AutoTaSC: Model Driven Development for Autonomic Software Engineering

Yousef M. Abuseta

A thesis submitted in partial fulfilment of the requirements of Liverpool John Moores University for the degree of Doctor of Philosophy

School of Computing and Mathematical Sciences
Liverpool John Moores University
Liverpool, UK,

July 09

BY REQUEST OF THE UNIVERSITY

THE FOLLOWING ITEMS HAVE NOT BEEN DIGITISED.

FIGURE 3.1 P23

FIGURE 8.23 P101

FIGURE D.1 P134

FIGURE D.2 P134

FIGURE D.3 P135

ABSTRACT

Whilst much research progress has been achieved towards the development of autonomic software engineering tools and techniques including: policy-based management, model-based development, service-oriented architecture and model driven architecture. They have often focused on and started from chosen object-oriented models of required software behaviour, rather than domain model including user intentions and/or software goals. Such an approach is often reported to lead to "misalignment" between business process layer and their associated computational enabling systems. This is specifically noticeable in adaptive and evolving business systems and/or processes settings. To address this long-standing problem research has over the years investigated many avenues to close the gap between business process modelling and the generation of enactment (computation) layer, which is responsive to business changes. Within this problem domain, this research sets out to study the extension of the Model Driven Development (MDD) paradigm to business/domain model, that is, how to raise the abstraction level of model-driven software development to the domain level and provide model synchronisation to trace and analyse the impact of a given model change.

The main contribution of this research is the development of a MDD-based design method for autonomic systems referred to as *AutoTaSC*. The latter consists of a series of related models, where each of which represents the system under development at a given stage. The first and highest level model represents the abstract model referred to as the Platform Independent Model (PIM). The next model encapsulates the PIM model for the autonomic system where the autonomic capabilities and required components (such as monitor, sensor, actuator, analyser, policy, etc.) are added via some appropriate transformation rules. Targeting a specific technology involves adding, also via transformation rules, specific information related to that platform from which the Platform Specific Model (PSM) for the autonomic system is extracted. In the last stage, code can be generated for the specific platform or technology targeted in the previous stage, web services for instance. In addition, the *AutoTaSC* method provides a situated model synchronisation mechanism, which is designed following the autonomic systems principles. For instance, to guarantee model synchronisation each model from each *AutoTaSC* stage has an associated policy-based feedback control loop, which regulates its

reaction to detected model change. Thus, *AutoTaSC* method model transformation approach to drive model query, view and synchronisation. The *AutoTaSC* method was evaluated using a number of benchmark case-studies to test this research hypothesis including the effectiveness and generality of *AutoTaSC* design method.

ACKNOWLEDGMENTS

I would like to take this opportunity to express my deep thanks and appreciation to my supervisor, Professor A. Taleb-Bendiab, for his invaluable and unlimited support throughout the course of this research. He was always encouraging me to keep and remain focused on achieving my goals. His useful critical comments and great ideas indeed helped me shape and establish the overall direction of my research and build on these comments and observations to take it further and deeper.

I would like to thank all of my family for their support during my study specially my father and mother. Also thanks go to all of my friends, who used to ask, support and care about me.

Special thanks and appreciation goes to my wife for her vital support, understanding and patience, without which this work would not be possible.

Yousef M. Abuseta

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivations

The advances in networking and computing technology have led to the explosive growth seen today in distributed applications and information services that impact all aspects of our life. The nature of these applications and services is extremely complex, heterogeneous and dynamic. As these systems and applications continue to grow in terms of the complexity, dynamics and scale, current tools and techniques are reaching the limits of their effectiveness. Researchers, therefore, had to consider alternative approaches to address this problem. They have adopted a set of strategies employed by the biological systems and their effort has resulted in the emergence of the autonomic computing paradigm. This paradigm was first introduced by IBM to the National Academy of Engineers at Harvard University in March 2001. Such a paradigm is inspired by the functioning of the human nervous system and its fundamental objective is to design software systems that exhibit autonomic and self managing capabilities. In such systems, the management and configuration tasks which are used to be performed by the system administrators are delegated to the software itself, supplied only with high-level policies, and thus shielding the administrators from carrying out these notoriously difficult tasks.

Although the autonomic systems are very promising and their benefits and values for the software design industry are evident, the support and design process for such systems is still unclear. This is probably due to the nature and unique features, stated above, that distinguish these systems from the ordinary ones which make the already existing software development processes, such as the *waterfall process*, not suitable or appropriate for designing these systems. Therefore, there is required work and research to be conducted and undertaken to find a good candidate for an autonomic system design process.

Despite the well defined approaches proposed so far, including the IBM, to address and guide the autonomic systems design, this important aspect has not been fully addressed yet by any of these approaches. Therefore, the primary aim of this research is to design and develop a modelling framework for autonomic systems, which will guide and help designers develop such systems. This design method distinguishes itself from the existing ones by providing a complete lifecycle for the autonomic systems development where the designer is guided and supported from the very early stage of the design process (the requirement stage) and up to the last stage when a set of useful skeletons of code is generated. More importantly, unlike other approaches which also have adopted the Model Driven Development (MDD) paradigm, this design method has raised the abstraction level one level higher by not committing itself from the beginning to any specific technology. The other approaches tend to adopt a particular technology from the early stage of the development process, the object oriented technique for instance (UML class diagrams).

In addition to the MDD paradigm, this design method is based on well-accepted and widely recognised paradigms such as the Service Oriented Architecture (SOA), Business Process Oriented Architecture (BPOA) and the Extensible Modelling Language (XML).

1.2 Challenges

To design and develop such systems, a number of theoretical and practical aspects need to be addressed which form the basis and requirements of a comprehensive and well engineered an autonomic systems reference design method. These challenges can be addressed as follows:

- **Reference models:** these include the reference design method that defines and establishes the high level stages of the autonomic systems development process, internal reference models such as the service-monitor-controller model, model synchronisation framework and model transformation engines.
- **Mechanisms:** these include approaches and tools related to the issue of providing a well-defined and resilient design method for autonomic or self-managing systems. To this end, a variety of services, tools and a framework need to be designed, developed and implemented providing a set of utilities including:

- The definition of the necessary and appropriate stages that comprise the proposed autonomic development lifecycle process. This development process supports and guides the system designer all the way from the requirement stage to the code generation stage.
- A model transformation engine for writing the transformation rules that are responsible for transforming one model in one stage to another model in another stage.
- The definition of the required and necessary elements that have to be injected into a core system in order to introduce and bring in the autonomic capabilities.
- Using software design patterns to establish and define some relationship between involved elements in a particular situation, the monitoring of a specific parameter for instance.
- The definition and establishment of an appropriate way that brings adaptability and modifiability characteristics to the system to be made autonomic and evolvable.
- The handling of the model synchronisation issue, which is crucial in a design method claiming to be an MDD compliant.
- The development of an environment in a high level language such as Java or C# to enable the autonomic system designer to demonstrate and experience the feasibility of the proposed development process or design lifecycle.

1.3 Approach

The work described in this thesis aims to develop a design method for the autonomic systems. Here, we provide a support for an autonomic system designer throughout the whole lifecycle of the system development, from the system requirement phase to the implementation phase where useful skeletons of code injected with autonomic capabilities are created. To bridge the gap between the high level models aimed at domain experts and low level ones especially suitable for designers and programmers, we employ the MDD technique. Therefore, we divide the design method lifecycle process into a number of stages according to the MDD approach. Three fundamental stages can be identified, namely the Platform Independent Model (PIM), Platform Specific Model (PSM) and Code generation.

For theoretical support, this research draws on a number of research results emerging from related fields, including:

- Autonomic systems: using some proposed models and concepts that enable a system to manage itself by monitoring, analysing, planning for corrective actions and applying and issuing such actions.
- Model Driven Development: using the MDD approach to raise the abstraction level of the software systems. This is particularly important here to abstract away from any specific platform details in the early stages of the software development, which helps and enables the elimination of the effect of the technology change.
- Service Oriented Architecture (SOA): using some proposed models and features of the SOA paradigm to enable the autonomic and self-managing systems. SOA paradigm is well suited for addressing such systems due to the features and concepts offered by this paradigm. The loose coupling between interacting parties (services) and dynamic binding are crucial and beneficial features to the autonomic systems.
- Design patterns: using some well understood and appropriate design patterns to model and design some important relationships between involved parties in some scenarios or situations. The observer design pattern, for example, is used to establish the relationship between the monitor and the monitored or managed element.

This research also involves an experimental approach, in which a number of models and transformations are proposed. These models and transformations are then designed, implemented and evaluated via some case studies.

1.4 Contributions

The main novel contributions to knowledge resulting from conducting this research are outlined as follows:

- *A complete lifecycle for autonomic systems design*: this lifecycle describes the required stages that a system designer should go through should he or she intends to design and produce systems that exhibit autonomicity. This lifecycle guides and supports the system designer from the very early stage of

the design method where the designer sets up and defines the requirements of the system in question until the last stage where a useful piece of code is automatically generated. A set of design principles is adopted and employed throughout the lifecycle. The separation of concern principle, for example, is applied here to keep the abstract model of the system under development separate from the Meta system that consists of the required elements for exhibiting the autonomic capabilities. That makes the system easy to evolve and extend to accommodate new capabilities and add useful future desirable aspects.

- *The task model* for system requirement definition: this is the starting point of the lifecycle of the proposed design method. Here, the system requirements are captured and presented in terms of tasks. *A task is a high level goal that should be addressed by the system in question.* Here, unlike other approaches that adopt the MDD, our approach raises the abstraction level even higher by not binding to any specific technology, the object oriented paradigm for instance. A UML use case diagram is used to specify the tasks. These tasks are defined at very high level in a way that they can be broken down and expressed in terms of services. These services responsible for fulfilling and addressing a specific task are generated by using the UML sequence diagram, which shows the relationship between those interacting services. This process is repeated for each task in the system under study. The set of tasks involved in a particular system, and thus the whole system, is encapsulated in an entity called a *domain.*

- *A Model Driven Development (MDD) compliant design method*: the design method proposed in this research is supported and based on the Model Driven Development (MDD) paradigm. This results in raising the abstraction level of software development and thus eliminating, or at least minimising, the technology change effect.

- *Transformation rules engine*: this includes a set of required files used to encapsulate the transformation rules necessary to generate one model from another. These files have been coded in Java classes and Java DOM APIs are used to parse the various involving models since they are all expressed and

represented in XML documents. Here, transformation files can be, generally, classified into:

  - An abstract to Platform Independent Model (PIM) autonomic model transformation.
  - A Platform Independent Model (PIM) autonomic to Platform Specific Model (PSM) autonomic model transformation.
  - A Platform Specific Model (PSM) autonomic to code transformation.

- *Aspects profiles*: these profiles are defined to specify and encapsulate the required elements and components for the various aspects that the system under development might accommodate. Aspects include, the security and Quality of Service (QoS), the four autonomic capabilities (*self-healing, self-protecting, self-configuring and self-optimising*).

- *Autonomic design styles*: In our approach to designing the autonomic systems, we have proposed and employed two different design styles to introduce and inject the autonomic capabilities, namely the *Service-Monitor-Controller (SMC)* and the *design by contract* styles.

- *Adaptable automatically generated code*: the autonomic code skeletons and artifacts generated by the transformation rules have been made adaptable in that a new artifact or component can be easily added or removed from the system in question. Such autonomic code skeletons follow a particular design style and pattern that helps and supports the issue of system adaptability.

- *Model synchronization mechanism:* A mechanism for realising the synchronisation between the involved models throughout the MDD based development process lifecycle is proposed and implemented. This Java based framework synchronises the involved models and keep the whole system in a consistent status should any changes or modifications have been performed on any part of the system.

1.5 Research objectives

In this study of ways to extend the Model Driven Development (MDD) paradigm to support business process oriented architecture, the project's objectives include:

- The development of a model-driven design method for autonomic systems, which starts from business domain models and facilitates both intentions and aspect injection.
- The development of model transformation rules and associated engine to support autonomic software engineering.
- The design and definition of an autonomic systems profile including associated metamodels and frameworks.
- The development of model synchronisation framework to support software evolution and/or process and aspect injection.
- The evaluation of the developed method.

1.6 Thesis Organisation

This thesis comprises of nine chapters, which are outlined below:

- Chapter 1: will introduce the main motivations of the work presented in this thesis, challenges, contributions and thesis structure.
- Chapter 2: will introduce the relevant background theories, principles and technologies relevant to the work presented in this thesis. Such technologies and approaches include Model Driven Development (MDD), Autonomic systems, Service oriented Architecture (SOA) and Web Services
- Chapter 3: will present a review of previous works on autonomic systems design that have been conducted so far by which the proposed design method presented in this thesis was inspired. In particular, this chapter reviews some of the research that has been reported in the literature relevant to model-based and model-driven development methods for autonomic software, some of which have adopted a range of paradigms, theories and/or architectural models.
- Chapter 4: will introduce our approach and design method for modelling and engineering autonomic systems from a conceptual perspective. Only the architecture, concepts and high level design decisions and ideas will be introduced and described here.

- Chapter 5: will present a detailed description of our design method. Here, a number of aspects are covered including the whole lifecycle of our design method and justification for design decisions and technology choices. The whole design method lifecycle process is described here in a number of stages according to the MDD approach.
- Chapter 6: will introduce and elaborate on the model transformation framework that have been proposed and developed. The three fundamental components of this framework are presented and described in detail. These components include the abstract to autonomic transformer, the PIM autonomic to PSM autonomic transformer and the autonomic code generation transformer.
- Chapter 7: will introduce and describe the model synchronisation framework that is proposed and developed to synchronise the different models of the proposed lifecycle once a change has been performed on any of these models.
- Chapter 8: will present an evaluation of the autonomic design method introduced in this thesis. The evaluation contains qualitative analysis of the design method proposed here. This included an evaluation of the design method lifecycle process in general in addition to some critical analysis to some models that have been adopted and developed in some particular stages throughout the MDD different stages.
- Chapter 9: will conclude the thesis where a summary, what has been achieved and contributions are presented as well as some proposed future work.

CHAPTER 2

BACKGROUND

This chapter provides a background description of the underpinning concepts and methods used in this study, and should provide a gentle introduction of the motivations for the work and the positioning of the proposed design model.

2.1 Model Driven Development Paradigm

The history of software development is a history of raising the abstraction level [1, 2]. Each level allows the system developer to focus more directly on solving the problem at hand rather than implementation details. This drive for higher abstraction has led to the development of high-level programming languages including Java, C++, C # and Smalltalk, and software development techniques such as Model Driven Development (MDD) paradigm.

The idea promoted by the MDD is to use models at different levels of abstraction for developing systems. Thus, the model plays a central role in this paradigm. In contrast to the non-MDA approaches, the model here is not used only for documentation purpose, as in the waterfall approach, but is regarded as a central artifact from which other intermediate models and source code are derived in an automated or semi automated manner. The transformations between different models are automated by using formal models of the software system and well-established mappings between different models.

2.2 Model Driven Architecture

The Model Driven Architecture (MDA) was proposed by the Object Management Group (OMG) -- in early 2001 -- as a support for model driven software development approach. The method is based on the separation of domain and platform modelling and information from the core software behaviour specification, then code is automatically generated for a given domain and platform using automated model-based transformation techniques. Hence, the models are considered as the backbone of the development process and used not only for documentation purposes but also as a building tool. Each activity of the system development life-cycle involves taking some models as input and

producing target models as output. Therefore, the process of building a software system can be viewed as a series of model transformations which lead, in the end, to delivering the final and target system [3].

2.2.1 Model abstraction

The MDA approach is based on ideas of raising the abstraction level, automated software generation and platform independence [2]. In [4], the MDA is described to be "... *motivated by integration and interoperability at the enterprise scale. It utilizes models and a generalized idea of architecture standards to address integration of enterprise systems in the face of heterogeneous and evolving technology and business domains ...*". In other words, there are three fundamental goals of the MDA approach, namely *interoperability*, *portability* and *reusability*. These are reflected in the model layering outlined below [5]:

- Computational Independent Model (CIM): This is the model of highest level of abstraction with which the MDA process starts. It presents only high-level concepts of the system under development and, thus, facilitates communication between the domain experts and the system architects. This model may be depicted in use cases diagrams to show the user interaction with the system and activity diagrams to present the high level activities required for achieving such a system.

- Platform Independent Model (PIM): A platform Independent Model views a system from platform independence point of view. No specific platform design decisions are made in this model, so it can be applied to a number of different platforms of similar type. This is very similar to Java slogan: "write once and run everywhere". The significant value of this approach is that reusability and extensibility can indeed be achieved here since many models aiming for many different platforms, such as .NET, EJB, CORBA, etc., can be produced from just one model.. Since this model is intended to be a platform independent, Unified Modelling Language (UML) is seen to be the best choice for this modelling task. However, an extension to the standard notations of this language is needed to introduce some domain specific concepts. This extension normally takes the form of UML profiles, which consist of predefined stereotypes and tagged values.

- Platform Specific Model (PSM): A platform Specific Model (PSM) represents a model of a system for a specific platform, which can be obtained by applying, manually or automatically, appropriate transformation and mapping rules on a Platform independent Model. Target platforms may include .NET, CORBA, J2EE, EJB, etc.

2.2.2 Model Transformation

The model transformation plays a key role in Model Driven Development [6]. A model transformation takes as input a model conforming to a given metamodel and produces as output another model conforming to a given metamodel. The following subsections introduce the proposed model transformation techniques.

2.2.2.1 Model Transformation techniques

Since the model transformation process is the key to a successful application of MDD [7], a number of techniques and approaches have been proposed and adopted so far to address this aspect. In [8], such techniques are categorised into "Model to code transformation" and "Model to model transformation". We agree with the authors in that it would be better to call it Model to text instead of model to code since non code artifacts, XML for instance, can be generated. These two kinds of approaches are presented in the two following subsections.

2.2.2.1.1 Model to text approaches

Such a category can be further classified into visitor based and template based approaches outlined below:

- *Visitor based approaches:* This kind of text (code) generators provides a visitor mechanism to traverse the internal representation of a model and write code into a text stream. Jamda [9] is an example of this kind of approaches, which is defined as an object oriented framework consisting of a set of classes to represent UML models, an API to manipulate models and a visitor mechanism to generate code. Inheritance is used here to introduce new model elements where existing Java classes representing the predefined model elements types are subclassed.
- *Template based approaches:* Most of the existing MDD tools adopt the template approach to accomplish the model to code generation. A set of popular tools include JET [10], FUUT-je [11], Codagen Architect [12],

AndroMDA [13], ArcStyler [14], OptimalJ [15], StringTemplate [16] and Apache velocity [17]. In [8] *a template usually consists of the target text containing splices of metacode to access information from the source and to perform code selection and iterative expansion.*

Compared to a visitor-based transformation, the structure of a template resembles more closely the code to be generated. Templates lend themselves to iterative development as they can be easily derived from examples. Since the template approaches discussed in this section operate on text, the patterns they contain are untyped and can represent syntactically or semantically incorrect code fragments. On the other hand, textual templates are independent of the target language and simplify the generation of any textual artifacts, including documentation.

2.2.2.1.2 Model to Model Approaches

Model-to-model transformations translate between source and target models, which can be instances of the same or different metamodels. All of these approaches support syntactic typing of variables and patterns. Most existing MDA tools, such as AndroMDA[13], provide only model-to-code transformations, which they use for generating PSMs (in this case being just the implementation code) from PIMs. When bridging large abstraction gaps between PIMs and PSMs, it is easier to generate intermediate models rather than go straight to the target PSM. For example, when going from a class diagram to an EJB implementation, tools such as OptimalJ [15] would generate an intermediate EJB component model, which contains all the necessary information to produce the actual Java code from it. This makes the transformations more modular and maintainable. Also, intermediate models may be needed for optimization and tuning, or at least for debugging purposes. In addition to PIM-to-PSM transformation, model-to-model transformations are useful for computing different views of a system model and synchronizing them.

2.2.3 Model Synchronization

The usage of different levels of abstraction and the separation of concerns, promised and addressed by the MDD approach, on the one hand reduce the complexity of the overall specification, but on the other hand the increasing number of used models very often leads to a wide range of inconsistencies. Model transformation technique can be used to overcome this problem as stated earlier where a source model is transformed to a target model according to a set of transformation rules. However, the software development

process is a quite iterative with frequent modifications to the involved models. Thus, some mechanisms and solutions should be introduced to address the issue of model synchronisation [18].

2.3 Service Oriented Development Approach

The concept of Service Oriented Architecture (SOA) has received significant attention within the software design and development community [19]. It is based on the idea of modelling software systems in terms of services which can be used by other systems. In other words, it is an architectural style of building software systems or applications which offers loose coupling between components, services in this case. *A service is a self-contained software module that is intended to provide a specific functionality and does not depend on the context or state of other services* [20]. Another definition by [21] defines SOA as a distinct approach for *separation of concerns*. Here, the logic required to solve a large problem can be better constructed, conducted and managed if it is decomposed into a collection of smaller, related pieces (services). This makes these components, services, reusable by other systems. However, reusability is not the real reason behind the emergence of the SOA approach since reusability has already been addressed by a great deal of existing component-based technologies such as COM+, CORBA and RMI. Instead, the SOA approach is aiming to address the interoperability issue which emerged because each component-based vendor wanted their solutions to be the market dominant.

2.3.1 Characteristics of Service Oriented Development Approach

The following characteristics can be identified in the SOA approach:

- Services are the basic building blocks of SOA. These services publish their interfaces, to be used by other applications, in: a platform, language and operating system independent way.
- Clients or service consumers can dynamically discover services. In other words, the location of the server is transparent to the client and is only known at runtime.
- Software (services) should be able to interoperate with other services running on other platforms.

Here, service providers register their services in a public registry or broker which later could be used by the service consumers to look up a particular service that match specific criteria. Provided that the service is available, the registry provides the consumer with a contract and an endpoint address for that service [20].

The key to SOA is Web standards, including XML, WSDL, SOAP, and UDDI. Each service has a standard-based interface. These common standards enable the services to discover, communicate, and interoperate with one another, independent of their underlying operating system, platform, or programming language. That is, for example, services written in C# running on .NET platforms and services written in Java, running on Java EE platforms, can both be used in a common composite application. Consequently, services can be deployed on various distributed platforms and be accessed across the network. Due to this flexibility and reusability of services, they can be easily created and rearranged to meet new business needs, providing business agility. With SOA, building and deploying IT systems becomes easier and faster. It provides the best of both worlds; it reuses IT assets and enables IT to flexibly change to better support business change [22].

Most of today's mission critical business applications and systems are being built using multiple distributed software components. However, as pointed out in [23], these software components systems fall short when it comes to the issue of developing self-managing systems, as:

- It is very difficult to upgrade software components to accommodate new functionalities without shutting down the running system.
- Extending the functionalities of the current software system does not occur in an automatic way and requires plenty of additional programming works and reconfiguration of systems by experienced IT professionals.
- In contrast, as stated earlier, the software service is self described and can be discovered and invoked at runtime through the Internet which offers a new and beneficial approach for composing software systems. Thus, the service concept and SOA can play a crucial role in designing and addressing the autonomic systems design issue [23, 24].

2.3.2 Web Services

Web services are a technology that can be used to implement SOA and are increasingly becoming the SOA implementation of choice [25]. In a definition by the World Wide Web Consortium (W3C) presented in [26], the web service is:

"... a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML-based messages conveyed by Internet protocols ..."

A definition by webservice.org presented in [27] states that: *"... web services are encapsulated, loosely coupled contracted functions offered via standard protocols ..."*

2.3.2.1 Web Services Programming Stack

The web services programming stack is a collection of protocols and Application Programming Interfaces (APIs) that enables individuals and applications locate and use web services. In [28], six layers for the stack can be identified and described as follows:

- *The network layer*: This is regarded as the foundation of the web services programming stack since all web services must be available over some network. Here, the most used network protocol is the HTTP. Other protocols, however, such as IIOP can also be used.
- *XML-based messaging layer*: This layer is responsible for facilitating communications between web services and their clients. The protocol used here is called SOAP, short for Simple Object Access Protocol.
- *Service description layer*: At this layer, the web service description takes place. The WSDL (Web Services Description Language) is used here to describe web services to clients in the form of XML documents.
- *Service publication layer*: Any form of action that makes the WSDL document available to a potential client can be regarded as a publication of service. However, the more popular way to carry out this task here is to publish the WSDL of a particular web service in a UDDI registry to be located and discovered by interested clients and developers.
- *Service discovery layer:* Any action that enables clients to access the WSDL for a particular web service is considered as a service discovery. This action

can be simple, such as accessing a file or URL containing the WSDL, or complex, such as querying a UDDI registry and using the retrieved WSDL files to select the most suitable web service.

- *Service workflow layer*: This layer is concerned with the process of composing web services into workflow. Put another way, this layer is responsible for modelling the interactions between a set of web services which often form a useful entity referred to as a business process. Modelling languages proposed for this task include WSFL from IBM, XLANG from Microsoft and, the most popular one, WS-BPEL.

2.4 Summary

This chapter introduced and overviewed the related and relevant technologies and paradigms to the research issue and methodologies presented in this thesis. Relevant technologies and paradigms presented here include:

- Autonomic computing
- Model driven development (MDD)
- Service Oriented Architecture (SOA)
- Web services

CHAPTER 3

AUTONOMIC SYSTEMS DESIGN

This chapter starts with a detailed description of autonomic systems design. This will be followed by a literature review of the state-of-the-art relevant to model driven tools and techniques for autonomic software development.

3.1 Autonomic Computing

The continually growing complexity and cost of today's software systems has made their management and maintenance task extremely difficult to achieve [29]. This is due to the distributed, dynamic, open and heterogeneous nature of these systems in which applications are made up of different components written by different vendors and implemented in different platforms [30]. In such systems, the task of troubleshooting technical problems can tie up systems and Information Technology (IT) professionals for significant periods of time while affecting the business performance in a negative way. Estimates show that from 30-70% of resources are used by IT professionals, in medium to large companies, for troubleshooting problems and the outrage costs per hour on business critical systems can range from thousands to millions of dollars [31]. To make the situation even worse, there is a shortage in the number of highly skilled people who can manage and handle the complexity of such computing environments. Since the existing tools and methodologies are incapable of managing the complexity of these systems and could not meet their requirements, an appropriate solution had to be adopted; otherwise the IT sector is heading for what it has been termed as a *software crisis* [32].

To overcome such a crisis and address the software management complexity, researchers had to consider alternative approaches based on strategies employed by the biological systems. As a consequence, the autonomic computing – as coined by IBM -- has emerged as a new paradigm and approach to the design of complex distributed systems. It is inspired by the mechanisms of the human nervous system and its main objective is to reduce the cost and expertise required for managing the complexity of the IT systems

[29, 33]. This was first introduced by IBM to the National Academy of Engineers at Harvard University in March 2001[32]. In such a paradigm, software systems will have the responsibility for managing themselves, given only high-level policies, and therefore shielding the human user or the system administrator from handling and performing this rather difficult task. Here, software systems should have the ability and the quality required to exhibit the self-healing, self-configuring, self-optimising and self-protecting capabilities.

The fundamental characteristics of an autonomic system are presented in [32], as follows:

- An autonomic computing system needs to "know itself". In other words, in order to govern itself, an autonomic system needs to gain detailed knowledge of its components, current status, ultimate capacity and all connections with other systems.
- An autonomic computing system must be able to configure and reconfigure itself under varying, including unpredictable, conditions.
- An autonomic computing system never settles for the status quo; it always looks for ways to optimise its workings. It monitors its constituent parts and fine-tunes workflow to achieve predetermined system goals.
- An autonomic computing system must be able to recover from routine and unusual events that might cause some of its parts to malfunction, and discover problems and then find an alternate way of using resources or reconfiguring the system to keep functioning smoothly.
- An autonomic computing system must be an expert in self-protection. It must detect, identify and protect itself against various types of attacks to maintain overall system security and integrity.
- An autonomic computing system must know its surrounding environment and act accordingly. It will find and generate rules for how best to interact with neighbouring systems. It should negotiate the use by other systems of its utilized elements, changing both itself and its environment in the process of adapting.
- While an autonomic computing system independent in its ability to manage itself, it must function in a heterogeneous world as well.

- An autonomic computing system should be able to anticipate the optimised resources required while keeping its complexity hidden without the involvement of the user in that implementation.

The fundamental capabilities of the autonomic or self-managing systems as described by IBM are:

- *Self-healing*: an autonomic system should be able to detect and diagnose occurring problems. Typical problems can be of a low level type (hardware failure) such as memory chip bit errors, or of a high level type (software aspect) such as an error in a directory service. The minimum requirement of the self healing process is to fix the detected problem and not to introduce any further harm to the system.
- *Self-optimisation*: an autonomic system should be able to optimise and tune its resources in order to improve performance and provide improved QoS. Such optimization actions may include reallocating resources in response to dynamically changing workloads or ensuring that a particular business transaction can be accomplished in a timely fashion. Changes may be initiated proactively as opposed to the reactive behaviour.
- *Self-configuration:* an autonomic computing system should be able to configure itself driven by high level goals. This enables the system to adapt dynamically to changing enticements and user requirements. The self configuration process can take the form of installing new components or the removal of existing ones.
- *Self-protecting:* an autonomic system should be able to protect itself from malicious attacks and also from costly mistakes committed by the system end users which may take the form of deleting important files. The autonomic system should be augmented with the necessary tools and resources to achieve security, privacy and data integrity. The ideal self protecting system should be able to anticipate security attacks and try to prevent them from occurring rather than to take a reactive action [29, 32, 34].

3.2 IBM Autonomic Reference Model

To realise autonomicity, IBM has proposed a reference model for autonomic control loops which is sometimes referred to as MAPE-K (Monitor, Analyse, Plan, Execute, and Knowledge) loop. . It is been pointed out in [29] that the MAPE-K control loop resembles and is probably inspired by the generic agent model proposed by Russel et al [35] . In such a model an intelligent agent observes its environment via sensors, and uses these observations to plan actions to be executed on the environment.

Below is a brief description of the components that are involved in the above control loop. Abstractly speaking, the autonomic element is composed of two primary components, namely the *autonomic manager* and *managed element.*

- *The managed element:* it is the entity that is monitored by the autonomic manager to observe some desirable or undesirable behaviour. Such an element may take the form of software resource such as a database or a directory service, or hardware resource such as a CPU or a printer [36].
- *Sensors and Effectors:* the sensors are used by the monitor component of the autonomic manager to monitor and collect the required information about the managed element. Similarly, the effectors are used by the execute component to carry out the possible changes to the managed element.
- *Monitor:* it is responsible for collecting, aggregating, filtering and reporting collected data from a managed element.
- *Plan:* the plan component uses monitoring data obtained from the sensors to create a series of changes which are performed on the managed element. A typical form of such a component may take the form of Event-Condition-Action (ECA) rules. Existing policy languages and applications in autonomic computing can be found in [37] [38, 39] [40, 41] [42, 43].
- *Execute:* this contains the business logic that drives the execution of a specific plan.

3.3 Existing Implementations of the IBM Control Loop

A number of tools and applications have been developed to implement the above described reference model. Below is a brief description of some of these implementations:

- *The Autonomic Toolkit*: this prototype implementation, referred to as the Autonomic Management Engine, of the MAPE-K control loop was developed by IBM as part of its developerWorks Autonomic Computing Toolkit. This tool is not meant to be a "ready to use" application for developing autonomic managers but rather it serves as a practical framework and reference implementation for injecting autonomic capabilities into software systems [44]. Such a framework is implemented in Java but can be communicated with by other applications through XML messages.
- *Kinesthetics eXtreme:* this Java based implementation of the MAPE-K control loop was developed by to introduce autonomic capabilities to legacy systems that were not designed in the first place with the autonomic capability in mind. Refer to [45] [46] for more information.
- *ABLE:* this IBM's toolkit offers the autonomic management in the form of multi agent architecture where each autonomic manager is implemented as an agent. Such a toolkit is developed in Java. See [47] for more information on ABLE.

3.4 Model Driven Development

In this section, we review the approaches and attempts that have been made to address the autonomic systems design problem using the Model Driven Development (MDD) paradigm.

Gracanin *et al* [48] believe that agent based systems and architectures offer a strong foundation for the design and development of an autonomic system. The researchers pointed out that the key challenge here is the selection and efficient use of effective agent architecture. They also believe that the Model Driven Approach accommodates the underlying architecture that could automate the development process. In other words, the MDA can be used here as a basis for system composition and automatic code generation of autonomic systems. The outcome of this approach is a framework for the agent-based, model-driven architecture for autonomic applications development. The whole work presented by the researchers is based entirely on a distributed agent architecture called COUGAAR, short for the Cognitive Agent Architecture. The COUGAAR is an open source, distributed agent architecture based on Java.

Also a similar approach, based partially on MDA, by Bulter *et al* [49] tackles the issue of the accommodation of unforeseen changes. The researchers in this work address the

complexity and the high cost of such an issue via employing autonomic techniques in a model driven approach to system change. The researchers identified two aspects of system change. The first one is a design based change over the development cycle which is best characterized by incrementing software version numbers. The second aspect deals with dynamic change over software systems in order to react intelligently to a dynamically evolving environment. The researchers address the first aspect of system change through the adoption of the Model Driven Architecture techniques where the focus is on the importance of modelling in managing change from a design led focus. Here, the development of systems is organized around a set of models by forcing a series of transformations between models, organized into an architectural framework of layers and transformations. Pena *et al* [50] also employed the MDA to model and design autonomic systems. More precisely, the researchers have dealt with the autonomic systems as policy based self management software systems where they use the MDA to model the policy to avoid any unnecessary platform specific details at the abstraction level of a policy. They, then, apply the transformations to the different models to bring the policy through all the level that has to go through until reaching the final level which represents the implementation.

3.5 Model-Based Design Approaches

In this section, we review a set of approaches to designing the autonomic systems which have adopted the model based technique to tackle and address this issue. In addition to some approaches proposed and adopted by the research team at the Computing and Mathematical Sciences School of Liverpool John Moores University, a pool of other approaches is presented as well.

