 3 to 12 add a new type, called *InterfaceTypeImpl*, which extends *InterfaceType* and adds an implementation element of the already defined abstract *Implementation* type. Note that the new type, *InterfaceTypeImpl*, may have an infinite number of abstract implementations.

Listing 4.6: XML-Schema definition of the abstract Implementation and the InterfaceType Implementation hook-up point

```
1  <xsd:complexType name="Implementation" abstract="true"/>
2
3  <xsd:complexType name="InterfaceTypeImpl">
4    <xsd:complexContent>
5      <xsd:extension base="archtypes:InterfaceType">
6        <xsd:sequence>
7          <xsd:element name="implementation" type="Implementation"
8                       minOccurs="0" maxOccurs="unbounded"/>
9        </xsd:sequence>
10     </xsd:extension>
11   </xsd:complexContent>
12 </xsd:complexType>
```

Finally, to set a concrete implementation for an *InterfaceTypeImpl*, the abstract *Implementation* type has to be extended. Listing 4.7 presents the *Java Implementation*, which extends the abstract *Implementation* type. Thus it can be used for all types that have an abstract *Implementation* element defined, like *InterfaceTypeImpl*.

Listing 4.7: XML-Schema definition of the concrete Java Implementation schema

```
1 <xsd:complexType name="JavaImplementation">
```

[2] http://www.w3.org/2001/XMLSchema

```
2     <xsd:complexContent>
3       <xsd:extension base="archimpl:Implementation">
4         <xsd:sequence>
5           <xsd:element name="mainClass" type="JavaClassFile"/>
6           <xsd:element name="auxClass"  type="JavaClassFile"
7                        minOccurs="0" maxOccurs="unbounded"/>
8         </xsd:sequence>
9       </xsd:extension>
10    </xsd:complexContent>
11  </xsd:complexType>
12
13  <xsd:complexType name="JavaClassFile">
14    <xsd:sequence>
15      <xsd:element name="javaClassName" type="JavaClassName"/>
16      <xsd:element name="url"           type="archinst:XMLLink"
17                   minOccurs="0" maxOccurs="1"/>
18    </xsd:sequence>
19  </xsd:complexType>
20
21  <xsd:complexType name="JavaClassName">
22    <xsd:simpleContent>
23      <xsd:extension base="xsd:string"/>
24    </xsd:simpleContent>
25  </xsd:complexType>
```

In our work we extended the abstract *Implementation* type to add the discussed message-centric system aspects (see Section 3.5) to the range of possible xADL extensions.

4.4 Message-Centric xADL Extension

As Figure 4.5 depicts, the *xArchADT / xADL 2.0 Document* module is the central point in our implementation. It gets filled with the architecture information from the ArchStudio editors and is then used by the *MSG Launcher* component. Afterwards, the *Consistency Checking* module checks the consistency of the architecture document and finally the *Architecture-to-Configuration transformation* module will transform it.

As already discussed, our approach adds several extensions to the xADL data structure. We extended it by four additional concrete implementations, named *Channel Implementation*, *Endpoint Implementation*, *JMS Implementation* and *Mule Implementation*. Each of them represents one of the introduced extensions of Section 3.5.1.5. Figure 4.6 extends Figure 3.4 with the new concrete implementations (colored in grey).

The following will describe the properties of the extensions. In addition to the mandatory properties discussed in Section 3.5.1.5 we also included optional properties that can be helpful but not required for a messaging system. Appendix A contains the XML-schemas of the suggested xADL extensions.

Channel Implementation (Listing A.1) This is the concrete implementation of the extension *Channel configuration* described in the approach and applies to an xADL Connector ele-

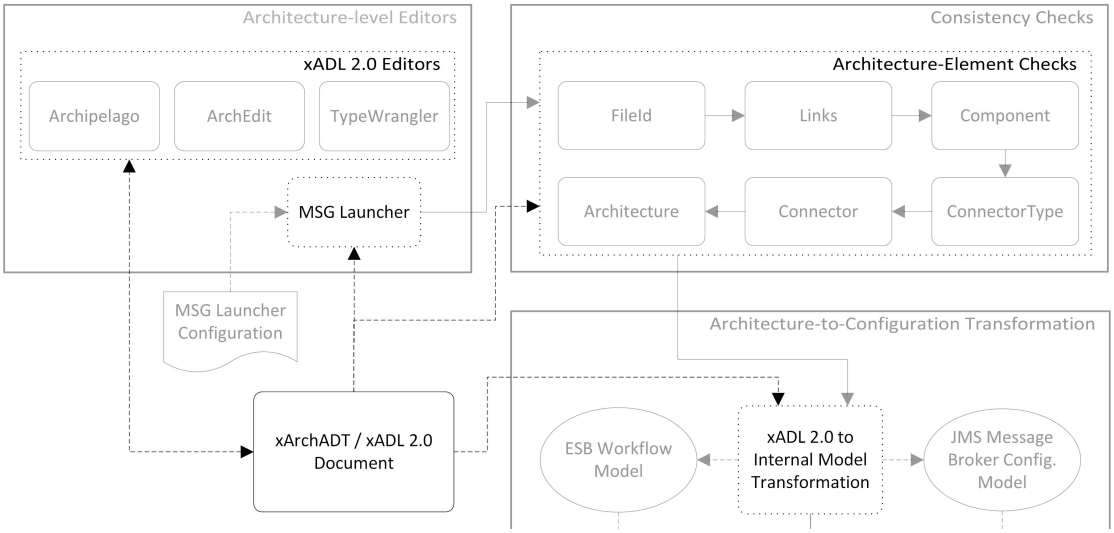


Figure 4.5: xADL Model and associated components

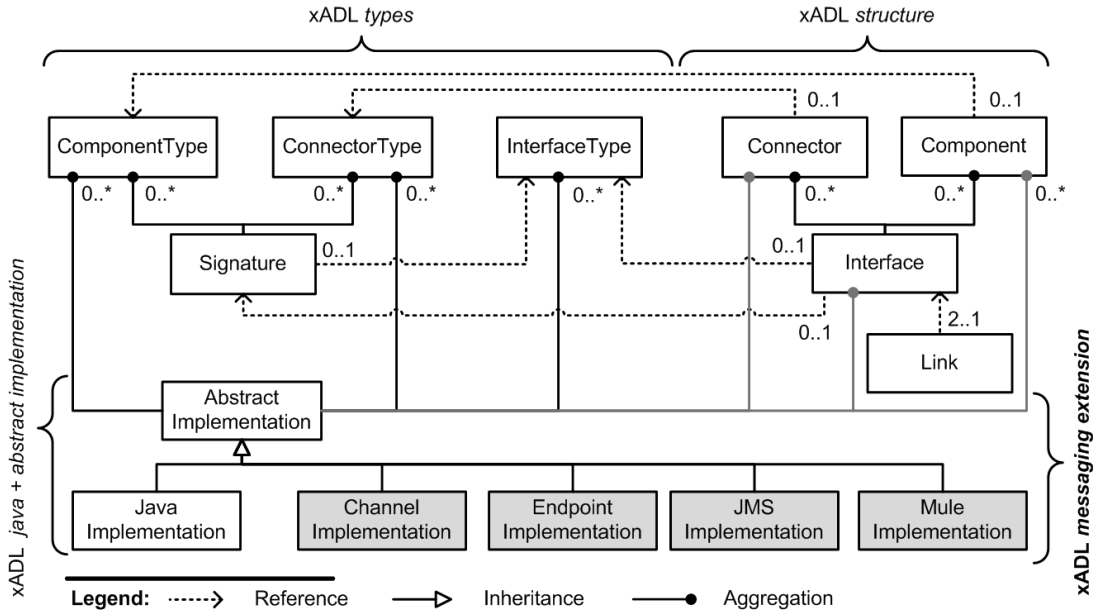


Figure 4.6: Relation between the structure, type, implementation elements and the new extensions (grey) [16].

ment. The extension adds two properties, called *Topic_Configuration* and *Queue_Configuration*. The first one identifies the channel as a publish-subscriber (topic) and the second as a point-to-point (queue) channel. Both of them have a string attribute to define the channel name.

Endpoint Implementation (Listing A.2) This is the concrete implementation of the extension *Service endpoint configuration* described in the approach and applies to an xADL Interface element associated with a component. Note that if it is used for an interface associated with a connector, it will be ignored. This implementation indicates that an interface is an endpoint of a Mule workflow. The extension adds four properties *Durable_Name*, *Reply_To_Queue*, *Connection_To_Request_Endpoint* and *Endpoint_Position_No*. The optional property *Durable_Name* sets the name of the subscriber. The properties *Reply_To_Queue*, *Connection_To_Request_Endpoint* are used to indicate that the interfaces are used within a request-reply pattern. The usage of the properties can be found in Section 3.5.1.5. For the implementation, the property *Reply_To_Queue* represents the property *ToReplyInterface* of the description and *Connection_To_Request_Endpoint* represents *ToRequestInterface*. Although our suggested implementation has an algorithm to sort the interfaces automatically, sometimes a special order is desired or the algorithm can't order the interfaces automatically. Therefore, we included the additional property *Endpoint_Position_No* to the concrete implementation. It is optional, takes an integer value, and can be used to define the order of the interfaces in a Mule workflow manually. Note that if this property is used all interfaces composed by a component have to be ordered by the *Endpoint_Position_No* property. The first interface gets the smallest number. The number has to be increased for each following interface.

JMS Implementation (Listing A.3) This is the concrete implementation of the extension *MOM configuration* described in the approach and applies to a xADL ConnectorType element. It defines that a JMS ActiveMQ server is used as message broker. As defined in Section 3.5.1.5 it has a property to define the URL of the message broker. This property is called *Transport_Configuration* and specifies at least one ActiveMQ connection endpoint URL. Furthermore we added the property *JMS_Specification_Version* and *Persistence_Configuration*. The property *JMS_Specification_Version* can be used to change the JMS protocol version (default version 1.1) and the property *Persistence_Configuration* holds information about the persistence configuration of the message broker. At the current stage, it defines the persistence adapter, ActiveMQ uses kahaDB as default, and the storage directory. Finally the implementation has the attribute *file_id* which holds the id of the ActiveMQ configuration file, in which the configuration should be saved. This id will be mapped to the id discussed in Section 4.2.2. All ConnectorTypes with the same *file_id* will end up in the same ActiveMQ configuration file.

Mule Implementation (Listing A.4) This is the concrete implementation of the extension *Service configuration* described in the approach and applies to an xADL Component element. It indicates that the component represents a service and is specified by a Mule workflow. Also this extension has the attribute *file_id* to specify where the workflow should be saved. All components with the same *file_id* end up in the same Mule workflow configuration file. As defined in Section 3.5.1.5, the extension is only used to identify the component as a service and doesn't need additional information. Nevertheless, we decided to include the property *Additional_Configuration*. The property can be used to define a name/value pair which will be stored as a XML attribute in the workflow element. This property is optional

and can be used an infinite number of times. But our prototype will only copy the pair into the Mule workflow and won't check the availability and correct use of the attribute. Thus, the architect or developer has to check the correctness of the name/value pair.

Natively, xADL provides implementation extensions only for the types *ConnectorType*, *ComponentType* and *InterfaceType*, though our solution also needs the ability to add implementations to the structure elements *Connector*, *Component* and *Interface*. Therefore, our extensions add implementation types for the remaining structures. Those implementation types are also included in the listings in the appendix A.

4.5 Consistency Check

The *Consistency Check* is the second step in the message system generation workflow. As shown in Figure 4.7, it is launched by the *MSG Launcher* component and it uses the *xArchADT / xADL 2.0 Document* module to access the messaging system relevant parts, stored in the *xADL architectural document*, and checks them for inconsistencies and informs the user of existing ones. Provided that the system is consistent, the *Architecture-to-Configuration transformation* is launched.

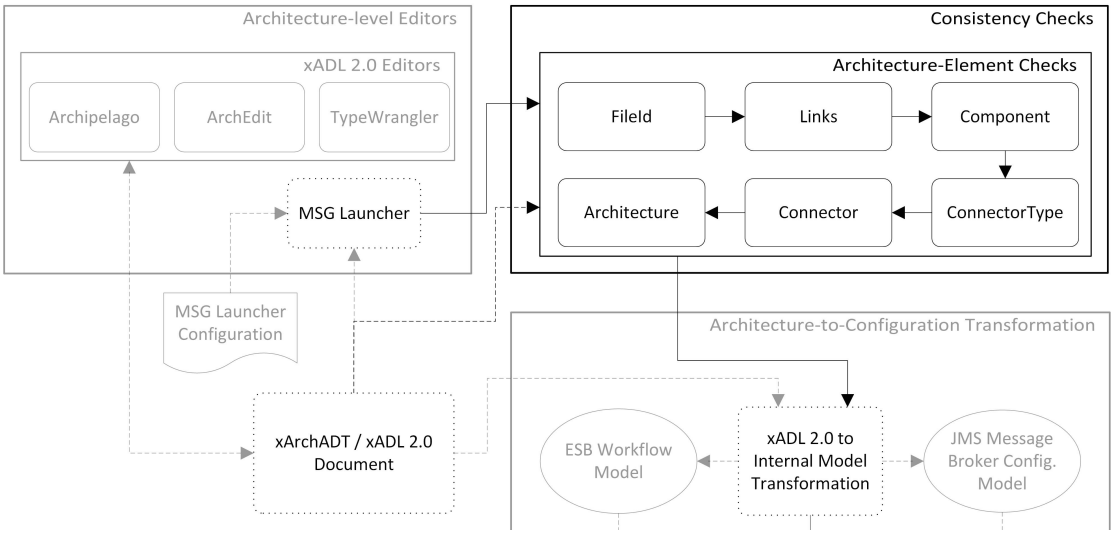


Figure 4.7: Consistency Check Module and associated components.

4.5.1 Consistency Check Definition

To support the development, we defined a set of soft and hard consistency checks. Those checks include the general faults discussed in Section 3.4 and add some Mule ESB and ActiveMQ specific ones. If an inconsistency can be found the system will inform the user of it. In addition to the report of sever violations, the system also gives warnings and recommendations on how

consistency flaws can be rectified.
In the following, the soft and hard consistency checks will be discussed in detail.

Soft consistency checks They detect flaws where the transformed system is functional but maybe doesn't have the intended behaviour. For these kind of violation we issue warnings, including recommendations on how to solve them, and let the user decide if the system should be transformed or the process aborted. Table B.1 shows the checks that are accomplished to ensure that the transformed system is consistent.

Hard consistency checks They detect flaws where the transformation can't be accomplished. In those cases the transformation process won't start and the user will be informed. Table B.2 shows the hard consistency checks.

In addition to the listed checks, ArchStudio 4 provided several basic validations like type compatibility.

4.5.2 Consistency Check Class Structure

The consistency check module is composed of six classes (see Figure 4.8), where each of them is responsible for a specific part.