In addition to the IBM approach to designing autonomic systems, many other approaches and models have been proposed to handle such an issue. These approaches have adopted and applied different paradigms, technologies and system theories in their proposed autonomic models. Such paradigms include some techniques and models adopted from the artificial intelligent field, agent based systems, system theories such as the Soft System Methodology (SSM) and Viable System Model (VSM), and the MDA.

Law *et al* [51], adopted the VSM approach to designing autonomic systems. Based on this approach, the researchers developed a reference architecture, known as J-reference,

which was underpinned by deliberative and normative models adopted from the artificial intelligent field.

Also, Bustard *et al* [52] have applied the Viable System Model (VSM) [53, 54] in combination with the Checkland's Soft System Methodology (SSM) [55, 56]. The latter was employed to extract and define the system requirements, which offers a systematic and systemic structure with which to unravel complex situations using basic principles of system thinking. This design model, adopts a top-down approach in a set of stages, including:

- **Environment Design:** This is achieved using the SSM method, which is refined using the VSM model.
- **System design:** which is achieved using a combination of methods including SSM and other computing-oriented modelling techniques such as; UML or Other Design Technique (ODT).

Based on the above design model, Taleb-Bendiab *et al* [57] have taken a further step and proposed a development process lifecycle. In fact, this research was partially and to some extent inspired and built on the model presented here. A high level model of this proposed approach is depicted in Figure 3.1.

Figure 3.1: High level guidelines for Autonomic System Design [57].

Badr *et al* [58] extended current models of self-adaptive software and reflective middleware with deliberative control mechanisms, which resulted in proposing a novel

autonomic control middleware. This is to support the design and lifetime management of deliberative middleware and application services. The developed approach was used as a reference model to facilitate a normative self-governance model that supports the safe self-adaptation of distributed applications for lifetime application management.

Pereira [59] used the VSM model to describe the fundamental requirements for a software framework and associated middleware services in order to develop on demand application services based on employing self-healing capability. Moreover, the researcher provided better understanding of software self-healing requirements for autonomic distributed software engineering, where she presented a reference model for self-healing capability.

Omar [60] proposed and developed a self-management reference model to specify and design autonomic distributed applications. In such a model, the management and control functionalities are encapsulated in middleware services that support and help deploy, discover, invoke and manage the planetary scale resources.

Reilly [61] has studied the instrumentation need for distributed systems management and the manner in which this instrumentation may be applied. The notion of *on demand distributed software instrumentation* was investigated here. As in[60], the management aspect (instrumentation here) is promoted as a new middleware service. The outcome of this research was the development of a dynamic software instrumentation framework.

Herring [62] ,in his PhD thesis entitled "The intelligent control paradigm for adaptable and adaptive architecture", proposed a viable software architecture which is totally based on the viable system model (VSM). The result of his work is a model-based architecture for developing adaptive, at runtime, software systems. These systems are referred to as viable systems. In his work, Herring adopted the Product Line Approach (PLA) to develop the viable system approach. The PLA contains a meta-level on which some domain-specific PLAs are based. This meta- level comprises a reference-meta model, meta-architecture, meta-framework, and a meta-component transfer protocol. Here, these elements are modelled using the Viable System Model (VSM).

In[63], the approach of designing self managing systems is based on the use of requirements goal models. In such an approach, unlike in [62] and [51], a large space of possible behaviour of the system under consideration is defined at design time. Here, the functional stakeholder goals are modelled in terms of hard goals where the goals either

satisfied or denied. In contrast, Non functional stakeholders' goals are modelled as soft goals where goals tend to be qualitative and hard to define formally. In general, the proposed architecture for autonomic software systems here is based upon the association of each goal in the goal model with an autonomic element whose objective is to address and achieve that goal. Although this approach is based on a well established and tested design principle, it is not particularly suitable for designing autonomic systems. This is due to the open and dynamic nature of such systems in which new and better alternative behaviours are likely to present and emerge during the runtime. Moreover, some of the already (at design time) identified and implemented behaviour may turn out to be impractical in some situations. Therefore, new alternatives and opportunities will have to be discovered and implemented by the system at runtime. So, this approach will not always succeed in addressing the requirements and behaviour of autonomic systems and thus an autonomic system based on it will not survive in many situations.

3.6 Aspect Oriented Programming based Techniques

This section introduces the techniques that were proposed and implemented using the Aspect Oriented Programming approach [64] for designing and modelling autonomic systems.

Dantas *et al* [65] proposed a framework to provide adaptation in software systems using AspectJ. Such a framework is composed of a number of components, namely the base application, adaptability aspects, auxiliary classes, adaptation data provider, context manager. While the base application contains the core system code, the other components coordinate and work together to provide the adaptability capability at runtime.

Another work by Yang *et al* [66] was carried out to provide dynamic adaptation using AOP. Two concepts are used here, namely the join points and rules. The former decides where the adaptation should be applied in the code whereas the latter specifies the conditions when an adaptation should take place.

Greenwood *et al* [67] proposed and developed a framework which introduces adaptive behaviour to applications using a combination of AOP and policies. Adaptability is achieved here by defining the appropriate policies using the Event-Condition-Action rules.

Duzan *et al* [68] proposed an aspect-based approach to programming QoS adaptive applications that separates the QoS and adaptation concerns from the core application code and middleware services.

Also, Falacarin [69] developed a framework for dynamically enabling applications to adapt and evolve using runtime aspect oriented programming. In such a framework, a system designer or administrator can control the architecture of an application by dynamically inserting and removing code extensions.

3.7 Model Transformation Techniques

This section reviews the state of the art and current approaches to the model transformation field. Here, we distinguish between two kinds of approaches, namely the model to model approach and the model to code approach.

"*... A transformation of two or more models may be described by specifying how a model conforming to its metamodel is translated into a corresponding model conforming to the other metamodel ...*" [70]. Refer to Section 2.2.2 for more information on model transformation paradigm.

A great deal of techniques and approaches have been proposed and developed in the field of model transformations. Below we introduce and review some of these approaches:

UMT (UML Model Transformation Tool), proposed and developed by SINTEF [71], is a general purpose UML transformation tool that is designed and intended to perform different kind of transformations. It is a tool to support model transformation and code generation based on UML models in the form of XMI[1][72] documents. The XMI models are imported by the tool and then converted into a simpler intermediate format which forms the basis for further validation and generation towards different target and specific platforms, Web services for instance. The intermediate format, referred to in UMT as light XMI, is represented in XML format. The transformation rules here are done and encapsulated in XSLT files where such transformers are run on intermediated models (light XMI) to target and generate the code for one specific platform. In addition, the transformation engine here does also support writing transformers in other technologies such as Java.

AndroMDA [13] is an open source code generation framework that adheres to the Model Driven Architecture (MDA) approach. It takes a UML model from a CASE-tool and generates classes and deployable components (J2EE or other) specific for your application architecture. AndroMDA comes with an array of ready-made cartridges for

[1] XMI, short for XML Metadata Interchange, is an Object Management Group (OMG) standard for exchanging metadata information via XML documents.

common architectures like Java, Web services, Spring, EJB, .NET, Hibernate, JSF, Struts and XSD. Similar to the UMT, the design model is imported to this tool in the form of an XMI file where specific transformer is applied to it which generates code for specific platform. However, the model to model transformation rules are here written in Java language and also, recently, in the QVT-like Atlas Transformation Language (ATL). The code generation process which is the ultimate goal of the AndroMDA tool is done via the use of templates. Such templates are written and coded using well known template engines such as velocity and FreeMarker. A worth mentioning point here is that the Platform Specific Model (PSM) is the same as the code generation stage since the transformation process goes from the Platform Independent Model (PIM) straightaway to the code.

SiTra [73, 74] is a simple model transformer written in Java language. This model transformation framework was designed to help advances programmers to start using the concepts of model transformation and also for the academic researchers to experiment with the creation of prototypes of implementation of their transformations. The underlying idea of SiTra is to focus primarily on the implementation of model transformations rather than on the specification language, maintenance and documentation aspects of such transformations.

3.8 Summary and Discussion

The autonomic system design approaches presented in this chapter have adopted different paradigms and taken several directions. For the simplicity sake, these approaches are classified in this discussion into MDD and non-MDD based approaches. In general, non-MDD based approaches tend to focus on the implementation aspect of the autonomic system development process where usually the platform, technology or architecture is already decided on in the early stages. This in fact makes it very difficult for the system designer or developer to modify or change the system in question in order to target and migrate to a different or new emerging platform or technology. Rather than concentrating on the problem at hand, the system designer here is distracted by the solution or implementation details. Also, there is no a supporting tool to transform the system requirements which normally take the form of UML diagrams to executable code. As for the MDD based approaches, their primary goal is to start at a higher level when designing and developing autonomic systems. This enables a thorough specification and

more controllable engineering of the system requirements since the system designer is not rushed too early into the implementation and technology details and thus is more focused on the problem at hand. Then, via supporting tools and transformers, the system at the abstract level is transformed into a set of software components for one specific platform. However, these approaches have shown some drawbacks which need to be addressed. Firstly, although these approaches claim to be adhering to the MDD principles and starting the development process at the PIM stage, they in fact make an early commitment to specific architectures and platforms. A typical example of this trend is the adoption of the object orientation paradigm via normally the use of the UML class diagrams. Consequently, the domain expert, whose involvement is very important regarding the aspect of expressing the system under study in a platform independent manner, is neglected here and not included in the development process loop. Also, current approaches that adopt the SOA paradigm and the web services in particular as the target platform, use the WS-BPEL 2.0 to define and specify the service interactions for one specific business process (the task in this thesis terminology). This tight and explicit coupling between business processes and their associated services often renders and works against the agile adaptation of information systems to very likely changing business processes and users requirements. Secondly, some of these approaches bypass the PSM stage and go straightaway from the PIM stage to the code dealing with the latter as the PSM. This leads to the issue of the semantic gap[2][75] which has direct impact on the code generators since the latter will be of high complexity to bridge the detail gap from the PIM to the code. Thirdly, the handling of the model modification impact seems to be unaddressed. There is no mechanism in place to synchronise the models that make up the development process lifecycle once a change or modification has been undertaken.

These drawbacks and unaddressed issues are handled and dealt with in the work presented in this thesis. The next chapter provides a conceptual description of the proposed and developed design method that accommodate the necessary stages, components and frameworks for designing the autonomic systems in general and overcoming and addressing the deficiencies and weaknesses that are exhibited by the so far proposed approaches.

[2] The semantic gap issue refers to the situation where there is a large semantic gap or distance between the input and target language.

CHAPTER 4

CONCEPTUAL DESCRIPTION OF PROPOSED DESIGN METHOD

The primary goal of this chapter is to introduce the conceptual and architectural description of the proposed design method for engineering autonomic systems. The detailed description of the proof-of-concept implementation of the method and associated supporting techniques will be covered in the following chapters.

4.1 An Overview of Proposed Design Method

Following the MDD principles, the proposed novel method is intended to facilitate the seamless autonomic software development starting from domain modelling and business process requirement models (intention) all the way to the generation of required software. The general requirements of the proposed design method include:

- Design a system with the domain experts in mind.
- Provide a mechanism to automate the transition from a domain specific model to a platform specific model, assuming the code is a model too.
- Adding the autonomic capabilities to the system in question in a way that maintains the separation of concern design principle.
- Creating and producing skeletons of code that exhibit autonomicity which can then be used by programmers or system developers to fill out some of the required business logic.
- Through the model abstraction principle – the method makes no commitment to platform specific design decisions at early stages in order to target a broad range of platforms and technologies.

Following the conventional MDD principles, the initial arrangement for the proposed development process can be constructed as illustrated in Figure 4.1. Since it is evident that such an arrangement is not capable of accommodating and addressing some of the above listed requirements, an extended and modified version of this arrangement is required. This is shown and introduced in the next section.

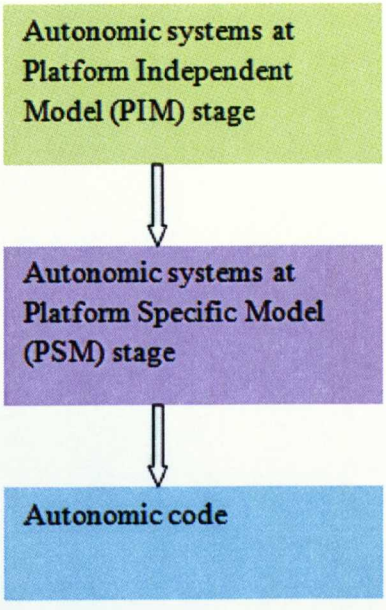

Figure 4. 1: A high level architecture of proposed autonomic development process.

4.1.1 An Extended MDD arrangement for Autonomic Systems design

As illustrated in Figure 4.1, the arrangement of the proposed development process is mapped to the standard MDD stages. . However, in order to engineer the autonomic capabilities in a way that maintains the separation of concerns design principle, the first stage of the arrangement depicted in Figure 4.1 should be split up into two distinct stages, one for specifying the core system requirements and the other for injecting the autonomic capabilities into the core system components. In relation to the separation of concerns principle, the autonomic aspect is seen here as the concern that should be engineered in a way that has no effect on the core system under study. The latter should be able to target and accommodate new desirable non functional requirements and not just limited and bound to the autonomic capabilities (or just one autonomic capability such as the self healing). The extended and revised arrangement of the conventional MDD stages presented in Figure 4.1 is shown in Figure 4.2. As it can be seen, the new arrangement is composed of four primary stages or models, namely: the Platform Independent Model, Autonomic Platform Independent Model, Autonomic Platform Specific Model, and Autonomic code.

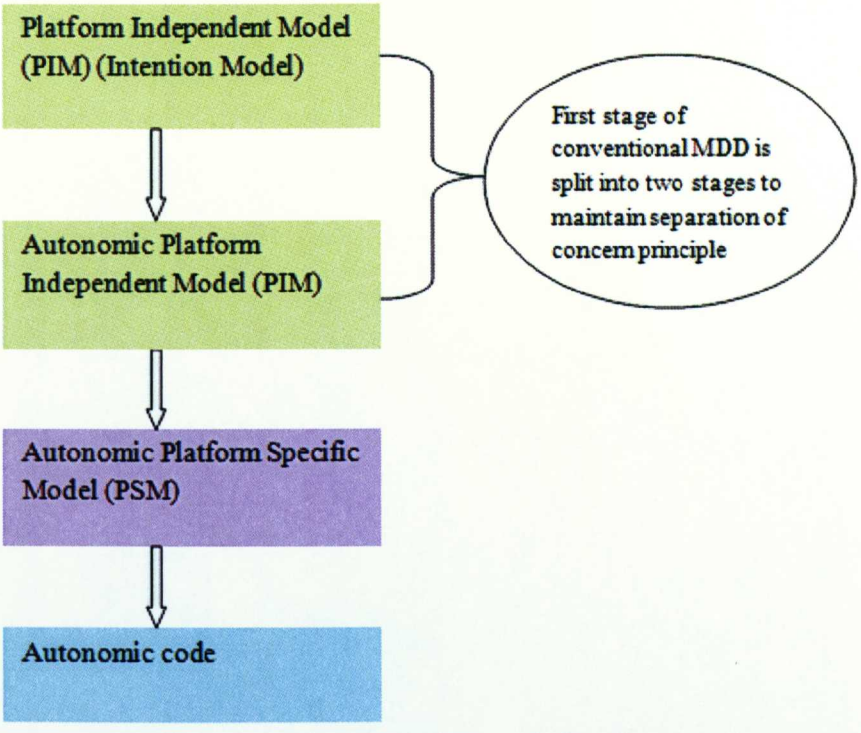

Figure 4.2: A revised development process lifecycle for autonomic systems.

The models shown in Figure 4.2 are described as follows:

- The Intention Model: In this stage of the development process lifecycle, the system requirements are described and defined using some concepts and terminology close to and understood by the domain experts. This helps in introducing practically those stakeholders into the automated development process loop. The system requirements being modelled here are of a functional nature, which represent the core system or the Intention model. The outcome of this stage is a platform independent or jargon free model.

- The Autonomic Model: To achieve the separation of concern principle, the autonomic capabilities were added at later stage and kept separate from the core system obtained in the first stage. This frees the core system from been bound to one specific non functional capability, the autonomic in this case, and thus not being able to evolve or target a new capability. The autonomic model contains the core system (Intention model) in addition to the autonomic components which are encapsulated in an entity called the Assurance. As the name suggests, the Assurance takes the responsibility for assuring that the core system is working and working well according to some predefined policies and reports coming from components like sensors and monitors.

- The PSM Autonomic Model: This model contains the same components as the ones in the previous model with the terminology and data types for specific platform added into these components.
- The Autonomic code: This model contains the business logic and workable autonomic code which takes the form of components or services (SOA), and their interactions as well, that work together and exhibit the autonomic capabilities injected in earlier stages. However, some business logic or code needs to be filled in by programmers to make the code fully working. Examples of such code include Web Services, EJB, CORBA, etc.

4.1.2 Model Transformation

The transformation from one model to another at any one stage is performed via a generative model transformer equipped with level specific templates. Thus, in this case, three different model transformers are required, namely: the Autonomic profile, Platform metadata injector, and code generator. The responsibilities and tasks of theses transformers can be described as follows:

- The autonomic transformer is responsible for adding and injecting the autonomic capabilities into the abstract model residing at the highest stage (PIM).
- The Platform metadata injector transformer contains the artifacts and necessary processes to add the required terminology and data types to target a specific platform, such as Web Services, EJB, C#, etc.
- The Code generator transformer produces the final and workable autonomic code which takes the form of service or components -- following Services Component Architecture (SCA) paradigm -- that work together and exhibit the autonomic capabilities injected in earlier stages. Examples include Web Services, EJB, CORBA, etc.

More detailed descriptions of these transformers are presented in Chapter 6.

4.2 Fundamental Concepts

The proposed autonomic development process is based on some fundamental concepts, namely:

- *Domain:* The domain here is the system under consideration which comprises a number of tasks.
- *Task*: Each task, in turn, contains a set of services responsible for addressing and achieving that task.
- *Service*: These services, later at the code generation stage, are mapped into software components such as Web Services, CORBA, Java, .NET, etc.
- *Composite*: the services of a particular task coordinate with each other to address the purpose of that task. Such coordination, which involves a set of interactions, is encapsulated in an entity called *composite*.

The proposed design method presented here extends the emerging *Service Component Architecture (SCA)*[3] [76] standard in that, it provides support starting from the business process (task) level all the way to the code generation stages. Hence, we refer to it as Autonomic, Task, Service and Component architecture (AutoTaSC) method.

4.3 Model Synchronisation issue

Model synchronisation in MDD can be characterised as a mechanism to facilitate model consistency management, which can be triggered by model change due to either upstream or downstream model transformations and/or other typical software changes made further downstream in the MDD processes. Hence, the absence of a robust mechanism for MDD model synchronization can lead to situations where any source and target models can change in an uncontrolled and inconsistent way. Thus, as depicted in Figure 4.3, the proposed method puts a major emphasis on this aspect and applies the autonomic design principles sensors-effectors (or event condition action) as the underlying mechanism to controlled model synchronisation. In other words, any modification made on one model would trigger appropriate actions on the other models to reflect and propagate the changes being carried out and leave the system in a consistent state. Therefore, a robust network of four collaborative change or modification controllers, one for each model, should be constructed and formed to keep the whole lifecycle in a consistent state.

In particular, this synchronisation framework distinguishes between changes taking the form of adding (or removing) new services or other elements where their definitions are

[3] The Service Component Architecture is a set of specifications which describes a model for building applications and systems using a Service-Oriented Architecture.

already included in the metamodel and changes requiring the introduction of new concept to the metamodel of a considered system under development. We also distinguish between changes or modifications conducted on core services or components and those performed on autonomic services or components responsible for injecting autonomic capabilities into core systems. These kinds of change have a direct impact on the way and manner the above mentioned change controllers communicate and interact. For instance, if a change has occurred at the autonomic code model, two scenarios can be identified:

- Autonomic component related changes: changes of this kind trigger the PSM and PIM autonomic model controllers but not the abstract (intention) model controller.
- Core component related changes: changes performed on this kind of components require change notification messages to be sent to the remaining models' controllers including the abstract or intention change controller.

Below is a description of the necessary controllers and their communication channels when the change is connected with adding or removing some components whose definitions are already included in the metamodel of the system in question.

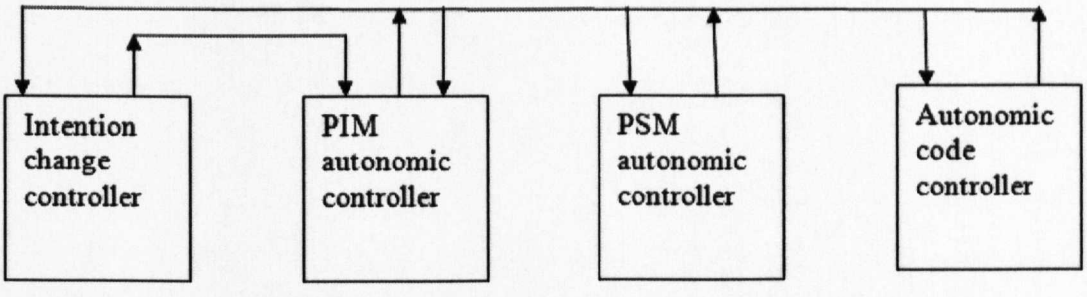

Figure 4.3: A network of cooperative change controllers for core components change.

4.3.1 The Intention Model Change Controller

Any changes or modifications made on the model at this stage of the system development lifecycle should be reported so other models at the other stages can modify themselves accordingly. This notification task is performed by the Intention model change controller. In this case, this controller should only notify the autonomic change manager and in turn the latter notifies the two remaining change controllers, the PSM autonomic and autonomic code controllers. The change notification for the controllers beyond the autonomic controller cannot be sent directly by this controller since at that point of time

the critical parameters (parameters of particular interest to monitor) of the new added services would not have been defined and added by the system designer. Critical parameters and policy definitions are accomplished in the PIM autonomic stage, so it is better off delegate the notification task to the change controller of this stage.

4.3.2 The PIM Autonomic Change Controller

Changes and modifications conducted at the PIM autonomic model result in notification messages sent by the change manager to the three remaining controllers, the Intention change controller, the PSM autonomic change controller and the autonomic code change controller.

4.3.3 The PSM Autonomic Change Controller

Changes and modification performed on the model at the PSM autonomic stage cause the controller to send change notification messages to each of the change controllers located at the other stages of the development process lifecycle.

4.3.4 The Autonomic Code Change Controller

Any change or modification of the working code will result in notification messages sent by this controller to the remaining change controllers in the development process lifecycle. Changes may take the form of adding or removing service components such as Web Services. Also changing the workflow of interacting services or components can be considered as another form of model change.

Figure 4.3, shows the above change controllers as well as the communication channels that connect them when the change is carried out on the core components of the system in a certain model of the lifecycle process.

As for those changes performed on the autonomic components, the communication channels will be slightly different from those depicted in Figure 4.3. The appropriate and correct communication channels of such a scenario can be shown in Figure 4.4. As it can be seen from the diagram, there is no direct connection or messaging channel between the Intention model change controller and the other change controllers of the network

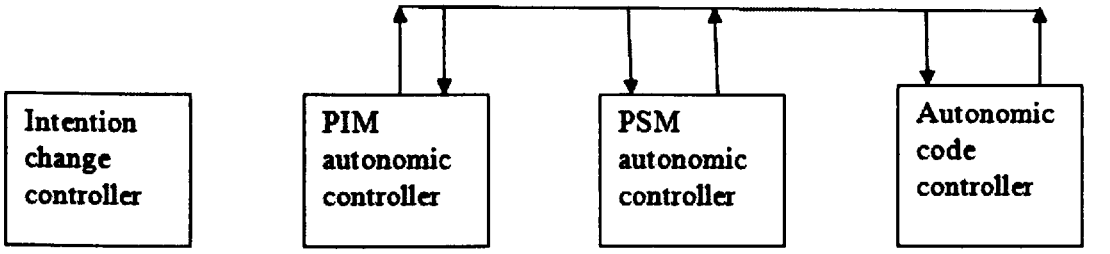

Figure 4.4: A network of controllers for autonomic components change.

A detailed description of this synchronisation framework and how the autonomic computing principles can be used to handle and address the MDD change issue is presented in Chapter 7.

4.4 Summary

This chapter introduced our approach and design method for modelling and engineering autonomic systems from a conceptual perspective. Put another way, only the architectures, concepts and high level design decisions ideas were introduced and described here. These aspects include the fundamental stages that comprise the lifecycle of the proposed development process, the model transformers required to transition from one model to another and the model synchronisation mechanism adopted to keep the system in a consistent state should any modifications or changes occur.

CHAPTER 5

PROPOSED AUTONOMIC DESIGN METHOD

In this chapter, we introduce in detail our design method and approach to designing autonomic systems. Here, we present the design process lifecycle as well as some justifiable design decisions and technology choices.

5.1. Autonomic Design Method and Model Lifecycle

The proposed autonomic design method adopts the MDD paradigm to gain some valuable benefits which are very crucial in designing distributed systems in general and autonomic systems in particular. Raising the abstraction level and separation of concerns are among those benefits. Achieving those design principles will result in a system design that can be a future proof and would survive in a world of rapidly changing system requirements and technologies. Comparing with the three fundamental models of MDD, these three models are present in our method. At the very high level, we present our system requirements in a computation independent model where a set of UML use cases is used to identify and define the required tasks in a particular domain. From the first stage of expressing our system in an XML file to the stage where autonomic capabilities are added and encapsulated in a separate XML file, we are still at the Platform Independent Model (PIM) stage. However, the Platform Specific Model (PSM) starts at the point of adding the specific elements and terms for a specific platform, Java for instance. Also the code generation stage is supported in our design method which is also done via encapsulating the transformation rules in Java classes.

The fundamental stages of the proposed design method lifecycle are depicted in Figure 5.1. Also, the three transformation engines or model transformers necessary for transition from one stage to another are shown. These involving stages and required model transformers are described in more detail in the subsequent sections. For more information on the proposed autonomic design method, please refer to [77, 78].

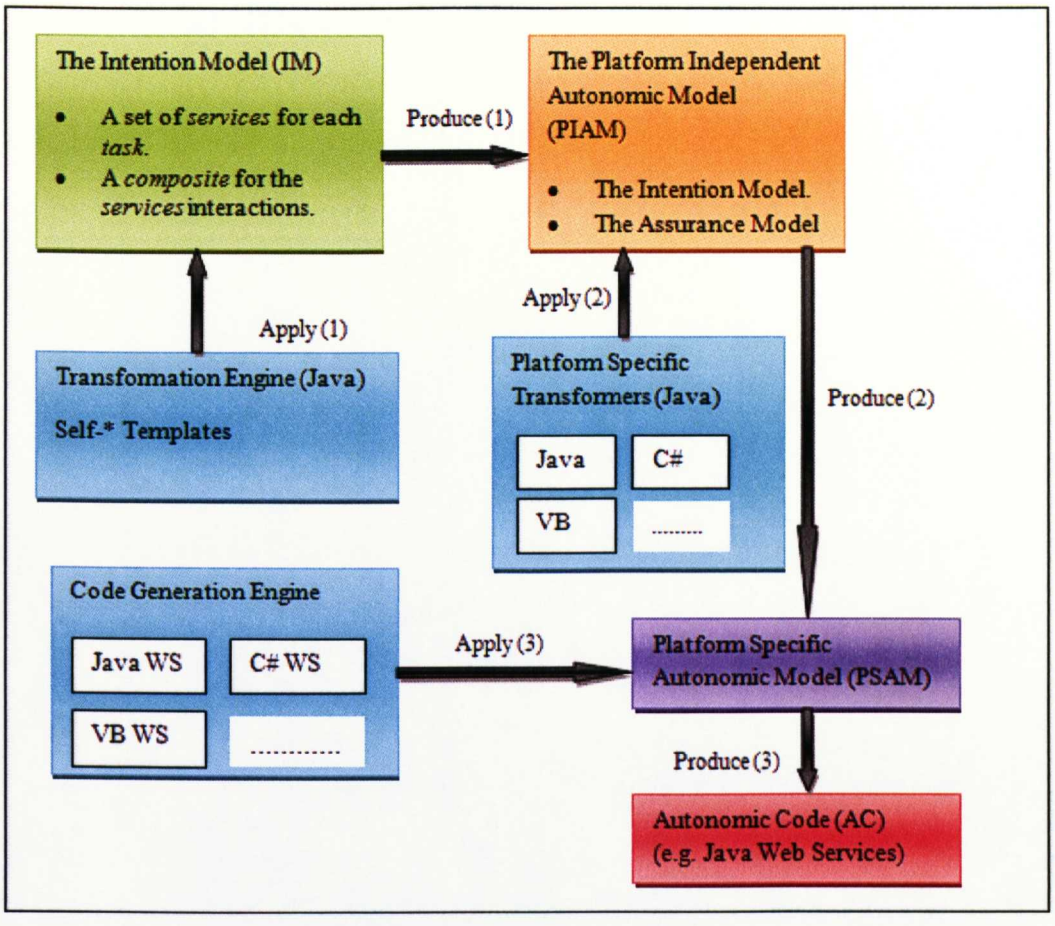

Figure 5.1: A simplified diagram for proposed design method lifecycle.

5.1.1 The Intention Model Capturing

Capturing the intention model which contains the system requirements takes the following steps:

- The functional system requirements are captured and presented in terms of tasks. *A task is a very high level goal that has to be addressed in order to address the overall system requirements.* A use case diagram is used in this stage. These tasks belong to a specific domain such as the universities, healthcare, manufacturing, etc. Figure 5.2 shows this relationship. These tasks are defined at very high level in a way that can be broken down and expressed in terms of services. *A service is an abstraction of a software or hardware entity that has a role to play in addressing the task goal.* In our terminology, tasks always start with *verbs* while *nouns* are used to represent services. For example, *book flight* is a task while *flight selection process* is considered as one of the services to enable this task.

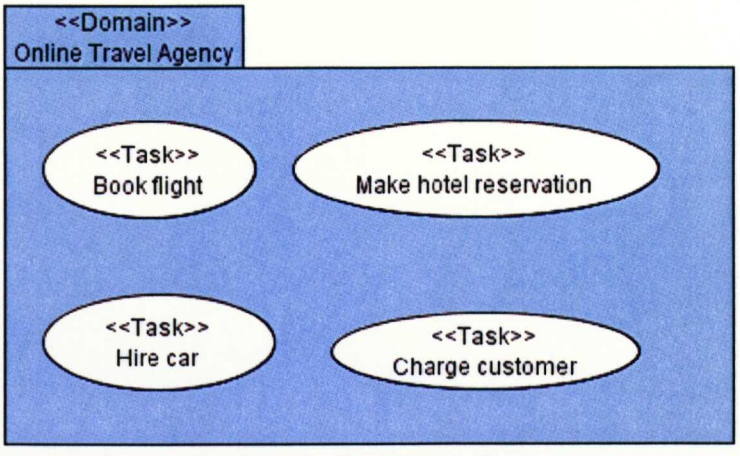

Figure 5.2: A set of tasks for the Online Travel Agency domain.

- Each task obtained in the previous step is realised by a set of services. A sequence diagram can be used to achieve such services. This process is repeated for each single task. Figure 5.3 presents this step.

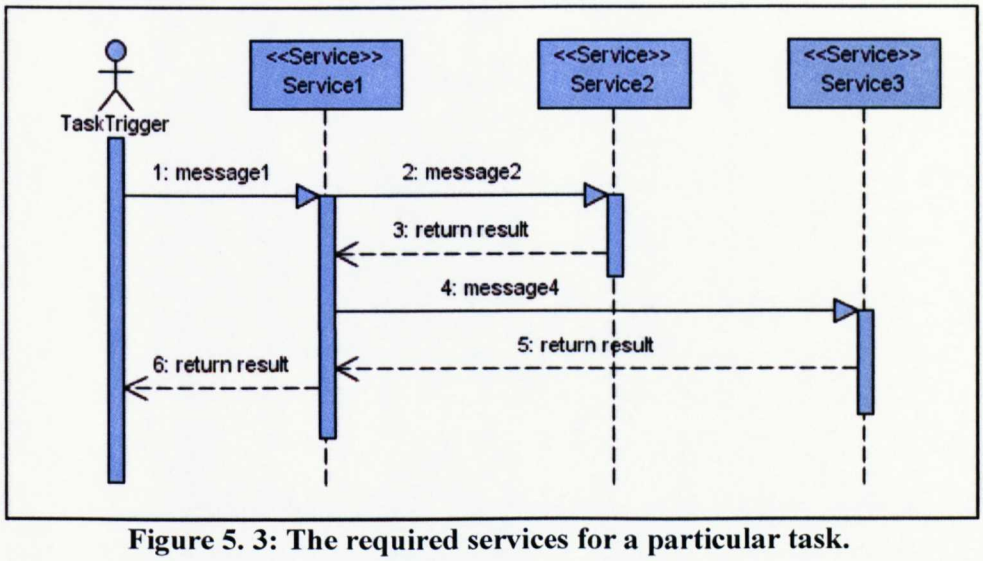

Figure 5. 3: The required services for a particular task.

- Then, each task and its services will be presented in terms of an XML file. The latter represents the intention model which contains the core services. In this stage, there is no kind of decision or terminology about the target platform. The XML file obtained here should be complied with a particular metamodel which contains some rules and specific structure that control and specify the allowed XML elements to be included in such an XML file. This is achieved here using an XML schema which is depicted in Figure 5.4 using a UML class diagram.

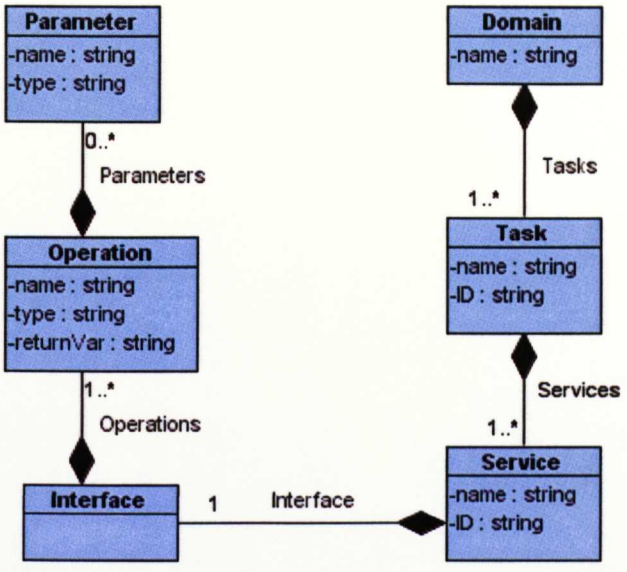

Figure 5. 4: Metamodel for XML based Intention Model.

- Also, out of the sequence diagrams presented above, the service interactions for each task can be extracted. Such a task interaction is referred to as a *composite,* thus, a set of composites can be obtained. Each *composite* defines the interactions between participating services as well as the sequence in which these interactions must be executed. These composites are saved in a separate XML file which takes the following form: *domainName_composite.xml*. Each composite element contains a set of *Interaction* elements and in turn each *Interaction* element defines the two participating services. In our terminology we call these services the *calling party* and *called party*. Figure 5.5 shows the XML schema structure for the composite file using the UML class diagram. The set of services and the composite defining their interactions and the order, in which these interactions must be fired, form the concept of the Business Process (BP).

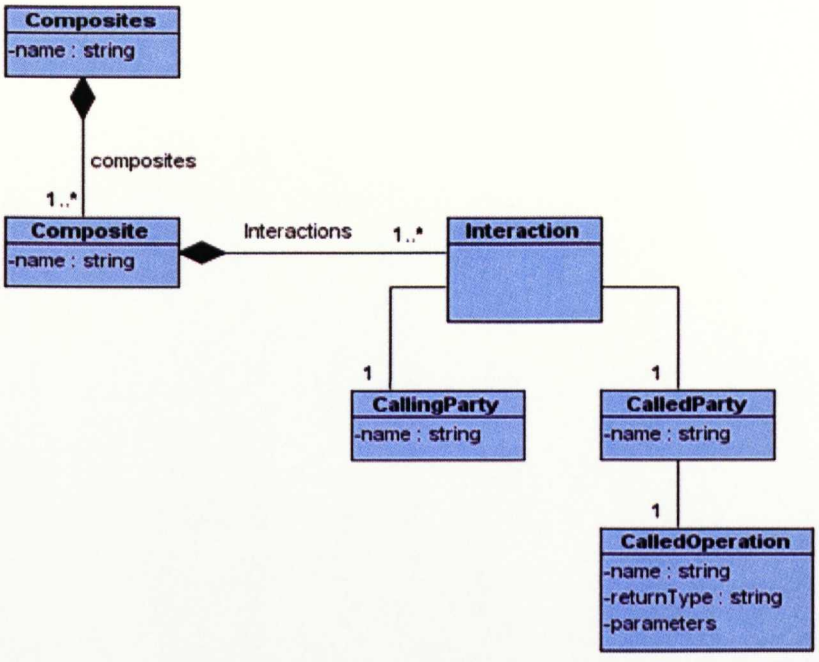

Figure 5.5: The class diagram for XML schema of the composite file.

5.1.2 The Platform Independent Autonomic Model Capturing

The process of transforming the Intention model obtained in the previous stage to an autonomic model may take the following stages:

- Each service belongs to a specific task will have the option of having the required elements and components for realising autonomicity. These elements are encapsulated in a profile in the form of XML file. See Figure 5.6 for the proposed UML self-healing profile.
- The autonomic elements introduced in this profile, the monitor for instance, are injected into the core system using a Java file (*AutonomicProfile.java)* containing the required transformation rules (templates). These elements or components are encapsulated inside an element called *Assurance*. The assurance element here is responsible for ensuring the functionality and integrity of the Intention model obtained in the previous stage. Here, the autonomic functionality is achieved using the Service–Monitor-Controller style (see Section 5.3.2). However, in order to achieve it using the other proposed style (design by contract) (see Section 5.3.1), the

ContractProfile.java file is applied to the intention model. Refer to Section 6.2 for more detailed description of the autonomic transformer.

The output of this step is a Platform Independent Autonomic Model (PIAM).

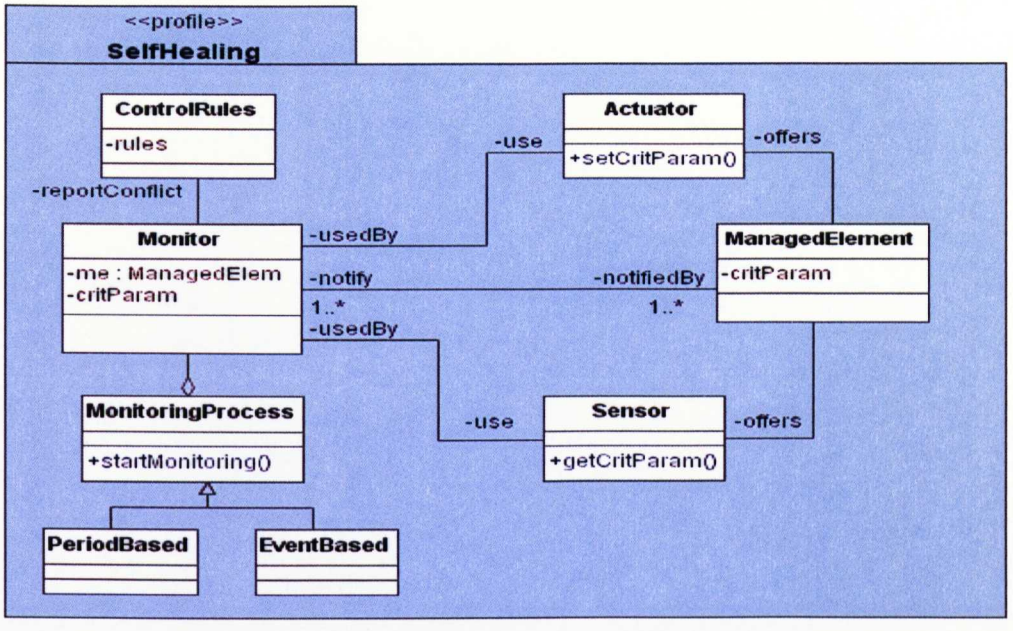

Figure 5. 6: Simplified UML profile for self healing systems.

5.1.3 The Platform Specific Autonomic Model Capturing

The process of obtaining the platform specific autonomic model can be explained as follows:

- To target a specific platform such as Java or C#, a model transformer is applied to the model obtained in the previous stage. This Java based transformer is responsible for injecting the right and necessary concepts and terminology of a particular platform. The output of this step is a Platform Specific Autonomic Model (XML format). Figure 5.7 shows a simplified form of the Java file that encapsulates the transformation rules (templates) required for targeting Java platform, *JavaPlatform.java*. Notice that this Java class implements an interface called *Platform*. The latter contains the necessary methods that each new platform, such as C#, should implement. Please refer to Section 6.3 for more details on the platform metadata transformer.

```
public class JavaPlatform  implements Platform{
private HashMap<String, String> abst2Java = new HashMap<String,
String>();
```

```
    private ObjectOutputStream outStream;
    private String javaMapFile = "c:/Users/LA_Abuseta/javamap.txt";

    public JavaPlatform () {
    // add new entry to the map
    public void addEntry (String absType, String javaType) {
        Abst2Java.put (absType, javaType);
    }
//retrieve corresponding Java term to abstract term
    public String getJavaTerm (String absTerm) {
        String javaTerm = (String) abst2Java.get (absTerm);
        return javaTerm;
    }
    //retrieve the whole abstract to Java map
public HashMap getDictionary () {
        return abst2Java;
    }
// serialize the map for later retrieval
    public void saveMap() {
        try {
            outStream = new ObjectOutputStream (
                    new FileOutputStream (javaMapFile));
            outStream.writeObject (abst2Java);
        } catch (Exception excp) {
            System.out.println (excp);
        }
    }
//read the map back from external file
    public void readMap () {
        try {
            ObjectInputStream inStream = new ObjectInputStream (
                    new FileInputStream (javaMapFile));
            abst2Java = (HashMap) inStream.readObject ();
        } catch (Exception excp) {
            System.out.println (excp);
        }
    }
}
```

Figure 5. 7: Transformation rules for Java metadata.

5.1.4 The Autonomic Code Generation

Generating autonomic code is performed at the last stage of the AutoTaSC process; where the appropriate transformer is run for the autonomic code generation for a particular platform. Two Java based transformers are used here, one for generating the code for the core services and another to generate the autonomic components. To target Java Web services platform, for instance, the *JavaCodeGenerator.java* file is applied to the *JavaWebServiceTemplate.java* to generate the core Java web services. Likewise, the *AutonomicJavaGenerator.java* file is applied to the *AutonomicJavaWSTemplate.java* in

order to generate the autonomic web services. Please refer to Section 6.4 for more information about the autonomic code generator.

5.2 Design Patterns and Architectural Style Support

To design and generate clean and easy to use, extend and maintain software systems (code), the application of design patterns seems to be a must. In the case study of the On Line Travel Agency presented in this Chapter 8, a number of design patterns have been applied in order to achieve some specific design principles such as the separation of concerns and loosely coupling associations. In such a case study, the *smart proxy pattern* (Appendix D), for example, has been used to encapsulate the business logic responsible for calling and invoking a particular web service. Thus, a web service client is able to call a remote web service operation as it is on the same computer which frees the client from being bound to any specific communication protocol. Had it not been for the proxy pattern, a client would have to change the code whenever the web service undergoes changes regarding the communication protocol. Also, the web service functionality can be extended via adding some operations to the proxy service interface. For example, to control and manage a web service, a sensor and actuator components which comprise the control interface should be provided by the web service. Since these components do not belong to the web service responsibilities, making them part of the proxy web service can be a wise and beneficial design decision. The sensor and actuator may take the form of getter and setter methods respectively. The *observer pattern* (Appendix D) is also applied here in which the monitor plays the role of the observer and the web service, in fact the proxy, takes the subject role. The monitor registers its interest of observing the state (a parameter) of the web service with the proxy and on the other hand the web service (the proxy) notifies the monitor whenever its state undergoes a change or being assigned a new value. Figure 5.8 shows the interaction model for autonomic Java web services. Also, to achieve the separation of concern design principle, the Model-View-Controller (MVC) design pattern (Appendix D) is applied in our generated Java web services code. Here, the web service invocations business logic is encapsulated in JSP files. The call or invocation is dispatched to the *web service (*via the proxy service), which represents the *Model* entity in this case. The *proxy service* business logic is encapsulated in a *Java Bean* file and called from within the JSP file instead of cluttering its code and tangle it with the JSP presentation code which is in fact regarded as a bad practice. JSP technology was developed in the first place to separate the presentation logic and business logic concerns.

The *view* entity of this design pattern is represented by one or more JSP files where the appropriate user interface or model information are shown and displayed to the system user. The *controller* role is also played by a JSP file where this file takes the responsibility of directing requests to the appropriate JSP file (*view*) and interacting with the *Model*, also via a JSP file. All of these artefacts (proxies, JSP files and web services) are automatically generated using a set of transformation templates encapsulated in Java files.

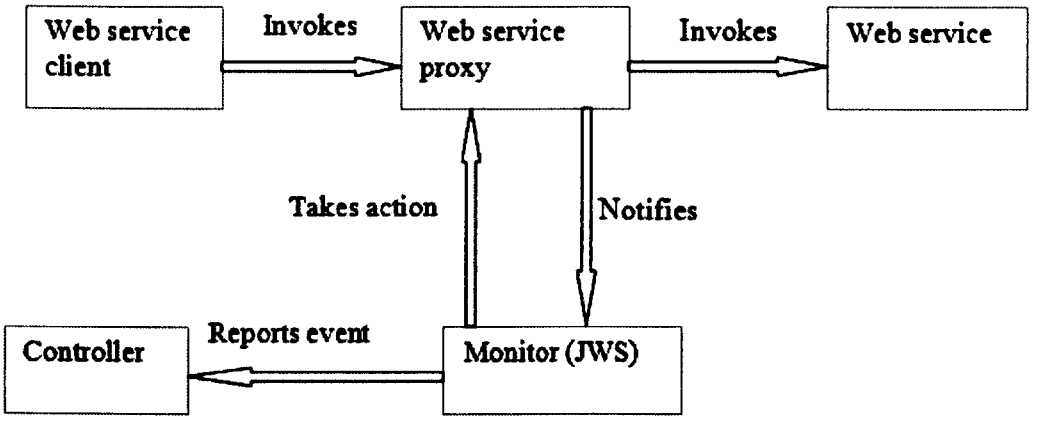

Figure 5.8: An interaction model for autonomic Java web services (proxy with embedded sensor and actuator).

5.3 Design Styles for Autonomic Capabilities Provisioning

In our approach to designing the autonomic systems we have employed two different styles to introduce and inject the autonomic capabilities, namely the *Service-Monitor-Controller (SMC)* and the *design by contract* styles.

5.3.1 The Design by Contract style

The design by contract (DBC) technique employs some concepts such as preconditions, postconditions and variants to build quality and reliable software systems. These concepts explicitly specify what a function or operation in a system requires to accomplish its task and what it guarantees or ensures to provide in return to its client. Reliability is defined as a system's ability to perform its job according to some specification (correctness) and to handle abnormal situations (robustness). This major component of quality is very important and beneficial for addressing autonomic or self-managing systems.

In our design method, we define the relationship between a particular service or component and its clients (within a particular business process or task) in an XML element called *contract.* Such a contract contains three children (in XML terminology) called *client*, *require* and *ensure*. Descriptions of these elements are presented as follows:

- Client: this element represents a service client in a particular task or business process and may take the form of another service or human user interacting with or triggering the business process.
- Require: this element encapsulates the conditions and the thresholds that must not be violated for a service to perform and accomplish its job as expected and desired.
- Ensure: As long as the service client complies with the rule set in the require element, the service will always guarantees providing what is set in this XML element.

Both XML elements *require* and *ensure*, include three similar children namely *parameter, operator and expression.* Figure 5.9 shows the contract element in the Online Travel Agency domain encapsulated in XML file.

```
<Domain name="OnlineTravelAgency">
- <Task name="BookFlight" id="1">
  - <Service name="FlightSelectionProcess" id="1">
    - <Interface>
      - <operation name="getAvailableFlights" returnedVar="avalableFlights" returnType="list">
        - <params>
            <param name="sourceAirport" type="String" />
            <param name="destinationAirport" type="String" />
            <param name="dateOfTravel" type="Date" />
          </params>
        - <contract>
            <client>any</client>
          - <require>
              <parameter>dateOfTravel</parameter>
              <operator>GreaterThan</operator>
              <expression>TodayDate</expression>
            </require>
          - <ensure>
              <parameter>flightNo</parameter>
              <operator>not</operator>
              <expression>null</expression>
            </ensure>
          </contract>
        </operation>
      - <operation name="reserveSeatOnFlight" returnedVar="done" returnType="boolean">
        - <params>
            <param name="fName" type="String" />
            <param name="lName" type="String" />
            <param name="flightNo" type="String" />
          </params>
        </operation>
      </Interface>
    - <Require>
      - <Service name="PaymentCardValidatorr">
          <operation name="verifyPaymentCard" />
        </Service>
      </Require>
    </Service>
```

Figure 5. 9: The Contract Element defined in an XML file.

5.3.2 The Service-Monitor-Controller Style

In this design style, three fundamental components can be identified, namely the *service*, *monitor* and *controller*. Figure 5.10 depicts these components as well as the communication channels that show the control flow between them. What follows is a description of these components and the messages exchanged between them:

- *Service*: This component contains the code that is responsible for addressing one specific functional requirement of the system under study. Within the context of autonomic or self-managing systems, this component is referred to as the managed element and such management can be achieved via monitoring its state and behaviour against some predefined thresholds or conditions. The service notifies interested parties in its state by sending a signals or message that indicates some event has happened, a variable state change for example.

- *Monitor:* the process of monitoring a particular service is accomplished by this component. Upon receiving an event notification from the service, the monitor starts reading the value of the monitored variable. The obtained value which is often obtained via a *sensor* is compared to a predefined value or threshold. In case of violating the threshold or the condition set, a conflict signal is raised and an event notification is sent to the controller.

- *Controller:* this is responsible for selecting and issuing the corrective actions that are necessary when a conflict event is received from the monitor. The controller usually makes such action selection based on a set of situations or events and attached corrective actions. Whenever the event sent by the monitor matches a situation in the controller, the corresponding action is taken. Such action is performed via an interface offered by an *actuator* which is able to access and adapt the target service variable.

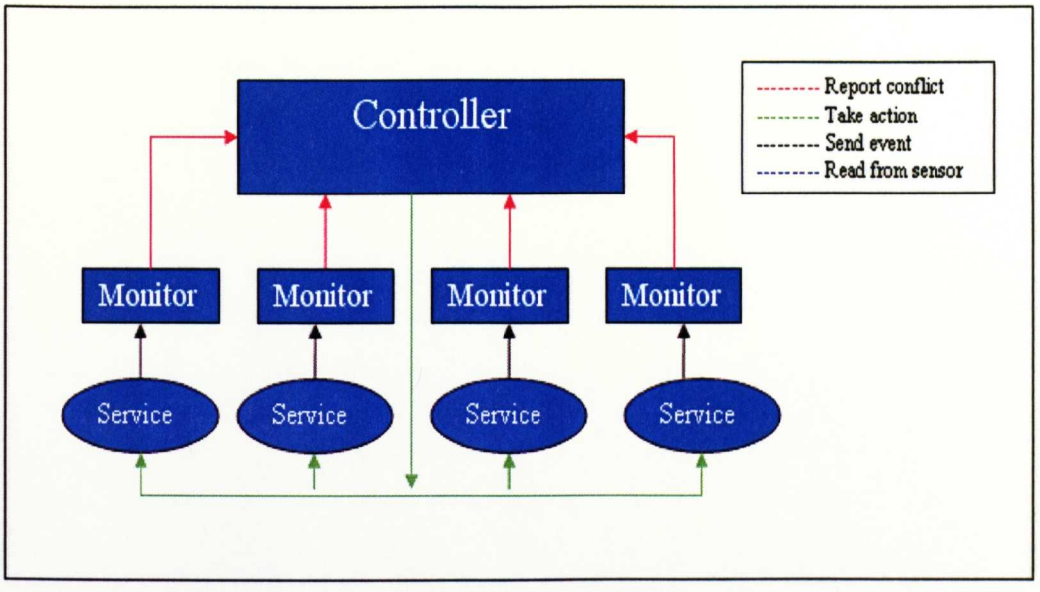

Figure 5.10: The Service-Monitor-Controller (SMC) style for autonomic systems.

5.4 Summary

In this chapter, we have introduced in detail our design method and approach to designing autonomic systems. In particular, we have presented the design process lifecycle as well as some justifiable design decisions and technology choices. Since our method is complied with the MDD paradigm, the design process lifecycle is presented in terms of the stages and models that exist and form such a paradigm. These stages or models include the *Platform Independent Model* (PIM), *Platform Specific Model* (PSM) and the *code*. However, we have proposed four stages due to the non-functional requirements, the autonomic capabilities in this case, that are injected to the core system. Such stages or models include the Intention Model (IM), Platform Independent Autonomic Model (PIAM), Platform Specific Autonomic Model (PSAM), and the Autonomic Code (AC) generation. The transformation rules needed for obtaining one model from another throughout the design process lifecycle are also presented and shown in each stage. These transformers are encapsulated in Java files and applied to the different XML based models.

Also, some design patterns have been adopted and implemented to introduce and inject the autonomic or self-managing capabilities. The proxy design pattern, for instance, were adopted and applied to introduce some kind of loosely coupling relationship between the monitor service and the monitored or managed service. In addition, the observer design pattern is used here to again draw or establish a relationship between the monitor and

monitored service. The monitor service here takes on the role of the observer while the monitored service plays the role of the subject.

CHAPTER 6

MODEL TRANSFORMATION FRAMEWORK

This chapter presents the model transformation framework developed to support the proposed AutoTaSC design method.

6.1 Model Transformation Framework Components

The proposed model transformation framework comprises of a set of transformers described as follows:

- The autonomic transformer: This transformer is responsible for adding and injecting the autonomic capabilities into the abstract model residing at the highest stage (PIM). Such autonomic capabilities are provided by autonomic components such as the monitor, sensor, actuator and policy.
- The Platform metadata injector transformer: This transformer contains the artifacts and necessary processes to add the required terminology and data types to target a specific platform, such as Web Services, EJB, C#, etc.
- The Code generator transformer: This transformer produces the final and workable autonomic code which takes the form of components or services (SOA) that work together and exhibit the autonomic capabilities injected in earlier stages. Examples include Web Services, EJB, CORBA, etc. To demonstrate and show the feasibility of our design method, we introduce here a set of case studies, namely the *Online Travel Agency* and *Pet Store* applications. Here, we show via these concrete examples the required XML and Java files that involve in the lifecycle of the autonomic systems design according to our method. Also, the generated autonomic skeletons of code are shown here.

The subsequent sections describe in detail these framework components.

6.2 The Autonomic Transformer

As stated earlier in Chapter 5, there are two deign styles adopted in the proposed design method (AutoTaSC) for introducing the autonomic capabilities to the intention model,

namely the Service-Monitor-Controller and Design by Contract. An autonomic system designer here can adopt one of these two styles at the first stage of the transformation framework. The two following subsections are dedicated to describe the autonomic capabilities injection process using these two design styles.

6.2.1 The Service-Monitor-Controller Based Transformer

This model transformer contains the transformation rules necessary to introduce the autonomic capabilities (encapsulated in the assurance element as described in Section 5.1.2) into the intention (core) model. Thus, this Java based model transformer is the first to apply in our proposed development process lifecycle. Prior to applying the transformation rules containing in this transformer, the critical parameters to be monitored should be defined as well as the policies, residing at the controller component described in Section 5.3.2, that control and drive the behaviour of these parameters. This process is described in Figure 6.1.

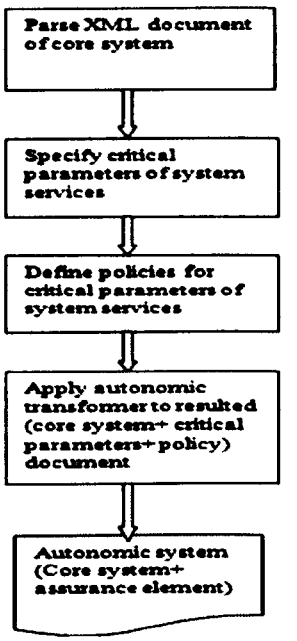

Figure 6.1: Application of autonomic transformer to core system.

The user interface for performing the two crucial steps, specifying the critical parameters and policies that control them are shown in Figure 6.2. Listings 6.1 and 6.2 below show the autonomic transformer components that contain the Java code responsible for defining the critical parameters for the monitor as well as the policy that controls the behaviour of these parameters. The process of applying the transformation rules (transformer) to the abstract model is carried out via the Java DOM transformer where

the result document is saved into an XML file takes the form of *domainName_autonomic_controllerStyle.xml*. The latter process is shown below in Listing 6.3.

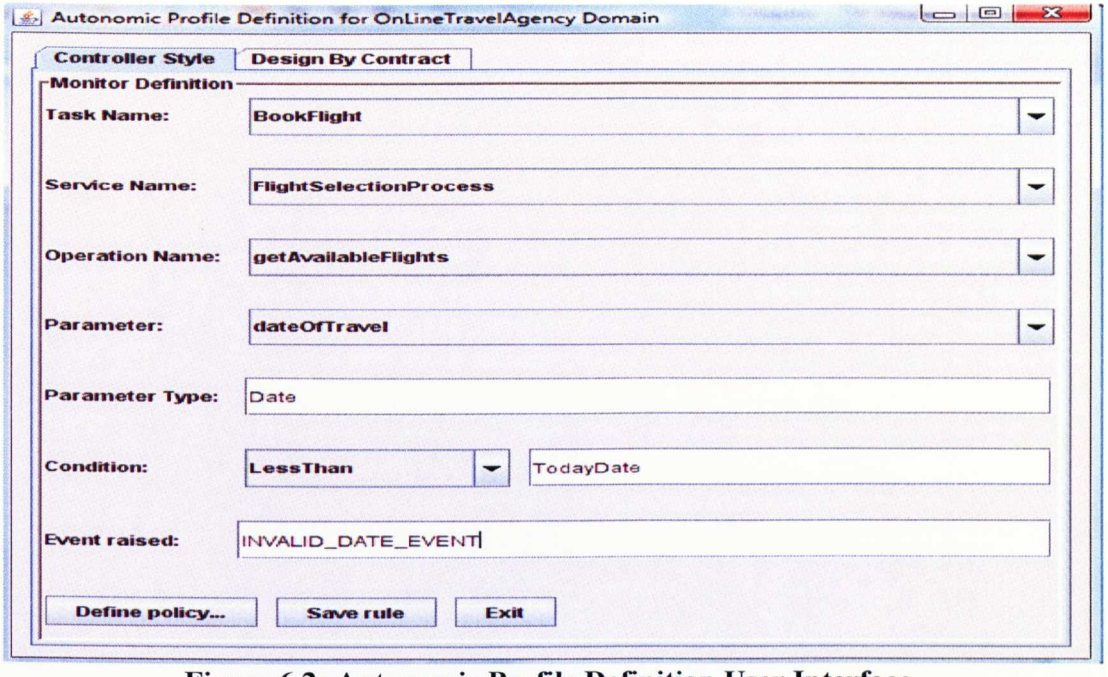

Figure 6.2: Autonomic Profile Definition User Interface.

Listing 6.1: Monitor definition for critical parameters

```
private void saveMonitor() {
task =
(Element)xmlDoc.getElementsByTagName("Task").item(taskList.getSelected
Index());
        service =
(Element)task.getElementsByTagName("Service").item(serviceList.getSele
ctedIndex());
        Element monitor = xmlDoc.createElement("Monitor");
        monitor.setAttribute("name",
"Monitor_"+service.getAttribute("name"));
        Element operation = xmlDoc.createElement("MonitorOperation");
        operation.setAttribute("name", "startMonitoring");
        operation.setAttribute("ReturnType", "void");
        monitor.appendChild(operation);
        Element moniVar = xmlDoc.createElement("MonitoredVar");
        moniVar.setAttribute("name",
paramList.getSelectedItem().toString());
        moniVar.setAttribute("type", typeText.getText());
        moniVar.setAttribute("Event", eventText.getText());
        Element thrushold = xmlDoc.createElement("Thrushold");
        thrushold.setAttribute("operator",
operator.getSelectedItem().toString());
        thrushold.setAttribute("value",conditionText.getText() );
```

```
        monitor.appendChild(moniVar);
        monitor.appendChild(thrushold);
        service.appendChild(monitor);
        xmlDoc.normalize();
    }
```

Listing 6.2: Policy Definition for critical parameters

```
private void savePolicy(){
        Element policy = xmlDoc.createElement ("Policy");
        Element event = xmlDoc.createElement("Event");
        event.setAttribute ("name", "event");
        Element pre = xmlDoc.createElement("PreCondition");
        pre.setAttribute ("value", preText.getText());
        Element post = xmlDoc.createElement("PostCondition");
        post.setAttribute ("value", postText.getText());
        Element action = xmlDoc.createElement("CorrectiveAction");
        action.setAttribute ("value", actionText.getText());
        policy.appendChild (event);
        policy.appendChild(pre);
        policy.appendChild(post);
        policy.appendChild(action);
        }
```

Listing 6.3: Model Transformation process

```
private void writeToXmlFile() {
        try {
            TransformerFactory tFactory =
TransformerFactory.newInstance();
            Transformer transformer = tFactory.newTransformer();
            Source source = new DOMSource(xmlDoc);
            Result dest = new StreamResult(new File(xmlOut));
            transformer.transform(source, dest);
        } catch (Exception ex) {
            ex.getMessage();
        }
    }
```

6.2.2 The Design by Contract Based Transformer

In such a design style, the autonomic or self management capabilities are part of the service interface. In particular, the '*require* 'and '*ensure'* elements of the contract are considered as part of the service operation. The system designer is provided with the necessary user interface to define the *contract* element for each operation of the available services in each task of the intention model. Figure 6.2 shows the 'Design by Contract' pane for this option. The business logic responsible for attaching these contracts to the intention model is encapsulated in the Java file *ContractProfile.java.* The model obtained

from this operation is saved into a file of the format *domainName_autonomic_ContractStyle.xml*

6.3 The Platform Metadata Injector Transformer

Such a transformer contains the transformation rules responsible for adding the terminology and data types of a specific platform to the autonomic model obtained from applying the *autonomic transformer* introduced above. The outcome of applying this transformer takes the form of a platform specific autonomic model. This process is depicted in Figure 6.3.

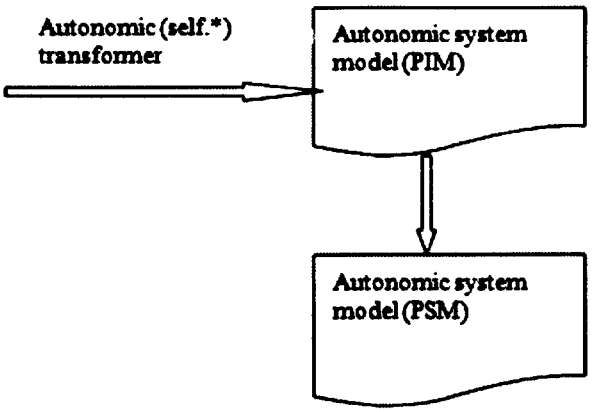

Figure 6.3: Extraction of Platform Specific Autonomic Model

This transformer includes a set of transformation templates, one for each concept or data type. A specific template is, for instance, defined for the service concept which maps it into a class or web service. There is a dedicated transformer for each platform (Java, C#, etc.) which contains the required code for adding these platforms' data types and terminologies. Such Java based transformers hold a map or dictionary with a set of entries in which one specific abstract term maps to one particular platform term. This map, which resides at a specific Java code, can be accessed and managed by the transformer. Adding, deleting and modifying entries are among those operations that can be performed on the map. Listing 6.4 shows the Java class (*JavaPlatform*) that is responsible for storing and managing the *abstract to Java* map, named as *abst2Java* in the class. Likewise, a Java class (*C_SharpPlatform*) for storing and managing the *abstract to C#* map, named as *abst2CSharp* in the class, is depicted in Listing 6.5. The last two classes, or any class for a specific platform, implement an interface called *platform* which defines the necessary interface for each abstract to specific platform map. Listing 6.6 shows the definition of this Java interface.

Listing 6.4: Abstract to Java Platform Map

```
public class JavaPlatform  implements Platform{
    private HashMap<String, String> abst2Java = new HashMap<String,
String>();
    private ObjectOutputStream outStream;
    private String javaMapFile = "c:/Users/LA_Abuseta/javamap.txt";
    public JavaPlatform () {
    }
    public void addEntry (String absType, String javaType) {
        abst2Java.put (absType, javaType);
    }
    public String getJavaTerm (String absTerm) {
        String javaTerm = (String) abst2Java.get (absTerm);
        return javaTerm;
    }
    public int getDictionaryLength () {
        return abst2Java.size ();
    }
public HashMap getDictionary () {
        return abst2Java;
    }
    public void saveMap() {
        try {
            outStream = new ObjectOutputStream (
                    new FileOutputStream (javaMapFile));
            outStream.writeObject (abst2Java);
        } catch (Exception excp) {
            System.out.println (excp);
        }
        //Close the ObjectOutputStream
        try {
            if (outStream! = null) {
                outStream.flush ();
                outStream.close ();
            }
        } catch (IOException ex) {
            ex.printStackTrace ();
        }
    }
    public void readMap () {
        try {
            ObjectInputStream inStream = new ObjectInputStream (
                    new FileInputStream (javaMapFile));
            abst2Java = (HashMap) inStream.readObject ();
        } catch (Exception excp) {
            System.out.println (excp);
        }
    }
}
```

Listing 6.5: Abstract to C# Platform Map

```
public class C_SharpPlatform implements Platform{
     private static HashMap abst2CSharp = new HashMap();
     Private String cSharpMapFile  =
EnvironmentConstants.PLATFORM_FILE_PATH+  "CSharpMap.txt";
    private ObjectOutputStream outStream;

    public C_SharpPlatform() {
    }
    public void addEntry(String absType, String csharpType) {
         abst2CSharp.put(absType, csharpType);
         saveMap();
    }
    public String getCSharpTerm (String absTerm) {

        String cSharpTerm = (String)  abst2CSharp.get(absTerm);
        return  cSharpTerm ;
    }
    public int getDictionaryLength() {
        return  abst2CSharp.size();
    }
    public HashMap getDictionary() {
        return  abst2CSharp;
    }
    public  void saveMap(){
        try{
      outStream  = new  ObjectOutputStream(
                  new FileOutputStream(cSharpMapFile));
      outStream.writeObject(abst2CSharp);
    }catch(Exception excp){
      System.out.println(excp);
    }
            try {
                if (outStream != null) {
                    outStream.flush();
                    outStream.close();
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            } } }
    public  void readMap(){
        try {
            ObjectInputStream inStream = new ObjectInputStream(
                    new FileInputStream(cSharpMapFile));
           abst2CSharp  = (HashMap) inStream.readObject();
        } catch (Exception excp) {
            System.out.println(excp);
        } } }
```

Listing 6.6 : Java Interface for abstract to platform map.

```
package platforms;
import java.util.HashMap;

public interface Platform {
     public void addEntry(String absType, String platformType);
    public String getPlatformTerm (String absTerm);
    public int getDictionaryLength();
    public HashMap getDictionary();
    public void saveMap();
    public void readMap();
}
```

The actual transformer that transforms the PIM autonomic model to the PSM autonomic model is shown below in Listing 6.7. Each transformation rules for a specific aspect of the system (tasks, services, operations, etc.) are encapsulated in a Java method.