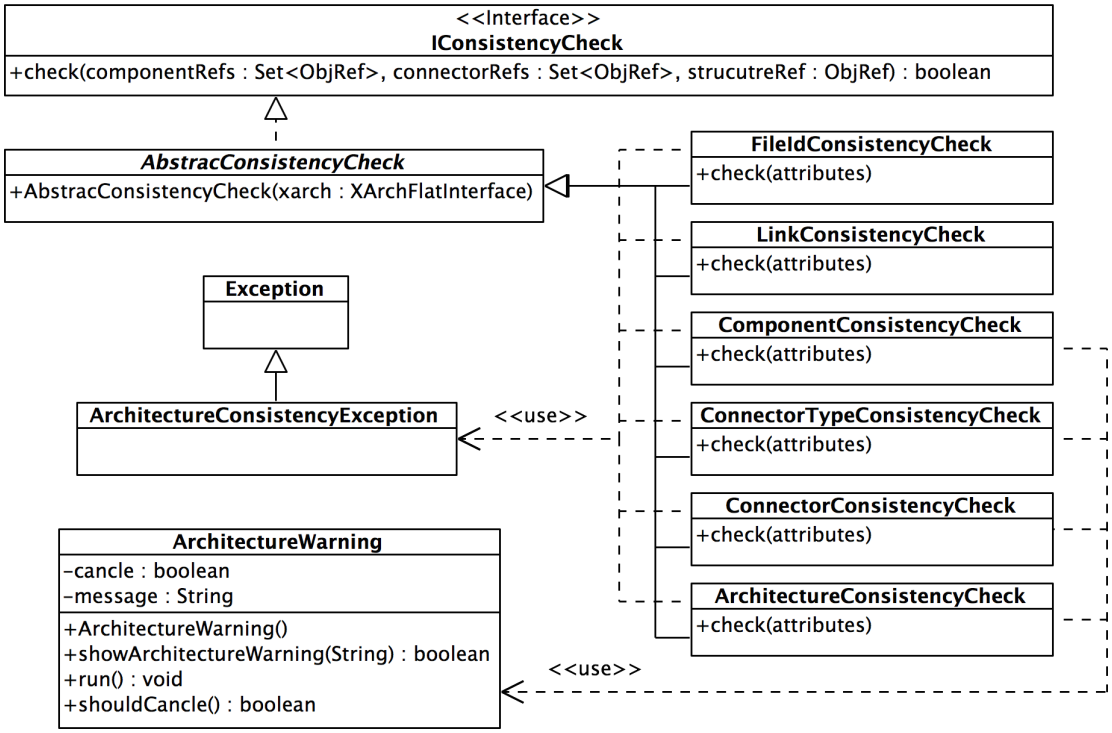


Figure 4.8: Class structure of consistency check classes

1. **FileIdConsistencyCheck:** This is responsible for checking the correct use of the FileId property.

2. **LinkConsistencyCheck:** Checks the correct use of the Links and the connected Interfaces. E.g. Are both ends of a Link connected to an Interface?

3. **ComponentConsistencyCheck:** This is responsible for Component checks. E.g. Do all Components have an implementation?

4. **ConnectorTypeConsistencyCheck:** This is responsible for the ConnectorType checks. E.g. Is the transport property defined?

5. **ConnectorConsistencyCheck:** This is responsible for the Connector checks. E.g. Do all Connectors define a Topic_Configuration or Queue_Configuration property?

6. **ArchitectureConsistencyCheck:** The checks of this class are concerned with architecture of the system. E.g. Check if a publish-subscriber channel is used for Request-Reply Pattern. These checks require that all other checks (FileId, Link, Component,...) are already processed. This is mandatory because those checks assume that certain constraints are fulfilled, e.g. All interfaces need a direction.

Our implementation calls the check methods in sequential order, to be specific the order of the list above.

As can be seen in Figure 4.8 all consistency checks can throw an *ArchitectureConsistencyException*. This exception is thrown when a hard consistency check is violated and the architecture can't be transformed. The figure also depicts that the ConnectorType, Connector, Component and Architecture checks can produce warnings. Warnings will be produced if a soft consistency check is violated and the architecture can be transformed but may not have the desired behaviour.

To read the xADL elements we used the xADL infrastructure tool *xArchADT*, which was discussed in Section 2.2.4.1.

4.6 Architecture-to-Configuration Transformation

The architecture-to-configuration transformation is concerned with the generation of the connector skeleton for Mule ESB and the configuration for the message broker by using the information given by the architecture description of the system. As depicted in Figure 4.9, the transformation is started after the Consistency Checks. Thus it is guaranteed that the architecture of the system is consistent and that the transformed system will be functional. During the transformation process, the xADL architecture model will be transformed into a Mule ESB Workfow model and a JMS message broker model. Those two models will then be used to generate, respectively to update, the XML Mule ESB code files and the ActiveMQ configuration files. In the following, we will first explain the models more deeply (Section 4.6.1), then we will describe the transformation from the xADL architecture document to the models (Section 4.6.2) and finally the transformation from the models to the output files (Section 4.6.3).

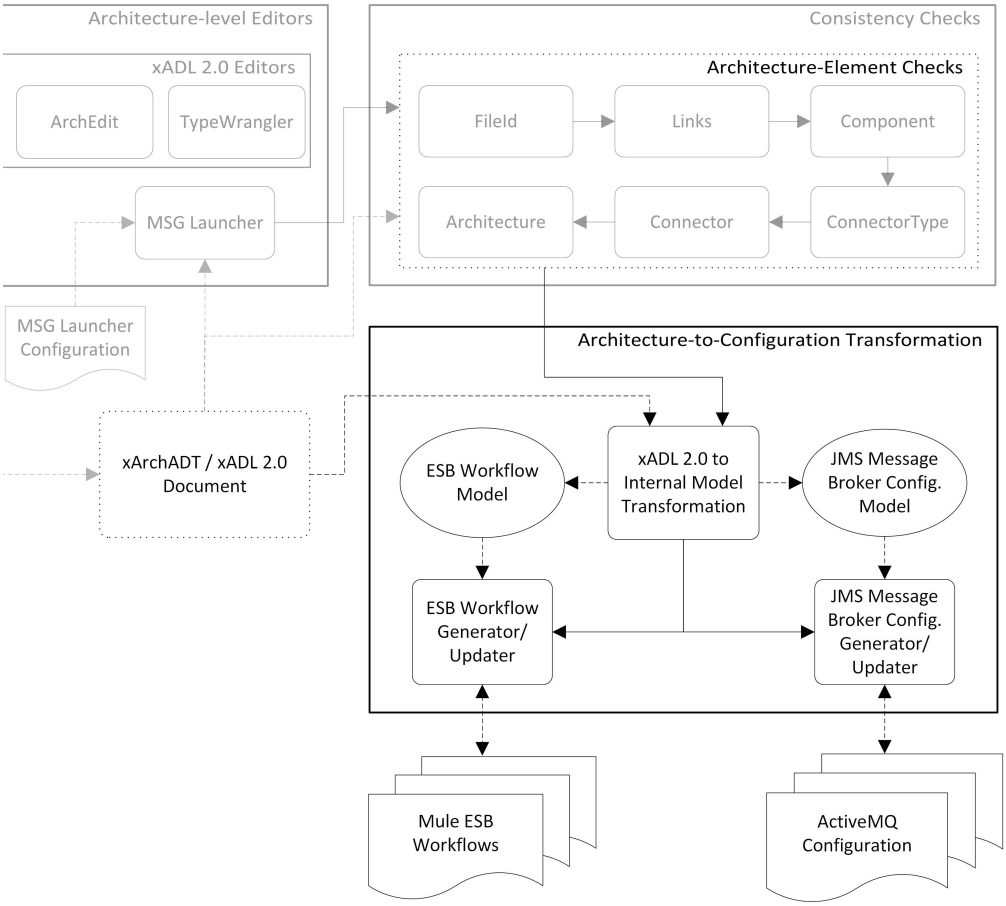


Figure 4.9: Architecture-to-Configuration Transformation Module and associated components.

4.6.1 Transformation Models

The transformation models are the internal models that get filled during the transformation process and then used to generate the output files. The following two sections will discuss those models in more detail.

4.6.1.1 Mule ESB Workflow Transformation Model

The Mule ESB Workflow transformation model[3] (depicted as *ESB Workflow Model* in Figure 4.9) holds the information that is required to generate the Mule ESB workflow configuration files. Figure 4.10 depicts the class structure of that model. The model represents one workflow including all endpoints and mandatory connection information. In the following we will discuss the classes of Figure 4.10.

[3]From know on we will call it workflow transformation model.

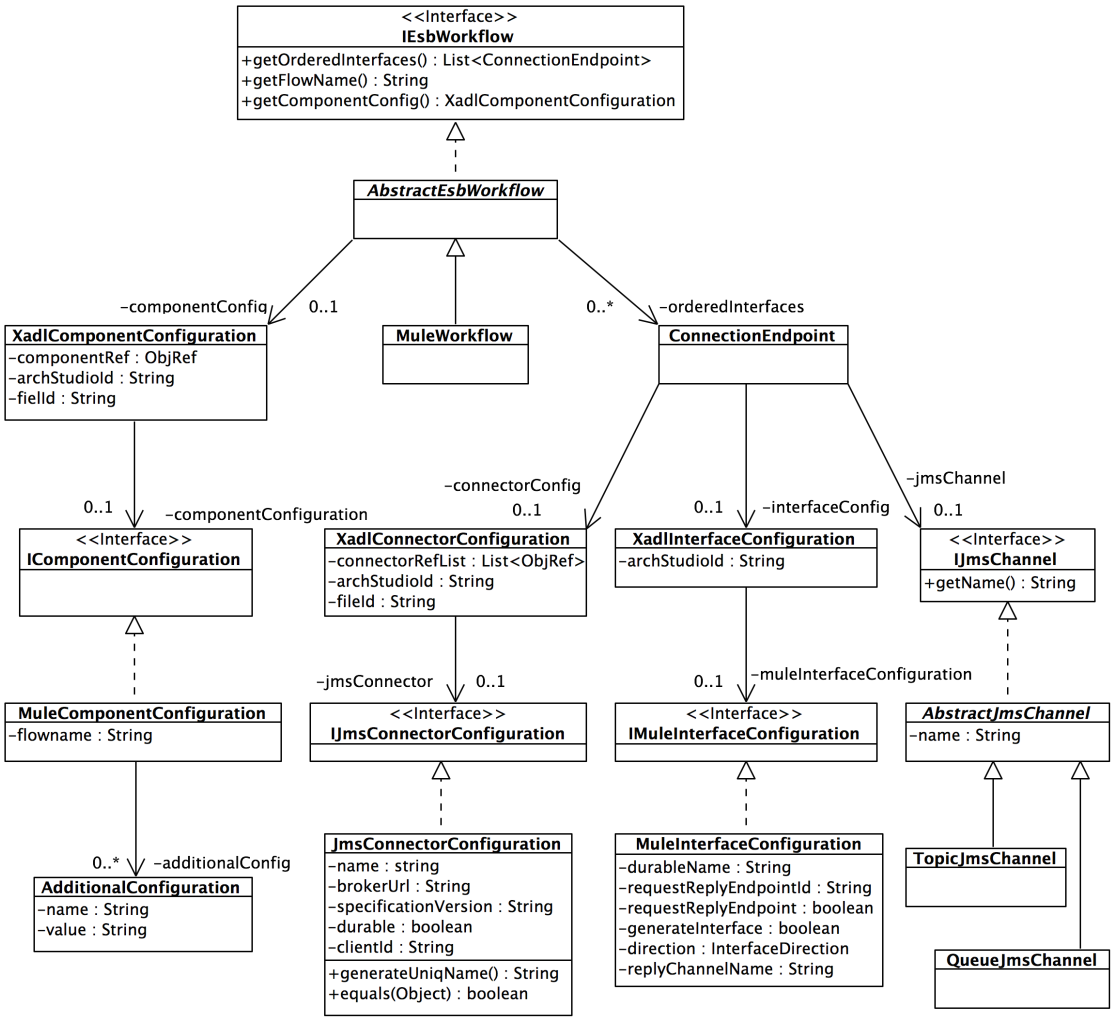


Figure 4.10: Workflow transformation model class diagram. *Note:* For a clearer diagram the getter and setter methods to access the attributes are left out.

IEsbWorkflow Represents the interface of the whole workflow.

Branch starting with XadlComponentConfiguration This branch holds all information to generate the Mule ESB workflow XML element. For a clearer separation of the classes we chose to split the information that is directly necessary for the Mule ESB XML elements, respectively JMS message broker XML elements, from information that are only needed for the transformation, respectively the output file generation. The information, which is only required for the transformation, is located in classes whose names start with *Xadl* (*XadlComponentConfiguration*, *XadlConnectorConfiguration*, *XadlInterfaceConfiguration*). For this branch of the diagram the class *XadlComponentConfiguration* holds the archStudioId, which will be required to propagate changes, and the File ID, which is used

to define the path of the output file. The class *MuleComponentConfiguration* holds the information that is mandatory for the workflow XML element, to be specific the name of the workflow. The class *AditonalConfiguration* is optional. It holds a key-value pair to define additional attributes for the workflow XML element.

Branch starting with ConnectionEndpoint This branch represents an endpoint in the workflow. It holds all information that is required to generate the XML elements that are responsible for the communication with the message broker, specifically *jms:activemq-connector* and *jms:[inbound | outbound]-endpoint* (recall Section 4.3.1). It is separated in three sub-branches:

Branch starting with XadlConnectorConfiguration This branch holds the information that is required to establish a connection to the message broker. The attributes of this branch flow into the XML element *jms:activemq-connector* in the Mule ESB workflow configuration file.

Branch starting with XadlInterfaceConfiguration This branch is used to hold the workflow endpoint information. Therefore all attributes are used to configure the XML elements *jms:[inbound | outbound]-endpoint*.

Branch starting with IJmsChannel The last branch includes the name and the type (Topic or Queue) of the JMS Channel. This information also ends up in the XML elements *jms:[inbound | outbound]-endpoint*.

4.6.1.2 JMS Message Broker Configuration Model

The JMS Message Broker transformation model[4] (depicted as *JMS Broker Configuration Model* in Figure 4.9) holds the information that is finally used to generate the configuration file for the ActiveMQ message broker. Figure 4.11 depicts the class structure of the model. Due to the fact that this configuration information is the same information that is also required to establish a connection from the workflow endpoint to the message broker our implementation uses the same class for it, to be specific *XadlConnectorConfiguration*, *IJmsConnectorConfiguration*, *JmsConnectorConfiguration*. The additional class *PersistenceAdditionalConfiguration* holds the persistence information.

4.6.2 xADL 2.0 to Internal Transformation Model

As depicted in Figure 4.9, the *xADL 2.0 to Internal Transformation Model* component is the start point for the transformation and is therefore responsible for the transformation from the xADL architectural document to the internal transformation models. After the transformation into these models, the component starts the generation/update of the output files.