Listing 6.7: The Abstract to Platform Transformer.

```
public class Abst2PlatformTransformer {

    Platform platform;
    Document doc;

    public Abst2PlatformTransformer(Platform p) {
        platform = p;
        platform.readMap();
    }
    public void setPIMAutonomicModel(Document model) {
        doc = model;
    }
    public void startTransformer() {

        if (doc != null) {

            Element root = doc.getDocumentElement();
            NodeList tasks = doc.getElementsByTagName("Task");
            for (int i = 0; i < tasks.getLength(); i++) {
                root.appendChild(transformTask((Element)
tasks.item(i)));
            }
            for (int i = tasks.getLength() - 1; i >= 0; i--) {
                root.removeChild(tasks.item(i));
            }
        }
    }
    public Element transformTask (Element task) {

      //Creating a corresponding task element
      Element corres2Task =
doc.createElement(platform.getPlatformTerm("Task"));
```

```
    //Retrieving the set of attributes of the task element
      NamedNodeMap attrs = task.getAttributes();

   //Appending task attributes  element to the corresponding platform
    //element after transforming attributes to paltform specific terms
          for (int i = 0; i < attrs.getLength(); i++) {

            Attr attr = (Attr) doc.importNode(attrs.item(i), true);
                corres2Task.getAttributes().setNamedItem(attr);
              }
     //Retrieving the set of services in this task
      NodeList services = task.getElementsByTagName("Service");

  /*calling Service transformation template to include platform
specific service in the platform specific task element*/

       for (int i = 0; i < services.getLength(); i++) {
       corres2Task.appendChild(transformService((Element)
services.item(i)));
              }
         return corres2Task;
   }
    public Element transformService(Element service) {
        Element corres2Service =
doc.createElement(platform.getPlatformTerm("Service"));
       NamedNodeMap attrs = service.getAttributes();

       //copying of attributes
       for (int i = 0; i < attrs.getLength(); i++) {

           Attr attr = (Attr) doc.importNode(attrs.item(i), true);
           corres2Service.getAttributes().setNamedItem(attr);
       }

       NodeList operations =
service.getElementsByTagName("operation");

       //calling operation transformation template
       for (int i = 0; i < operations.getLength(); i++) {

           corres2Service.appendChild(transformOperation((Element)
operations.item(i)));
       }
      return corres2Service;
   }
```

Since the transformer code is quite long, only the Java methods for transforming the *task* and *service* elements are shown above.

6.4 The Code Generation Transformer

Since the code generation process here adopts the template approach, three fundamental components, namely the *data model, template* and *code generator,* should be available. The data model here takes the form of the PSM autonomic model, be it in Java, C# or another Language, which is the output of the transformer presented in the previous section. The template component contains a set of templates, one for each targeted platform. A template is responsible for formatting the data model into the output code and such a template contains references to necessary entities, which belong to the data model. The code generator performs the transformation process and produces the desirable output code. Such a component takes as input the data model and one template for a specific platform. Figure 6.4 shows this template based code generation process. The code generator should, prior to replacing the references contained in the template with the corresponding data, have some mechanism to import the data from the data model.

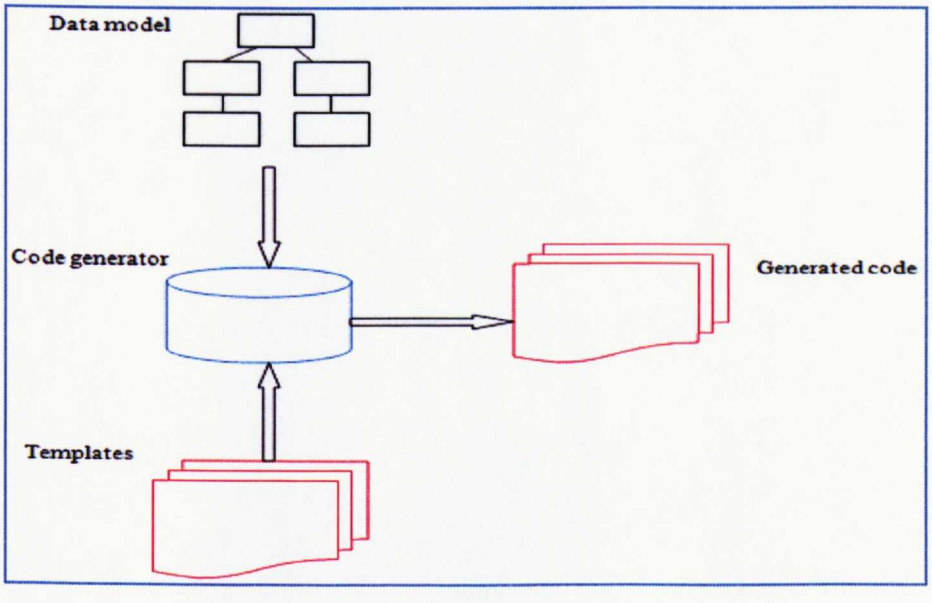

Figure 6.4: The Template based code generation process.

The description of code generator is divided into two primary aspects, one aspect for generating code for core services, such as Java web services, and another for generating the autonomic services, such as the monitor service and policy.

6.4.1 Code generator for core services

Here, the data model elements are encapsulated in Java classes. Data model elements in Java, for example, include *JavaSystem*, *package*, *class*, *method*, *parameter*, etc. The

model data importer is then used to populate the Java classes with the parsed XML based data model. Listing 6.8 to 6.11 show the Java classes for the data model elements that serve as containers for the retrieved data. As it can be seen, these classes are specified for Java platform. Other platforms, however, can be targeted in the same manner. Also, the model data importer, the *JavaDataImporter* class in this case, is depicted in Listing 6.12.

Listing 6.8: A set of packages for Java application

```
public class JavaSystem {

   private String name;
   private ArrayList <PackageDescriptor> packages = new ArrayList
<PackageDescriptor>();

    public JavaSystem() {
    }
    public String getName (){
        return this.name;
    }

    public void setName (String name){
        this.name = name;
    }
    public void addPackage (PackageDescriptor packageDesc){
        packages.add (packageDesc);
    }

   public ArrayList getPackages(){
        return packages;
    }
}
```

Listing 6.9: The package descriptor Java class

```
public class PackageDescriptor {

    private String name;
    private ArrayList <ClassDescriptor> classes = new ArrayList
<ClassDescriptor>();

    public PackageDescriptor() {
    }

   // Retrieval of package name
    public String getName(){
        return name;
    }
   // set the package name using the retrieved name attribute
    public void setName(String name){
        this.name = name;
    }
    // Adding retrieved class to package
    public void addClass(ClassDescriptor classDesc){
        classes.add(classDesc);
    }
// Retrieval of classes of specific package
```

```
    public ArrayList getClasses(){
        return classes;
    }
}
```

Listing 6.10: The class descriptor Java class.

```
public class ClassDescriptor {
    private String name;
    private ArrayList <MethodDescriptor> methods = new ArrayList
<MethodDescriptor> ();

    public ClassDescriptor() {
    }
  // Retrieval of the classDescriptor name
    public String getName(){
        return this.name;
    }
    // set the classDescriptor name using the retrieved name attribute
value
    public void setName (String name){
        this.name = name;
    }
    // Add retrieved method to specific classDescriptor
    public void addMethod(MethodDescriptor method){
        methods.add(method);
    }
   // Retrieval of methods belonging to specific classDescriptor
    public ArrayList getMethods(){
        return methods;
    }
}
```

Listing 6.11: The Method descriptor Java class

```
public class MethodDescriptor {
    private String name;
    private String returnType;
    private String returnVariable;
    private ArrayList<ParameterDescriptor> parameters = new
ArrayList<ParameterDescriptor>();

    public MethodDescriptor() {
    }
    public String getName() {

        return this.name;
    }
    public void setName(String name) {

        this.name = name;
    }
    public String getReturnType() {

        return returnType;
    }
```

```
    public void setReturnType (String reType) {
        returnType = reType;
    }
    public String getReturnVariable() {
        return returnVariable;
    }
    public void setReturnVariable(String reVariable) {
        returnVariable = reVariable;
    }
    public void addParameter (ParameterDescriptor parameter) {
           parameters.add(parameter);
    }
    public ArrayList getParameters() {
        return parameters;
    }
}
```

Listing 6.12: The Java Data Importer for retrieving data from data model.

```
public class JavaDataImporter implements DataImporter {
    private JavaSystem js;
    private PackageDescriptor pd;
    private ClassDescriptor cd;
    private MethodDescriptor md;
    private ParameterDescriptor pad;
    private Document dataSource;

    public JavaDataImporter() {
    }
    public void setDataSource (Document doc){
        dataSource = doc;
    }
    public void startImporting(){

       String systemName =
dataSource.getDocumentElement().getAttribute("name");
        js = new JavaSystem();
        js.setName (systemName);

        NodeList packages = dataSource.getElementsByTagName("package");
        for (int i = 0; i < packages.getLength(); i++) {
           Element packageName = (Element) packages.item(i);
            js.addPackage (getPackage (packageName));
        }
        for (int i = 0; i < js.getPackages().size(); i++) {
           pd = (PackageDescriptor)js.getPackages().get(i);
        }
    }
    public PackageDescriptor getPackage(Element pa){
        pd = new PackageDescriptor();
        pd.setName (pa.getAttribute("name"));
        return pd;
```

```
    }
public ClassDescriptor getClass(Element cl){
        cd = new ClassDescriptor();
        cd.setName(cl.getAttribute("name"));
        NodeList methods = cl.getElementsByTagName("Method");

        for (int i = 0; i < methods.getLength(); i++) {

            Element currentMethod = (Element) methods.item(i);
            cd.addMethod(getMethod (currentMethod));
        }
        return cd;
    }
}
```

Only the Java methods responsible for importing the package and class elements data are shown here. The above shown importer implements an interface called DataImporter which contains two methods, *setDataSource and startImporting*, to be implemented by every platform data importer. Such an interface is shown in Listing 6.13.

Listing 6. 13: Data Importer Interface.

```
public interface DataImporter {

    public void startImporting();
    public void setDataSource(Document doc);

}
```

As for the template component for the Java platform, a Java class, referred to as *JavaClassTemplate.java* is used to represent it. Such a template takes the form of a string output it into a string container like the *StringBuffer* or *BuilderBuffer*. The template here spreads over a number of Java methods, where each method is responsible for producing the part belonging to one specific element, such as the package, class, method and parameter. References to these elements are included in the template, which are replaced at runtime with real values. Setters methods in this class are used to set these write only variables. Listing 6.14 introduces the Java class of the template, called JavaClassTemplate here, which is served as input to the Java code generator.

Listing 6.14: The JavaClassTemplate used by the generator.

```
public class JavaClassTemplate {

    private String packageName;
    private String className;
    private String methodName;
    private String returnVariable;
```

```
    private String returnType;
    private HashMap<String, String> parameters = new HashMap<String,
String>();
    private StringBuffer javaClassTemplate = new StringBuffer();
    private JavaInitialValues jiv ;
    public JavaClassTemplate() {
private void insertEmptyLines(int lines) {

        for (int i = 0; i < lines; i++) {
            javaClassTemplate.append("\n");
        }
    }
    private void insertSpace(int spaces){
        for (int i = 0; i < spaces; i++) {

            javaClassTemplate.append(" ");
        }
    }
    public void createTemplate(){
        insertEmptyLines(2);
        javaClassTemplate.append("package "+packageName+";");
        insertEmptyLines(2);
        javaClassTemplate.append("public class " +className +" {");
        insertEmptyLines(2);
        insertSpace(4);
        javaClassTemplate.append("private "+returnType+"
"+returnVariable+";");
        insertEmptyLines(2);
        createMethod();
        javaClassTemplate.append("}");
    }
    public void createMethod() {
        insertSpace(4);
        javaClassTemplate.append("public " + returnType + " " +
methodName + getParameters()+"{");
        insertEmptyLines(3);
        insertSpace(6);
        javaClassTemplate.append("return " + returnVariable + ";");
        insertEmptyLines(2);
        insertSpace(4);
        javaClassTemplate.append("}");
        insertEmptyLines(2);
    }
    public StringBuffer getParameters(){
        StringBuffer constructedString = new  StringBuffer();
        int no_of_parameters = parameters.size();
        Set set = parameters.entrySet();
        Iterator i = set.iterator();
        //Start of parameters construction
        constructedString.append("( ");
        while (i.hasNext()) {
            Map.Entry po = (Map.Entry) i.next();
            constructedString.append((String)po.getValue());
```

```
            constructedString.append(" ");
            constructedString.append((String)po.getKey());
            no_of_parameters--;

constructedString.append(insertPossibleComma(no_of_parameters));
        }
        constructedString.append(")");
        // End of parameters construction
        return constructedString;
    }
    private String insertPossibleComma(int currentParamNo) {
        String comma = "";
        if (currentParamNo > 0) {
            comma = ",";
        }
        return comma;
    }
```

The Java code generator, which contains the business logic that applies to the Java template shown above and controls the way the generated Java files (web services in this case) look like, is illustrated in Listing 6.15.

Listing 6.15: The Java code generator class.

```
public class JavaCodeGenerator extends CodeGenerator{
    private Document xmlDoc;
    private JavaClassTemplate jct;
    private JavaDataImporter jdi;

    public JavaCodeGenerator(String autoPSM_output) {
        xmlDoc = XmlParser.getXmlDocument(autoPSM_output);
    }
    public  void importModelData(){
        jdi = new JavaDataImporter();
        jdi.setDataSource(xmlDoc);
        jdi.startImporting();
    }
    public  void startGenerator(){
         jct = new JavaClassTemplate();
        JavaSystem js = jdi.retrievePackages();
        ArrayList packages = js.getPackages();

        for (int i = 0; i < packages.size(); i++) {
            PackageDescriptor pd = (PackageDescriptor) packages.get(i);
            jct.setPackageName(pd.getName());
            jct.createPackageBody();
            generatePackageData(pd);
            jct.clearTemplateStream();
        }
    }
    private void generatePackageData(PackageDescriptor packageDesc){

        ArrayList classes = packageDesc.getClasses();

        for (int i = 0; i < classes.size(); i++) {
            ClassDescriptor cd = (ClassDescriptor) classes.get(i);
            jct.setClassName(cd.getName());
```

```
            jct.createClassBody();
            generateClassData(cd);
            jct.markClassEnd();
            jct.save();
            jct.clearTemplateStream();
            jct.createPackageBody();
        }
    }
   private void generateClassData(ClassDescriptor classDesc){
       ArrayList methods = classDesc.getMethods();
       for (int i = 0; i < methods.size(); i++) {
           MethodDescriptor md = (MethodDescriptor) methods.get(i);
           jct.setMethodName(md.getName());
           jct.setReturnType(md.getReturnType());
           jct.setReturnVariable(md.getReturnVariable());
           generateMethodData(md);
           jct.createMethodBody();
       }
   }
    private void generateMethodData(MethodDescriptor md){
        jct.clearParameters();
        ArrayList params = md.getParameters();
        for (int j = 0; j < md.getParameters().size(); j++) {
            ParameterDescriptor pad = (ParameterDescriptor)
params.get(j);
            jct.addParameter(pad.getName(), pad.getType());
            }
    }
}
```

Listing 6.16 and 6.17 show two skeletons for Java Web Services generated by the code generator introduced above.

Listing 6.16: A generated skeleton for FlightSelectionProcess Web Service.

```
package BookFlight;

import java.util.*;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService()
public class FlightSelectionProcess {

    @WebMethod()
    public List getAvailableFlights( Date dateOfTravel, String
destinationAirport, String sourceAirport ){

    List availableFlights= null;


    //Implementation code goes here...


    return availableFlights;

  }
```

```
    @WebMethod()
    public boolean reserveSeatOnFlight( String flightNo, String lName,
String fName ){

    boolean done= false;


    //Implementation code goes here...


    return done;
  }
}
```

Listing 6.17: A generated skeleton for PaymentCardValidator Web Service.

```
package BookFlight;

import java.util.*;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService()
public class PaymentCardValidator {

    @WebMethod()
    public boolean verifyPaymentCard( String cardNo, String securityNo
){

    boolean successful= false;


    //Implementation code goes here...


    return successful;
  }
}
```

As stated earlier, targeting a new platform can be achieved in the same way as it is been shown with Java platform. To target and generate Web services in C#, for instance, the concepts and components (Task, Service, operation, etc.) of the autonomic abstract model have to be converted into C# terms first. This is conducted, as illustrated in Listing, by passing the 'Platform instance' (in this case the C#) into the *Abst2PlatformTransformer.java* class. The obtained file here takes the following format: *domainName_Autonomic_CSharp.xml*. Then, prior to generating the code, the model data should be imported to the code generator, so the latter can use it to substitute the references in the template with these values. This is done by the

CSharpDataImporter.java class which is shown in Listing 6.18. The C# Web Services template and code generator are depicted in Listings 6.19 and 6.20 respectively.

Listing 6. 18: The C# Data Importer class.

```
public class CSharpDataImporter implements DataImporter {
    private Document dataSource;
    private CSharpSystem css;
    private NamespaceDescriptor nsd;
    private ClassDescriptor cd;
    private MethodDescriptor md;
    private ParameterDescriptor pad;

    public CSharpDataImporter() {
    }
    public void setDataSource(Document doc) {
        dataSource = doc;
    }
    public void startImporting() {
        String systemName =
dataSource.getDocumentElement().getAttribute("name");
        css = new CSharpSystem();
        css.setName(systemName);
        NodeList namespaces =
dataSource.getElementsByTagName("namespace");

        for (int i = 0; i < namespaces.getLength(); i++) {
            Element currentNamespace = (Element) namespaces.item(i);
            css.addNameSpace(getNamespace(currentNamespace));
        }
    }
    private  NamespaceDescriptor getNamespace(Element pa){

        nsd = new NamespaceDescriptor();
        nsd.setName(pa.getAttribute("name"));
        NodeList classes = pa.getElementsByTagName("class");

        for (int i = 0; i < classes.getLength(); i++) {
            Element currentClass = (Element)classes.item(i);
            nsd.addClass(getClass(currentClass));
        }
        return nsd;
    }
    private  ClassDescriptor getClass(Element cl){
        cd = new ClassDescriptor();
        cd.setName(cl.getAttribute("name"));
        NodeList methods = cl.getElementsByTagName("Method");

        for (int i = 0; i < methods.getLength(); i++) {

            Element currentMethod = (Element)methods.item(i);
            cd.addMethod(getMethod(currentMethod));
        }
        return cd;
    }
    private MethodDescriptor getMethod(Element method) {
        md = new MethodDescriptor ();
        md.setName(method.getAttribute("name"));
        md.setReturnType(method.getAttribute("returnType"));
        md.setReturnVariable(method.getAttribute("returnedVar"));
```

```
        NodeList params = method.getElementsByTagName("param");
        for (int i = 0; i < params.getLength(); i++) {
            Element currentParam = (Element)params.item(i);
            md.addParameter(getParameter(currentParam));
        }
        return md;
    }
    private ParameterDescriptor getParameter(Element param) {
        pad = new ParameterDescriptor();
        pad.setName(param.getAttribute("name"));
        pad.setType(param.getAttribute("type"));
        return pad;
    }
    public CSharpSystem retrieveNamespaces(){
        return css;
    }
}
```

Listing 6. 19: Java class for C# Web Services template.

```
public class CSharpWebServiceTemplate {

    private String nameSpace;
    private String className;
    private String methodName;
    private String returnVariable;
    private String returnType;
    private HashMap<String, String> parameters = new HashMap<String,
String>();
    private StringBuffer cSharpWSTemplate = new StringBuffer();
    private CSharpInitialValues cSharpiv ;

    public CSharpWebServiceTemplate() {
        cSharpiv = new CSharpInitialValues ();
        cSharpiv.readMap();
    }
    public void setNameSpace(String namespace){
        nameSpace = namespace;
    }
    public void setClassName(String clasName) {
        className = clasName;
    }
    public void setMethodName(String mthdName) {
        methodName = mthdName;
    }
    public void setReturnType(String reType) {
        returnType = reType;
    }
    public void setReturnVariable(String reVariable) {
        returnVariable = reVariable;
    }
    public void addParameter(String paramName, String paramType) {
        parameters.put(paramName, paramType);
    }
    private void getPossibleReturn() {
        if (!returnType.equals("void")) {
            cSharpWSTemplate.append("return " + returnVariable + ";");
        }
    }
    private void insertEmptyLines(int lines) {
```

```
        for (int i = 0; i < lines; i++) {
            cSharpWSTemplate.append("\n");
        }
    }
    private void insertSpace(int spaces) {
        for (int i = 0; i < spaces; i++) {
            cSharpWSTemplate.append(" ");
        }
    }
    public void markClassEnd() {
        cSharpWSTemplate.append("}");
    }
    public void createNamespaceBody() {
        addDeveloperInfo();
       cSharpWSTemplate.append("namespace " + namespace+ ";");
    }
    public void createClassBody() {
        insertEmptyLines(2);
        importStandardPackages();
        insertEmptyLines(2);
        cSharpWSTemplate.append("[WebService]");
        insertEmptyLines(1);
        cSharpWSTemplate.append("public class " + className +
":WebService {");
        insertEmptyLines(2);
    }
    public void createMethodBody() {
        insertSpace(4);
        cSharpWSTemplate.append("[WebMethod]");
        insertEmptyLines(1);
        insertSpace(4);
        cSharpWSTemplate.append("public " + returnType + " " +
methodName + getParameters() + "{");
        insertEmptyLines(2);
        insertEmptyLines(3);
        insertSpace(4);
        cSharpWSTemplate.append ("//Implementation code goes here...");
        insertEmptyLines(3);
         insertSpace(4);
        getPossibleReturn();
        insertEmptyLines(2);
        insertSpace(2);
        cSharpWSTemplate.append("}");
        insertEmptyLines(2);
    }
    public StringBuffer getParameters() {
        StringBuffer constructedString = new StringBuffer();
        int no_of_parameters = parameters.size();
        Set set = parameters.entrySet();
        Iterator i = set.iterator();
        //Start of parameters construction
        constructedString.append("( ");
        while (i.hasNext()) {
            Map.Entry po = (Map.Entry) i.next();
            constructedString.append((String) po.getValue());
            constructedString.append(" ");
            constructedString.append((String) po.getKey());
            no_of_parameters--;

constructedString.append(insertPossibleComma(no_of_parameters)+" ");
        }
```

```
        constructedString.append(")");
        // End of parameters construction
        return constructedString;
    }
    public void clearParameters() {
        parameters.clear();
    }
    public void clearTemplateStream() {
        cSharpWSTemplate.delete(0, cSharpWSTemplate.length());
    }
    private String insertPossibleComma(int currentParamNo) {
        String comma = "";
        if (currentParamNo > 0) {
            comma = ",";
        }
        return comma;
    }
    private void addDeveloperInfo() {
        insertEmptyLines(2);
        cSharpWSTemplate.append("/*****************************");
        insertEmptyLines(1);
        cSharpWSTemplate.append("*Developer: Yousef Abuseta***");
        insertEmptyLines(1);
        cSharpWSTemplate.append("*University: LJMU (2009)    *");
        insertEmptyLines(1);
        cSharpWSTemplate.append("C# WEB SERVICES");
         insertEmptyLines(1);
        cSharpWSTemplate.append("*****************************/");
        insertEmptyLines(2);
    }
    private void importStandardPackages() {
        cSharpWSTemplate.append ("<%@ WebService language=\"C#\"
class=\""+className+"\"%>");
        insertEmptyLines(2);
        cSharpWSTemplate.append("using System;");
         insertEmptyLines(1);
        cSharpWSTemplate.append("using System.Web.Services;");
    }
    }
    public void save() {
        Writer output = null;
        try {
            output = new BufferedWriter(new
FileWriter(EnvironmentConstants.CSHARP_FILES + className + ".asmx"));
            output.write(cSharpWSTemplate.toString());
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        try {
            if (output != null) {
                output.close();
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Listing 6.20 shows a generated C# web service (the customer charger) using the code generator shown above.

Listing 6.20: A Generated C# Web Service.

```
/*****************************
*Developer: Yousef Abuseta***
*University: LJMU (2009)    *
C# WEB SERVICES
*****************************/



<%@ WebService language="C#" class="CustomerCharger"%>

using System;
using System.Web.Services;

[WebService]
public class CustomerCharger:WebService {

    [WebMethod]
    public long calculateTicketCost( string flightNo ){



    //Implementation code goes here...


    return ticketCost;

  }

    [WebMethod]
    public boolean chargeCustomer( string refNumber ){



    //Implementation code goes here...


    return done;

  }

}
```

6.4.2 Code generator for Autonomic services

This code generator contains counterpart components of the core services generator described above, namely the *data importer, template* and *code generator*. The data importer functionality here is encapsulated in a Java file which takes the following format: *AutonomicPlatformDataImporter.java*. Listing 6.21 shows the autonomic data importer class for Java platform. It also implements the DataImporter interface. As seen in the Listing, the autonomic data model elements are described using two Java classes

called *MonitorDescriptor* and *CriticalParamDescriptor*. Listings 6.22 and 6.23 show these two Java classes.

Listing 6.21: Autonomic Java Data Importer.

```
public class AutonomicJavaDataImporter implements DataImporter{

    private MonitorDescriptor md;
    private CriticalParamDescriptor cpd;
    private Document dataSource;

    public void startImporting() {

      //implementation goes here

    }

    public void setDataSource(Document doc) {

        dataSource = doc;

    }
```

Listing 6.22: Monitor Descriptor Java Class.

```
public class MonitorDescriptor {

    private String monitorName;
    private ArrayList <CriticalParamDescriptor> criticalParameters =
            new ArrayList <CriticalParamDescriptor> ();

    public MonitorDescriptor() {
    }

    public void setName (String name) {

        monitorName = name;

    }
    public String getName () {

        return monitorName;
    }

    public void addCriticalParameter (CriticalParamDescriptor cpd) {

        criticalParameters.add (cpd);
    }

    public ArrayList getCriticalParameters () {
        return criticalParameters;
    }
}
```

Listing 6.23: Critical Parameters Descriptor Java Class.

```
public class CriticalParamDescriptor {

    private String name;
    private String type;
    private String event;
    private String condition;

    public CriticalParamDescriptor() {
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }
    public String getCondition() {
        return condition;
    }
    public void setCondition(String condition) {
        this.condition = condition;
    }
    public String getEvent() {
        return event;
    }
    public void setEvent(String event) {
        this.event = event;
    }
}
```

6.5 Summary

This chapter introduced the transformation framework that was proposed and developed to handle the transformation activities that occur throughout the lifecycle of the autonomic design method presented in this thesis. The developed framework is composed of three fundamental components namely, the Autonomic Transformer, Platform Metadata Injector Transformer and the Code Generation Transformer. These three transformers were introduced and shown in detail.

CHAPTER 7

MODEL SYNCHRONISATION FRAMEWORK

This chapter describes the model synchronisation framework proposed and developed in this project to solve and overcome the possible inconsistency issue raised by performing some changes or modifications on a particular model of the proposed development lifecycle process. Prior to the introduction of the proposed synchronisation framework, a classification of the modifications and changes that are required and supported in our approach is provided.

7.1 The Model Modification Concerns

Systems and applications cannot be fully and completely designed and implemented at once and stay untouched forever. Rather, systems need to be modified and changed by one of the stakeholders involved in the lifecycle of AutoTaSC process. Stakeholders include domain experts, designers, developers and programmers. The need to these changes and modifications surfaces as a response to possible and desirable user requirements changes. This section is concerned with mechanisms to enable software evolution and/or design refinement. For instance, where services or tasks can be added and injected into an existing domain – of the system under consideration. In such situation, at set scenarios of system/models modification and/or extension can be defined including:

- *Introducing (or removing) a new service or task to an existing domain*: At some time, the system might need be modified and extended by adding and injecting either a new service into a particular task or a new task into a particular domain. This process, unlike the process of introducing a new concept presented later, requires and triggers the system models synchronisation operation which is applied to only the models involved throughout the software development lifecycle. In our proposed lifecycle process, these models represent the XML files but not the schema files that define the metamodels for those models. This is because no new concept or

item has been introduced which should be defined in the metamodel of a particular model. To accomplish this task, the QVT (Query/View/Transform) approach [79] is mimicked and applied here. The system designer or developer can *query* the system under development and retrieve (*view*) the available services and tasks of a particular domain. He or she can then add and inject a new service or even a complete task (a set of services) in the right place and apply the *transformation* rules required to reflect and propagate the changes in the subsequent models.

- *Introducing (or removing) a new concept to the metamodel of the domain*: In this scenario, a new concept or metamodel element may need be added and introduced later to the system metamodel.
- *Introducing (or removing) a new operation*: In order to extend the capability of a particular service, a new functionality in the form of an operation can be introduced. Likewise, a particular operation of a service might be seen as a redundant or misplaced and thus needs to be removed by one of the stakeholders. Therefore, two main modifications can be identified here, namely adding to and removing operation from services.
- *Introducing (or removing) a new parameter*: At any time, one of the stakeholders involving in the lifecycle of the proposed development process may see that a particular parameter should be added or removed from one operation's signature or its type should have a different value.
- *Introducing (or removing) a new policy to the policy engine*: Since the systems that are designed and modelled using the approach proposed and developed in this thesis are of autonomic nature, policies and controller rules that manage and control these systems can never be static. Therefore, an appropriate interface should be put in place and offered to the stakeholders so they can modify policies (add to or remove from policy engine).

The next subsections introduce the model synchronisation framework that is proposed and developed to respond and handle the effect of the above mentioned changes.

7.2 Proposed Model Synchronisation Framework

In this section, we present the proposed framework for synchronising a set of models at different level of abstraction in an MDD based software system. This framework is based on the idea of autonomic systems where a particular component (the model in this case) is monitored and managed by an autonomic manager or controller. Since the activities taking place here are basically connected with modifying and changing the involving models by adding or removing a new service or task, the autonomic manager is better named as a change manager. What follows is a high level view of the architecture and layers that comprise the proposed and developed framework as well as the interactions between these layers.

7.2.1 A High Level View of Proposed Framework

Figure 7.1 shows a high level view of our proposed framework for the model synchronisation realisation. As it can be seen from the Figure, the framework architecture contains three fundamental layers namely: the Managed Model layer, Sensing and Actuation layer and Manager or Controller layer. These three layers are described as follows:

- **The Managed Model Layer:** This layer represents the core component of the framework proposed here which contains a set of models that involves in designing and developing a software system complying with the MDD principles. Here, such models take the form of XML documents. The decision for representing our models in an XML format was driven by the flexibility of XML documents and ease of modification. Adding, removing and editing elements can be performed very smoothly and easily. Furthermore, XML format is a standard one for representing data and thus encapsulating systems in it will offer the opportunity to express systems in a platform independent manner which is again a crucial requirement for MDD complaint systems.
- **The Sensing and Actuation Layer:** The manipulation layer contains the required APIs for querying and modifying the models residing at the model layer. This layer serves as a sensor or monitor of model change which notifies the model change manager of such events.
- **The Model Change Manager Layer:** This layer contains the software components responsible for checking for specific events (model changes)

reported by the manipulation APIs layer and dispatching corresponding responsive actions.

In our proposed framework, we adhere to the Query, View and Transformation (QVT) technique proposed by the Object Model Group (OMG) to realise and address the model synchronisation issue. The functionality and required components spread over the above three layers. Querying the model is carried out via the APIs provided by Sensing and Actuation layer as well as viewing the models to the system designer in an appropriate way. The transformation process, which is performed to propagate the changes to all related models, is conducted by the software components offered by the model change manager layer.

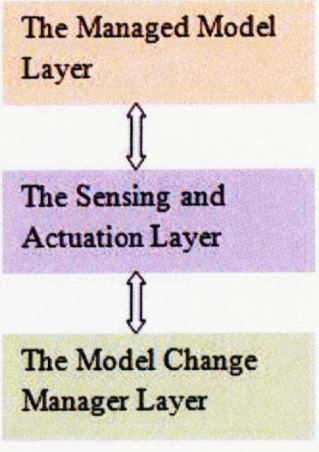

Figure 7.1: Model Synchronisation Framework Layers.

7.2.2 Detailed Synchronisation Framework

The proposed model synchronisation framework presented in this thesis is detailed and depicted in Figure 7.2. As shown in the Figure, the three layers for each stage of our MDD based design method are presented. The first layer, the change manager or controllers contains four components, one for each stage: the PIM Controller, Autonomic PIM Controller, Autonomic PSM Controller and Autonomic Code Controller. This network of controllers is responsible for synchronising the managed models residing at the lower level of the framework stack. Each controller accesses and manipulates its own managed element via the manipulation APIs in the second layer. Such a layer consists of a sensor which senses and reports any interesting events (adding or removing services or tasks) to the controller and an actuator that is used by the controller to dispatch and issue any responsive and corrective actions on the managed element.

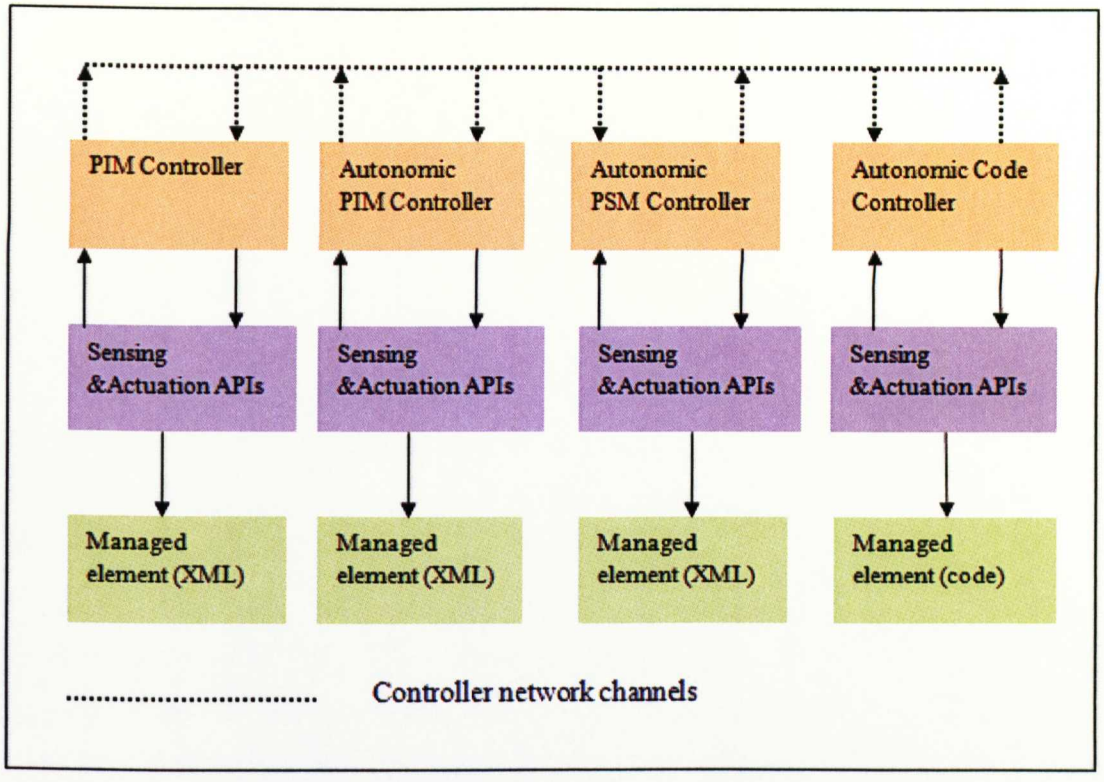

Figure 7. 2: More detailed synchronisation framework.

7.3 The observer design pattern for the controllers' network

To design and generate clean and easy to use, extend and maintain software systems (code), the application of design patterns seems to be a must. To define and establish the relationship between the controllers involving in the framework, it is believed that such a relationship is best modelled and designed using the observer design pattern. In such a design pattern, two entities can be identified: the observer which can be triggered by a change in a particular component or entity and the subject whose states are monitored and of particular importance to the observer. In our proposed framework, each controller plays both the roles of the observer and subject at the same time. A PIM Controller, for instance, is seen as an observer when it registers an interest of notification of changes occurring at the other controllers. However, it is regarded as a subject when taking the role of notifying the other controllers of any changes or modifications taking place at its own managed model. The same applies to the rest of the controllers' network. Each controller must implement two Java interfaces, namely the *ModelChangeNotifier* and *ModelChangeObserver* in order to play the roles of a subject and observer as well. Listing 7.1 shows these two interfaces with the required methods to be implemented.

Listing 7.1: The change notifier and observer interfaces.