The transformation from the xADL architecture document to the transformation model can be divided into three parts. The first one is the transformation of structural information. The second part is concerned with the transformation of attributes that can be transformed directly into

[4]From now on we will call it message broker transformation model

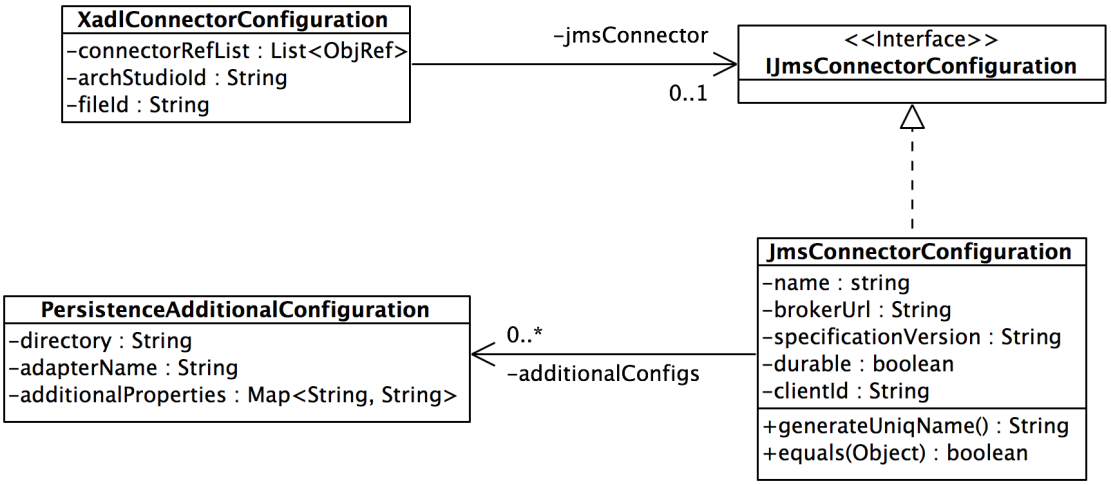


Figure 4.11: Message broker transformation model class diagram. *Note:* For a clearer diagram the getter and setter methods, to access the attributes, are left out.

the model and the third part is concerned with model attributes that are not directly transformed from the architecture.

4.6.2.1 Structural Transformation

The workflow transformation model represents one workflow with all workflow specific configuration attributes, enclosed endpoints and connection information. Furthermore one message broker transformation model contains the configuration information of one JMS communication channel. Therefore the whole architecture is transformed in a collection of workflow transformation models and a collection of message broker transformation models. The models reflect the architecture structure by the workflows, specifically the combination of the brokerUrl attribute *JmsConnectorConfiguration* and the class which implements the *IJmsChannel* interface, and the JMS message broker configuration is also taken from the *JmsConnectorConfiguration* class.

4.6.2.2 Direct Attribute Transformation

Table 4.5 shows the trivial mapping of the xADL attributes to the corresponding class attributes. Those attributes can be adopted directly from the architecture document.

xADL 2.0		**Workflow transformation model**	
xADL Element	Attribute	Class	Attribute
Component	id	XadlComponentConfiguration	archStudioId
Component	description	MuleComponentConfiguration	flowname
MuleImplementation	file_id	XadlComponentConfiguration	fileId
Additional_Configuration	name, value	AdditionalConfiguration	name, value
Interface	id	XadlInterfaceConfiguration	archStudioId
Interface	direction	MuleInterfaceConfiguration	direction
EndpointImplementation	Durable_Name	MuleInterfaceConfiguration	durableName
Connector	file_id	XadlConnectorConfiguration	fileId
Queue_Configuration	name	QueueJmsChannel	name
Topic_Configuration	name	TopicJmsChannel	name
ConnectorType	id	XadlConnectorConfiguration	archStudioId
Transport_Configuration	transportConnector	JmsConnectorConfiguration	brokerUrl
Persistence_Configuration	adapter, directory	PersistenceAdditionalConfiguration	adapter, directory
Jms_Specification_Version	value	JmsConnectorConfiguration	specificationVersion

Note: The class *PersistenceAdditionalConfiguration* is only used for the JMS message broker model.

Table 4.5: Mapping between xADL extensions and resulting tool attributes.

4.6.2.3 Indirect Attribute Transformation

Some of the class attributes of our model can't be directly transformed from the architecture document or be filled with additional information that isn't stored in the xADL document, e.g. the attribute *name* in the class JmsConnectorConfiguration. Such attributes and the origin of the data is listed in table 4.6.

Workflow transformation model		**Origin**
Class	Attribute	
JmsConnectorConfiguration	name	Has to be unique and is generated automatically. The format is "JMS_Connector_" + <unique string>.
JmsConnectorConfiguration	durable	Is set to true (default is false) if one of the interfaces is set as durable (it has a durable name configured).
JmsConnectorConfiguration	clientId	Has to be unique and is generated automatically, if it is required. The format is "Client_Id_" + <unique string>.
MuleInterfaceConfiguration	requestReplyEndpointId	Is used to couple the request and reply interface of a request-reply pattern together. It is generated automatically for the first transformed interface.
MuleInterfaceConfiguration	requestReplyEndpoint	Is set to true if the interface is part of a request-reply pattern.
MuleInterfaceConfiguration	generateInterface	If it is set to false (default is true) the interface won't be generated.
MuleInterfaceConfiguration	replyChannelName	Holds the name of the replying channel. The attribute is set for requesting interfaces and is acquired through the replying interface.

Table 4.6: Class attributes that are not directly filled with attribute values of the xADL document.

4.6.2.4 Endpoints Ordering

Due to the fact that a workflow is sequentially processed the endpoints have to be ordered. As already said, our xADL extension has a special attribute to order the endpoints manually, the attribute *Endpoint_Position_No*. But we also implemented an algorithm that can order the endpoints automatically. This algorithm uses the simple constraints that are defined by Mule ESB:

- A workflow can only start with an endpoint which has the direction *in* or *inout*.
- Endpoints which are combined by a *request-reply* pattern, or with the direction *inout*, can be everywhere in the workflow.
- A workflow can only end with an endpoint which has the direction *out*.

Note that this constraint means the direct start or end of a workflow, i.e. if there is, for example, a user-defined code after the *out* endpoint, then this isn't the direct end.

With these constraints in mind we constructed the algorithm, which returns a list of ordered interfaces where the order is:

1. *in*- or *inout*-endpoint
2. Remaining *request-reply* participating endpoints and *inout*-endpoints
3. *out*-endpoint

As can be seen, the algorithm can only guarantee the right order for a simple combination of endpoints, that is: maximal one *in*-endpoint or one *inout*-endpoint, one *request-reply* combination and one *out*-endpoint. If there are more than those endpoints the algorithm can't guarantee the right order and the user should use the *Endpoint_Position_No* to order all interfaces manually. As can be seen in the Section 4.5 our *Consistency Checking* algorithm checks if the algorithm can order the interfaces. If it can't order them an error message will be prompted with the advice to use the *Endpoint_Position_No* attribute. Algorithm 4.1 shows the ordering algorithm in pseudocode. .

4.6.3 Transformation Model to Output Files

In the following we discuss the transformation process from the internal model to the Mule ESB workflow and Apache ActiveMQ configuration files. This process is executed if the files, which should be generated, do not exist. If they already exist, the change propagation algorithm, discussed in Section 4.7, will be executed.

4.6.3.1 Mule ESB Workflow Transformation

As depicted in Figure 4.9, this step is done by the *ESB Workflow Generator/Updater* component. Table 4.7 lists the mapping between the internal model and the XML-elements of a Mule ESB workflow configuration. For each workflow object, an XML-element workflow is generated, in the defined Mule ESB workflow file, with the included endpoints. As discussed in Section 4.3.1, each endpoint needs a reference to an *activemq-connector* element, which specifies the connection information to the message broker, by the attribute *connector-ref*. This *activemq-connector* element must exist for each Mule ESB Server instance and can be used from several endpoints. This means that if, for example, there is only one message broker, all endpoints, regardless of whether they are in the same workflow file or in another one, can use this connector element as long as they are on the same Mule Server instance. However, we decided to add this element

Input: *List of unordered endpoints hold in* all-endpoints
Output: *Ordered list of endpoints hold in* ordered-endpoints

```
1  foreach endpoint in all-endpoints do
2  |  if endpoint direction is IN && endpoint is not part of a request-reply-pattern then
3  |  |  Add endpoint to ordered-endpoints;
4  end
5  if no endpoint with direction IN found then
6  |  foreach endpoint in all-endpoints do
7  |  |  if endpoint direction is INOUT then
8  |  |  |  Add endpoint to ordered-endpoints;
9  |  end
10 foreach endpoint in all-endpoints do
11 |  if endpoint direction is OUT && endpoint is not part of a request-reply-pattern then
12 |  |  saved-out-interface ← endpoint;
13 end
14 foreach endpoint in the set all-endpoints do
15 |  if endpoint is not used above then
16 |  |  Add endpoint to ordered-endpoints;
17 |  |  if endpoint is part of request-reply-pattern then
18 |  |  |  foreach second-endpoint in all-endpoints do
19 |  |  |  |  if second-endpoint ID is searched ID then
20 |  |  |  |  |  Add endpoint to ordered-endpoints;
21 |  |  |  end
22 end
23 if outInterface is set then
24 |  Add saved-out-interface to ordered-endpoints;
```

Algorithm 4.1: Order Endpoint interfaces in a workflow.

to each transformed Mule ESB configuration file and give them different reference id, because, as already said, we want to offer the possibility to use the separated workflow files on different Mule Servers. If we would include those connector elements only once then each server would have to load all workflow files. Therefore the transformation algorithm will check for each endpoint if the correct *activemq-connector* element is already set in the current workflow file. If it is, then the new endpoint will refer to this connector. If it isn't, then a new *activemq-connector* element will be generated for this workflow file, with a random UID for the *name* attribute, and the new endpoint element will get a reference to this connector. Furthermore the transformation algorithm transforms the archStudioId, which is used for the mapping between the xADL element and the transformed element, into a XML-Comment which is a XML-Child of the corresponding messaging system element, e.g. an XML-Child of the workflow XML-Element *flow* which holds the xADL id of the corresponding xADL architectural Component.

In addition to the directly transformed attributes, our transformation algorithm also adds some automatic generated values. Table 4.8 shows those values, including the discussed activemq-

connector *name* attribute and endpoint *connector-ref* element. The attributes doc:description and doc:name are not mandatory for the function of the workflow, specifically the endpoint, but increase the readability of the transformed workflows.

Workflow transformation model		**Mule ESB workflow configuration file**	
Class	Attribute	XML Element	XML Attribute
XadlComponentConfiguration	archStudioId	flow	*reference comment*
MuleComponentConfiguration	flowname	flow	name
AdditionalConfiguration	name, value	flow	name, value
XadlInterfaceConfiguration	archStudioId	endpoint	*reference comment*
MuleInterfaceConfiguration	durableName	endpoint	durableName
QueueJmsChannel	name	endpoint	queue
TopicJmsChannel	name	endpoint	topic
XadlConnectorConfiguration	archStudioId	activemq-connector	*reference comment*
JmsConnectorConfiguration	brokerUrl	activemq-connector	brokerURL
JmsConnectorConfiguration	specificationVersion	activemq-connector	specification

Table 4.7: Mapping between Workflow transformation model and Mule ESB workflow configuration file.

XML Element	**XML Attribute**	**Value**
activemq-connector	name	*Random generated UID*
endpoint	connector-ref	*Refer to activemq-connector name*
endpoint	doc:description	"Generated by ArchStudio"
endpoint	doc:name	"JMS"
flow	doc:description	"Generated by ArchStudio"
flow	doc:name	*Same as flow name*

Table 4.8: Automatically generated and predefined Mule ESB workflow configuration attributes.

4.6.3.2 Apache ActiveMQ Transformation

As depicted in Figure 4.9 this step is done by the *JMS Message Broker Configuration Generator/Updater* component. Table 4.9 describes the mapping between the internal message broker model and the transformed ActiveMQ configuration file. Note that the class attribute *adapterName* from the class *PersistenceAdditionalConfiguration* is transformed to an XML-Element name (which is an XML-Child of the *persistenceAdapter* element), rather than to a XML-Attribute. If another persistence adapter should be used, then the XML-Element, which defines this persistence adapter, has to be used, e.g. in Section 4.3.2 we used the KahaDB and therefore the XML-Element name is *kahaDB*.

Workflow transformation model		ActiveMQ configuration file	
Class	Attribute	XML Element	XML Attribute
XadlConnectorConfiguration	archStudioId	transportConnector	*reference comment*
JmsConnectorConfiguration	brokerUrl	transportConnector	uri
PersistenceAdditionalConfiguration	adapterName	persistenceAdapter	*adapter name*
XadlConnectorConfiguration	brokerUrl	*adapter name*	*reference comment*
PersistenceAdditionalConfiguration	directory	*adapter name*	directory

Note: The attribute *adapterName* gets transformed to a XML-Element name, rather then to an attribute name.

Table 4.9: Mapping between ActiveMQ transformation model and ActiveMQ configuration file.

4.7 Change Propagation

As outlined above, our approach propagates architecture changes to already transformed configuration files. In this section, we describe the change propagation algorithm which is based on the defined algorithm in Section 3.7.

The change propagation is part of the *ESB Workflow Generator/Updater* and *JMS Message Broker Configuration Generator/Updater* components which are depicted in the overview Figure 4.9. Those two components first check if the files that should be generated, already exist. If they don't exist, the steps discussed in Section 4.6.3.1 and Section 4.6.3.2 will be performed. If they exist, the files will be loaded and transformed into a DOM-tree on which the change propagation algorithm will be executed. This will be discussed in the following section. As can be seen in the overview figure, the update algorithm uses the information that is stored in the internal transformation model and that the system is checked for inconsistencies, by the consistency checking module, before the update is performed.

The tree structure of the Mule ESB configuration, respectively ActiveMQ configuration, XML-files (recall Section 4.3.1 and 4.3.2) makes the challenge of finding the right XML-element easier, since we could iterate through the XML-elements until we reach the desired element. As defined in the approach we used the unique xADL id, that is assigned by ArchStudio to each xADL element, for the mapping between the xADL element and the transformed configuration element. Furthermore the algorithm has to handle three different types of changes, which will be discussed in the following: (1) a new architecture element is added, (2) an existing architecture element is updated and (3) an architecture element is deleted.