```
public interface ModelChangeNotifier {
 public void addChangeListener (ModelChangeObserver mco);
 public void removeChangeListener (ModelChangeObserver mco);
}
public interface ModelChangeObserver {
     public void synchronise (ModelChangeNotifier mcn);
}
```

Listing 7.2 shows a snippet of Java code required to establishing the relationship between the PIM controller as a subject (and as an observer too) and the other controllers with the observer role. Notice that the PIM controller implements both the *ModelChangeNotifier* and *ModelChangeObserver* interfaces. The controller here uses the *synchronise method* when playing the observer role to be notified by other controllers (ModelChangeListener_APIM and ModelChangeListener_APSM). An appropriate action is then taken accordingly. However, it executes the *notifyObservers ()* method to notify registered observers of model changes when it takes the subject role.

Listing 7.2: Java class for the PIM controller.

```
public class ModelChangeListener_PIM implements ModelChangeNotifier,
ModelChangeObserver{
  private  ModelChangeListener_APIM mcl_APIM;
  private ModelChangeListener_APSM mcl_APSM;
  private ModelChangeListener_CODE mcl_CODE;
  private Vector <ModelChangeObserver> observers = new
Vector<ModelChangeObserver> ();
   public ModelChangeListener_PIM (ModelChangeListener _APIM  mcl_APIM)
{
         this.mcl_APIM = mcl_APIM;
         this.mcl_APIM.addChangeListener (this);
    }
    public ModelChangeListener_PIM (ModelChangeListener_ APSM mcl_APSM)
{
        this.mcl_APSM = mcl_APSM;
        this.mcl_APSM.addChangeListener (this);
    }
public void addChangeListener (ModelChangeObserver mco) {
    observers.add (mco);
}
```

```
 public void removeChangeListener (ModelChangeObserver mco)
{
          observers.remove (mco);    }
//executed when controller plays the observer role
public void synchronise (ModelChangeNotifier mcn) {
     if (mcn == mcl_APIM)  // take appropriate action
     if (mcn == mcl_APSM)  // take appropriate action
     }
//executed when controller plays the subject role
public void notifyObservers () {
     for (int i=0; i< observers.size (); i++)
        observers.get (i).synchronise (this)
}
```

For each change controller, there should be a change plan engine which is composed of a set of actions that should be executed upon receiving a notification of change from another controller in the network. Listing 7.3 shows a simplified Java class for the change plan engine for the APIM controller. Only the change plan part of injecting new service is shown.

Listing 7.3: Java code for APIM change plan engine.

```
public class ChangePlanEngine_APIM {
public void triggerPlanEngine(String domain, int taskNo, Element
service){
       domainName = domain;
       this.taskNo = taskNo;
       injectedService = service;
       System.out.println("change happened at: "+ domainName+"
Domain");
       System.out.println("at task no: "+this.taskNo);
       System.out.println("Service added:
"+injectedService.getAttribute("name"));
       updateModel_APIM();
       writeToXmlFile();
   }
    public void triggerPlanEngine(String domain, Element task){
    }
    private void updateModel_APIM() {
       xmlFile =
EnvironmentConstants.TRANSFORMER_PATH+domainName+"_autonomic.xml";
        doc = XmlParser.getXmlDocument(xmlFile);
        if(doc != null){
            currentTask =
(Element)doc.getElementsByTagName("Task").item(this.taskNo) ;
            Node temp = doc.importNode (injectedService, true);
            currentTask.appendChild(temp);
            System.out.println("Model APIM: "+
doc.getDocumentElement().getAttribute("Domain"));
        }
    }
```

7.4 Summary

This chapter was dedicated to describe the model synchronisation framework proposed and developed in this thesis. Such a framework was developed to solve and overcome the possible inconsistency issue raised by performing some changes or modifications on a particular model of the proposed development lifecycle process. A classification of the modifications and changes that are required and supported in our approach was presented. The framework architecture is composed of three fundamental layers namely: the *Managed Model layer*, *Sensing and Actuation layer* and *Manager or Controller layer*. A network of cooperating controllers is formed from the last layer of each stage or model. To define and establish the relationship between the controllers involving in the framework, we have found that such a relationship is best modelled and designed using the observer design pattern. In such a design pattern, two entities can be identified: the *observer* which can be triggered by a change in a particular component or entity and the *subject* whose states are monitored and of particular importance to the observer. In our proposed framework, each controller plays both the roles of the observer and subject at the same time. Also, a snippet of Java code of the classes and implemented interfaces has been presented to show how such a framework and controllers' relationship can be implemented.

CHAPTER 8

EVALUATION

8.1 Introduction

This chapter is dedicated to present an evaluation of the autonomic design method introduced in this thesis. The evaluation task is by no means easy to perform and conduct. This can be put down to the lack of known or clear metrics or benchmarks to evaluate the MDD technique and let alone the autonomic systems.

However, this evaluation contains both quantitative and qualitative analysis of the design method proposed here. This includes an evaluation of the design method lifecycle process in general in addition to some critical analysis to some models that have been adopted and developed in some particular stage throughout the MDD different stages. In general, the evaluation presented here evaluates the proposed method in terms of the productivity and time to market issues which conducts a comparison between MDD based systems and other software systems developed via different techniques.

The remainder of the chapter outlines the evaluation methodology, followed by qualitative evaluations of the work. Finally, the chapter concludes with a critical analysis and general discussion of the results.

8.2 Methodology

The evaluation has been designed to demonstrate the use and effect of the developed and implemented design method for the self-management systems from the qualitative perspective.

8.2.1 Objectives

The main objective of the evaluation process is to show the feasibility of using the proposed and developed design method and models and technologies used and adopted in such a method. To achieve this goal, four test example applications have been developed utilising autonomic computing services, developed frameworks and reference models. These test applications are: the Online Travel Agency, Pet Store, Intelligent Office, and Salt World which represent the evaluation from a qualitative perspective. However,

quantitative evaluation is very difficult to apply here due to the lack of any appropriate metrics that can be used to draw any comparison between MDD and non-MDD based systems.

8.2.2 Approach

The approach taken to conduct the evaluation process involves measuring and testing the proposed design method against a number of metrics. For each metric or aspect, one specific case study is used. The following Section introduces these qualitative metrics.

8.3 Qualitative evaluation

A general evaluation of the MDD based design method proposed and introduced in this thesis is described in this section. Three case studies are employed here to evaluate qualitatively the proposed design method. These applications or case studies make use of the proposed method to inject the autonomic capabilities and required models and architecture and thus be autonomic or self managed. A set of qualitative metrics is chosen here to carry out such an evaluation:

- Functionality: it concerns with the applicability and feasibility of the proposed design method. In other words, the proposed method is measured against the complete support for the tested applications throughout the lifecycle of the development process (from the requirement engineering stage all the way to producing an autonomic code).
- Generality: this metric is used to test whether the proposed design method is generic enough to accommodate any desirable platform or technology and not limited to any specific case study, programming language or platforms (software or hardware) and architectures. This is achieved here via adopting the MDD approach as the basis for designing and engineering autonomic systems. Furthermore, using the XML format to express and represent system models helps and supports the proposed method to be generic enough.
- Adaptability: this metric is used to test and evaluate the proposed design method against its ability of being adaptable and easy to modify. Such a metric may take the form of adding or removing new service or task to or from a particular domain (the system under consideration). Also adding or removing parameters to or from a specific service can be considered as a form of adaptability. The adaptability metric is provided and supported here via a

synchronisation framework based on the underlying concepts proposed in the autonomic computing model.

- Ease of aspect accommodation: this metric is used to evaluate the proposed design method against the accommodation, modelling and designing of a new aspect or non functional requirements. Here, the self organisation systems were chosen as the new aspect to be targeted.

The subsequent sections provide evaluation of the proposed design method against the qualitative metrics introduced above.

8.3.1The Online Travel Agency case study- Functionality Evaluation

To evaluate the proposed design method in terms of functionality and feasibility, a practical case study referred to as Online Travel Agency (OTA) is used here. Using this case study, we show below how the proposed method can be used to direct and guide the system designer from the early stage of the development process (Requirement engineering) and all the way to the last stage where the necessary code artifacts for autonomic systems are produced. In this particular case study, the target platform for the produced code artifacts is Java Web Services (JAX-WS).

The travel agency here is providing its services through some interfaces provided as a web site. These interfaces can be used by customers to apply and submit their requests and enquiries to make use of the offered services. The agency here is committed to providing some services to its customers related to booking flights, reserving hotels and hiring cars.

8.3.1.1 Task and service definitions

For the owner of this agency to provide such services, a set of tasks (according to our design method terminology) has to be accomplished. These tasks include *Book flight*, *Make hotel reservation* and *Hire car*. Figure 8.1 shows these fundamental tasks. Each task shown in Figure 8.1 should be addressed by a set of services. The process of generating the required services is performed using UML sequence diagrams, one for each task. Here, for example, the task *Book flight* is addressed by the following services: *flight selection process and payment card validator.* Figures 8.2, 8.3 and 8.4 show the UML sequence diagrams for the tasks "*Book flight* ", "*MakeHotelReservation*" and "*HireCar*" respectively.

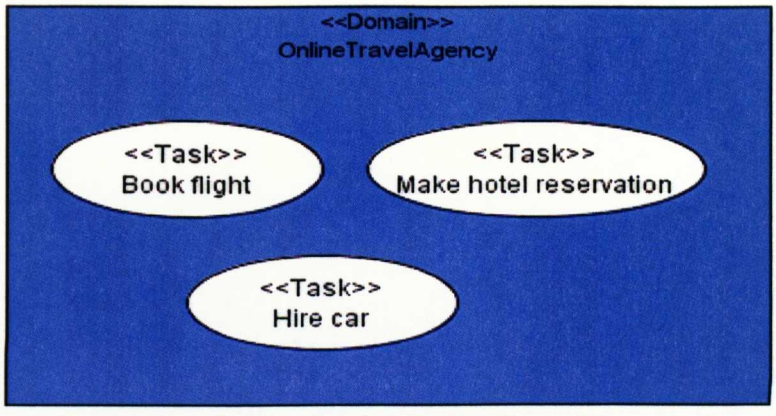

Figure 8. 1: A set of tasks for the Online Travel Agency domain.

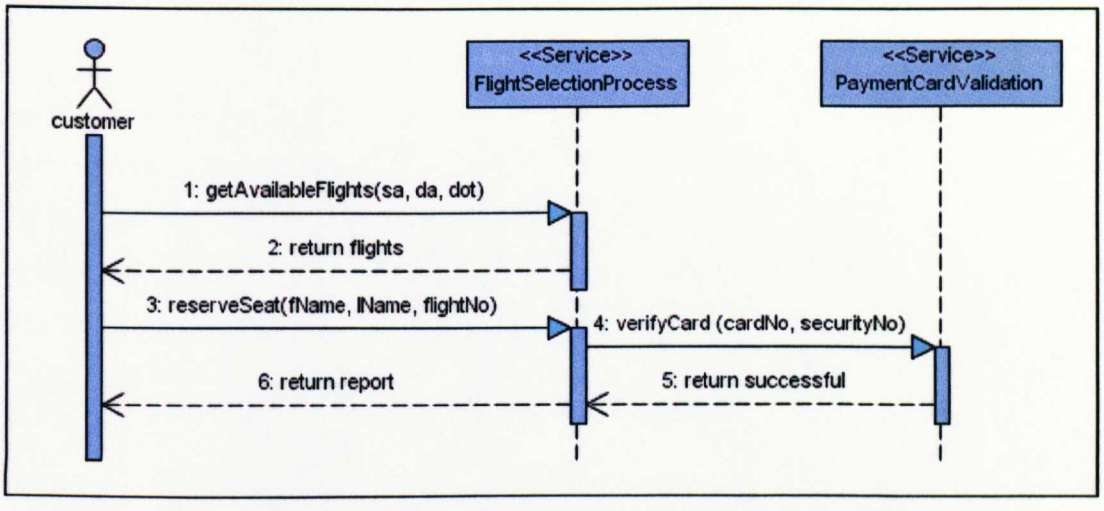

Figure 8. 2: The required services for the 'Book flight' task.

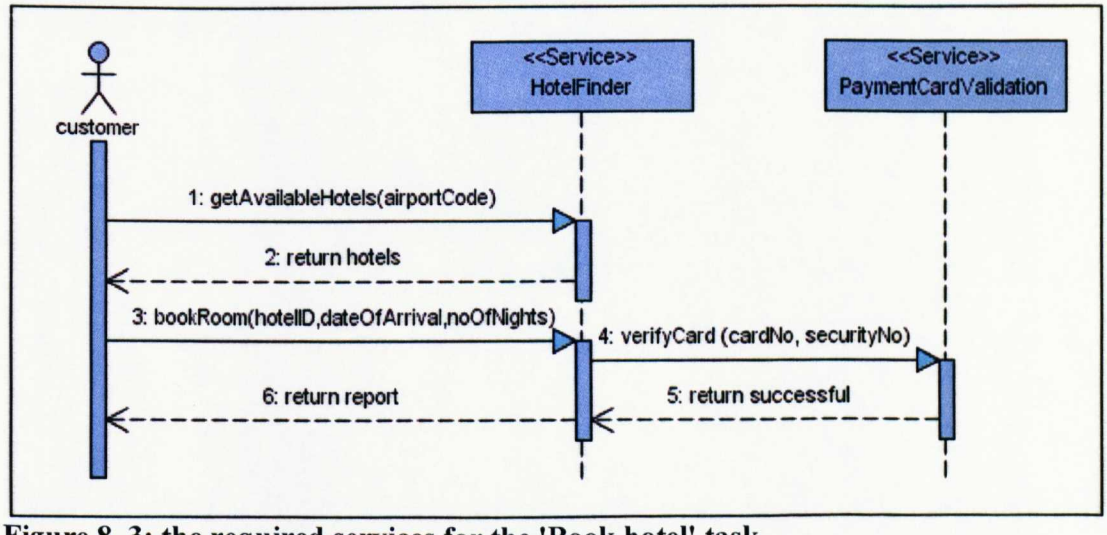

Figure 8. 3: the required services for the 'Book hotel' task.

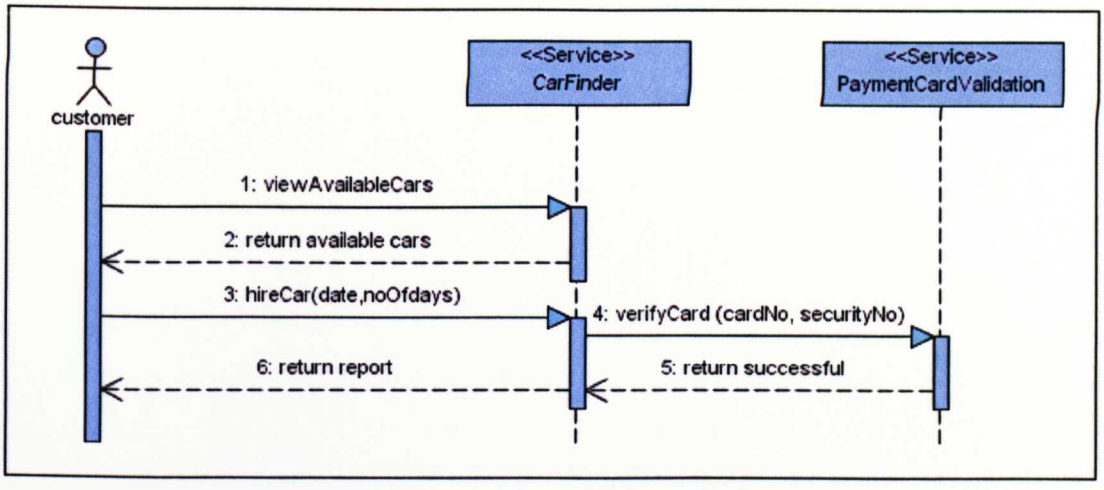

Figure 8. 4: The required services for the 'Hire car' task.

8.3.1.2 The Intention Model definition

The services involved in the online travel agency domain, which are generated in the Figures above, are saved into an XML file. This file, *OnLineTravelAgency.xml*, represents the real start point which contains the intention (abstract) model of this domain from which the other models are derived. A chunk of this file is depicted in Figure 8.5. Please refer to Section 5.1 for more information. As stated in Section 5.1.1, in addition to the above XML file, there exists a separate XML file for encapsulating the system services composites. The latter, saved as *OnLineTravelAgency_composite.xml*, is depicted in Figure 8.6. This, in fact, represents the task realisation which takes the form of service interactions. The full versions of these two files can be found in Appendix E.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Domain name="OnlineTravelAgency">
        <Service id="1" name="FlightSelectionProcess">
            <Interface>
                <operation name="getAvailableFlights" returnType="list"
returnedVar="availableFlights">
                    <params>
                        <param name="sourceAirport" type="Text"/>
                        <param name="destinationAirport" type="Text"/>
                        <param name="dateOfTravel" type="Date"/>
                    </params>
                </operation>
.....................................................................
            </Interface>
        </Service>
.....................................................................
        <Service id="1" name="HotelSelectionProcess">.........
        </Service>
</Domain>
```

Figure 8. 5: The Online Travel Agency Intention Model.

```
<?xml version="1.0" encoding="UTF-8" ?>

<composites domain="OnLineTravelAgency">

   <composite name="BookFlight">
        <Interaction>
            <callingParty name="user"/>
            <calledParty name="FlightSelectionProcess">
                <operation name="getAvailableFlights"/>
            </calledParty>
        </Interaction>
.....................................................
        <Interaction>
            <callingParty name="FlightSelectionProcess"/>
            <calledParty name="PaymentCardValidator">
                <operation name="verifyPaymentCard"/>
            </calledParty>
        </Interaction>
    </composite>

    <composite name="MakeHotelReservation">
        <Interaction>
            <callingParty name="user"/>
            <calledParty name="HotelSelectionProcess">
                <operation name="getAvailableHotels"/>
            </calledParty>
        </Interaction>
        <Interaction>
            <callingParty name="user"/>
            <calledParty name="HotelSelectionProcess">
                <operation name="reserveRoom"/>
            </calledParty>
        </Interaction>
        <Interaction>
.................................................
.............................................
        </Interaction>
    </composite>
</composites>
```

Figure 8. 6: Composite file for Online Travel Agency domain.

8.3.1.3 The Platform Independent Autonomic Model (PIAM)

The next step is to operate on this file by applying the transformation rules contained in the Java file (*AutonomicProfile.Java*). The purpose of this step is to inject the autonomic capabilities into the core system model obtained in the first stage. This process is accomplished through applying the *AutonomicProfile.Java* file to the *OnLineTravelAgency.xml* file. The result autonomic model is saved in the *OnLineTravelAgency_autonomic.xml* file. This file contains the core services of our

system as well as the Assurance service. The latter has the necessary services for ensuring the behaviour of the core system (the set of services). Such services include the *monitor and policy.* Figure 8.7, shows the user interface for specifying some *critical* parameters of the *Online Travel Agency* domain to be monitored by the monitor. Also, the policy which controls and specifies the corrective actions can be defined here.

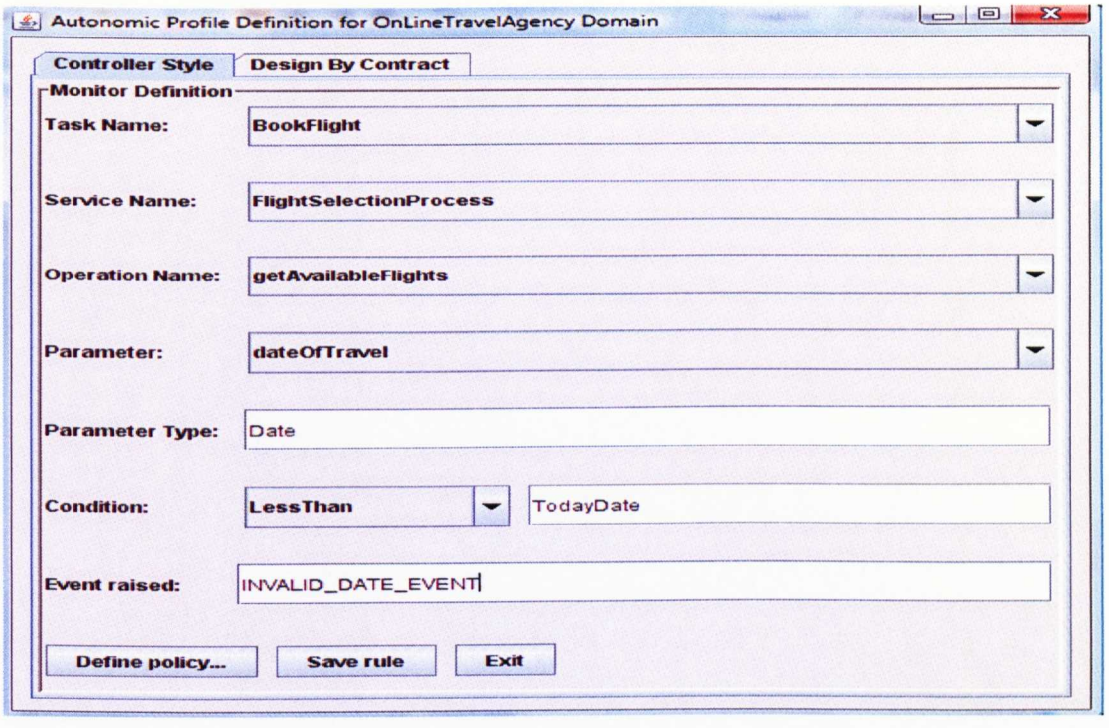

Figure 8. 7: User Interface for Autonomic Profile Definition for OTA domain.

Figure 8.8 depicts just a small portion of the resulted autonomic model which shows two services with their autonomic components, the *monitor* and *policy*.

```
<? xml version="1.0" encoding="UTF-8" standalone="no"?>
<Domain name="OnlineTravelAgency">
    <Task id="1" name="BookFlight">
        <Service id="1" name="FlightSelectionProcess">
            <Interface>
                <operation name="getAvailableFlights" returnType="list"
returnedVar="availableFlights">
                    <params>
                        <param name="sourceAirport" type="Text"/>
                        <param name="destinationAirport" type="Text"/>
                        <param name="dateOfTravel" type="Date"/>
                    </params>
                </operation>
.....
            </Interface>
            <Monitor name="Monitor_FlightSelectionProcess">
                <MonitorOperation ReturnType="void"
name="startMonitoring"/>
                <MonitoredVar Event="BAD_DATE_EVENT"
name="dateOfTravel" threshhold="&lt;Today" type="Date"/></Monitor>
```

```
            <Policy name="BAD_DATE_EVENT">
                <Event name="event"/>
                <PreCondition value="NA"/>
                <PostCondition value="NA"/>
                <CorrectiveAction value="BAD_DATE_ACTION"/>
            </Policy>
        </Service>
.....
        <Service id="3" name="PaymentCardValidator">
            <Interface>
                <operation name="verifyPaymentCard"
returnType="boolean" returnedVar="successful">
                    <params>
                        <param name="cardNo" type="Text"/>
                        <param name="securityNo" type="Text"/>
                    </params>
                </operation>
            </Interface>
        </Service>
    </Task>
    <Task id="2" name="MakeHotelReservation">
        <Service id="1" name="HotelSelectionProcess">
            <Interface>
                <operation name="getAvailableHotels" returnType="list"
returnedVar="hotels">
                    <params>
                        <param name="airportCode" type="Text"/>
                    </params>
                </operation>
                <operation name="reserveRoom" returnType="void"
returnedVar="void">
                    <params>
                        <param name="hotelID" type="Text"/>
                        <param name="dateOfArrival" type="Date"/>
                        <param name="noOfNights" type="int"/>
                    </params>
                </operation>
            </Interface>
            <Monitor name="Monitor_HotelSelectionProcess">
                <MonitorOperation ReturnType="void"
name="startMonitoring"/>
                <MonitoredVar Event="BAD_VALUE_EVENT" name="noOfNights"
threshhold="&lt;1" type="int"/></Monitor>
            <Policy name="BAD_VALUE_EVENT">
                <Event name="event"/>
                <PreCondition value="NA"/>
                <PostCondition value="NA"/>
                <CorrectiveAction value="BAD_VALUE_ACTION"/>
            </Policy>
        </Service>
    </Task>
</Domain>
```

Figure 8. 8: A Simplified Version of the Core System and the Assurance Element.

8.3.1.4 The Platform Specific Autonomic Model (PSAM)

The last stage before the code generation is to add the terms and data types for the specific platform, Java for instance. To target Java for example, we apply the Java file