New architecture element: If this is the case, a distinction has to be made between a Component, an Interface, a Connector or a ConnectorType.

a) *Component*: If the corresponding Mule ESB configuration file doesn't exist, it has to be generated and the new workflow can be included. If the file already exists, the new workflow has to be added to the mule root element (see Section 4.3.1 for a closer description of the structure). In the mule root element, the order of the workflows isn't of importance, so it can be added as the last element of the mule element.

b) *Interface*: The Interface information is used to create the [inbound | outbound]-endpoints of a workflow. To add a new endpoint the enclosing workflow has to exist. Due to the fact that the workflow will be generated before the endpoints, this will definitely be the case. But in contrast to the order of the Components the order of the Interfaces isn't arbitrary. To maintain the order, our update algorithm uses the list of ordered interfaces, which is generated during the transformation, as discussed in Section 4.6.2.4. The update algorithm sequentially steps through this list of interfaces and processes each one after the other. If a new interface has to be included, which isn't the first or the last interface, the update algorithm will add the interface directly after the last updated/added interface. If it is the first interface it will be added at the beginning of the workflow and if it is the last one it will be added at the end of the workflow. However, this method also has a restriction. If the workflow already contains user-defined code the new interface can't be placed somewhere into that code automatically, because the new interface will be placed directly after an interface or at the begin/end of the workflow. Note that if the position of the inserted endpoint isn't correct it can be rearranged after the transformation/update manually.

c) *Connector*: As discussed in Section 3.5.1, the Connector holds the channel information. This information is finally stored in the endpoints of the workflows. If the endpoints, which are connected to the Connector, are new, the channel information will be added by them (see b)). If they already exist the information will be updated in them (see "Updated architecture element").

d) *ConnectorType*: All stored information in the ConnectorType is directly transformed into the configuration file for the JMS server. If the corresponding configuration file doesn't exist, it has to be generated with the information stored in the ConnectorType. With an existing configuration file the update algorithm has to check if the needed sub-elements, of the root element exist (e.g. transportConnectors for the transport information). For detailed definition of the structure see Section 4.3.2. If it exists, the new information has to be placed in that sub-element and if it doesn't exist it has to be created with the new information.

Updated architecture element: If the updated element can be found in the Mule ESB or ActiveMQ configuration file, all parameters that are configured in xADL will override the existing parameters in the configuration files. Changes to the parameters that are done in the configuration file will be overwritten. At the end, the algorithm checks if the endpoint order was changed. If this is the case the endpoints are rearranged automatically.

Deleted architecture element: In that case there are two solutions: (1) delete the element and all of its sub-elements or (2) leave the element and delete only the reference to the xADL element. The second solution can be useful if the element includes user-defined code that shouldn't be deleted. This isn't recommended, because this can lead to an inconsistency of the architecture and the implementation, even though in some cases it could be necessary. We decided to leave the decision, if the whole element should be deleted or only the reference to the element in the architecture specification, up to the user.

4.8 Implementation

In the following we will discuss the installation and usage of our ArchStudio 4 extension. For this explanation we assume that the reader has a basic knowledge of the functionality of Eclipse and a basic knowledge of the standard ArchStudio 4 usage. The interested reader can find detailed information about the usage of ArchStudio 4 on the project homepage [20] and in the paper [11].

4.8.1 Installation

To compile and execute ArchStudio 4, including our messaging system generation extension, a working ArchStudio 4 is required. In the following we will only discuss the installation and execution of our extension, an installation manual for the standard ArchStudio 4 can be found on the project website [20]. For our work and this explanation we used an Eclipse Classic version 4.2.2 and the ArchStudio 4 plugin version 4.1.50 [5].

After ArchStudio 4 is installed the source code of our extension can be imported by using the standard import wizard from Eclipse. The source code of our extension is available at `http://goo.gl/eSsbUF`. The downloadable code contains a full ArchStudio 4 version, including our extension composed of the following sub-projects:

- *at.ac.tuwien.infosys.msa.archstudio4.comp.msglauncher:* Contains all classes for the *MSG Launcher* view. Discussed in Section 4.2.2.
- *at.ac.tuwien.infosys.msa.archstudio4.comp.msacc:* Contains the consistency check classes. Discussed in Section 4.5.
- *at.ac.tuwien.infosys.msa.archstudio4.comp.msgenerator:* Contains the classes for the Architecture-to-Configuration Transformation. Discussed in Section 4.6.
- *at.ac.tuwien.infosys.msa.xadl:* Contains the extended xADL schema documents, from Section 4.4.
- *at.ac.tuwien.infosys.msa.example:* Contains the ongoing example, which was used for several explanations above. Presented in Section 3.3.
- *at.ac.tuwien.infosys.msa.evaluation:* Contains the architecture files that we used for the evaluation of the system. Presented in Chapter 5.

After the source code is imported the compilation can be started by launching it as an Eclipse application or by using the included Eclipse launch script that can be found in the project *edu.uci.isr.archstudio4* in the directory *res/eclipse*. This will start a new Eclipse instance with the ArchStudio 4 plugin including our extension installed.

[5] http://www.isr.uci.edu/projects/archstudio-4/updatesite-4.2/

4.8.2 Usage

After the new ArchStudio 4 instance is running the standard functionality including our extension can be used. In the following we will discuss the usage of our extension. The basic usage of ArchStudio can be studied at the project homepage [20]. To discuss the usage of our system, we will use the ongoing example, which was presented in Section 3.3. The example can also be found in the source code of our extension in the sub-project *at.ac.tuwien.infosys.msa.example*.

First of all, the example has to be imported as a project into the current workspace. This can be done by using the Eclipse import system.
The imported project contains two files:

options.xml is the current configuration file that contains the paths and the IDs for the output files (the content of the file is also depicted in the example box 15).

thesis_example.xml is the xADL 2.0 file for the ongoing example.

The architecture can now be shown and altered with the standard ArchStudio tools *Archipelago*, *ArchEdit* and *Type Wrangler*. Note that the standard functionality of *ArchEdit* is now extended with our schema extension described in Section 4.4. To transform the architecture into the Mule ESB and ActiveMQ configuration files the xADL file has to be opened with our extension named *MSG Launcher*. Figure 4.12 depicts the view of the *MSG Launcher* with the information of the *options.xml* file. The view is divided into four parts (depicted by four red rectangles in the figure):

1. The topmost part loads and saves the configuration file. By using the *Load Config* button a new configuration file can be loaded and by using the *Save Config* button the current output file definition can be saved to a configuration file. For the figure we loaded the *option.xml* file.

2. The second part presents the IDs and file paths for the Mule ESB output files. By using the *Add Entry* button a new entry can be created. If a new entry is created the system adds a new entry to the table left of the button and sets an unique ID. This ID can be altered by selecting the table cell. Furthermore the file path can be changed by selecting the desired cell and using the button that appears at the end of the cell. To delete an entry the button *Delete Entry* can be used. By using the button the current selected row will be deleted.

3. The third part presents the IDs and file paths for the ActiveMQ output files. The usage is the same as before.

4. The bottommost part contains the button that starts the consistency checks and the transformation process.

After the configuration file is loaded the use of the *Generation* button starts the consistency check discussed in Section 4.5. As discussed before if the consistency of the architecture is violated a warning is shown and the user can decide whether the transformation shall be continued or canceled for soft consistency violations. For hard consistency violations an error is shown and the transformation is canceled. After all consistency checks are done, the transformation process will start.

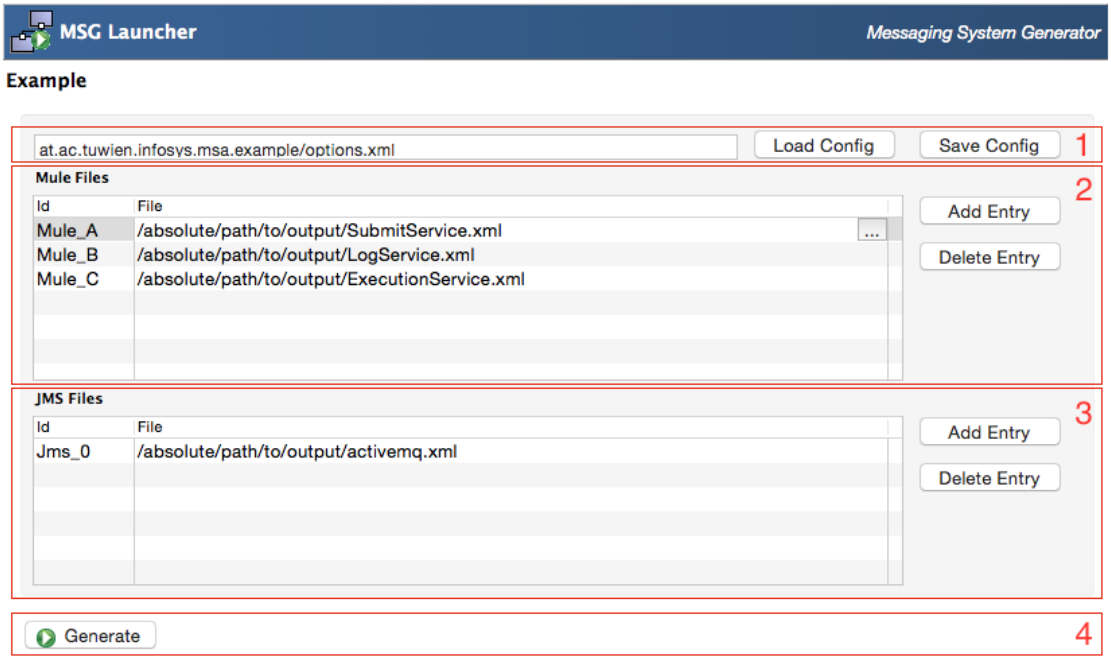


Figure 4.12: MSG Launcher view

CHAPTER 5

Evaluation

This chapter presents the evaluation of the approach and prototype implementation of this thesis. We will evaluate each part of our work separately, starting with the *Architecture-level Editor* followed by *Consistency Checking* and the *Architecture-to-Configuration transformation*. At the end we will evaluate the *Change propagation* mechanism. As evaluation scenario we will use the real world problem "Parking Management System", which was presented in Section 1.2.1.

5.1 Objectives

The overall purpose of the evaluation is to prove that the approach (Chapter 3) and our suggested implementation (Chapter 4) can support a development team that has to deal with the challenges that arise during the development of a service-centric system, as discussed in the introduction of this thesis (Section 1.1). To do this, we will prove that the approach fulfils the defined goals in Section 1.3. The goals of the work can be split into the following segments: (1) offering a method to model the high-level architecture of service-centric system by the use of components and connectors, (2) offering a method to enrich the model with specific configuration information for the MOM and the service endpoints, (3) check the consistency of the described architecture, (4) use architecture-to-configuration transformation to generate the messaging skeleton and MOM configuration, and (5) propagate changes in the architecture to the transformed system. The following sections will focus on these segments and discuss the advantage of our approach and the implemented prototype.

First of all, we will use our prototype to model and configure the evaluation scenario. This step can't be evaluated directly. However, we will use this modelled architecture as a basis for the other evaluations and thereby we can prove that our xADL extensions are enough to model and configure a real world problem. Furthermore we can prove that the resulting xADL document has enough information to perform consistency checks, architecture-to-configuration transformation and change propagation on it.

Second, we validate the automatic *Consistency Checker* which helps the development team to detect inconsistencies in the architecture, as well as, in the configuration. To validate, it we

first analyze the checks, defined in the approach (Section 3.4), and compare the results with the effort that has to be done by a development team to do the same checks without the automatic support. After proving the approach we will validate our implementation by proving that all defined checks are covered by the implementation and prove if they and the extended checks (Mule ESB and ActiveMQ specific checks) are implemented correctly.

Third, we will validate the architecture-to-configuration transformation. The transformation is evaluated by first creating a checklist that includes all elements and their attributes that have to be included in the output-files. After completing the checklist we will use our tool to transform the architecture to Mule ESB and ActiveMQ configuration files and finally we will check these output-files with the predefined checklist.

Last of all, the change propagation mechanism will be validated. We will evaluate this mechanism by changing the evaluation scenario in a way that all cases (defined in Section 4.7) are executed, one after another. By using this method we can prove that all changes will be propagated in the anticipated way.

As evaluation scenario we will use the real world scenario *Parking Management System*, presented in Section 1.2.1. The scenario reflects a real world project that was developed during an internship. For this thesis we reduced (by removing the business logic of the services) and generalised the real world system to the one we represented in the Section 1.2.1. However, the problem and general structure of our scenario is the same and can be easily transferred back to the original one. We included the xADL file of the *Parking Management System* scenario and all evaluation tests in the downloadable source code of the project. The source code is available at `http://goo.gl/eSsbUF`.

5.2 Evaluation Scenario and Preparation

This part of the evaluation focuses on the modelling and configuration utility of our approach. Our approach, respectively the implemented tool, uses xADL as the underlying architecture description language and ArchStudio 4 as tool support to generate and edit the architecture document.

Figure 5.1 depicts the architecture of the *Parking Management System* evaluation scenario, planned with *Archipelago*.

The example contains a Filter Service, an Aggregator Service, a POS Service, each of them a composition of different components, and five message queues/topics. Note that in a real world usage of the scenario there would be more services than one of each kind, but for our evaluation one of each is enough. The Filter Service is composed of two components. The first component, called *Process Dynamic Change Events*, is responsible for processing dynamic change events from parking sites (e.g. amount of free parking slots) and the second, called *Process Structural Events*, is responsible for processing structural events (e.g. amount of overall parking slots). An Aggregator Service is a composition of three components. The first, called *Subscribe and Aggregate Structural Data*, aggregates structural data from the Filter Service. The second, called *Check Structural Changes and Publish Updates*, verifies received structural data for changes and if changes occurred, it will provide the new information to the POS Services. The third component, called *Provide Initial Data*, is responsible for providing initial data (e.g. Structural

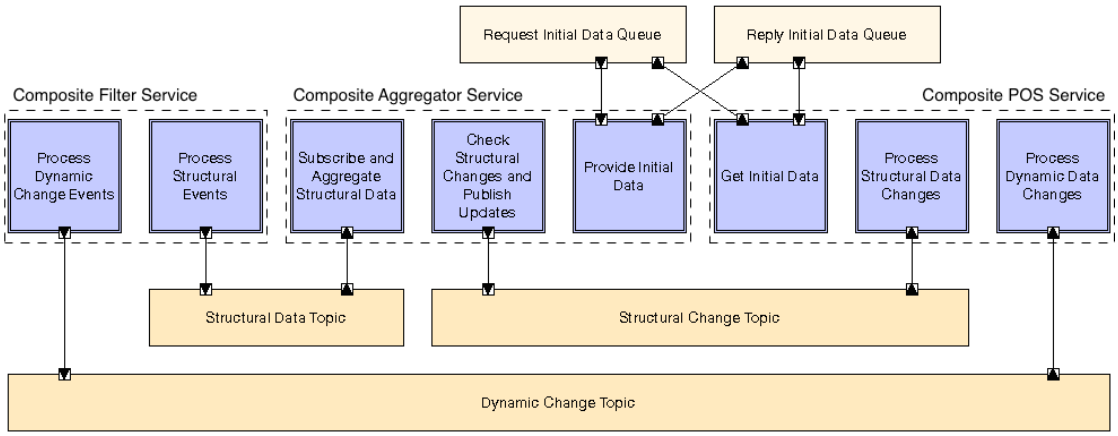


Figure 5.1: "Parking Management System" evaluation scenario modeled in Archipelago. Service components depicted in blue and message broker connectors in beige.

information of a car park) to new POS Services. A POS Service is composed of three components. One, called *Get Initial Data*, to request the structural data from the Aggregator Service. One, called *Process Structural Data Changes*, to subscribe the topic where the Aggregator Service provides updates of the structural data and one, called *Process Dynamic Data Changes*, to receive and process the dynamic data provided by the Filter Services. Each of these components will end up as a separated Mule ESB workflow. The actual business logic of each workflow is irrelevant at this architectural level and for our approach and therefore also for the evaluation.

For the evaluation we only used one message broker, consequently all message queues and topics got the same xADL *ConnectorType*. For the components we used three different xADL *ComponentTypes*, one type with one *out*-interface, one type with one *in*-interface and one type with one *in* and one *out*-interface. Additionally for each Interface and Signature we used the same InterfaceType, called *MessageType*. Furthermore each xADL *Component* had the implementation *MuleImplementation* and all xADL *Interfaces* the implementation *EndpointImplementation*.
In the following the configuration of the Component, the surrounded Interfaces, the ConnectorType and Connectors is listed.

Output-File definition

Mule ESB:
Filter_Services
Aggregator_Service
POS_Service

Message Broker:
Jms_broker

Component: Process Dynamic Change Events

MuleImplementation: file_id=Filter_Services

Component: Process Structural Events
MuleImplementation: file_id=Filter_Services
Component: Subscribe and Aggregate Structural Data
MuleImplementation: file_id=Aggregator_Service
In-Endpoint: EndpointImplementation
Durable_Name: name=content_static_data
Component: Check Structural Changes and Publish Updates
MuleImplementation: file_id=Aggregator_Service
Component: Provide Initial Data
MuleImplementation: file_id=Aggregator_Service
In-Endpoint: EndpointImplementation
Reply_To_Queue: reference to Out-Endpoint
Out-Endpoint: EndpointImplementation
Connection_To_Request_Endpoint: reference to In-Endpoint
Component: Get Initial Data
MuleImplementation: file_id=POS_Service
In-Endpoint: EndpointImplementation
Connection_To_Request_Endpoint: reference to Out-Endpoint
Out-Endpoint: EndpointImplementation
Reply_To_Queue: reference to In-Endpoint
Component: Process Dynamic Data Changes
MuleImplementation: file_id=POS_Service
Durable_Name: name=pos_dynamic_data
Component: Process Structural Data Changes
MuleImplementation: file_id=POS_Service
Durable_Name: name=content_static_data_changes
ConnectorType: JMS Connector Type
Persistence_Configuration: adapter=kahaDB
directory=connect/kahadb
Transport_Configuration: transportConnector=tcp://127.0.0.1:61616
Connector: Dynamic Change Topic
Topic_Configuration: name=dynamic.change.topic
Connector: Structural Change Topic
Topic_Configuration: name=structural.change.topic
Connector: Structural Data Topic
Topic_Configuration: name=structural.data.topic
Connector: Reply Initial Data Queue
Queue_Configuration: name=reply.initial.data.queue
Connector: Request Initial Data Queue
Queue_Configuration: name=request.initial.data.queue

5.3 Consistency Check Evaluation

The automatic *Consistency Checks* help the development team by detecting inconsistencies as soon as possible. Even in a smaller architecture like our evaluation scenario, several consistency checks were done (recall Section 3.4 and 4.5).

5.3.1 Evaluation Method

This evaluation was divided into two parts: first, we evaluated if our approach can detect the general inconsistencies defined in Section 3.4 and we analyzed what effort an architect would have to make to do the same checks manually. In the second part we proved that our implementation includes all the consistency checks defined in the approach and that these and the additional checks defined in Section 4.5 were implemented correctly.

Therefore we analyzed the following inconsistencies for the first part:

1. Two different channels have the same channel name.
2. Two interfaces, that are connected over a message channel, have the same direction.
3. A messaging service endpoint isn't connected to a channel.
4. Misspelling in the channel name of a messaging service endpoint.
5. A publish-subscriber channel is used for publishing messages within the scope of a request-reply pattern.
6. Several services are listening to a point-to-point connection.
7. Two endpoints, connected to the same channel, are configured for different channel types.
8. The connection configuration of the interfaces is inconsistent.
9. The request-reply service endpoints are not defined correctly.

For the second part we evaluated the implemented consistency checks. For each check we used the *Parking Management System* as a base and changed the system in a way that it had the current inconsistency. The following list shows the consistency checks defined in Section 4.5 and in Table B.1 and B.2:

10. Is there a link between two connectors or two components?
11. Are the interfaces, used for a request-reply group, defined correctly?
12. Do two interfaces, that are connected by a link, have the same direction?
13. Can the interfaces, on a component, be arranged automatically?
14. Does the component have more than one *in* or one *out*-interface that is not part of a request-reply group?

15. Is the channel name and the channel type (publish-subscriber or point-to-point) set?
16. Is the link set correctly?
17. Do two different connectors, with the same connector type, have the same channel name?
18. Is an *inout*-interface connected to a publish-subscriber channel?
19. Does an *out*-interface of a queue have several connected links?
20. Does a component have more than one link at an *in* or *out*-interface?
21. Is a publish-subscriber channel used for publishing a message within the scope of a request-reply pattern?
22. Do all used connectors and components have an implementation?

5.3.2 Result & Discussion

The following list discusses the consistency checks defined in the approach. As described above we analyzed what effort an architect would have to make, to search manually for inconsistencies in the *Parking Management System* scenario. The list refers to the inconsistencies defined before and specifies which implemented consistency check searches for those kind of inconsistencies and how much effort is required for the manual checking.

1. **Check**: Two different channels have the same channel name.
   Implemented by check no.: 17.
   Effort for manual checking: Due to the fact that the channel names are declared in the endpoints of the Mule workflow and not predefined in the ActiveMQ configuration, an architect has to traverse all Mule workflow configuration files and identify the message endpoints. Subsequently he has to pairwise compare those information across all workflows. Specifically, the architect has to do $n*(n-1)/2$ checks, i.e. he has to do 45 checks for the 10 interfaces in our example.

2. **Check**: Two interfaces, that are connected over a message channel, have the same direction.
   Implemented by check no.: 12.
   Effort for manual checking: Again the architect has to traverse all Mule workflows and identify the message endpoints and finally he has to pairwise compare them, i.e. he must again do 45 checks.

3. **Check**: A messaging service endpoint isn't connected to a channel.
   Implemented by check no.: This check is part of future work where we will use the ArchStudio 4 tool *Archlight* to implement several separate checks.
   Effort for manual checking: For this check the architect has to traverse all Mule workflows and check the message endpoints. Therefore he has to check 10 endpoints.

4. **Check**: Misspelling in the channel name of a messaging service endpoint.
   Implemented by check no.: Can't happen because of the transformation.
   Effort for manual checking: Needs the same effort as defined in check 1 and 2. Specifically, it requires the architect to perform 45 checks.

5. **Check**: A publish-subscriber channel is used for publishing messages within the scope of a request-reply pattern.
   Implemented by check no.: 21.
   Effort for manual checking: This check requires the architect to verify all requesting endpoints and check if the configured channel is not a topic. In our scenario an architect would only need to check one interface.

6. **Check**: Several services are listening to a point-to-point connection.
   Implemented by check no.: 19., 20.
   Effort for manual checking: To validate this an architect has to traverse all Mule workflows and identify the message endpoints that are listening to a queue. Subsequently he has to check if each queue name can only be found once. Our scenario uses primarily topics and has only 2 queues. Therefore the architect has to check 2 message endpoints.

7. **Check**: Two endpoints, connected to the same channel, are configured for different channel types.
   Implemented by check no.: Can't happen because of the transformation.
   Effort for manual checking: This validation requires, as in the first consistency check, to traverse all Mule workflows and identify the message endpoints. This information can then be used to find the endpoints that are connected to the same channel. Those channels, then have to be checked if they are connected to the same connector element, or at least to connector elements with the same configuration. This again, leads to 45 checks for our scenario.

8. **Check**: The connection configuration of the interfaces is inconsistent.
   Implemented by check no.: Can't happen because of the transformation.
   Effort for manual checking: This requires an architect to identify all message endpoints in the Mule workflows. Then he has to check the referred message broker connector element and the configuration attributes of it. For our scenario he has to check at least 10 endpoints including their connectors. Due to the fact that our scenario has three different Mule workflow files and three different connectors, an architect has to check those three connectors and the 10 connected interfaces, i.e. he has to do 13 checks.

9. **Check**: The request-reply service endpoints are not defined correctly.
   Implemented by check no.: 11., 18.
   Effort for manual checking: This check requires to first identify all request-reply endpoints and validate their configuration. In our scenario at least 4 endpoints have to be checked.

As can be seen, manual consistency checking takes a lot of time and requires a deep knowledge of the used technology and is therefore highly error-prone. In fact if the architect performs

one check after another he has to do at least 210 checks to make sure the *Parking Management System* is free of inconsistencies. Note that the amount of checks can be reduced if some checks are done in parallel, or information of previous checks are saved and used for new checks. Yet the amount of checks will still be high, take a lot of time and is error-prone. The list also shows that all inconsistencies are checked by our implemented consistency checking algorithm or prevented by using the automatic architecture-to-configuration transformation, which will be evaluated in Section 5.4.

The second part of the evaluation was concerned with the implemented consistency checks. We evaluated the checks listed above by changing the *Parking Management System* in a way that it exhibits the inconsistencies, one after another. Note that after each check the altered scenario was reset to the original base. Table 5.1 presents the result of the evaluation and if each defined consistency check is implemented.

No.	Result	No.	Result
10.	✓	17.	✓
11.	✓	18.	✓
12.	✓	19.	✓
13.	✓	20.	✓
14.	✓	21.	✓
15.	✓	22.	✓
16.	✓		

Table 5.1: Evaluation of the implemented Consistency Checks. A checkmark sign that the Consistency Check is implemented and the tool detected the corresponding inconsistency.

It can be said conclusively that the defined *Consistency Checks* of our approach can support the planning and development of a message-based system by offering an algorithm that can detect inconsistencies at an early stage. Additionally, we proved that our implemented *Consistency Checks* fulfill the checks that we defined in the approach and even extend them.

5.4 Architecture-to-Configuration Transformation Evaluation

In the following the Architecture-to-Configuration transformation will be evaluated. We used our tool to transform the *Parking Management System* into Mule ESB workflows and to generate the corresponding configuration for Apache ActiveMQ. This scenario needs the complete range of functions of our approach/implementation, including Queues, Topics, a request-reply combination and data persistence configurations. Thus we can safely draw the conclusion that our solution is capable of supporting development teams in real world architecture-to-configuration transformation.

5.4.1 Evaluation Method

As said above, we used our tool to transform the "Parking Management System" into several Mule ESB workflows and to generate the corresponding Apache ActiveMQ message broker configuration. The transformation generated three separate Mule ESB workflow files (one for the Filter Services, one for the Aggregator Services and one for the POS Services) and one Apache ActiveMQ configuration file.

Furthermore, we evaluated the endpoint order algorithm specifically. Unfortunately in the *Parking Management System* scenario the components have a at most two endpoints, but to test the algorithm we needed three or more endpoints. Therefore, we had to change some components for this test. Specifically, we added two *inout*-endpoints to the *Provide Initial Data* Component and one *in*-endpoint and one *out*-endpoint to the *Get Initial Data* Component. Additionally, we added a new Connector with two *inout*-interfaces. This new Connector is required so that the two *inout*-endpoints of the *Provide Initial Data* Component can be connected to them. The new *in*-endpoint of the Component *Get Initial Data* was connected to the *Structural Change Topic* Connector and the *out*-endpoint to the *Dynamic Change Topic* Connector. These endpoints should be ordered as follows: (1) *in*-endpoint, (2) the original endpoints, which are combined by a request-reply pattern and (3) the *out*-endpoint. With these additional endpoints, the altered *Get Initial Data* Component had an endpoint combination which the algorithm should be capable to order and the altered *Provide Initial Data* Component had a combination where the algorithm reached its limit, because it can't know which *inout*-endpoint should be the first one. Thus we had two components which exhausted the ordering algorithm. But due to the fact that before the changes are propagated the consistency check is executed and during these checks also the interface combination is tested, an error message should be shown and the transformation canceled. Note that we did these extensions only to test the algorithm, with regards to the behavior of the system, and that they don't have any usage for the original scenario.

In the first part, of the evaluation, we analyzed the structure of the XML-based output files and in the second part we checked if the transformation includes all the required information. To validate this we first defined a check-list which we then used to check the transformed output-files. In the end, we loaded the workflow files into Mule ESB, compiled it there and started the message broker ActiveMQ with the generated configuration file, to test if the files are runnable. For the evaluation we used MuleStudio version 3.5 and Apache ActiveMQ version 5.9.

5.4.2 Result & Discussion

Figures 5.2a to 5.2d depict the structure and important attributes of the output files. For a clearer representation we only included the required attributes, i.e. the attributes that are mandatory for a working messaging system, and the xADL ids, that are needed to map an output element back to the xADL element.

The following depicts the check-list, which was created before the evaluation and which holds all information that should be included in the output-files. A checkmark, at the end of a line, marks that the element or attribute is included in the output-file, an X marks that the element isn't included or it was transformed incorrectly.

Workflow: Process Dynamic Change Events

Workflow is in the output-file *Filter_Service.xml* ✓
Workflow has the name *Process_Dynamic_Change_Events* ✓
Has a XML-child with the xADL id of the xADL Component ✓
Has an *out*-endpoint ✓
out-endpoint:
Is connected to topic *dynamic.change.topic* ✓
Has a reference to the *activemq-connector* ✓
Has a XML-child with the xADL id of the xADL Interface ✓

Workflow: Process Structural Events
Workflow is in the output-file *Filter_Service.xml* ✓
Workflow has the name *Process_Structural_Events* ✓
Has a XML-child with the xADL id of the xADL Component ✓
Has an *out*-endpoint ✓
out-endpoint:
Is connected to topic *structural.data.topic* ✓
Has a reference to the *activemq-connector* ✓
Has a XML-child with the xADL id of the xADL Interface ✓

Workflow: Subscribe and Aggregate Structural Data
Workflow is in the output-file *Aggregator_Service.xml* ✓
Workflow has the name *Subscribe_and_Aggregate_Structural_Data* ✓
Has a XML-child with the xADL id of the xADL Component ✓
Has an *in*-endpoint ✓
in-endpoint:
Is connected to topic *structural.data.topic* ✓
Has the durable name *content_static_data* ✓
Has a reference to the *activemq-connector* ✓
Has a XML-child with the xADL id of the xADL Interface ✓

Workflow: Check Structural Changes and Publish Updates
Workflow is in the output-file *Aggregator_Service.xml* ✓
Workflow has the name *Check_Structural_Changes_and_Publish_Updates* ✓
Has a XML-child with the xADL id of the xADL Component ✓
Has an *out*-endpoint ✓
out-endpoint:
Is connected to topic *structural.change.topic* ✓
Has a reference to the *activemq-connector* ✓
Has a XML-child with the xADL id of the xADL Interface ✓

Workflow: Provide Initial Data
Workflow is in the output-file *Aggregator_Service.xml* ✓
Workflow has the name *Provide_Initial_Data* ✓

Has a XML-child with the xADL id of the xADL Component	✓
Has an *in*-endpoint	✓
in-endpoint:	
Is connected to queue *request.initial.data.queue*	✓
Has the exchange pattern *request-response*	✓
Has a reference to the *activemq-connector*	✓
Has a XML-child with the xADL id of the xADL Interface	✓