that encapsulates the transformation rules responsible for adding Java terminology and data types. This file is referred to as *JavaPlatform.java* and was introduced in more detail in Section 6.1.2. The resulted XML file, *OnLineTravelAgency_autonomic_java.xml*, which contains the Java autonomic system, can be seen in Figure 8.9.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Domain name="OnlineTravelAgency">
    <package id="1" name="BookFlight">
        <class id="1" name="FlightSelectionProcess">
            <Method name="getAvailableFlights" returnType="List"
returnedVar="availableFlights">
                <param name="dateOfTravel" type="Date"/>
                <param name="destinationAirport" type="String"/>
                <param name="sourceAirport" type="String"/>
            </Method>
            <Method name="reserveSeatOnFlight" returnType="boolean"
returnedVar="done">
                <param name="flightNo" type="String"/>
                <param name="lName" type="String"/>
                <param name="fName" type="String"/>
            </Method>
        </class>
        <class id="2" name="CustomerCharger">
            <Method name="calculateTicketCost" returnType="long"
returnedVar="ticketCost">
                <param name="flightNo" type="String"/>
            </Method>
        </class>
        <class id="3" name="PaymentCardValidator">
            <Method name="verifyPaymentCard" returnType="boolean"
returnedVar="successful">
                <param name="securityNo" type="String"/>
                <param name="cardNo" type="String"/>
            </Method>
        </class>
    </package>
    <package id="2" name="MakeHotelReservation">
        <class id="1" name="HotelSelectionProcess">
            <Method name="getAvailableHotels" returnType="List"
returnedVar="hotels">
                <param name="airportCode" type="String"/>
            </Method>
            <Method name="reserveRoom" returnType="void"
returnedVar="void">
                <param name="noOfNights" type="int"/>
                <param name="dateOfArrival" type="Date"/>
                <param name="hotelID" type="String"/>
            </Method>
        </class>
     </package> </Domain>
```

Figure 8. 9: Java Autonomic Online Travel Agency system.

8.1.3.5 The Autonomic Code Generation

To generate code for Java web services, we apply two transformation files, namely the *JavaWSGenerator.java* and *AutoJavaWSGenerator.java* to the

OnLineTravelAgency_Autonomic_java.xml file. The former is responsible for generating the core web services whereas the latter produces the code for the autonomic components, the *monitor* for example. A high level representation of these stages is depicted in Figure 8.10.

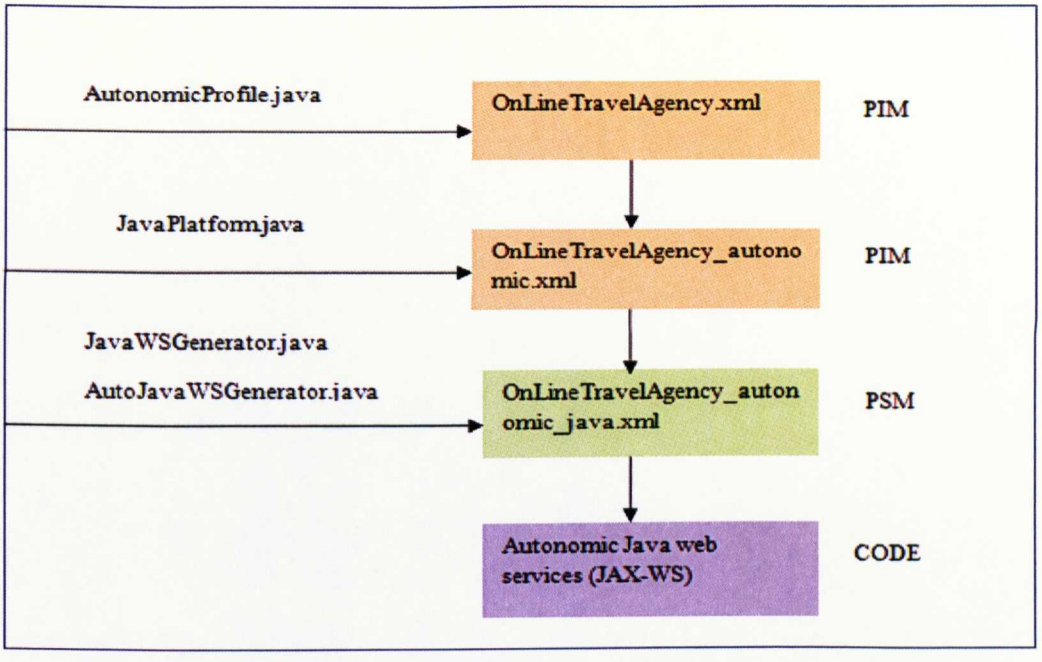

Figure 8. 10: Autonomic Java Web Services Lifecycle.

The generated Java files (proxy web service, actual web service and a JSP file) for this domain can be found in Appendix E. As shown in Appendix E (Listing E.1), the proxy web service contains the necessary business logic for calling the actual web service, the *PaymentCardValidator* in this case. Also, invoking the proxy web service which meant to be performed and dispatched to the actual web service is encapsulated in a JSP file as depicted in Listing E.2. The skeleton code for the actual Java web service is shown in Listing E.3. Here, the web services are coded according to the JAX-WS technology. To show part of the autonomic aspect of the case study, the automatically generated code for the monitor service is introduced in Listing E.4. The monitor service here is monitoring the *dateOfTravel* parameter of the *FlightSelectionProcess* service. The state change notification comes from the proxy as shown in the code and then the monitor responds by reading the new entered value from the sensor, the getter method in this case. The retrieved value is compared to a threshold, today date in this case, to decide whether or not to raise a conflict event. In case of sensing a value which reflects an old date, an INVALID_DATE_EVENT is raised and sent to the control rules (policy) component.

Here, the received event is used to search for a matching situation. In the case of finding a matching situation, the associated action is performed.

The following screenshots (Figures 8.11-8.14) show the user interactions with the Online Travel Agency application. In particular, it shows the required interactions involved in addressing the *Book Flight* task (Business process). The user interface is coded here in JSP files and the business logic and actual code is encapsulated in Java Web services.

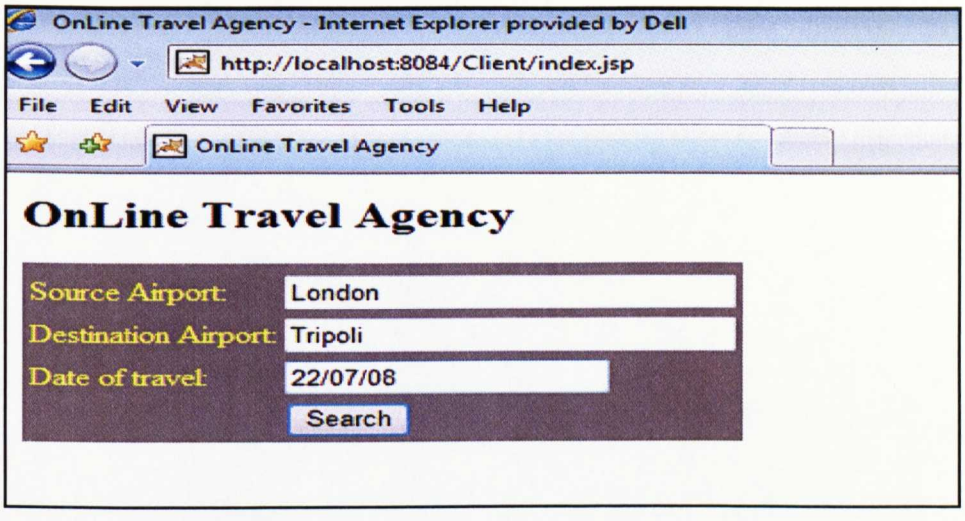

Figure 8.11: Flight details entry form.

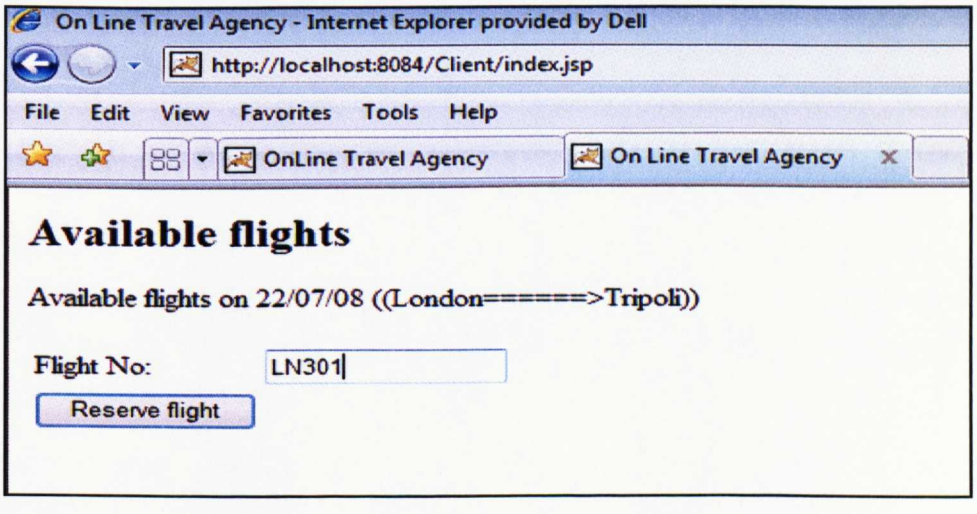

Figure 8. 12: Rendering of available flights.

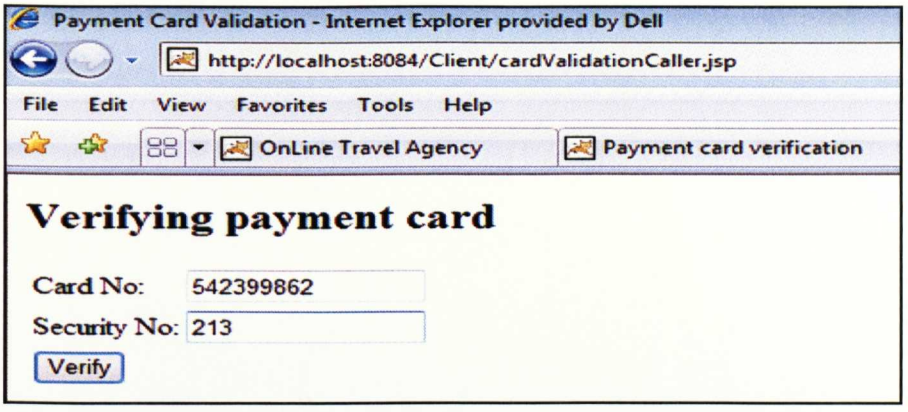

Figure 8. 13: The Payment Card Input Form.

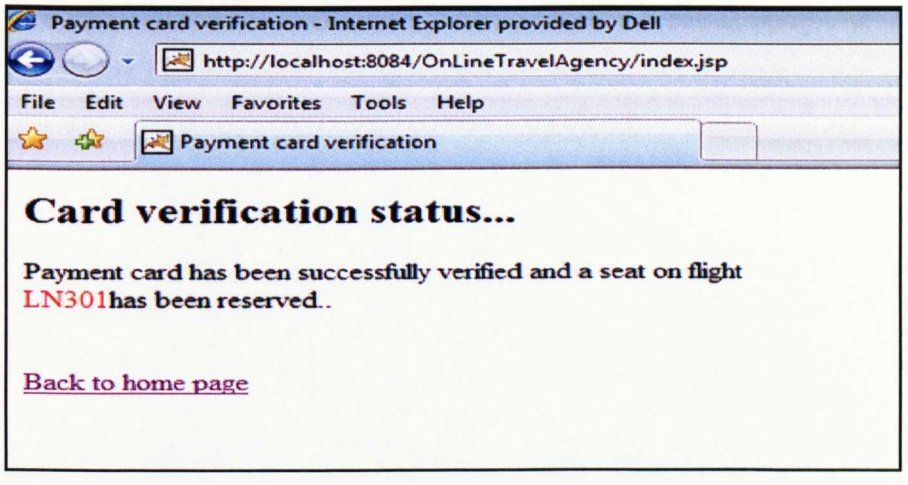

Figure 8. 14: Notification of card verification and seat reservation.

8.3.2 The Intelligent Office case study- Generality Evaluation

As stated earlier, the generality aspect indicates that the proposed design method should not be limited or tailored for one specific platform or case study. Therefore, a case study called the *intelligent office* is introduced here to show how the proposed design method can be used to accommodate a new system (domain) without carrying out any changes. Also the target platform here is the C# Web services.

The purpose of this hypothetical system is to design and develop *an intelligent office*. This intelligent office should address and meet the requirements set below:

- Security measures should be provided in terms of who is authorised to get in.
- Room temperature should be kept at a reasonable level.
- The power should be consumed reasonably and efficiently inside the office. For example, the lights should be automatically turned off when it is sunny and the office curtain is open.

These requirements or high level goals are mapped into tasks according to our design method. Thus, here we have four fundamental tasks, listed above, to be addressed in order to deliver and achieve the overall system functionality. The system under consideration here is called a *domain* following the terminology of our approach which is the *Intelligent Office.*

8.3.2.1 Task and service definitions

The *intelligent office* domain has four primary tasks as follows:

- *Enforce authorised access to office task:* this task is responsible for providing mechanisms and security measures to control who is entitled and authorised to get into the office.
- *Measure office temperature task*: this task is responsible for measuring the office temperature so it can be kept at a reasonable and specific value.
- *Ensure efficient power consumption task*: this task ensures and guarantees that power consumed efficiently and sensibly.

The Intelligent office domain and its fundamental tasks are depicted in Figure 8.15.

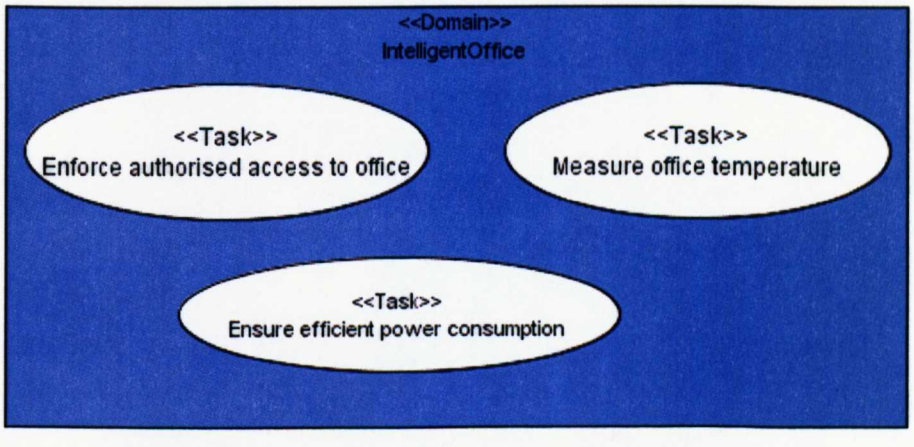

Figure 8. 15: A set of tasks for the Intelligent Office domain.

Each of the tasks above is then represented in terms of services. The following sequence diagrams show this process.

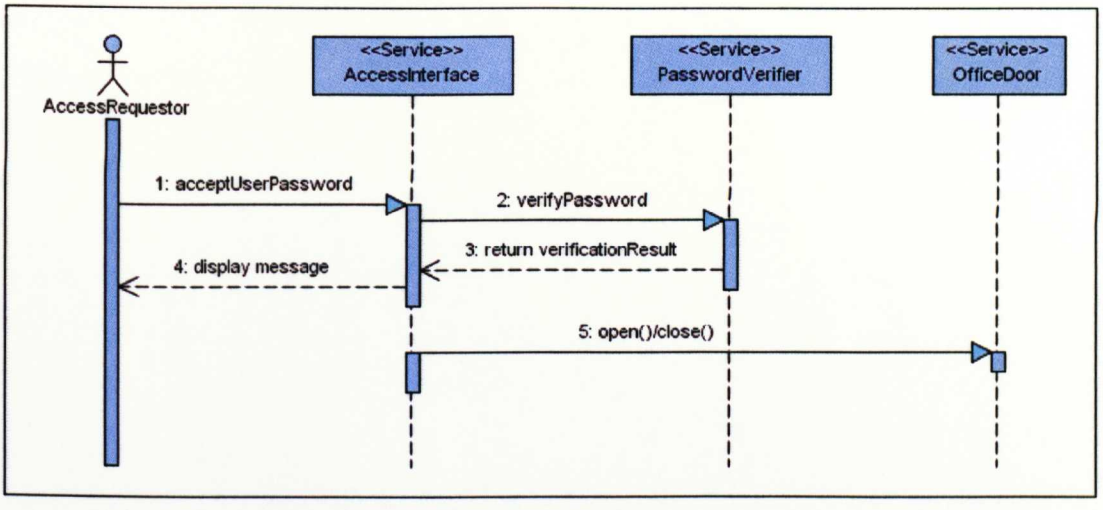

Figure 8.16: A set of services for the 'enforce authorised access to office' task.

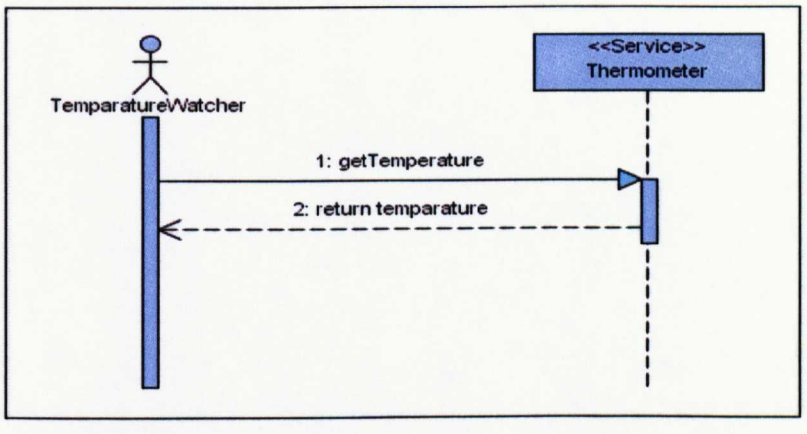

Figure 8. 17: The thermometer service for 'MeasureOfficeTemparature' task.

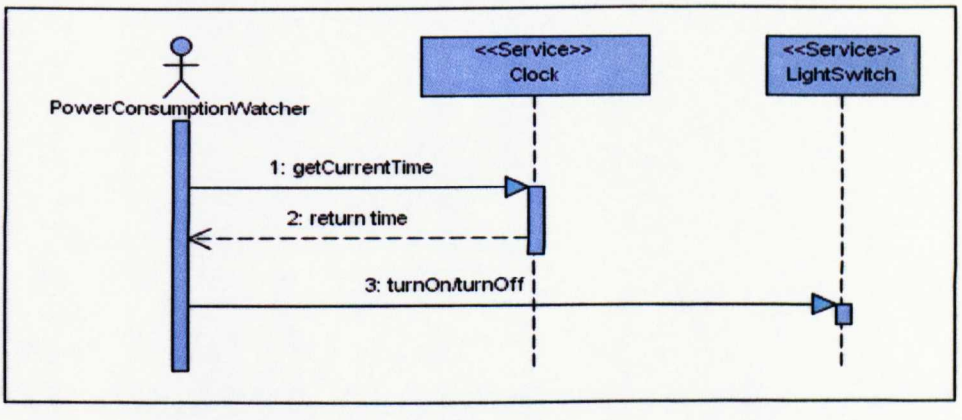

Figure 8.18: Services for 'Ensure efficient power consumption' task.

8.3.2.2 The Intention Model definition

Out of these sequence diagrams, the complete system (domain) can be obtained which is represented, at high level, in terms of tasks and their corresponding services. The

obtained model (Intention model) of the intelligent office is saved in an XML file called *IntelligentOffice.xml*. This file is shown in Figure 8.19. Also, the interactions between each task services can be extracted and modelled here. Such interactions are encapsulated and saved in a separate XML file called *IntelligentOffice_composite.xml*. Such a file is depicted in Figure 8.20.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Domain name="IntelligentDoor">
    <Task name="EnforceAutorisedAccess" id="1">
       <Service name="AccessInterface" id="1">
         <Interface>
             <operation name="acceptUserPassword" returnedVar="message"
returnType="Text">
                <params>
                      <param name="password" type="Text"/>
                </params>

              </operation>
         </Interface>
      </Service>
      <Service name="OfficeDoor" id="2">
         <Interface>
             <operation name="open" returnedVar="void"
returnType="void"/>
             <operation name="close" returnedVar="void"
returnType="void"/>
         </Interface>
      </Service>
   </Task>

   <Task name="" id="ControlOfficeTemperature">
       <Service name="thermometer" id="1">
         <Interface>
             <operation name="getTemperature" returnedVar="temperature"
returnType="float">

</operation>
         </Interface>
      </Service>
.....................................................................
   </Task>
 </Domain>
```

Figure 8. 19: The Intention Model for the Intelligent Office domain.

```
<?xml version="1.0" encoding="UTF-8" ?>

<composites domain="IntelligentOffice">

   <composite name=" EnforceAutorisedAccess ">
        <Interaction>
            <callingParty name="user"/>
            <calledParty name="AccessInterface">
                <operation name="acceptUserPassword"/>
            </calledParty>
        </Interaction>
```

```
        <Interaction>
            <callingParty name="AccessInterface"/>
            <calledParty name="PasswordVerifier">
                <operation name="verifyPassword"/>
            </calledParty>
        </Interaction>
        <Interaction>
            <callingParty name="AccessInterface"/>
            <calledParty name="OfficeDoor">
                <operation name="setDoorStatus"/>
            </calledParty>
        </Interaction>
    </composite>

    <composite name=" ControlOfficeTemperature ">
        <Interaction>
.....................................................
.....................................................
    </composite>
</composites>
```

Figure 8. 20: Composite Model for the Intelligent Office domain.

8.3.2.3 The Platform Independent Autonomic Model (PIAM)

The next step of the transformation process is to add and generate the autonomic model for the Intelligent Office domain. This can be achieved by applying the *AutonomicProfile.java* file to the intention model. The outcome of this operation is saved in an XML file called *IntelligentOffice_Autonomic.xml* which is illustrated (small part of it) in Figure 8.21.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Domain name="IntelligentDoor">
  <Task id="1" name="EnforceAutorisedAccess">
    <Service id="1" name="AccessInterface">
      <Interface>
        <operation name="acceptUserPassword" returnType="Text" returnedVar="message">
          <params>
            <param name="password" type="Text"/>
          </params>
        </operation>
      </Interface>
      <Monitor name="Monitor_AccessInterface">
        <MonitorOperation ReturnType="void" name="startMonitoring"/>
        <MonitoredVar Event="INVALID_PASSWORD" name="password" threshhold="!= admin"
type="Text"/>
      </Monitor>
.....
.....
.....
    </Service>
    <Service id="2" name="OfficeDoor">
      <Interface>
        <operation name="open" returnType="void" returnedVar="void"/>
        <operation name="close" returnType="void" returnedVar="void"/>
```

```
      </Interface>
    </Service>
  </Task>
  <Task id="ControlOfficeTemperature" name="">
    <Service id="1" name="thermometer">
      <Interface>
        <operation name="getTemperature" returnType="float" returnedVar="temperature"/>
      </Interface>
    </Service>
  </Task>
</Domain>
```

Figure 8. 21: The autonomic model for Intelligent Office model.

8.3.2.4 The Platform Specific Autonomic Model (PSAM)

Since the target platform here is the C# web services, the *Abst2PlatformTransformer.java* class is instantiated with the CSharpPlatform class instance passed to its constructor. This results in producing the platform specific autonomic intelligent office model. Such a model is stored in an XML file referred to as *IntelligentOffice_Autonomic_CSharp.xml* and depicted in Figure 8.22.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Domain name="IntelligentDoor">
    <namespace id="1" name="EnforceAutorisedAccess">
        <class id="1" name="AccessInterface">
            <Method name="acceptUserPassword" returnType="string"
returnedVar="message">
                <param name="password" type="string"/>
            </Method>
        </class>
        <class id="2" name="OfficeDoor">
            <Method name="open" returnType="void" returnedVar="void"/>
            <Method name="close" returnType="void" returnedVar="void"/>
        </class>
    </namespace>
.....................................................................
    <namespace id="ControlOfficeTemperature" name="">
        <class id="1" name="thermometer">
            <Method name="getTemperature" returnType="float"
returnedVar="temperature"/>
        </class>
.....................................................................
    </namespace>
</Domain>
```

Figure 8.22: The C# Autonomic Intelligent Office domain.

8.3.2.5The Autonomic Code Generation

The final stage of the development process is to generate the autonomic code. Here, two kinds of code generator are used. The first generator which is encapsulated in the file CSharpCodeGenerator.java is used to generate the core C# web services whereas the second generator is used to generate the autonomic C# web service and saved in the file

AutonomicCSharpCodeGenerator.java. Listings 8.1 and 8.2 depict two core C# web services, namely the Access Interface and Thermometer.

Listing 8.1: Generated C# code for Access Interface web service.

```
/*****************************
*Developer: Yousef Abuseta***
*University: LJMU (2009)    *
C# WEB SERVICES
*****************************/

<%@ WebService language="C#" class="AccessInterface"%>

using System;
using System.Web.Services;

[WebService]
public class AccessInterface:WebService {

    [WebMethod]
    public string acceptUserPassword( string password ){

    //Implementation code goes here...

    return message;
  }
}
```

Listing 8. 2: Generated C# code for Thermometer web service

```
/*****************************
*Developer: Yousef Abuseta***
*University: LJMU (2009)    *
C# WEB SERVICES
*****************************/

<%@ WebService language="C#" class="thermometer"%>

using System;
using System.Web.Services;

[WebService]
public class thermometer:WebService {

    [WebMethod]
    public float getTemperature( ){

    //Implementation code goes here...

    return temperature;
  }
}
```

8.3.3 The Pet Store case study-Adaptability Evaluation

To evaluate the proposed design method from the adaptability perspective, the case study *Pet Store* [80] is used here. This case study is a typical e-commerce application. It is an online pet store enterprise that offers services and sells animals to customers. This

application has a web site via which the customers can interact with and access services offered by this application. Application interfaces are also available to other users such administrators and suppliers to maintain inventory and perform managerial tasks. Each class of users has access to specific categories of functionality, and each interacts with the application through a specific user interface mechanism. A high level view of the real world problem intended to be solved by this application is depicted in Figure 8.23. As it can be seen from the Figure, a number of involving parties can be identified, including: the customers, order fulfilment centre, suppliers and credit card validation. According to our approach, we have to represent this application (*domain* in our terminology) in terms of tasks. A task, as stated in Section 4.2, is *a very high level goal that has to be addressed in order to address the overall system requirements.*

Figure 8.23: A high-level view of the major modules of pet store application [80]

8.3.3.1 Task and service definitions

The two following tasks (shown in Figure 8.24) can be identified in the Pet store domain:

- Order product from supplier: the pet store has established contracts with a number of suppliers to offer a varied collection of pets to target a broad range of customers. The trigger of this task is the selling point service.

- Sell products (pets, food): the trigger of this task is the customer whose primary intention here is to buy pets from the online pet store.

Each of the above listed tasks should be addressed and realised through a set of services following the process adopted in our design method.

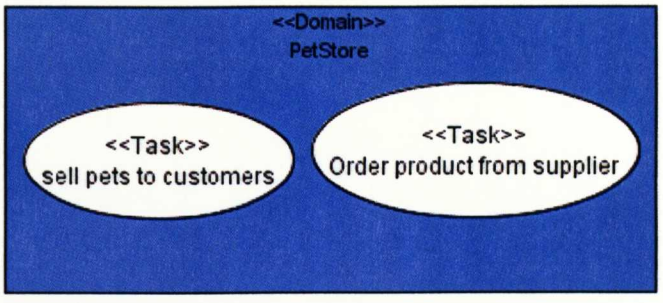

Figure 8. 24: Tasks for Pet Store domain.

To express and represent each task in terms of services, the two following UML sequence diagrams are used.

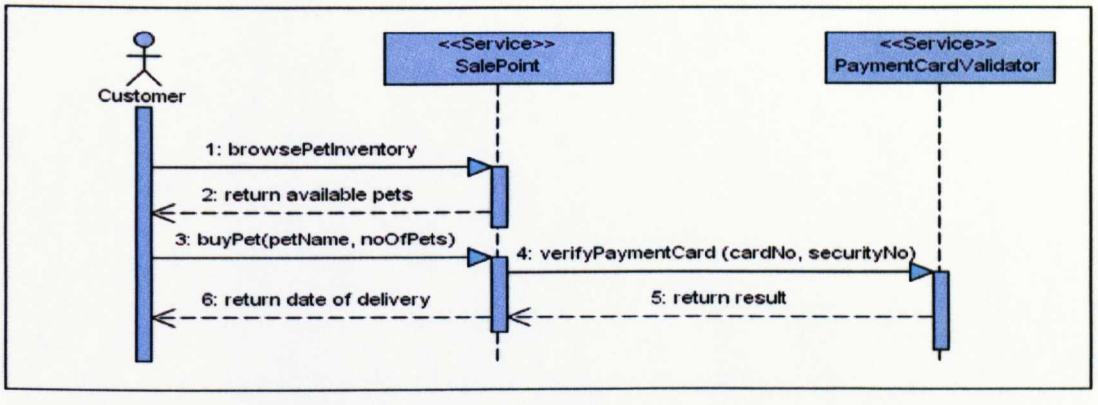

Figure 8.25: A set of services for 'the sell products to customer' task.

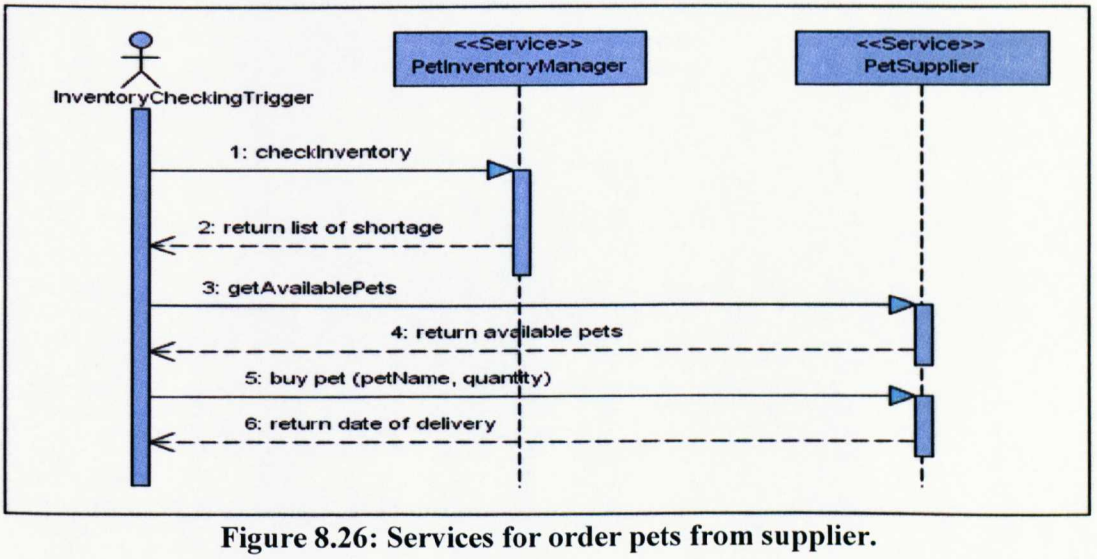

Figure 8.26: Services for order pets from supplier.

In this section, we demonstrate and show the feasibility of our proposed method by injecting a new service to the above task shown in Figure 8.25. The service to be injected into the task is called *OfferChecker*. Such a service is called by the *SalesPoint* service to check whether the customer is qualified and eligible for a special offer and

charge him or her accordingly. The special offer eligibility is evaluated based on the pet name, number of pets and the payment method.

Only the synchronisation between the PIM and APIM will be discussed and shown here. As stated earlier, the sensing and actuation layer is responsible for providing the facilities and means for the system designer to modify and accomplish the desirable changes (adding new service in this case) to the target model. Here, a development environment was developed in Java to serve as a supporting tool. A screenshot of this environment is shown in Figure 8.27.

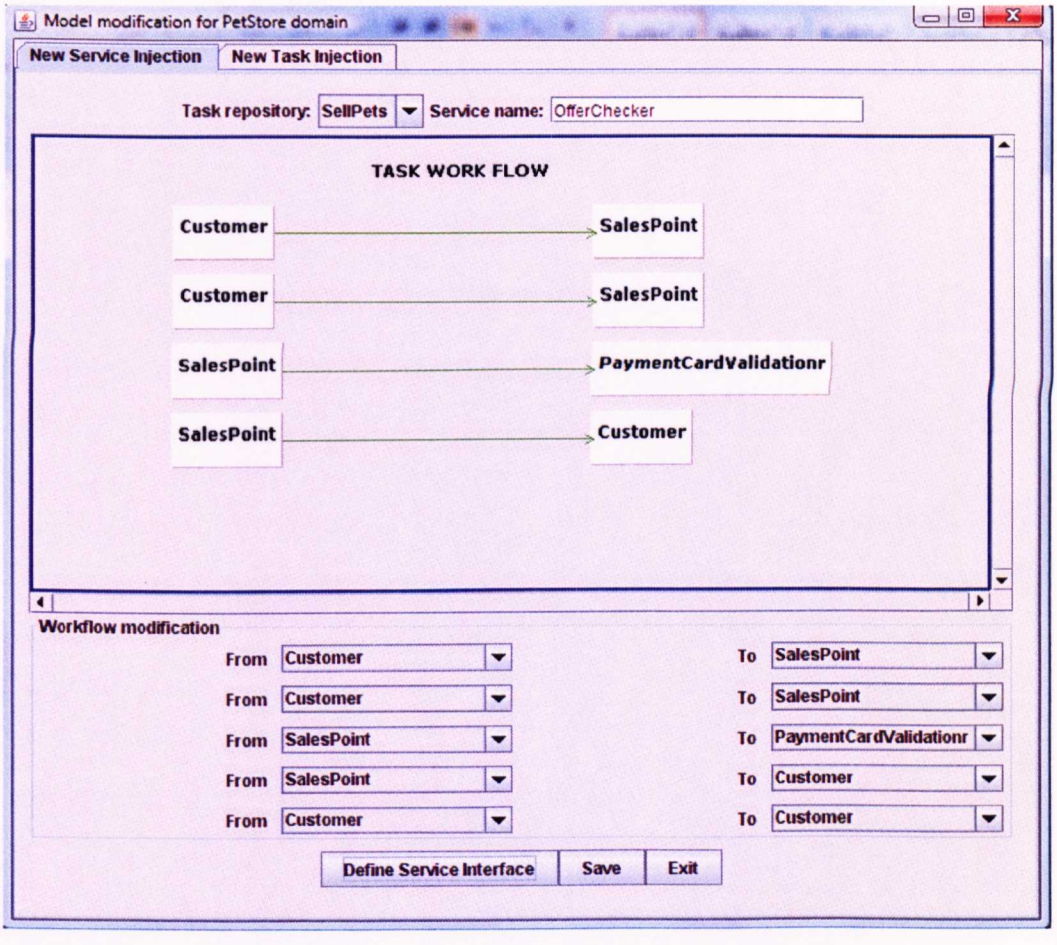

Figure 8. 27: New service injection user interface.

In the proposed synchronisation mechanism, the human is introduced to the loop and is offered the facilities and required user interface to accomplish the model modification task. However, as explained earlier, the model synchronisation process is carried out in an automated way with the coordination of the controllers that spread over the four MDD stages of our design method. As shown in Figure 8.27, the human user is provided by the required GUI to add the *OfferChecker* service (at PIM level) to the 'sell pets' task of the

Pet store domain. Also the workflow of this task is shown, so that the user knows where to plug in this new service and performs the required changes. In the service name field, the user has to enter the name of the service (OfferChecker in this case) to be injected and the workflow modification section can be used to change and rearrange the service interactions, which is a necessary step once a new service is introduced. Then, the user should define the service interface (its operations) which can be done via pressing the 'Define service interface' button. Such an action brings up the interaction dialog depicted in Figure 8.28. As soon as the user or system designer finishes defining the service interface, and confirms adding the service by pressing save and them exit, a notification of change message is sent to the APIM Change controller stating that a change has occurred at the PIM stage and the appropriate actions are being taken. This is depicted in Figure 8.29. Such changes will be reflected in the workflow of the '*sell pets*' task which is shown in Figure 8.30. Also, the sensor notifies the change manager of the event just taken place. The controller or change manager, in turn, notifies the other controllers of this event and passes the necessary information, so they can adapt their models accordingly. In this case, the information passed to the controller (APIM in this case) include the event name (add new service), task name and the service being added. The APIM change manager will have to take the corresponding action or actions found in its change plan engine which include: 1) adding the new service to the right task; 2) trigger the critical parameters definition process to enable designers specify the parameters to be monitored; 3) trigger the policy definition process which enables the human user to amend and add the control policies and 4) notify the APSM change manager of the change that has just taken place.

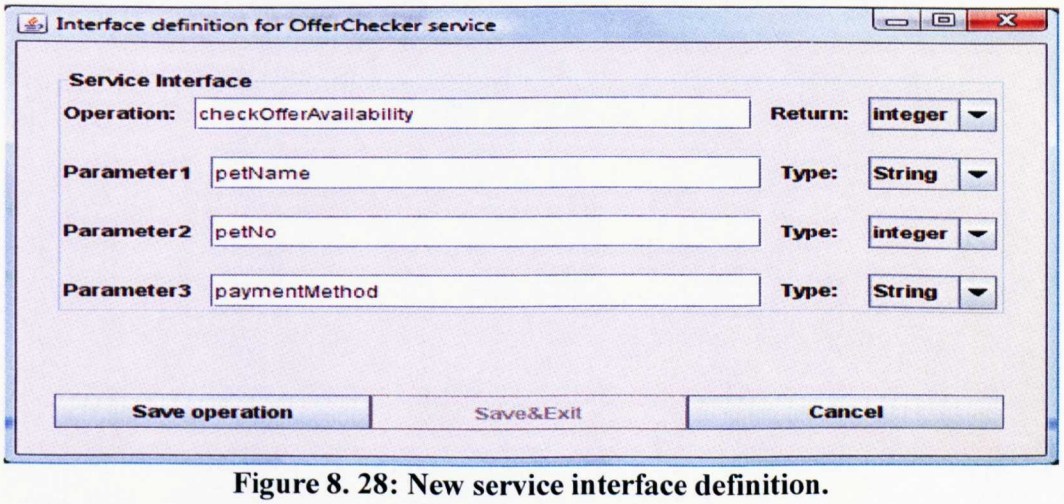

Figure 8. 28: New service interface definition.

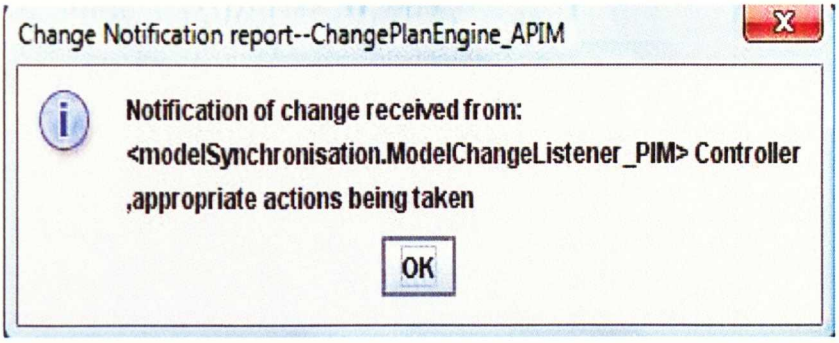

Figure 8.29: Notification of model change from PIM change controller.