Workflow: Get Initial Data

Workflow is in the output-file *POS_Service.xml*	✓
Workflow has the name *Get_Initial_Data*	✓
Has a XML-child with the xADL id of the xADL Component	✓
Add the message property MULE_REPLYTO with value *jms://reply.initial.data.queue* before the *request-reply* element	✓
Has an *request-reply* element with embedded *in*-endpoint and *out*-Interface	✓
In *request-reply* element: *out*-Interface before *in*-endpoint	✓
out-endpoint:	
Is connected to queue *request.initial.data.queue*	✓
Has a reference to the *activemq-connector*	✓
Has a XML-child with the xADL id of the xADL Interface	✓
in-endpoint:	
Is connected to queue *reply.initial.data.queue*	✓
Has a reference to the *activemq-connector*	✓
Has a XML-child with the xADL id of the xADL Interface	✓

Workflow: Process Dynamic Data Changes

Workflow is in the output-file *POS_Service.xml*	✓
Workflow has the name *Process_Dynamic_Data_Changes*	✓
Has a XML-child with the xADL id of the xADL Component	✓
Has an *in*-endpoint	✓
in-endpoint:	
Is connected to topic *dynamic.change.topic*	✓
Has the durable name *pos_dynamic_data*	✓
Has a reference to the *activemq-connector*	✓
Has a XML-child with the xADL id of the xADL Interface	✓

Workflow: Process Structural Data Changes

Workflow is in the output-file *POS_Service.xml*	✓
Workflow has the name *Process_Structural_Data_Changes*	✓
Has a XML-child with the xADL id of the xADL Component	✓
Has an *in*-endpoint	✓
in-endpoint:	
Is connected to topic *structural.change.topic*	✓

Has the durable name *content_static_data_changes*	✓
Has a reference to the *activemq-connector*	✓
Has a XML-child with the xADL id of the xADL Interface	✓

Message Broker Configuration

persistenceAdapter element is included	✓
persistenceAdapter element includes *kahaDB* element	✓
kahaDB element:	
Has the attribute *directory="connect/kahadb"*	✓
Has a XML-child with the xADL id of the xADL ConnectorType	✓
transportConnectors element is included	✓
transportConnectors element includes *transportConnector* element	✓
transportConnector element:	
Has the attribute *uri="tcp://127.0.0.1:61616"*	✓
Has a XML-child with the xADL id of the xADL ConnectorType	✓

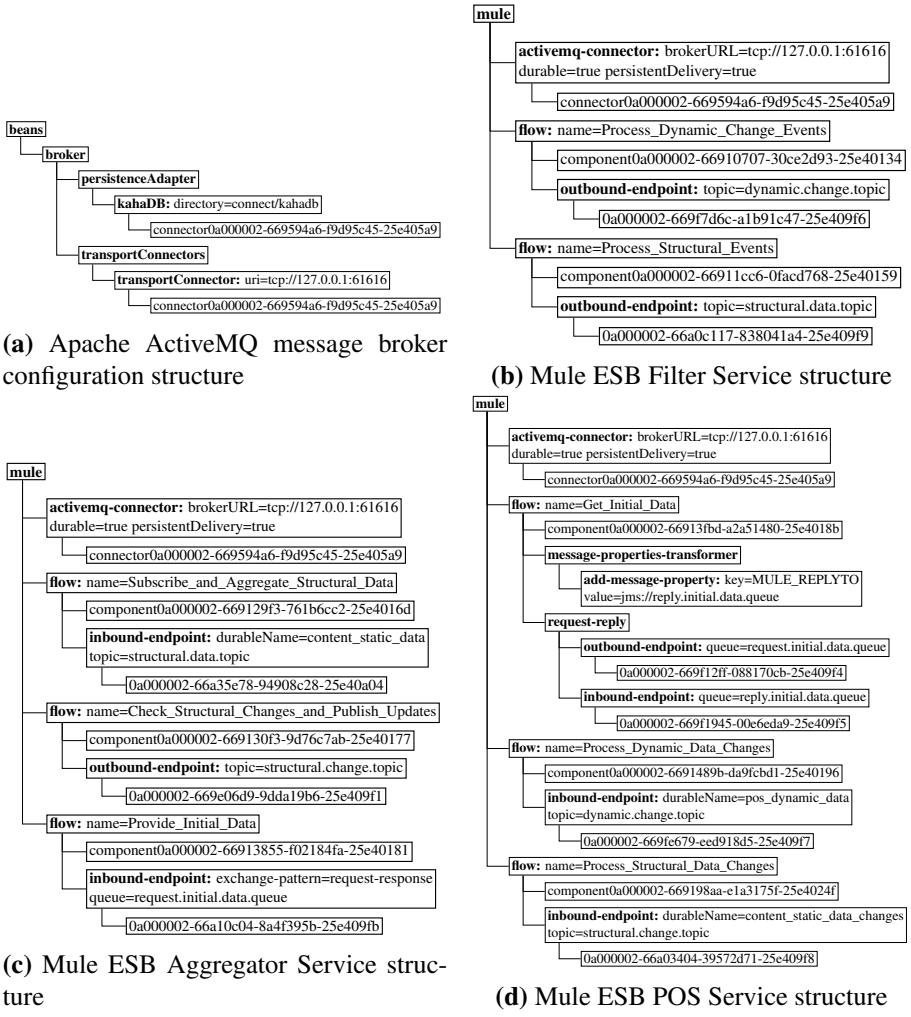
Configuration usableness

File *Filter_Service.xml* can be compiled, without errors, by Mule ESB	✓
File *Aggregator_Service.xml* can be compiled, without errors, by Mule ESB	✓
File *POS_Service.xml* can be compiled, without errors, by Mule ESB	✓
Apache ActiveMQ can be started with the configuration file *activemq.xml*	✓

Endpoint Ordering Algorithm

Get Initial Data endpoints order:	
First endpoint: New *in*-endpoint (*Structural Change Topic*)	✓
Second endpoint: Request-Reply combination	✓
Third endpoint: New *out*-endpoint (*Dynamic Change Topic*)	✓
Provide Initial Data endpoints order:	
First endpoint: Request-Reply combination	✗
Second endpoint: New *inout*-endpoint (publisher)	✗
Third endpoint: New *inout*-endpoint (subscriber)	✗
Show error message that the interface could not be ordered automatically.	✓
Provide Initial Data endpoints order with *Endpoint_Position_No*:	
First endpoint: Request-Reply combination	✓
Second endpoint: New *inout*-endpoint (publisher)	✓
Third endpoint: New *inout*-endpoint (subscriber)	✓

As can be seen in the check-list, all architecture elements, including our xADL extensions which are holding the configuration, have been transformed correctly. The only problem that we encountered was the automatic ordering of the extended *Provide Initial Data* Component. In this scenario the *Consistency Check* algorithm showed an error with the information that the interfaces can not be ordered automatically. But we expected that, because the algorithm didn't have enough information to order them correctly. After we gave the algorithm the required



(a) Apache ActiveMQ message broker configuration structure

(b) Mule ESB Filter Service structure

(c) Mule ESB Aggregator Service structure

(d) Mule ESB POS Service structure

Figure 5.2: XML-tree structure of the Mule ESB workflows.

ordering information, by using the additional *Endpoint_Position_No* attribute, it ordered the endpoints correctly. Furthermore the evaluation shows that the transformed output-files can be used in Mule ESB and as a message broker configuration for Apache ActiveMQ. Due to the XML-format (see Figure 5.2 for the XML tree structure) and the sequential processing of the workflows, those files can now be easily extended with additional business logic which finally completes each service. This also proves that our approach and implementation support the separation of responsibilities. A developer, who is responsible for the business logic, doesn't have to take care of the messaging relevant parts and an architect, who is responsible for the messaging system, doesn't have to take care of the business logic of each service.

5.5 Change Propagation Evaluation

In the following the change propagation of our approach and implementation will be evaluated.

5.5.1 Evaluation Method

To evaluate this aspect we performed several changes to the already transformed *Parking Management System* from Section 5.4. To get a clear result we tested all changes discussed in Section 4.7, by (1) performing the change in the xADL document, (2) then restarting the transformation on top of the already generated output-files and finally (3) checking if the changes have been propagated. The xADL basis for each test was the original *Parking Management System* scenario architecture document.

The following list defines the executed tests:

1. **Test:** Add new Component, Interface, Connector, ConnectorType.
   Procedure: This test adds a new Component, called "NewComponent", with one *in*-interface and a new Connector, called "NewConnector", which is a queue with the queue name "new.queue". For the Connector a new ConnectorType is generated, with the address "tcp://127.0.0.2:61616". Furthermore the Component *Process Dynamic Data Changes* gets an additional *out*-interface which is connected to the write interface of the new queue. The read interface of the queue is connected to the *in*-interface of the new Component. The new Component should be saved in the "POS_Service.xml" file and the configuration for the new ConnectorType in a new message broker configuration file.
   Expected Result: The new workflow and a new activemq-connecter should be generated in the "POS_Service.xml" file. The workflow should have an *in*-endpoint that is connected to the new queue and this endpoint should have a reference to the new activemq-connector. Furthermore the Component *Process Dynamic Data Changes* should have a new interface and a new message broker configuration file with the address "tcp://127.0.0.2:61616" should be generated.

2. **Test:** Change a channel name.
   Procedure: Change the name of the *Dynamic Change Topic* channel from "dynamic.change.topic" to "dynamic.change.topic.changed".
   Expected Result: The new channel name should be "dynamic.change.topic.changed". This should be changed in the Components *Process Dynamic Change Events* and *Process Dynamic Data Changes*.

3. **Test:** Reconnect an Interface.
   Procedure: Reconnect the *in*-interface of Component *Process Dynamic Data Changes* from the Topic *Dynamic Data Change* to *Structural Change Topic*.
   Expected Result: The *in*-endpoint of the workflow *Process Dynamic Data Changes* should be connected to the *Structural Change Topic* topic.

4. **Test:** Change order of the Interfaces.
   Procedure: This test is divided into two subparts. (1) Redo the steps of Test 1 (this

is necessary because we need a Component with two Interfaces which are not part of a request-reply pattern) and then (2) change the order of the Interfaces (from first *in*, second *out* to first *out* and second *in*) of Component *ProcessDynamic Data Changes* by using the extension *Endpoint_Position_No*.
Expected Result: The first endpoint of the workflow *Process Dynamic Data Changes* should be an *out*-endpoint, which is connected to the channel "new.queue", and the second one an *in*-endpoint, which is connected to the channel *Dynamic Change Topic*.

5. **Test:** Delete the reference (xADL id) to a xADL Component.
Procedure: This test should check the behaviour of the system if a component is deleted. This test will only delete the reference between the architecture document and the transformed system, i.e. delete the xADL id reference in the corresponding workflow. To do this, the Component *Process Dynamic Data Changes* will be deleted. The system should then ask if we want to delete the whole corresponding workflow or only the reference. For this test only the reference will be deleted.
Expected Result: The comment that holds the xADL id and that is a XML-child of the workflow should be deleted.

6. **Test:** Delete a Component.
Procedure: In contrast to the previous test, this test will delete the whole workflow and not only the reference to the architectural component. Therefore, the Component *Process Dynamic Data Changes* will be deleted. The system should then ask if we want to delete the whole corresponding workflow or only the reference. For this test the whole workflow will be deleted.
Expected Result: The whole workflow should be deleted.

7. **Test:** Delete a Connector.
Procedure: Same procedure as in the test before expect that during this test the Connector *Dynamic Data Topic* is deleted.
Expected Result: The tool should ask if we want to delete the *Process Dynamic Change Events* and *Process Dynamic Data Changes* Components, because they are no longer part of the messaging system.

Besides that we checked the results of the tests, we loaded the resulting output-files into Mule ESB and ActiveMQ to test if they are still runnable.

5.5.2 Result & Discussion

The following list presents the results of our tests:

1. **Test:** Add new Component, Interface, Connector, ConnectorType.
Result: The new workflow with the name "NewComponent" was added at the end of the POS_Service.xml Mule ESB output-file, including the *in*-endpoint connected to the queue "new.queue" and with a reference to the new activemq-connector, including the new address, which was also added to the POS_Service.xml file. The Component *Process*

Dynamic Data Changes got an additional *out*-endpoint as last child of the corresponding *flow* element. Furthermore the additional ActiveMQ configuration file was created and the new transport connector with the address *tcp://127.0.0.2:61616* was defined.

2. **Test:** Change a channel name.
   Result: All connected endpoints, i.e. the *out*-endpoint of Component *Process Dynamic Change Events* and the *in*-endpoint of Component *Process Dynamic Data Changes*, were changed to the new topic name "dynamic.change.topic.changed".

3. **Test:** Reconnect an Interface.
   Result: The *in*-endpoint of Component *Process Dynamic Data Changes* was changed to the new topic *Structural Change Topic*.

4. **Test:** Change order of the Interface.
   Result: The order of the endpoints of Component *Process Dynamic Data Changes* were changed to first *out* and second *in*.

5. **Test:** Delete the reference (xADL id) to a xADL Component.
   Result: Our tool asked if we want to delete the reference or the element. We chose to delete the reference, which resulted in the question if the endpoint, which is not needed anymore, should also be removed or only the reference to it. We also chose to delete only the reference. The tool then removed the xADL id stored in the workflow for the Component *Process Dynamic Data Changes* and the xADL id stored in the endpoint of the workflow.

6. **Test:** Delete a Component.
   Result: Our tool asked if we want to delete the reference or the element. We chose to delete the element. The tool then removed the whole *Process Dynamic Data Changes* workflow.

7. **Test:** Delete a Connector.
   Result: By deleting the Connector *Dynamic Data Topic* the connected Components *Process Dynamic Change Events* and *Process Dynamic Data Changes* were no longer required. Therefore our tool asked if we want to delete them or to remove only the reference to them, which resulted in the same behaviour as in test 5 and 6.

Finally all runnable tests, in which we loaded the output-files into Mule ESB and ActiveMQ, were positive.

In conclusion, we were able to prove that our approach and the suggested implementation is able to propagate all common changes to already available transformed systems. The change propagation, including the consistency checks which are performed before a change is propagated, makes our approach and the suggested implementation a good supporting tool for incremental development of message-based systems.

Nevertheless, the change propagation algorithm has one restriction that appears if user-defined code, in the already transformed workflow, exists. For example, if the transformed workflow has one interface and an user-defined code after that interface and during the change

a new interface should be added somewhere in the user-defined code. Due to the fact that the correct position in the user-defined code can't be specified in our approach, our algorithm will place the new interface directly after the already existing interface. In other words, our algorithm can only guarantee the right order of the interfaces but not the correct place in a workflow with user-defined code. But of course, the responsible developer can move the interface in the user-defined code to the correct position. Note that the update of an already transformed interface doesn't change the position in the user-defined code, as long as the update doesn't include a reordering of the interfaces.

5.6 Summary

As can be seen in the presented results, our approach and the corresponding implementation is able to handle the given evaluation scenarios. First of all, it can be used to plan and configure a message-based system with the help of xADL and ArchStudio 4, specifically the tools *Archipelago* and *ArchEdit*. The planning and configuration, per se, is hard to evaluate, because it depends on the experience that an architect has with the ArchStudio 4 tool support. But in the end, we were able to show that the planned xADL extensions were integrated in the implementation and that they are sufficient enough to model and configure a message-based system, due to the fact that we used the designed system from Section 5.2 for the consistency check evaluation (Section 5.3), transformation evaluation (Section 5.4) and change propagation evaluation (Section 5.5).

During the evaluation of the consistency checking, we first proved that the checks, defined in our approach, are able to find different common inconsistencies. This saves the architect a huge amount of time and takes over a tedious and error-prone task, especially for large systems and for changes at an advanced stage of the development process. But it has to be said that our defined consistency checks only take the architectural structure and the configuration into account and ignore the business logic of the services. This stands in contrast to other checking systems, which we discussed in the related work Section 2.2, which in fact require more information about the business logic, but therefore also offer a more fine-grained checking. But as proved, for our purpose, the provided consistency checks are enough and are an enormous benefit for the architecture team. At the end of this evaluation, we evaluated the implemented consistency checks. Due to the fact that the original "Parking Management System" scenario didn't contain any inconsistencies we had to change, for each implemented check, a part of the architecture to trick the system into this inconsistency.

In the second part, we evaluated the architecture-to-configuration transformation. With this evaluation we proved that the information that is given by the xADL, i.e. the structural information and the information that is given by our xADL extensions, are enough to transform the architecture to the Mule ESB workflow configuration files and the Apache ActiveMQ configuration file. At the current stage of the implementation the system is only capable of generating Mule ESB and ActiveMQ configuration files. However, the implementation can be easily extended to support other tools and output-file formats. The overall approach of the system remains valid also for other service messaging frameworks and tools. Even the basic consistency checks stay the same, only the output-file specific consistency checks have to be altered or extended. In

the end, we were able to prove that our solution offers a way for a clear separation of responsibilities. In a similar way as Zheng and Taylor did in their work [63], the usage of XML-based workflow configuration files promotes the separation of architecture-prescribed code from user-defined code and offers an elegant way for change propagation.

At the end, the evaluation of the change propagation aspect completed the analysis. During this evaluation we proved by adding, altering and removing different architecture-level elements that the propagation of changes from the architecture to an already transformed system is functional in our implemented prototype. Furthermore we proved that the xADL ids are enough for the mapping between the architecture elements and the transformed elements. But at the end, we also highlighted that our change propagation system has a limitation in the ordering of the interfaces. Specifically, if user-defined code is already included in the workflows, our algorithm can't guarantee that a new interface is placed at the right position in this user-defined code. The algorithm only guarantees that the order of the interfaces is correct. If this situation occurs, a developer has to move the interface to the correct position in the user-defined code after the transformation. Zheng and Taylor had a similar problem in their research [63]. They solved it by including a notification system which informs the developer, that he has to perform changes in the user-defined code to support the new included architecture-prescribed code. Such a notification system could be a worthwhile additional feature to our system and is considered as future work (Section 6.1).

CHAPTER 6

Conclusion and Future Work

In the course of this thesis, we investigated how architecture-centric systems can be used for the development of complex, message-based service systems. Our work focused on the specification and generation of message-driven, highly decoupled composite service systems with an uncertain number of fluctuating instances and without the need for a central controlling unit. The work was separated into two parts: (1) we first defined an approach which is able to model and configure such systems and perform consistency checking on a specified system, and (2) second we implemented a prototype for this approach.

For our approach we first investigated which inconsistencies can occur during the planning of such message-centric systems and how a system can detect those inconsistencies as early as possible. In the following we defined several consistency checks which issue warnings and errors to inform the architects. Furthermore we explored how the highly expendable architecture description language xADL can be extended to provide a basis for the specification of message-based systems. Specifically, our approach added several extensions to define Services, Communication Channels and MOM configuration parameters. This led to the question of how the planned architecture can be transformed to a working message routing skeleton and how the mapping between the transformed message system elements and the architecture-level elements can be achieved. Finally, we dealt with the task of change propagation. Specifically, we investigated how changes, done in the architecture, can be propagated to already transformed code, without altering user-defined code.

In the second part we implemented a prototype of our defined approach. For the underlying tool support we chose to use ArchStudio 4 to read and modify the xADL 2.0 documents and to generate configuration files for Mule ESB and the message broker Apache ActiveMQ. The first part was concerned with the ArchStudio extensions that we implemented, followed by the analysis of the data models of a Mule ESB workflow configuration file, an Apache ActiveMQ configuration file and the xADL data model. During these steps we also defined the mandatory information that is required by those tools. Afterwards, we implemented the xADL extensions, which finally added the ability to plan and configure a message-based system, which uses Mule ESB and Apache ActiveMQ as the underlying tools, to ArchStudio 4. Consequently, we were

able to implement the consistency checks, including the checks defined in the approach and specific checks for Mule ESB and Apache ActiveMQ. Finally we completed the implementation by the architecture-to-configuration transformation and the change propagation system.

Finally, we evaluated our approach and the implemented prototype by using it to develop the *Parking Management System* Scenario, discussed during the introduction.

6.1 Future Work

Our work, in the current state, is a fully functional prototype, but the development isn't considered as finished. The approach and implementation can be used as basis for future projects, theses or dissertations.

Our future work, on the theoretical part of the project, will focus on the following aspects:

- It is worthwhile evaluating how the EAI patterns, which are currently modelled in Mule ESB, can be supported. For example it can be considered to extend xADL in a way that it is possible to plan the EAI patterns or use some predefined patterns, that can be used and transformed into the ESB workflow.

- At the current stage the approach only propagates changes from the architecture to the output-files. This can be extended to propagating changes the other way around. Therefore, the system has to check if an element in the output-files doesn't have a xADL id. If this is the case, the configuration attributes of the element can be transformed into the xADL architecture document.

- Furthermore, the consistency analysis methods of the approach can be extended to include more consistency checks and architecture optimising algorithms. It is valuable to evaluate how an algorithm can be used to optimise the allocation of the channels, across different message-broker instances, to get the optimal usage.

- In the 1.x-way mapping [63] Zheng uses a notification system to inform the developers that changes which were propagated from the architecture to the code, also require changes in the user-defined code. Such a notification system would be an assisting support in our approach. For example, it can be used to notify the developers if the order of interfaces has changed and therefore also the user-defined code may have to be repositioned.

On the practical side, we will focus on the following aspects:

- First of all the implementation can be extended with the discussed approach extensions from above.

- At the current stage the implementation only supports Mule ESB and ActiveMQ. Due to the fact that our approach doesn't rely on those tools, the implementation can be extended to support other tools and protocols without changing the underlying approach. Even the basic consistency checks stay the same, only the output-file specific consistency checks

have to be altered or extended. For example an extension to support the advanced message queue protocol AMQP or another ESB system like Fuse ESB can be considered.

- Another supporting feature, that is worthwhile to be included, would be the ability to add comments to the architectural components, connectors and interfaces. These comments could be included in the transformation and therefore they would end up in the messaging system skeleton and the message broker configuration. By using such a feature the architect could specify some useful comments/information for the development team.

- xADL has the ability to define versions of the system and to specify architecture elements as optional. This allows the definition of several different system versions with different abilities. At the current state this functionality isn't included in our implementation, but it could be an efficient feature and is therefore worthwhile to be included.

- In addition to the introduced tools, *Archipelago*, *ArchEdit* and *TypeWrangler*, ArchStudio 4 offers the separate analyzing tool *Archlight*, which can be used to run selfcontained tests on the architecture of a system. By rewriting the introduced consistency checks into Archlight test cases, this tool can be used to perform consistency checks independent from the Architecture-to-Configuration transformation process, which we introduced in this work. This will be part of an upcoming project and will then be included in the project files of this thesis.

APPENDIX A

xADL Extensions

Listing A.1: XML-Schema of the *Channel Implementation* extension

```
1  <xsd:schema xmlns="at.ac.tuwien.
       infosys/arch/xArch/channelimplementation.xsd" xmlns:xsd="
       http://www.w3.org/2001/XMLSchema" xmlns:archtypes="http://www.ics.
       uci.edu/pub/arch/xArch/types.xsd" xmlns:archimpl="http://www.ics.
       uci.edu/pub/arch/xArch/implementation.xsd" targetNamespace="at.ac.
       tuwien.infosys/arch/xArch/channelimplementation.xsd"
       elementFormDefault="qualified" attributeFormDefault="qualified">
2
3    <xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/types.
         xsd" schemaLocation="http://www.isr.uci.
         edu/projects/xarchuci/ext/types.xsd"/>
4    <xsd:import namespace="http://www.ics.uci.
         edu/pub/arch/xArch/implementation.xsd" schemaLocation="
         http://www.isr.uci.edu/projects/xarchuci/ext/implementation.xsd"
         />
5
6    <!-- TYPE: ConnectorImpl -->
7    <xsd:complexType name="ConnectorImpl">
8      <xsd:complexContent>
9        <xsd:extension base="archtypes:Connector">
10         <xsd:sequence>
11           <xsd:element name="implementation" type="
                archimpl:Implementation" minOccurs="0" maxOccurs="
                unbounded"/>
12         </xsd:sequence>
13       </xsd:extension>
14     </xsd:complexContent>
15   </xsd:complexType>
16
17   <!-- TYPE: ChannelImplementation -->
```

```
18    <xsd:complexType name="ChannelImplementation">
19      <xsd:complexContent>
20        <xsd:extension base="archimpl:Implementation">
21          <xsd:sequence>
22            <xsd:choice>
23            <xsd:element name="Queue_Configuration" type="QueueConfig"/
                  >
24            <xsd:element name="Topic_Configuration" type="TopicConfig"/
                  >
25            </xsd:choice>
26          </xsd:sequence>
27        </xsd:extension>
28      </xsd:complexContent>
29    </xsd:complexType>
30
31    <xsd:complexType name="QueueConfig">
32      <xsd:attribute name="name" type="xsd:string"/>
33    </xsd:complexType>
34
35    <xsd:complexType name="TopicConfig">
36      <xsd:attribute name="name" type="xsd:string"/>
37    </xsd:complexType>
38  </xsd:schema>
```

Listing A.2: XML-Schema of the *Endpoint Implementation* extension

```
1   <xsd:schema xmlns="at.ac.tuwien.
        infosys/arch/xArch/endpointimplementation.xsd" xmlns:xsd="
        http://www.w3.org/2001/XMLSchema" xmlns:archinstance="http://www.
        ics.uci.edu/pub/arch/xArch/instance.xsd" xmlns:archtypes="
        http://www.ics.uci.edu/pub/arch/xArch/types.xsd" xmlns:archimpl="
        http://www.ics.uci.edu/pub/arch/xArch/implementation.xsd"
        targetNamespace="at.ac.tuwien.
        infosys/arch/xArch/endpointimplementation.xsd" elementFormDefault=
        "qualified" attributeFormDefault="qualified">
2
3     <xsd:import namespace="http://www.ics.uci.
          edu/pub/arch/xArch/instance.xsd" schemaLocation="http://www.isr.
          uci.edu/projects/xarchuci/core/instance.xsd"/>
4     <xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/types.
          xsd" schemaLocation="http://www.isr.uci.
          edu/projects/xarchuci/ext/types.xsd"/>
5     <xsd:import namespace="http://www.ics.uci.
          edu/pub/arch/xArch/implementation.xsd" schemaLocation="
          http://www.isr.uci.edu/projects/xarchuci/ext/implementation.xsd"
          />
6
7     <!-- TYPE: InterfaceImpl -->
8     <xsd:complexType name="InterfaceImpl">
9       <xsd:complexContent>
```

```
10        <xsd:extension base="archtypes:Interface">
11          <xsd:sequence>
12            <xsd:element name="implementation" type="
                  archimpl:Implementation" minOccurs="0" maxOccurs="
                  unbounded"/>
13          </xsd:sequence>
14        </xsd:extension>
15      </xsd:complexContent>
16    </xsd:complexType>
17
18    <!-- TYPE: EndpointImplementation -->
19    <xsd:complexType name="EndpointImplementation">
20      <xsd:complexContent>
21        <xsd:extension base="archimpl:Implementation">
22          <xsd:sequence>
23            <xsd:element name="Reply_To_Queue" type="
                  archinstance:XMLLink" minOccurs="0" maxOccurs="1"/>
24            <xsd:element name="Endpoint_Position_No" type="
                  EndpointPositionNoType" minOccurs="0" maxOccurs="1"/>
25            <xsd:element name="Duralbe_Name" type="DurableName"
                  minOccurs="0" maxOccurs="1"/>
26            <xsd:element name="Connection_To_Request_Endpoint" type="
                  archinstance:XMLLink" minOccurs="0" maxOccurs="1"/>
27          </xsd:sequence>
28        </xsd:extension>
29      </xsd:complexContent>
30    </xsd:complexType>
31
32    <xsd:complexType name="DurableName">
33      <xsd:attribute name="name" type="xsd:string" />
34    </xsd:complexType>
35
36    <xsd:complexType name="EndpointPositionNoType">
37      <xsd:attribute name="value" type="xsd:string" />
38    </xsd:complexType>
39  </xsd:schema>
```

Listing A.3: XML-Schema of the *JMS Implementation* extension

```
1  <xsd:schema xmlns="at.ac.tuwien.infosys/arch/xArch/jmsimplementation.
       xsd" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:archimpl="
       http://www.ics.uci.edu/pub/arch/xArch/implementation.xsd"
       targetNamespace="at.ac.tuwien.infosys/arch/xArch/jmsimplementation
       .xsd" elementFormDefault="qualified" attributeFormDefault="
       qualified">
2
3    <xsd:import namespace="http://www.ics.uci.
         edu/pub/arch/xArch/implementation.xsd" schemaLocation="
         http://www.isr.uci.edu/projects/xarchuci/ext/implementation.xsd"
         />
```

```
4
5   <!-- TYPE: JmsImplementation -->
6   <xsd:complexType name="JmsImplementation">
7     <xsd:complexContent>
8       <xsd:extension base="archimpl:Implementation">
9         <xsd:attribute name="file_id" type="xsd:string"/>
10        <xsd:sequence>
11          <xsd:element name="Transport_Configuration" type="
                TransportConfig" minOccurs="1" maxOccurs="unbounded"/>
12          <xsd:element name="Persistence_Configuration" type="
                PersistenceConfig" minOccurs="0" maxOccurs="1"/>
13          <xsd:element name="Jms_Specification_Version" type="
                JmsSpecificationType" minOccurs="0" maxOccurs="1"/>
14        </xsd:sequence>
15      </xsd:extension>
16    </xsd:complexContent>
17  </xsd:complexType>
18
19  <xsd:complexType name="TransportConfig">
20    <xsd:attribute name="transportConnector" type="xsd:string"/>
21  </xsd:complexType>
22
23  <xsd:complexType name="JmsSpecificationType">
24    <xsd:attribute name="value" type="xsd:string" default="1.1" />
25  </xsd:complexType>
26
27  <xsd:complexType name="PersistenceConfig">
28    <xsd:attribute name="adapter" type="xsd:string" default="kahaDB"/
        >
29    <xsd:attribute name="directory" type="xsd:string"/>
30  </xsd:complexType>
31 </xsd:schema>
```