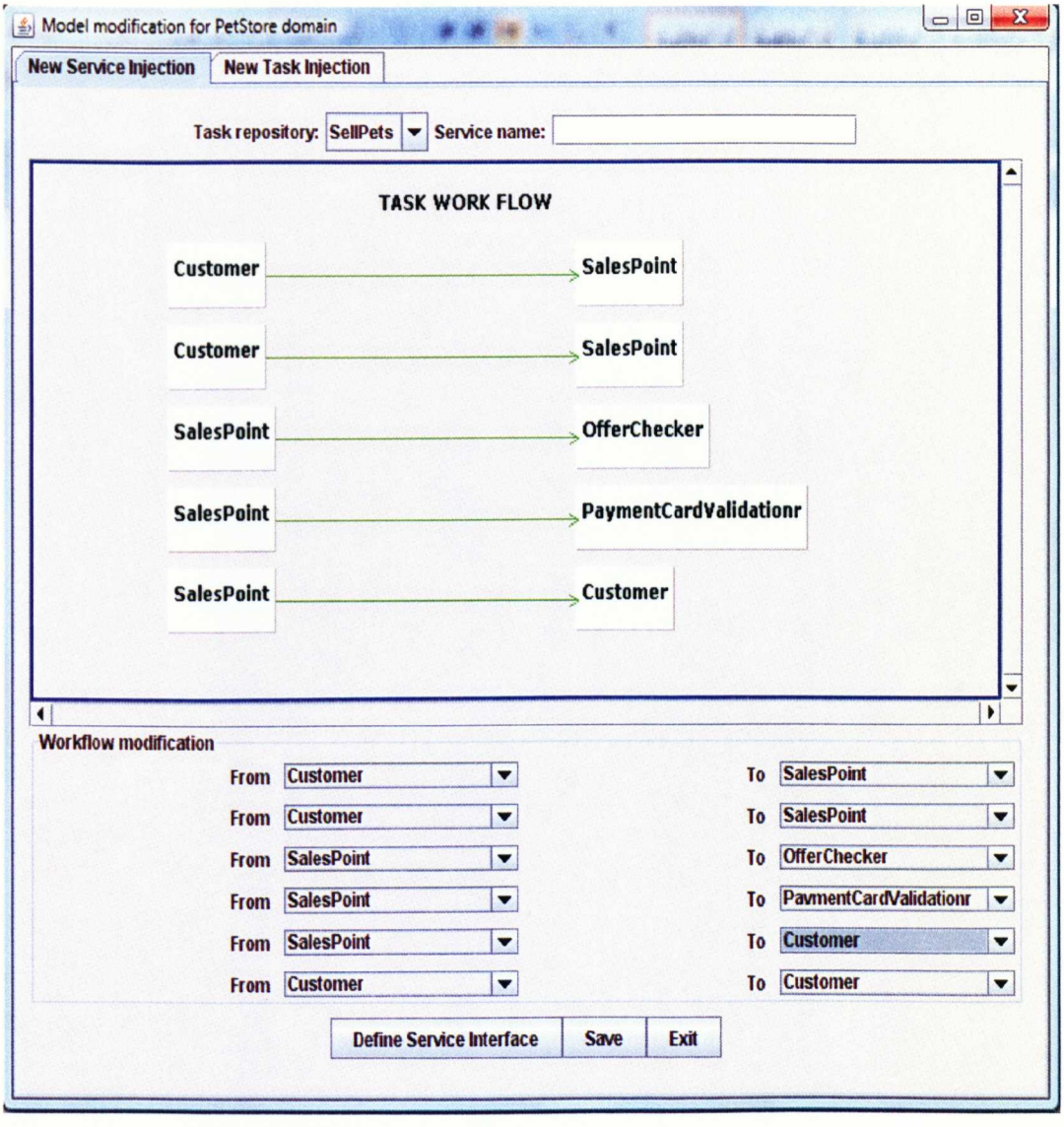

Figure 8.30: Task work flow after service injection.

8.3.4 The Salt World case study- self organising systems

In this section, the proposed design method is evaluated against the self organisation systems [81] . The Salt World case study [81] is used here for this purpose. The latter

contains two primary entities, namely the salt grains and salt carriers. In the initial situation, grains of salt are randomly distributed over a 2-D world which also contains randomly distributed salt carriers their job is to pick the grains and drop them to form a salt pile. The carriers move randomly within the world, pick up salt grains and drop them at the nearest empty space once another salt grain is encountered. This behaviour leads to the formation of highly concentrated connected salt pile. Therefore, the salt concentration event is the emergent behaviour that should be observed here. Subsequent subsections introduce the definitions and specifications of the different stages and models for the Salt World domain based on the proposed MDD based design method.

8.3.4.1 Task and service definitions

For the salt world application, the following tasks can be identified:

- Salt grain holding
- Salt grain dropping
- Salt concentration calculation

Figure 8.31 shows these tasks using a UML use case diagram.

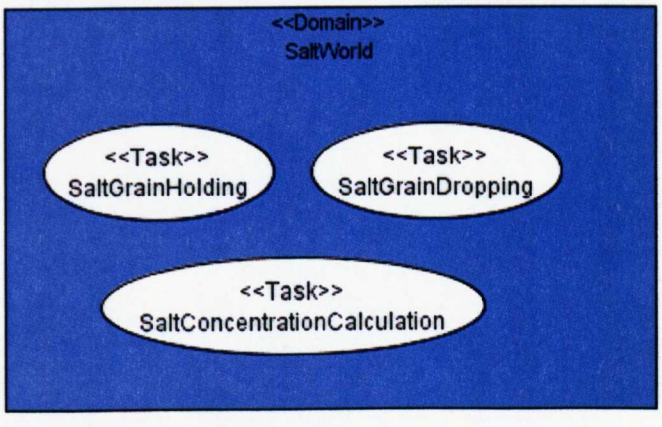

Figure 8. 31: A set of tasks for the Salt World domain.

For each task of the above tasks, a set of services is defined. A UML sequence diagram is used to define these services and their interactions. Below are the set of sequence diagrams (Figures 8.32-8.34) for each of these tasks.

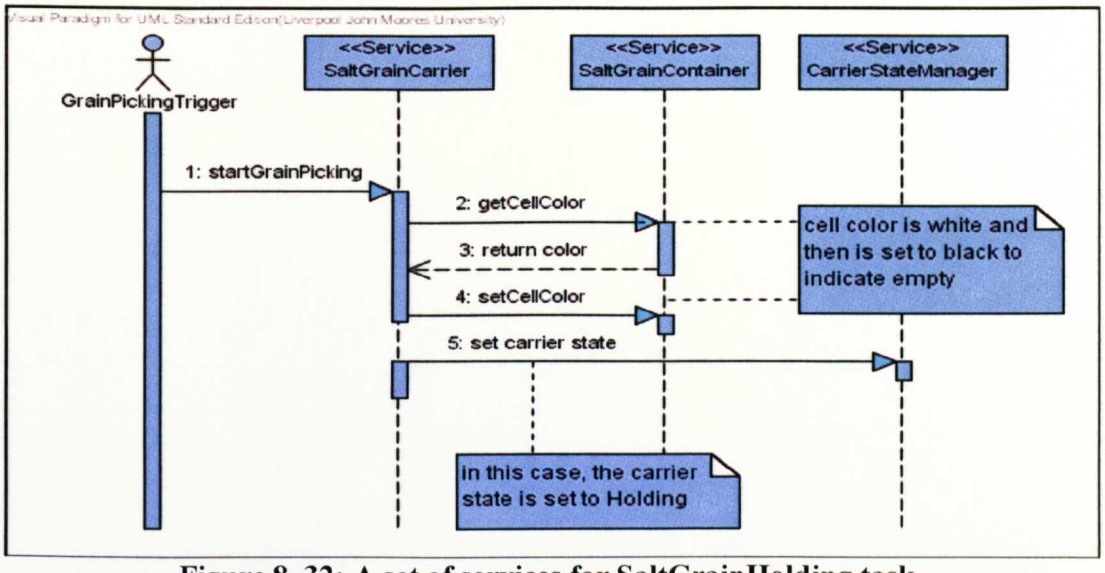

Figure 8. 32: A set of services for SaltGrainHolding task.

As it can be seen, the above task is addressed by a set of services, namely the *SaltGrainPicker*, *SaltGrainContainer* and *the CarrierStateManager*.

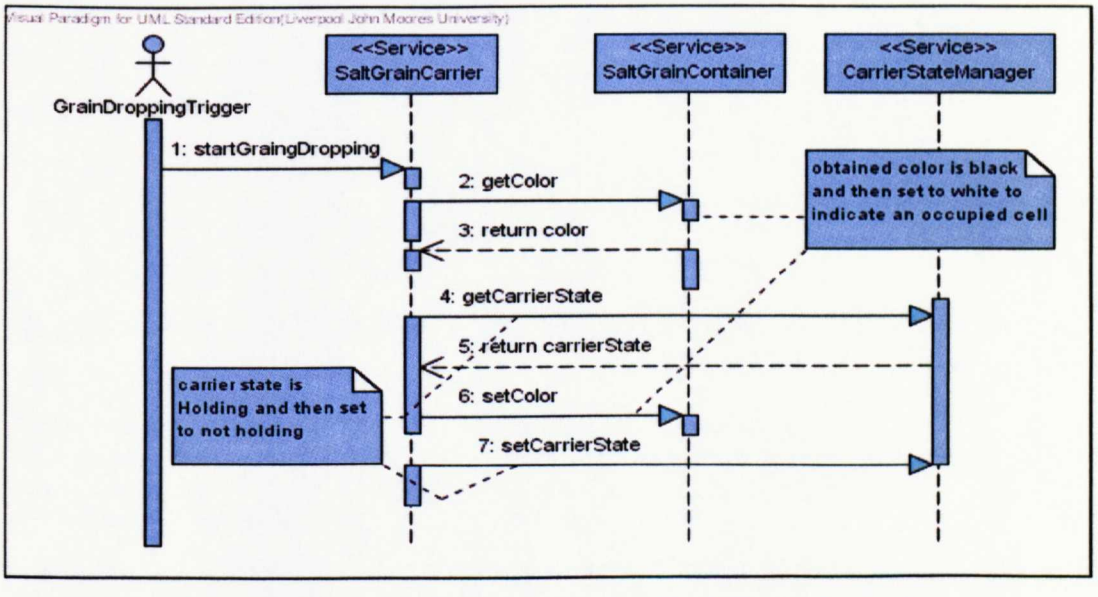

Figure 8. 33: A set of services for SaltGrainDropping task.

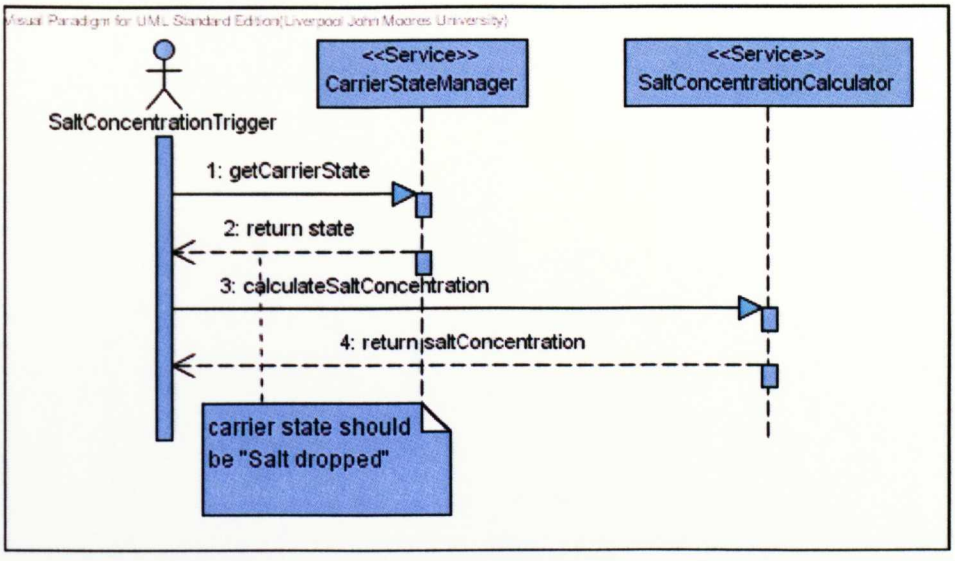

Figure 8.34: A set of services for SaltConcentrationCalculation task.

8.3.4.2 The Intention Model definition

The Intention Model for the SaltWorld domain is composed of the core services generated above using the UML sequence diagrams in addition to the interaction model that defines the service interactions for each task. The core services are encapsulated in an xml file in the form of *domainName.xml*. For the salt world, this file is saved as *SaltWorld.xml* as shown in Figure 8.35. As for the service interactions for each task, an xml file of the form *domainName_serviceComposites.xml*. The salt world tasks interactions are saved in a file called *SaltWorld serviceComposites.xml* as illustrated in Figure 8.36. A more visualised representation of these interactions is depicted in Figure 8.37.

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
   Document   : SaltWorld.xml

   Author     : LA_Abuseta
   Description:
      The Intention Model of the Salt world domain
-->
<Domain name="saltworld">
   <Service name="SaltGrainCarrier">
      <Operation name="startGrainPicking" returnType="none"/>
      <Operation name="startGrainDropping" returnType="none"/>
   </Service>
   <Service name="SaltGrainContainer">
      <Operation name="getColor" returnType="string" returnVar="color"/>
      <Operation name="setColor" returnType="none" returnVar="none">
         <param name="color" type="string"/>
      </Operation>
   </Service>
```

```
    <Service name="CarrierStateManager">
      <Operation name="setCarrierState" returnType="none">
        <param name="state" type="string"/>
      </Operation>
      <Operation name="getCarrierState" returnType="string">
      </Operation>
    </Service>
</Domain>
```

Figure 8. 35: The Intention Model for SaltWorld domain.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Document   : SaltWorld_serviceComposite.xml
    Created on : 11 May 2009, 12:23
    Author     : LA_Abuseta
    Description:
        The Task composites for the Salt World domain
-->
<Composites domain="SaltWorld">
    <Composite name="SaltGrainHolding">
      <Interaction>
        <CallingParty name="Trigger"/>
        <CalledParty name="SaltGrainCarrier">
          <CalledOperation name="startGrainPicking"/>
        </CalledParty>
      </Interaction>
      <Interaction>
        <CallingParty name="SaltGrainCarrier"/>
        <CalledParty name="SaltGrainContainer">
          <CalledOperation name="getCellColor"/>
        </CalledParty>
      </Interaction>
      <Interaction>
        <CallingParty name="SaltGrainCarrier"/>
        <CalledParty name="SaltGrainContainer">
          <CalledOperation name="setCellColor"/>
        </CalledParty>
      </Interaction>
      <Interaction>
        <CallingParty name="SaltGrainCarrier"/>
        <CalledParty name="CarrierStateManager">
          <CalledOperation name="setCarrierState"/>
        </CalledParty>
      </Interaction>
    </Composite>
    <Composite name="SaltGrainDropping">
      <Interaction>
        <CallingParty name="Trigger"/>
        <CalledParty name="SaltGrainCarrier">
          <CalledOperation name="startGrainDropping"/>
        </CalledParty>
      </Interaction>
      <Interaction>
        <CallingParty name="SaltGrainCarrier"/>
        <CalledParty name="CarrierStateManager">
          <CalledOperation name="getCarrierState"/>
        </CalledParty>
      </Interaction>
      <Interaction>
        <CallingParty name="SaltGrainCarrier"/>
```

```
        <CalledParty name="SaltGrainContainer">
          <CalledOperation name="getCellColor"/>
        </CalledParty>
      </Interaction>
      <Interaction>
        <CallingParty name="SaltGrainCarrier"/>
        <CalledParty name="CarrierStateManager">
          <CalledOperation name="getCarrierState"/>
        </CalledParty>
      </Interaction>
    </Composite>
    <Composite name="SaltConcentrationCalculation">
      <Interaction>
        <CallingParty name="trigger"/>
        <CalledParty name="CarrierStateManager">
          <CalledOperation name="getCarrierState"/>
        </CalledParty>
      </Interaction>
      <Interaction>
        <CallingParty name="trigger"/>
        <CalledParty name=" SaltConcentrationCalculator">
          <CalledOperation name="calculateSaltConcentration"/>
        </CalledParty>
      </Interaction>
    </Composite>
</Composites>
```

Figure 8. 36: Service Compositions for SaltWorld tasks.

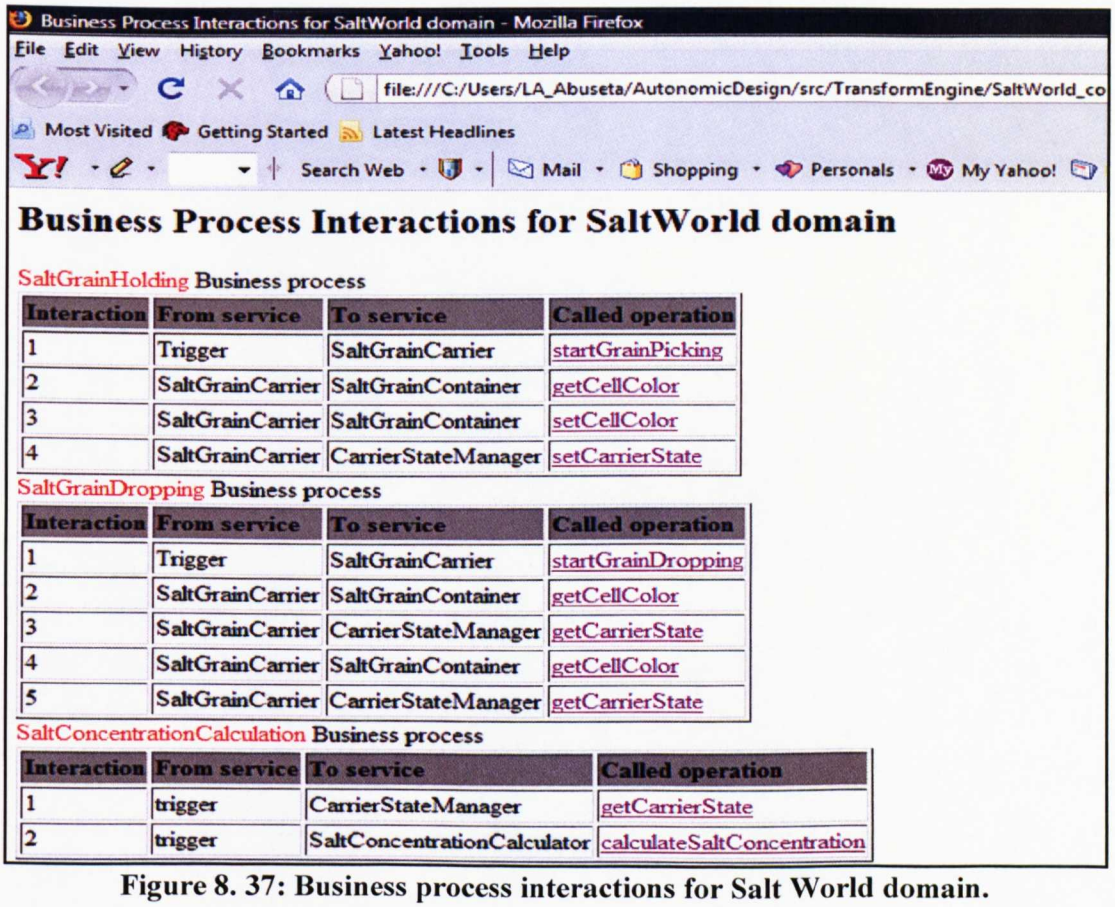

Figure 8. 37: Business process interactions for Salt World domain.

8.3.4.3 The Platform Independent Autonomic Model (PIAM)

The process of monitoring some interesting behaviour requires the definition and designing of an observer or controller component. This component will read the value of the monitored parameter and compare it with a threshold. If the read value violates the threshold, a conflict signal is raised to reflect or show an emergent behaviour. In the Salt world domain, the salt concentration is the parameter or behaviour to watch and monitor. In line with the autonomic systems terminology, the observer or the controller plays the autonomic manager role, while the managed element role is played here by the *salt concentration calculator* service. The design style depicted in Figure 8.38 shows the arrangement that is used to establish the relationship between the controller and the managed element, the salt concentration calculator service in this case.

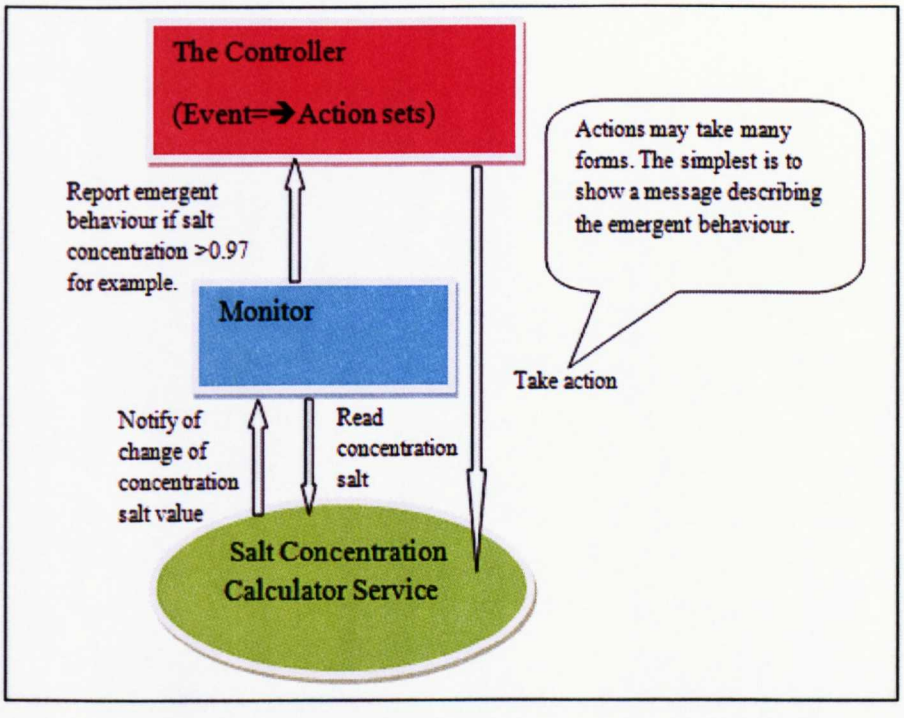

Figure 8. 38: The Service-Monitor-Controller design style for the salt world domain.

As illustrated in the figure, the service component notifies the monitor that a change on the monitored parameter has occured and the monitor responds by reading this new value. Based on the outcome of the comparison of the obtained value against the predefined threshold, a conflict signal might be raised and sent by the monitor to the controller which indicates the occurrence of an emergent behaviour. The controller then uses the received conflict signal to look up and dispatch the corresponding corrective action.

In the SaltWorld domain, the salt concentration value is calculated by the *salt concentration calculator* service. Once this process is completed, this service sends a signal to the monitor that reads the new value and then compares it with a threshold value of 0.97. If it is greater than 0.97, a signal is raised indicating high concentration behaviour. The monitor then notifies the controller of this event or behaviour and the controller takes the corresponding corrective action.

To inject the necessary autonomic components to the Intention Model, the autonomic transformer is applied. Such a transformer is encapsulated in a Java file called *AutonomicProfile.java*. The user interface used to specify the parameter to observe (salt concentration), the threshold (0.97) and the controller (policy engine) is shown in Figure 8.39.

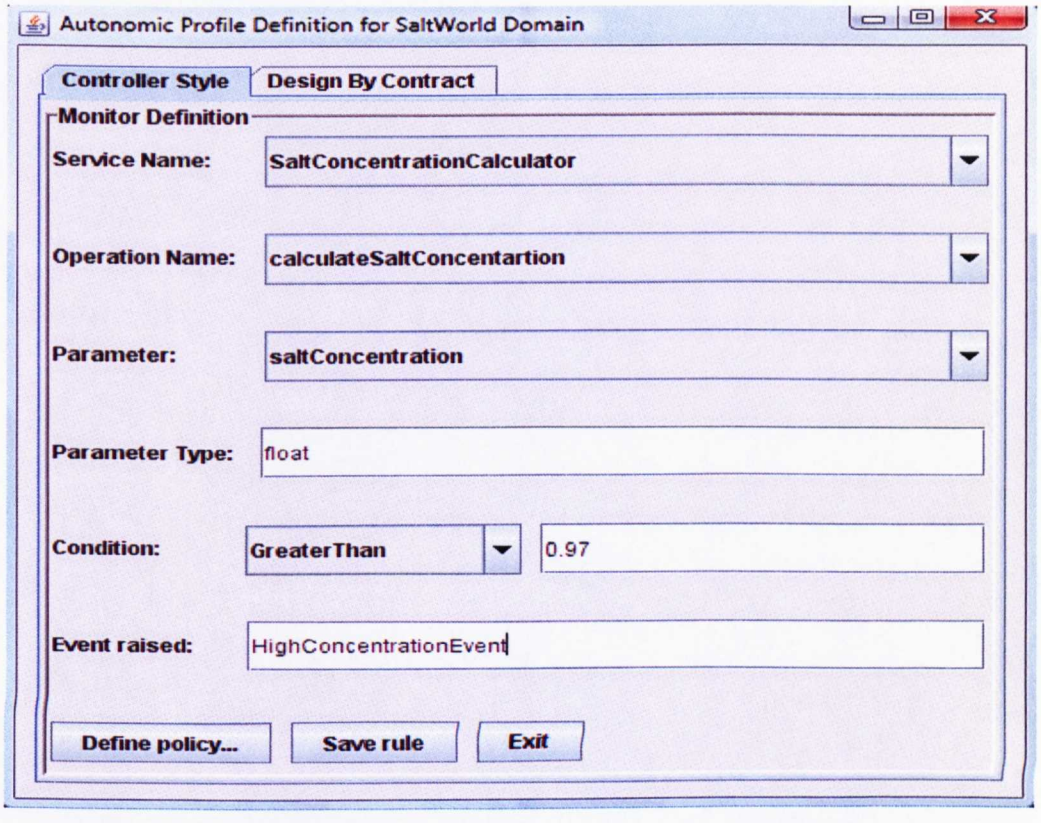

Figure 8. 39: Autonomic profile definition for Salt World domain.

The policy rule shown above follows the Event Condition Action (ECA) approach which takes the form of: on *event* if *condition,* do *action.* In the case of the monitor and service, for example, the monitor receives a change of value (salt concentration) event from the service and then evaluates the condition (concentration >0.97). If the condition holds, the monitor takes action in the form of sending a conflict signal to the controller. Figure 8.40

shows a simplified XML representation of the autonomic model for the salt world domain.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!--
   Document   : SaltWorld_autonomic.xml
   Created on : 16 May 2009, 12:21
   Author     : LA_Abuseta
   Description: The Autonomic Model of the Salt world domain
-->
<Domain name="saltworld">
   <Service name="SaltGrainCarrier">
      <operation name="startGrainPicking" returnType="none"/>
      <operation name="startGrainDropping" returnType="none"/>
   </Service>
   <Service name="SaltGrainContainer">
      <operation name="getColor" returnType="string" returnedVar="color">
      </operation>
      <operation name="setColor" returnType="none" returnedVar="none">
         <param name="color" type="string"/>
      </operation>
   </Service>
   <Service name="CarrierStateManager">
      <operation name="setCarrierState" returnedType="none">
         <param name="state" type="string"/>
      </operation>
      <operation name="getCarrierState" returnedType="string">
      </operation>
   </Service>
   <Service name="SaltConcentrationCalculator">
      <operation name="calculateSaltConcentartion" returnType="float" returnedVar="concentration">
         <param name="saltConcentration" type="float"/>
      </operation>
      <Monitor name="Monitor_SaltConcentrationCalculator">
         <MonitorOperation Type="void" name="startMonitoring"/>
         <MonitoredVar name="saltConcentration" type="float"/>
         <Thrushold operator="Equal" value="0.97"/>
         <Event>High_Concentration</Event>
      </Monitor>
      <Policy name="HighConcentration">
         <Action name="HighConcentraion_Action">
               <Operation name="takeAction" type="void"/>
         </Action>
      </Policy>
   </Service>
</Domain>
```

Figure 8. 40: A simplified version of the salt world autonomic model (PIM).

8.3.4.4 The Platform Specific Autonomic Model (PSAM)

In this phase, the model obtained in the previous stage is transformed into a platform specific one by adding and injecting the terms and concepts for that platform, Java for instance. The transformer used to accomplish this task is coded in Java and referred to as *Abst2PlatformTransformer.java*. Figure 8.41 shows a simplified version of the platform specific autonomic model for the salt world domain.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!--
   Document   : SaltWorld_autonomic_java.xml
   Created on : 16 May 2009, 12:21
   Author     : LA_Abuseta
   Description:
      The Autonomic Model (PSM) of the Salt world domain
-->
<Domain name="saltworld">
   <class name="SaltGrainCarrier">
      <Method name="startGrainPicking" returnType="void"/>
      <Method name="startGrainDropping" returnType="void"/>
   </class>
   <class name="SaltGrainContainer">
      <Method name="getColor" returnType="string" returnedVar="color"/>
      <Method name="setColor" returnType="void" returnedVar="void">
         <param name="color" type="String"/>
      </Method>
   </class>
   <class name="CarrierStateManager">
      <Method name="setCarrierState" returnedType="void">
         <param name="state" type="String"/>
      </Method>
      <Method name="getCarrierState" returnedType="String"/>
   </class>
   <class name="SaltConcentrationCalculator">
      <Method name="calculateSaltConcentartion" returnType="float" returnedVar="concentration">
         <param name="saltConcentration" type="float"/>
      </Method>
   </class>
.....
.....
</Domain>
```

Figure 8.41: A simplified version of the salt world autonomic model (PSM).

8.3.4.5 The Autonomic Code Generation

As described earlier, generating autonomic code requires the application of two code generators. Since the target platform here is the Java web services, two Java code generators are applied here. The first generator which is encapsulated in the file JavaWSCodeGenerator.java is used to generate the core Java web services whereas the second generator is used to generate the autonomic Java web services and saved in the file AutonomicJavaWSCodeGenerator.java. The generated artifacts for the salt world domain can be found in appendix E.

8.4 Summary and Discussion

This chapter was dedicated to evaluate the proposed design method introduced in this thesis. Since a quantitative evaluation is very hard to perform only qualitative evaluation was carried out. Four case studies were introduced here to show and demonstrate the feasibility of our proposed design approach, namely, the *Online Travel Agency*, *Pet*

Store, Intelligent office and *Salt World* applications. Evaluation metrics were used hare include functionality, generality, adaptability and Ease of aspect accommodation.

As stated earlier, it is rather difficult to carry out a quantitative evaluation and make comparison between MDD based systems and systems developed using different methods or approaches. However, the qualitative evaluation has provided positive indications that the proposed design method along with the developed frameworks and reference models seem to fulfil the requirements specified and are generic and flexible enough to support the autonomic system designer with this very hard task. The flexibility and generality has been mainly achieved by adopting and applying the MDD approach. Using the Online Travel Agency case study, it was shown and proven the feasibility and applicability of the proposed design method. UML diagrams, XML files and Java web services files were presented here. The Intelligent Office case study, however, was developed to show that generality of the proposed designed method where the target platform was the C# web services. As for the adaptability and ease of change issue, the Pet Store case study was employed to show how to modify (add/remove components) the system in question. A new service was injected into the Intention Model and corresponding corrective actions were shown. These actions included adding the new service into a specific task, modifying the service interaction (composite) and notifying the other change controllers of such an event. The Salt World case study was employed here to evaluate the proposed design method against the engineering of the self organisation systems. It was shown that such systems can be accommodated and designed easily using the proposed design method.

CHAPTER 9

CONCLUSIONS

9.1 Motivations and Approach Summary

The explosive growth seen today in distributed applications and information services that affects our life in many aspects has been put down to the advances and remarkable progress made in networking, computing technology and software tools. The nature of these applications and services is extremely complex, heterogeneous and dynamic. As these systems and applications continue to show a growth in terms of the complexity and scalability, existing tools and methodologies have failed to show the capability and support needed to manage such systems. Researchers, therefore, had to consider alternative approaches to address this problem. They have adopted a set of strategies employed by the biological systems and their effort has resulted in the emergence of the autonomic computing paradigm. This paradigm was first introduced by IBM to the National Academy of Engineers at Harvard University in March 2001. Such a paradigm is inspired by the functioning of the human nervous system and its fundamental objective is to design software systems that exhibit autonomic and self managing capabilities. In such systems, the management and configuration tasks which are used to be performed by the system administrators are delegated to the software itself, supplied only with high-level policies, and thus shielding the administrators from carrying out these notoriously difficult tasks.

However, despite the rapid advancements in autonomic systems research and development, their design and engineering support is still under active research. Based on an ongoing research work focusing on open-standard design support for runtime software adaptation, the work included in this thesis presented our findings towards the development of a novel Model Driven Development (MDD) method for autonomic systems.

To cater for the increasing rate of software evolution and change, the proposed MDD-based method was proposed to provide a complete design lifecycle support starting from

the system requirements in terms of tasks (intentions or goals) and ends with an autonomic code generated by way of model transformation mechanisms. In such a paradigm, the software development process was entirely based on using models to represent all relevant information at some specific phases. Hence, the models were considered as the backbone of the development process and used not only for documentation purposes but also as a building tool. Each activity of the system development life-cycle involved taking some models as input and producing target models as output. There were four models in the development lifecycle process, namely the *Intention Model* (IM), Platform Independent Autonomic Model (PIAM), Platform Specific Autonomic Model (PSAM) and Autonomic Code (AC). The proposed design method presented here extended the emerging *Service Component Architecture (SCA)* standard, in that it provided support starting from the business process level all the way to the code generation stages. Hence, we referred to it as Autonomic, Task, Service and Component architecture (AutoTaSC) method.

9.2 Achievements and contributions

The work presented in this thesis made a number of contributions towards a better and deeper understanding of the requirements of the autonomic or self managing software systems design. However, the primary contribution of the research presented in this thesis is the reference autonomic development process. This development process or design method is augmented with the MDD paradigm to overcome and face the effect of the technology change and avoid any unnecessary early commitment to a specific technology. In the proposed development process, the conceptual stages and artifacts that should exist to help and guide system designers to design autonomic systems are defined and detailed. More precisely, four stages have been proposed and defined which are described as follows:

- The Intention Model (IM): in such a PIM stage, each task defined in the previous stage is represented in terms of a set of services. Also, the interaction flow between the involved services of each task is modelled here as *composite* and saved in a separate file. The set of services and the workflow defining their interaction and the order, in which these interactions must be fired, form the concept of the Business Process (BP). The system model in this stage takes the format of an XML file.

- The Platform Independent Autonomic Model (PIAM): in this PIM stage, the autonomic capabilities and required elements are attached to the system model obtained in the previous stage. Such elements include, in addition to others, the *monitor*, *policy*, *sensor* and *actuator*. Also the critical variables to be monitored and the control rules or policies are specified and saved in this model.
- The Platform Specific Autonomic Model (PSAM): in this PSM stage, the appropriate terms and data types for a specific platform, Java for instance, are added to the model obtained in the previous stage.
- The Autonomic code generation stage: in such a stage, a set of autonomic code skeletons is produced here for the platform specified and targeted in the previous stage.