Listing A.4: XML-Schema of the *Mule Implementation* extension

```
1  <xsd:schema xmlns="at.ac.tuwien.infosys/arch/xArch/muleimplementation
       .xsd" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:archtypes
       ="http://www.ics.uci.edu/pub/arch/xArch/types.xsd" xmlns:archimpl=
       "http://www.ics.uci.edu/pub/arch/xArch/implementation.xsd"
       targetNamespace="at.ac.tuwien.
       infosys/arch/xArch/muleimplementation.xsd" elementFormDefault="
       qualified" attributeFormDefault="qualified">
2
3     <xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/types.
         xsd" schemaLocation="http://www.isr.uci.
         edu/projects/xarchuci/ext/types.xsd"/>
4     <xsd:import namespace="http://www.ics.uci.
         edu/pub/arch/xArch/implementation.xsd" schemaLocation="
         http://www.isr.uci.edu/projects/xarchuci/ext/implementation.xsd"
         />
```

```
 5
 6    <!-- TYPE: ComponentImpl -->
 7    <xsd:complexType name="ComponentImpl">
 8      <xsd:complexContent>
 9        <xsd:extension base="archtypes:Component">
10          <xsd:sequence>
11            <xsd:element name="implementation" type="
                  archimpl:Implementation" minOccurs="0" maxOccurs="
                  unbounded"/>
12          </xsd:sequence>
13        </xsd:extension>
14      </xsd:complexContent>
15    </xsd:complexType>
16
17    <!-- TYPE: MuleImplementation -->
18    <xsd:complexType name="MuleImplementation">
19      <xsd:complexContent>
20        <xsd:extension base="archimpl:Implementation">
21          <xsd:attribute name="file_id" type="xsd:string"/>
22          <xsd:sequence>
23            <xsd:element name="Additional_Configuration" type="
                  AdditionalConfig" minOccurs="0" maxOccurs="unbounded"/>
24          </xsd:sequence>
25        </xsd:extension>
26      </xsd:complexContent>
27    </xsd:complexType>
28
29    <xsd:complexType name="AdditionalConfig">
30      <xsd:attribute name="name" type="xsd:string"/>
31      <xsd:attribute name="value" type="xsd:string"/>
32    </xsd:complexType>
33  </xsd:schema>
```

APPENDIX B

Consistency Checks

Level	Validation	Description	Recommendation
Architecture	Is an *inout*-interface connected to a publish-subscriber channel?	This could lead to unwanted behavior in some Mule ESB versions and could again lead to an unclear architecture if the system is more complex.	A better and clearer way is to use an *out*-interface for publishing to a publish-subscriber channel and an extra receiving queue for the reply.
	Does an *out*-interface of a queue contain several connected links?	A queue is a point-to-point connection and doesn't support different receivers. The message will be sent once and only to one receiver. If there are several links on the *out*-interface it is undefined which one will receive the message.	Use several different queues, for each receiver one, or a publish-subscriber channel.

Level	Validation	Description	Recommendation
	Is a publish-subscriber channel used for publishing a message within the scope of a request-reply pattern?	This could lead to several response messages, because each subscriber could reply to the request. This could be desired but then the receiving service have to handle these multiple replies.	Use a point-to-point connection to send the requesting message.
Component Connection	Do all used connectors and components have an implementation?	It could be desired that a component or connection doesn't have an implementation (e.g. the component will be implemented in a later development phase), but it could also be a fault.	Add an implementation to the connector or component.

Table B.1: Soft consistency checks at architecture level.

Level	Validation	Description	Recommendation
Architecture	Do two different connectors, with the same connector type, have the same channel name?	This wouldn't be a problem on the final system, but it isn't recommended to use two different connectors with the same channel name, because this could lead to an unclear architecture.	Use different channel names or use only one Connector.

Level	Validation	Description	Recommendation
	Is there a link between two connectors or two components?	Two components could only be connected over a connector. This has to be done because the channel name of the connection is needed for the messaging system. Two connected connectors are not allowed.	The connection between two components has to be done over a connector and two connected connectors have to be combined in one connector.
	Are the interfaces, used for a request-reply group, defined correctly?	For the request-reply pattern the component needs at least two interfaces, one with the direction *in* and one with direction *out*. The requesting interface (interface with direction *out*) must refer to the receiving interface (interface with direction *in*) on the same component.	Use an *in*-interface and an *out*-interface and set the corresponding references.
	Do two interfaces, that are connected by a link, have the same directions?	If two interfaces are connected together and don't have the direction *inout*, they must have different directions, *in* to *out* or *out* to *in*.	If they have the same direction, change the direction of one interface or set both interface directions to *inout*.
	Does a component have more than one link at an *in*- or *out*-interface?	An interface can only be linked to one channel.	Use one interface for each channel.

Level	Validation	Description	Recommendation
Component	Can the interfaces, on one component, be arranged automatically?	If the component has more than one bidirectional interface or more than one request-reply interface combination the interfaces can't be arranged automatically.	Arrange the interfaces manually by adding the ordering configuration value.
	Does the component have more than one *in*- or one *out*-interface that are not part of a request-reply group?	Our system only allows one *in*- and *out*-interface at a component.	
Connector	Is the channel name and the channel type (publish-subscriber or point-to-point) set?	In the connector type it has to be defined if it is a publish-subscriber or point-to-point connection. Additional to the type a channel name has to be set.	Set the channel type and name.
Link	Is the link set correctly?	Each endpoint of a link has to be connected to a valid interface.	

Table B.2: Hard consistency checks at component level.

Bibliography

[1] Organization for the Advancement of Structured Information Standards (OASIS): Web Services Business Process Execution Language (WS-BPEL) Version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`, Accessed: October 2014.

[2] Robert J Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon School of Computer Science, 1997.

[3] Apache. Apache ActiveMQ. `http://activemq.apache.org`, Accessed: July 2014.

[4] Apache. Openjms. `http://openjms.sourceforge.net`, Accessed: October 2014.

[5] Luciano Baresi, Carlo Ghezzi, and Luca Mottola. On Accurate Automatic Verification of Publish-Subscribe Architectures. In *29th International Conference on Software Engineering (ICSE'07)*, pages 199–208. IEEE, May 2007.

[6] Adam Barker, Christopher D. Walton, and David Robertson. Choreographing Web Services. *IEEE Transactions on Services Computing*, 2(2):152–166, April 2009.

[7] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley, 2012.

[8] Mauro Caporuscio, P. Inverardi, and P. Pelliccione. Compositional verification of middleware-based software architecture descriptions. In *Proceedings. 26th International Conference on Software Engineering*, pages 221–230. IEEE Comput. Soc, 2004.

[9] David Chappell. *Enterprise service bus*. O'Reilly Media, 2004.

[10] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond, Second Edition*. Addison-Wesley Professional, 2nd edition, 2010.

[11] Eric M. Dashofy, Hazel Asuncion, Scott Hendrickson, Girish Suryanarayana, John Georgas, and Richard Taylor. ArchStudio 4: An Architecture-Based Meta-Modeling Environment. In *29th International Conference on Software Engineering (ICSE'07 Companion)*, pages 67–68. IEEE, May 2007.

[12] Eric M. Dashofy, A. van der Hoek, and R.N. Taylor. A highly-extensible, XML-based architecture description language. In *Proceedings Working IEEE/IFIP Conference on Software Architecture*, pages 103–112. IEEE Comput. Soc.

[13] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *Proceedings of the 24th international conference on Software engineering - ICSE '02*, pages 266–276, New York, New York, USA, 2002. IEEE.

[14] Eric M. Dashofy, André van der Hoek, and Richard N Taylor. A comprehensive approach for the development of modular software architecture description languages. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, volume 14, pages 199–245, 2005.

[15] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In *IEEE International Conference on Web Services (ICWS 2007)*, pages 296–303. IEEE, July 2007.

[16] Christoph Dorn, Philipp Waibel, and Schahram Dustdar. Architecture-Centric Design of Complex Message-Based Service Systems. In *Service-Oriented Computing - 12th International Conference, ICSOC 2014, Paris, France, November 3-6, 2014. Proceedings*, pages 184–198, 2014.

[17] Schahram Dustdar and BJ Krämer. Introduction to special issue on service oriented computing (SOC). *ACM Transactions on the Web*, 2(2):1–2, April 2008.

[18] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[19] Patrick Th. Eugster, Pascal a. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[20] Institute for Software Research at the University of California Irvine. Archstudio 4. `http://isr.uci.edu/projects/archstudio-4/www/archstudio/`, Accessed: July 2014.

[21] Institute for Software Research at the University of California Irvine. xadl 2.0. `http://isr.uci.edu/projects/archstudio-4/www/xarchuci/`, Accessed: July 2014.

[22] Institute for Software Research at the University of California Irvine. xadl 2.0 extensions overview. `http://isr.uci.edu/projects/archstudio-4/www/xarchuci/ext-overview.html`, Accessed: July 2014.

[23] The Apache Software Foundation. Apache activemq documentation. `http://activemq.apache.org/using-activemq-5.html`, Accessed: July 2014.

[24] The Apache Software Foundation. Kahadb. http://activemq.apache.org/kahadb.html, Accessed: July 2014.

[25] Joshua Garcia, Daniel Popescu, Gholamreza Safi, William G. J. Halfond, and Nenad Medvidovic. Identifying message flow in distributed event-based systems. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, page 367, New York, New York, USA, 2013. ACM Press.

[26] F.J. Garcia-Jimenez, M.a. Martinez-Carreras, and A.F. Gomez-Skarmeta. Evaluating Open Source Enterprise Service Bus. In *2010 IEEE 7th International Conference on E-Business Engineering*, pages 284–291. IEEE, November 2010.

[27] David Garlan. Software architecture: a Roadmap. In *Proceedings of the conference on The future of Software engineering - ICSE '00*, pages 91–101, New York, New York, USA, 2000. ACM Press.

[28] David Garlan, Serge Khersonsky, and Jung Soo Kim. Model Checking Publish-Subscribe Systems. *Model Checking Software*, pages 166—-180, 2003.

[29] David Garlan, Robert T. Monroe, and David Wile. ACME : An Architecture Description Interchange Language. In *In Proceedings of CASCON'97*, pages 169—-183, 1997.

[30] Red Hat. Hornetq. http://hornetq.jboss.org, Accessed: October 2014.

[31] Red Hat. Jboss fuse. http://fusesource.com/products/enterprise-servicemix/, Accessed: July 2014.

[32] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[33] Michael N. Huhns and Munindar P. Singh. Service-oriented computing: key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, January 2005.

[34] Nicolai M. Josuttis. *SOA in Practice The Art of Distributed System Design*. O'Reilly Media, Inc., 2007.

[35] Tobias Kruessmann, Arne Koschel, Martin Murphy, Adrian Trenaman, and Irina Astrova. High availability: Evaluating open source enterprise service buses. In *Proceedings of the ITI 2009 31st International Conference on Information Technology Interfaces*, pages 615–620. IEEE, June 2009.

[36] Rikard Land and Ivica Crnkovic. Existing approaches to software integration – and a challenge for the future. *integration*, 40:58—-104, 2004.

[37] Fredlund Lars-Ake. Implementing WS-CDL. *Proceedings of the second Spanish workshop on Web Technologies (JSWEB 2006)*, 2006.

[38] Youn Kyu Lee, Jae young Bang, Joshua Garcia, and Nenad Medvidovic. ViVA: a visualization and analysis tool for distributed event-based systems. In *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, pages 580–583, New York, New York, USA, 2014. ACM Press.

[39] David S Linthicum. *Enterprise application integration*. Addison-Wesley Professional, 1999.

[40] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[41] J Magee, N Dulay, S Eisenbach, and J Kramer. Specifying Distributed Software Architectures. In Wilhelm Schäfer and Pere Botella, editors, *Proceedings of the 5th European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[42] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology*, 11(1):2–57, January 2002.

[43] Nenad Medvidovic and RN Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[44] Falko Menge. Enterprise Service Bus. *Free and open source software conference*, 2:1–6, 2007.

[45] OpenESB Community Formerly Sun Microsystems. Openesb. `http://www.open-esb.net`, Accessed: July 2014.

[46] Nikola Milanovic and Miroslaw Malek. Current solutions for Web service composition. *IEEE Internet Computing*, 8(6):51–59, November 2004.

[47] MuleSoft. Understanding Enterprise Application Integration - The Benefits of ESB for EAI. `http://www.mulesoft.com/resources/esb/enterprise-application-integration-eai-and-esb`, Accessed: July 2013.

[48] MuleSoft. Mule ESB. `http://www.mulesoft.org/`, Accessed: July 2014.

[49] MuleSoft. Mule ESB Documentation. `http://www.mulesoft.org/documentation/display/current/Home`, Accessed: July 2014.

[50] Mangala Gowri Nanda, Satish Chandra, and Vivek Sarkar. Decentralizing Execution of Composite Web Services. *ACM SIGPLAN Notices*, 39(10):170, October 2004.

[51] Deakin Nigel, Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. *Java Message Service*. Number December. 2012.

[52] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing: A Research Roadmap. *International Journal of Cooperative Information Systems*, 17:223–255, November 2008.

[53] M.P. Papazoglou. Service-oriented computing: concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, volume 46, pages 3–12. IEEE Comput. Soc, 2003.

[54] Cesare Pautasso, Thomas Heinis, and Gustavo Alonso. JOpera: Autonomic Service Orchestration. *IEEE Data Engineering Bulletin*, 29(3):1–8, 2006.

[55] C. Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10):46–52, October 2003.

[56] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE '03*, page 267, New York, New York, USA, 2003. ACM Press.

[57] Stephen Ross-Talbot. Orchestration and Choreography: Standards, Tools and Technologies for Distributed Workflows. In *NETTAB Workshop-Workflows management: new abilities for the biological information overflow, Naples, Italy*, 2005.

[58] Thorsten Scheibler and Frank Leymann. A framework for executable enterprise application integration patterns. *Enterprise Interoperability III*, pages 485–497, 2008.

[59] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[60] Wil MP Van der Aalst and Arthur HM Ter Hofstede. YAWL: yet another workflow language. *Information Systems*, 30(4):245–275, June 2005.

[61] Ustun Yildiz and Claude Godart. Information Flow Control with Decentralized Service Compositions. In *IEEE International Conference on Web Services (ICWS 2007)*, number Icws, pages 9–17. IEEE, July 2007.

[62] Johannes Maria Zaha, Alistair Barros, Marlon Dumas, and Arthur ter Hofstede. Let's Dance: A Language for Service Behavior Modeling. *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, 4275:145–162, 2006.

[63] Yongjie Zheng and Richard N. Taylor. Enhancing architecture-implementation conformance with change management and support for behavioral mapping. *Proceedings of the 34th International Conference on Software Engineering*, pages 628–638, 2012.

[64] Yongjie Zheng and Richard N. Taylor. A classification and rationalization of model-based software development. *Software & Systems Modeling*, 12(4):669–678, June 2013.