In addition to the reference design method presented above, a number of other contributions were made which can be described below:

- *Transformation rules engine*: this includes a set of required files used to encapsulate the transformation rules necessary to generate one model from another. These files have been coded in Java. Here, transformation files can be, generally, classified into:
  - An abstract to Platform Independent Model (PIM) autonomic model transformation.
  - A Platform Independent Model (PIM) autonomic to Platform Specific Model (PSM) autonomic model transformation.
  - A Platform Specific Model (PSM) autonomic to code transformation.
- *Aspects profiles*: these profiles are defined to specify and encapsulate the required elements and components for the various aspects that the system under development might accommodate. Aspects include, in addition to others such as the security and Quality of Service (QoS), the four autonomic capabilities (*self-healing, self-protecting, self-configuring and self-optimising*).
- *Autonomic design styles*: In our approach to designing the autonomic systems we have proposed and employed two different design styles to introduce and inject

the autonomic capabilities, namely the *Service-Monitor-Controller (SMC)* and the *design by contract* styles.

- *Model synchronization mechanism:* A mechanism for realising the synchronisation between the involved models throughout the MDD based development process lifecycle is proposed and implemented. Our mechanism was inspired by the autonomic principles and particularly the manager and managed element relationship proposed by IBM blueprint. In such a mechanism, the managed element role is played by the XML models and the code while the management task is dedicated to the change manager or controller.

9.3 Thesis summary

This thesis comprises eight chapters and is organised as follows:

- Chapter 1: this chapter introduces the main motivations of the work presented in this thesis, challenges, contributions and thesis structure.
- Chapter 2: it introduces the relevant background theories, principles and technologies relevant to the work presented in this thesis. Such technologies and approaches include Model Driven Development (MDD), Autonomic systems, Service oriented Architecture (SOA) and Web Services, Business Process Oriented Development and its languages such as BPEL and BPMN, and Design by Contract (DBC).
- Chapter 3: This chapter was dedicated to reviewing the previous works on autonomic systems design that have been conducted so far by which the proposed design method presented in this thesis was inspired. In particular, this chapter reviewed some of the research that has been reported in the literature relevant to model-based and model-driven development methods for autonomic software, some of which have adopted a range of paradigms, theories and/or architectural models.
- Chapter 4: this chapter introduced the proposed approach and design method for modelling and engineering autonomic systems from a conceptual perspective. Only the architecture, concepts and high level design decisions and ideas were introduced and described here.

- Chapter 5: this chapter presented a detailed description of our design method. Here, a number of aspects were covered including the whole lifecycle of our design method and justification for design decisions and technology choices. The whole design method lifecycle process was described here in a number of stages according to the MDD approach.
- Chapter 6: this chapter introduced and elaborated on the model transformation framework that have been proposed and developed. The three fundamental components of this framework were presented and described in detail. These components included the abstract to autonomic transformer, the PIM autonomic to PSM autonomic transformer and the autonomic code generation transformer.
- Chapter 7: this chapter introduced and described the model synchronisation framework that was proposed and developed to synchronise the different models of the proposed lifecycle once a change has been performed on any of these models.
- Chapter 8: This chapter was dedicated to present an evaluation of the autonomic design method introduced in this thesis. The evaluation contains qualitative analysis of the design method proposed here. This included an evaluation of the design method lifecycle process in general in addition to some critical analysis to some models that have been adopted and developed in some particular stage throughout the MDD different stages.
- Chapter 9: the thesis concluded at this chapter where a summary, what has been achieved and contributions were presented as well as some proposed future work.

9.4 Conclusion and Discussion

Autonomic systems model has emerged to address the increasing systems' design and administration complexity. Since their emergent in 2001, numerous design and implementation approaches have been proposed. These approaches fall into a number of categories including:

- Reference models for autonomic systems (IBM blueprint, J-Reference).
- Aspect oriented programming based approaches.

- Design by contract based approaches.
- Model based approaches.
- Model driven development approaches.

To tackle some of the outstanding research issues, this research work proposed a design method for autonomic systems based on the Model Driven Development (MDD) approach. The latter provides support to the system designer throughout the lifecycle of the development process starting from the system requirements stage and all the way to the last stage where a set of useful skeletons of code is produced. Support is also provided with regard to the system modification where a set of changes and modifications can be performed by the system stakeholder, be it a designer, programmer or domain expert. Carrying out changes on one model of the lifecycle leaves the system in an inconsistent state unless a mechanism has been put in place to address this issue. This, too, is provided here via proposing a model synchronisation framework where a network of controllers is created in which each controller of a specific model has the responsibility of informing, and also taking actions, other controllers should any changes have been sensed.

9.5 Comparative evaluation of proposed design method

Below is a comparison between what has been achieved by conducting this research and some related works. The comparison is based on the two following aspects:

- The MDD based development process lifecycle.
- The model synchronisation mechanisms.

For the first aspect of the comparison, a few approaches have taken this trend and adopted the MDD for designing autonomic systems. However, these approaches tend to neglect the domain expert role and make an early commitment to one specific platform or programming language. This aspect was addressed in this thesis via introducing a higher level stage where the system requirements can be specified and defined in an abstract and platform neutral manner.

Also, existing MDD based approaches adopt the UML paradigm as the tool to express their systems in a platform independent manner. Then, such UML diagrams, typically class diagrams, are converted into XMI files, which are in turn processed by a specific program (code generator) to produce the artifacts for the target platform. Two drawbacks

can be identified here. The first is connected with the use of the XMI as the format of the system under study. Such a format is intended primarily for exchange UML diagrams between different CASE tools and this format is not guaranteed to be standard which obviously affects the code generator that is applied to it. The second drawback is related to the use of the UML diagrams for the Platform Independent Model (PIM). Consequently, runtime system modification is very difficult to accomplish since such a task enforces the domain expert or system designer to switch to different environment (UML CASE tool), make the required modifications and then convert it to produce the updated XMI format. These issues were addressed in this thesis by expressing the system at the PIM using the XML format instead of the UML. An appropriate user interface is provided to facilitate and simplify the system modification which eliminates the need for working in and switching between different environments to make only a small change. Since the XML is an open standard and can be interpreted in a similar way by different tools, an expected behaviour by these tools can be guaranteed. Also, necessary concepts and elements can be introduced easily using the XML and this is particularly important when it comes to defining autonomic profiles or other non functional system requirements.

For the second aspect, it is believed that there is not much work on the issue of model synchronisation. Even when it is addressed, approaches tend only to focus on and support one way synchronisation. Furthermore, the adoption of the autonomic computing principle for supporting the modification and change aspect in MDD has not been used elsewhere by any of the existing approaches. Using the autonomic systems ideas here is beneficial since the model modification occurs iteratively and is triggered by many actions. The obvious action of change comes from the software components (code) where one service is added or removed at runtime in response to one specific event. In the absence of such an autonomic synchronisation framework, it is very hard to sense these modifications and take the appropriate corrective and responsive action.

9.6 Future work

This section presents a suggested further works to extend the developed method and its associated tools, including:

- The proposed and developed design method along with proposed models and design patterns have been tested and evaluated using three case studies. These can be extended to include more applications and case studies.
- Extending the proposed *design method* and associated tools to include other non-functional aspects.
- Further testing the model for runtime injection of new services and aspects with a specific focus on the requirements of critical systems. In addition, the method could be studied to support the emerging Cloud computing model.
- Further study can be undertaken to investigate the practical software engineering aspect of the method in terms of flexibility, efficiency and effectiveness of the method when dealing with complex and critical systems development.
- Further study can focus on the user interface aspect to enhance the ease of use of the method and its associated tools when dealing with autonomic systems.

APPENDICES

APPENDIX A:

Design by contract

Design by Contract (DBC) is a software engineering technique of building quality and reliable software systems by explicitly specifying what each function or operation in a system requires in order to accomplish its task, and what it guarantees to provide in return to its client [82]. The term was coined by Bertrand Meyer in connection with his design of the Eiffel programming language and first described in various articles starting in 1986 and the two successive editions of his book *Object-Oriented Software Construction* [83]. In DBC, a class and its clients are bound by a contract. Such a contract states that the client must meet some conditions to call a method defined by the class and in return the class guarantees some properties that will hold upon completion of method execution [84, 85]. The contract here is often specified by *pre conditions* and *post conditions* to define the client's obligations and benefits respectively. These two conditions can be described as follows:

- *A precondition* states the constraints under which a routine will function properly. It is an obligation for the client and a benefit for the supplier.
- *A post condition* expresses properties of the state resulting from a routine's execution. It is an obligation for the supplier and a benefit for the client [86].

The third form of condition which can be used is the class *invariant*. It is a property that must be applied to every instance of the class [87]. More precisely, it is a condition that must hold before and after any public methods [88]. Unlike the pre and post conditions, the invariants can only be applied within the object-oriented approach [87]. In Eiffel, pre conditions and post conditions are represented by keywords *require* and *ensure* respectively. Listing A.1 shows a DBC representation in this language.

Listing A. 1: DBC in Eiffel

```
put_child (new: NODE) is
          --- Add new to the children of current node

     require
```

```
            new |= void
    do
            --- Insertion algorithm
     Ensure
                  new.parent = Current;
                    child-count = old child-count + 1

    end-- put_child
```

However, in other programming languages, different concepts and terms are used to represent such contract elements. In Java, for example, where the DBC capability was not supported from the beginning, a number of approaches have been proposed and developed each with its own terminology. Listings A.2, A.3 and A.4 show some of these approaches.

Listing A. 2: DBC in Jass (Java assertions)

```
public class A implements Cloneable {
protected int a;
public void addValue (int x) {
/** require x > 0; **/
a = a + x;
/** ensure Old.a > a; a > 0; **/
}
protected Object clone () {...}
/** invariant a > 0; **/
}
```

Listing A. 3: DBC in jContractor

```
class ClassName ... {
Object methodName (Object x, String key) {
/* method body */
}
protected boolean methodName_PreCondition (Object x, String key) {
/* Precondition of above method */
}
protected boolean methodName_PostCondition (Object x, String key) {
/* Postcondition of above method */
}
protected boolean className_ClassInvariant () {
/* Class invariant */
}}
```

Listing A. 4: DBC in iContract

```
/**
* @pre o != null;
* @post list.size () == list.size () @pre + 1;
*/
void append (Vector list, Object o) {...}
```

Benefits of Design by Contract

The benefits of Design by Contract include the following:

- A better understanding of the software systems and, more generally, of software construction.
- A systematic approach to building bug-free software systems.
- An effective framework for debugging, testing and, more generally, quality assurance.
- A method for documenting software components.
- A technique for dealing with abnormal cases, leading to a safe and effective language construct for exception handling.

Most of the benefits listed above are of great relevance to the research issue presented in this thesis, the autonomic or self-managing systems enabling.

APPENDIX B:

Business Process Oriented Modelling

As stated earlier in Section 2.3, in the Service-oriented Architecture (SOA) paradigm, software components are modelled and exposed as services. These services offer standard interfaces and can be invoked via message exchanging mechanisms. A number of relevant services can be coordinated to achieve and address specific business functionality. The coordination and integration role is played by a business process that takes the responsibility for orchestrating the involved services. A typical business process consists of a control flow, a number of service invocations and other activities for data processing, fault and transaction handling, and so on [89]. A definition of a business process presented in [90], states that:

> *"A business process is a collection of coordinated service invocations and related activities that produce a business result, either within a single organization or across several. "*

In another definition in [91]:

> *"A process is a type of complex activity that defines its own context for execution. Like other complex activity types, it is a composition of activities and it directs the execution of these activities. A process can also serve as an activity within a larger composition, either by defining it as part of a parent process or by invoking it from another process."*

Both definitions are concerned with the coordination between a number of parties participating in a set of activities in order to address one specific system functionality. However, the second definition is more generic since the first one defines the business process from the SOA perspective.

Business process classification

Business processes can be classified into two kinds, namely the *abstract and executable*. The abstract processes are similar to the APIs in that they only describe what the process does and its required input and expected output; they do not go into details and describe how things done. In contrast, the executable processes, consists of all of the execution steps required for representing a cohesive unit of work [92] .

Business process modelling languages

A number of modelling languages already exist for the purpose of defining and developing business processes. Such languages include Event-Driven Process Chains (EPC) [93], Business Process Modelling Notation (BPMN, WS-BPEL and XPDL[94].

BPEL4WS and WS-BPEL

The Business Process Execution Language for Web Services (BPEL4WS) is an orchestration language whose primary goal is to define an execution format for business processes operating on web services [21]. This language was inspired by previous variations such as IBM's Web Services Flow Language (WSFL) and Microsoft's XLANG specification. The first release of BPEL4WS specification appeared in July 2002, which was a result of a joint effort by IBM, Microsoft and BEA. In May 2003, less than a year later, BPEL4WS1.1 specification was released with other contributors joining from SAP and Siebel Systems. More attention and vendor support was given to this version which led to a number of commercially available BPEL4WS-complaint orchestration engines. An official and open standard version of the BPEL4WS was accomplished by the OASIS technical committee which led to the release of version 2.0 where this language was renamed to the Web Services Business Process Execution Language, or WS-BPEL [21]. This XML programming language has three fundamental components, including:

- Programming logic- **BPEL**
- Data types- **XSD (XML Schema Definition)**
- Input/output (I/O) - **WSDL (Web Services Description Language)**

A BPEL business process is made of the following three main entities:

- The partners that abstractly represent the services involved in the composition.
- The variables used to manipulate the data (SOAP messages) exchanged between partners and to hold states of the business logic. XPath expressions can be used to access a part of a variable or to test conditions.
- The activities that describe the business logic. They can be basic such as invoking a Web Service or assigning a value to a variable, or structured such as executing a set of activities in sequence or in parallel.

Business Process Modelling Notation

BPMN, first developed by the Business Process Management Initiative (BPMI), is a standardised graphical notation for drawing business processes in a workflow. It is being maintained by the Object Management Group (OMG) since its merger with the Business Process Management Initiative in 2005. The core objective of BPMN is to provide a standard notation that can be understood by all business stakeholders. These business stakeholders include the business analysts who create and refine the processes, the technical developers responsible for the processes implementation, and the business managers who monitor and manage the processes. In other words, BPMN was developed to serve as a common language to bridge the communication gap that frequently occurs between business process design and implementation [95, 96].

APPENDIX C:

Development Environment Description

Here, we introduce the development environment that was developed in Java to show and demonstrate the feasibility of our design method, the subject of this thesis. A set of screenshots and interactive dialogs is shown here which shows how the system user (autonomic designer) can interact with the development environment to generate the necessary models for the MDD based autonomic systems.

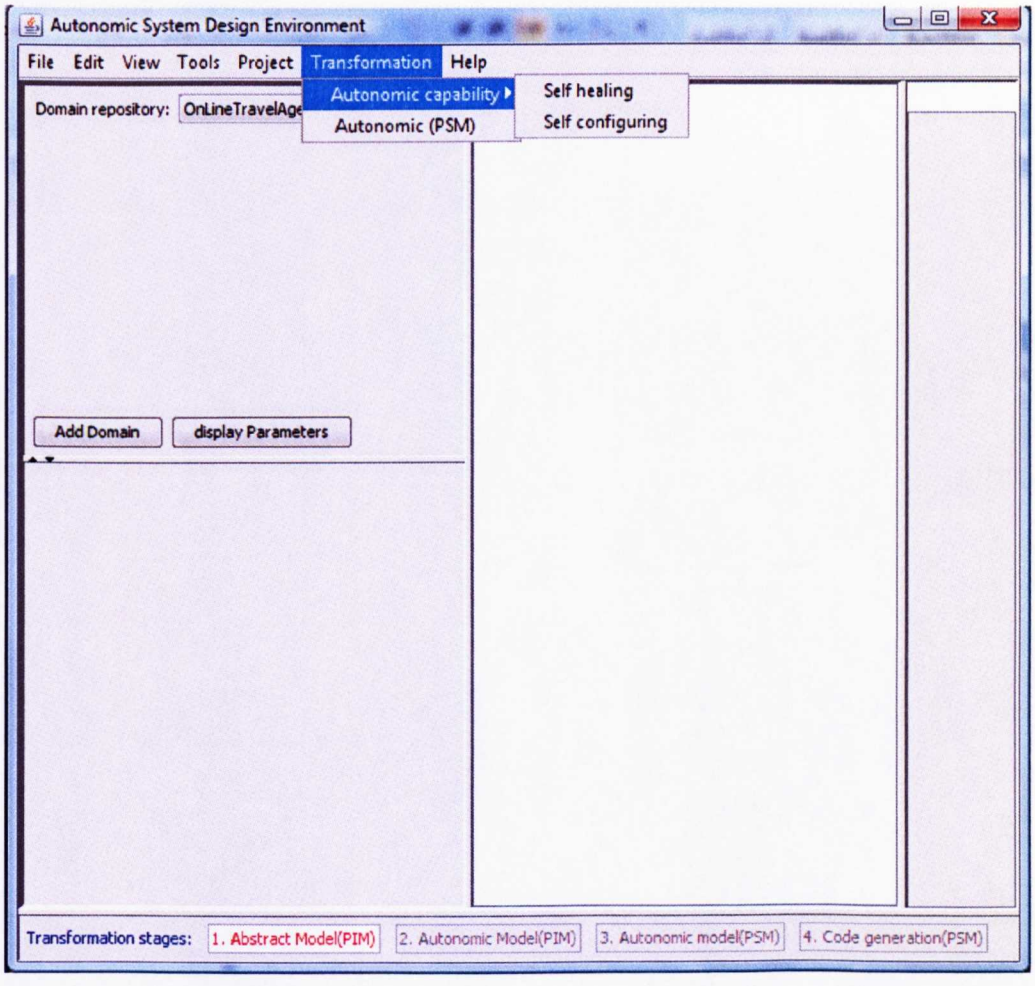

Figure C. 1: the start-up window of our development environment.

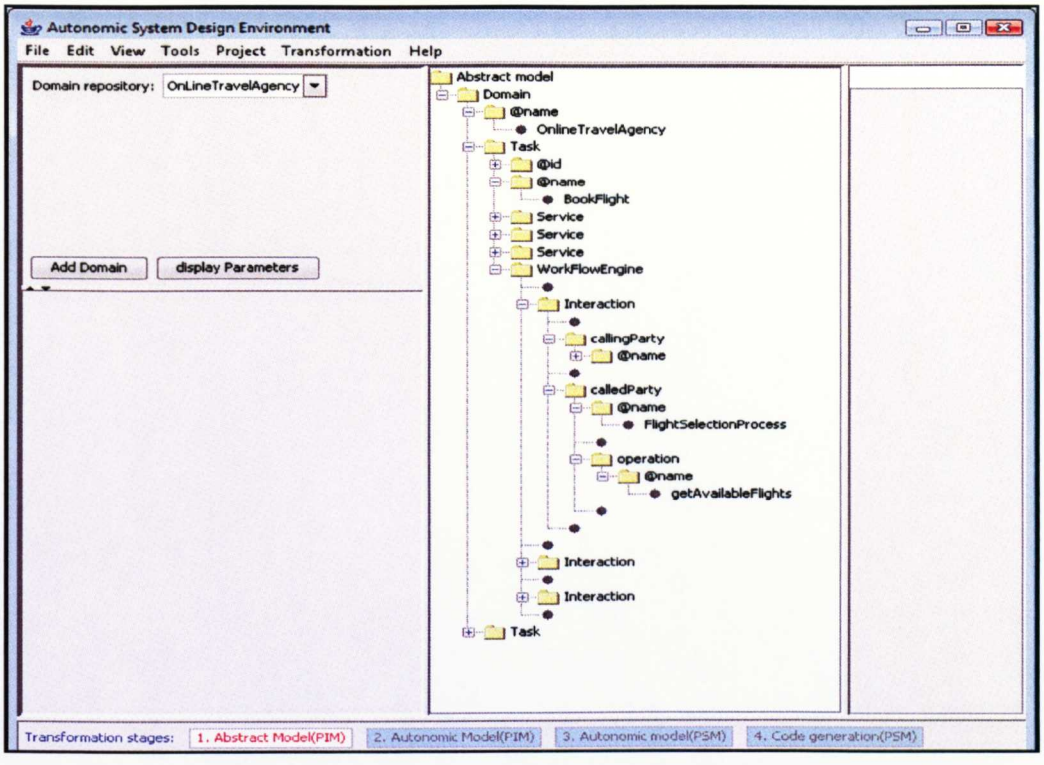

Figure C. 2: The Abstract model stage (The Intention Model) (PIM).

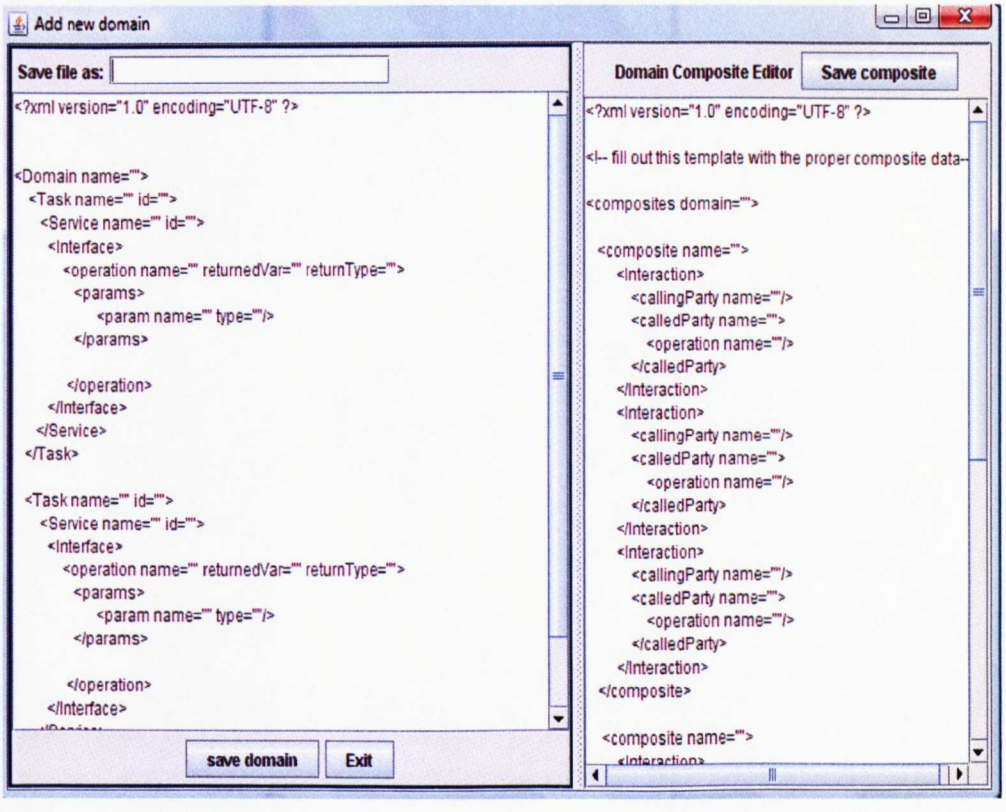

Figure C. 3: The process of creating new domain.

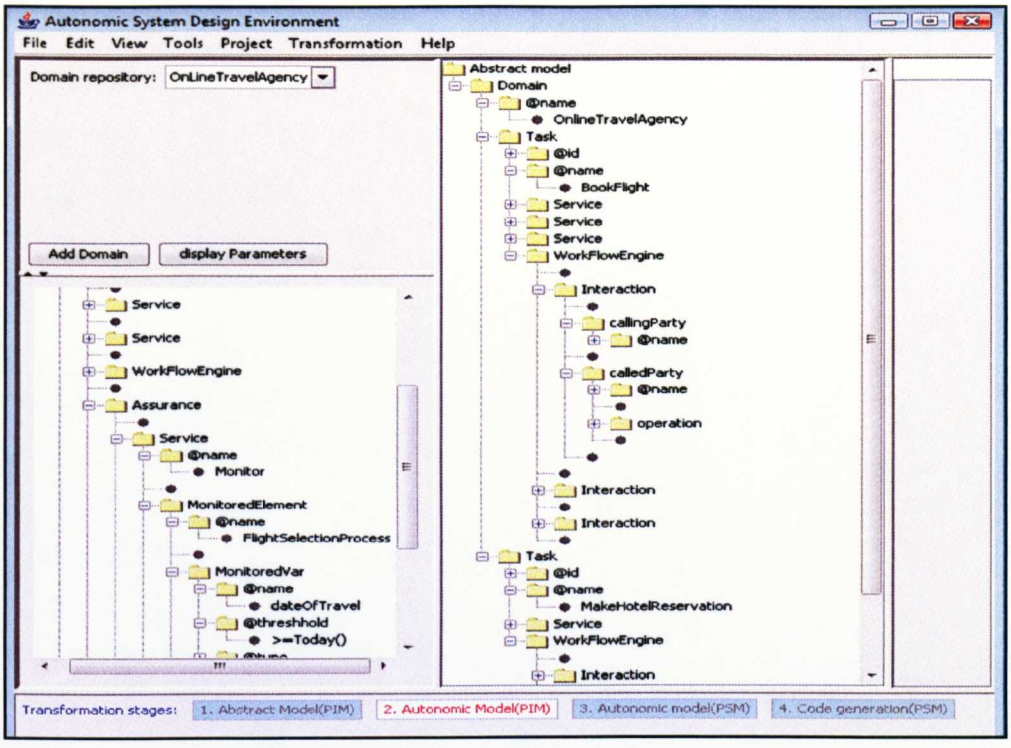

Figure C. 4: the Autonomic model (PIM).

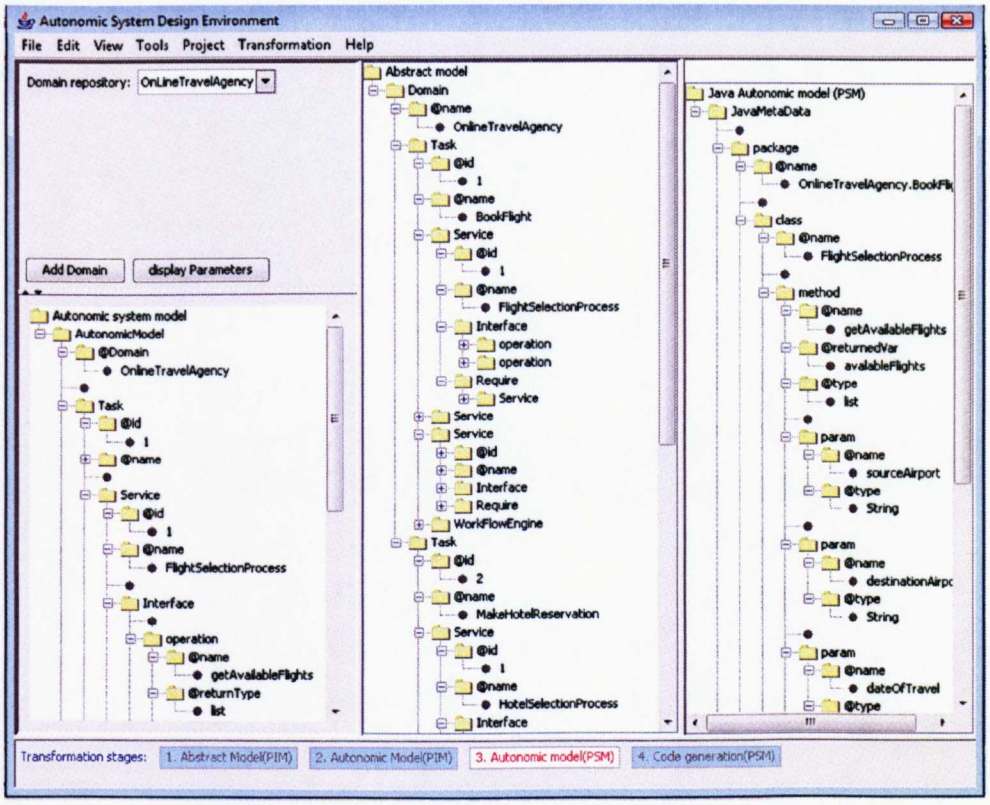

Figure C. 5: the autonomic model (PSM).

APPENDIX D:

Software Design Patterns

In [97], a *design pattern* is defined as "*a documented best practice or a core of a solution that has been applied successfully in multiple environments to solve a problem that recurs in a specific set of situations*". Below, a set of design patterns is introduced which was used throughout the project presented in this thesis in a number of models and architecture. These design patterns include the observer, smart proxy and Model-View-Controller (MVC).

- **The Observer Design Pattern**: it is a software design pattern in which an object maintains a list of its dependents and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems. Here, the set of dependents are referred to as observers and the object that they are dependent on is called the subject. The observer pattern belongs to the behavioural patterns group. To enable this mechanism and relationship, the following list of requirements should be met:

    - The *subject* should provide an interface for registering and unregistering change notifications.
    - One of the following two states must be true:
        - *The pull model*: here, the *subject* provides an interface for *observers* to query for the required state information so they can update their state.
        - *The push model*: *the subject* is responsible for sending the state information that the *observers* may be interested in.
    - Observers should provide an interface for receiving notifications from the subject [98]

The structure of the different classes along with their association, catering to the above listed requirements, is depicted in Figure D.1.

Figure D. 1: Class Diagram for Observer Design Pattern [99].

- **The Proxy Design Pattern:** The proxy design pattern is a structural design pattern that creates a surrogate, or placeholder class. Proxy instances accept requests from client objects, pass them to the underlying object and return the results. Proxies can improve efficiency and enhance functionality. This can improve the efficiency of access to objects that are too expensive to access directly either because they are slow to execute or are resource-intensive, or because extra functionality needs to be added [99, 100]. Proxy design pattern takes a number of forms depending on the issue that each form intends to address. Common used forms include *cache proxy, protection proxy, virtual proxy, remote proxy and smart proxy* [97, 99, 100].

Figure D. 2: Proxy Design Pattern Class Diagram [99].

- **Model-View-Controller (MVC):** The Model-View-Controller architecture is a widely used architectural approach for interactive applications. It divides

functionality among objects involved in maintaining and presenting data to minimize the degree of coupling between the objects [80]. Figure D.3 shows the three fundamental components of this architecture, the model, view and controller.

Figure D. 3: The Model-View-Controller Architecture [80].

APPENDIX E:

Generated Artifacts for Evaluation Case Studies:

Online Travel Agency (OTA) case study

Listing E.1: A proxy class for PaymentCardValidator Java web service.

```
public class PaymentCardValidator_Proxy {
    /*creates a new instance of PaymentCardValidator_Proxy */
        public PaymentCardValidatorr_Proxy(){
             }
       public boolean verifyPaymentCard(String cardNo,
        String securityNo)
             {
                boolean successful;
       try{//Call Web Service Operation
          PaymentCardValidatorrService service =  new
PaymentCardValidatorrService();
          PaymentCardValidatorr port =
service.getPaymentCardValidatorrPort();
          successful = port.verifyPaymentCard(String cardNo,
String securityNo);
           }
   catch(Exception ex){
                    }
          }
}
```

Listing E.2: A JSP file for PaymentCardValidator proxy invocation.

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<html>
    <head> <title>Payment card verification</title> </head>
    <body>
        <%-- start web service invocation -- %>
     <c:if test="
            <%=olta.PaymentCardValidator_Proxy.verifyPaymentCard
("2356 77","231") %>">
        <%out.println ("Payment card has been successfully
verified") ;%>
    </c:if>
 <%-- end web service invocation -- %>
   </body>
</html>
```

Listing E.3: A generated skeleton of code for PaymentCardValidator web service.

```
/*****************************
*Developer: Yousef Abuseta***
*University: LJMU            *
*****************************/
package BookFlight;

import java.util.*;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService()
public class PaymentCardValidator {

    @WebMethod()
    public boolean verifyPaymentCard( String cardNo, String securityNo
){
    boolean successful= false;

    //Implementation code goes here...

    return successful;
  }
}
```

Listing E.4: A generated skeleton for the FlightSelectionProcess monitor.

```
public class Monitor implements Observer {
private FlightSelectionProcess_proxy proxy;
public Monitor (FlightSelectionProcess_proxy proxy) {
this.proxy = proxy;
proxy.register(this);
}
public void receiveStateChange (Observable subject) {
    if (subject == this.proxy)
        Date dateOfTravel = this.proxy.getdateOfTravel();
        if (dateOfTravel< Date())
            //send INVALID_DATE_EVENT to controller
}
```

```
<? xml version="1.0" encoding="UTF-8" standalone="no"?>
<Domain name="OnlineTravelAgency">
        <Service id="1" name="FlightSelectionProcess">
            <Interface>
               <operation name="getAvailableFlights" returnType="list"
returnedVar="availableFlights">
                    <params>
                        <param name="sourceAirport" type="Text"/>
                        <param name="destinationAirport" type="Text"/>
                        <param name="dateOfTravel" type="Date"/>
                    </params>
            </operation>
```

```
            <operation name="reserveSeatOnFlight" returnType="boolean"
returnedVar="done">
                    <params>
                        <param name="fName" type="Text"/>
                        <param name="lName" type="Text"/>
                        <param name="flightNo" type="Text"/>
                    </params>
                </operation>
            </Interface>
        </Service>
        <Service id="2" name="CustomerCharger">
            <Interface>
              <operation name="calculateTicketCost" returnType="long"
returnedVar="ticketCost">
                    <params>
                        <param name="flightNo" type="Text"/>
                    </params>
                </operation>
            </Interface>
        </Service>
        <Service id="3" name="PaymentCardValidator">
            <Interface>
              <operation name="verifyPaymentCard" returnType="boolean"
returnedVar="successful">
                    <params>
                        <param name="cardNo" type="Text"/>
                        <param name="securityNo" type="Text"/>
                    </params>
                </operation>
            </Interface>
        </Service>
        <Service id="1" name="HotelFinder">
            <Interface>
                <operation name="getAvailableHotels" returnType="list"
returnedVar="hotels">
                    <params>
                        <param name="airportCode" type="Text"/>
                    </params>
                </operation>
                <operation name="reserveRoom" returnType="void"
returnedVar="void">
                    <params>
                        <param name="hotelID" type="Text"/>
                        <param name="dateOfArrival" type="Date"/>
                        <param name="noOfNights" type="int"/>
                    </params>
                </operation>
            </Interface>
        </Service>
</Domain>
```

Figure E. 1: Intention Model for Online Travel Agency domain.

```
<?xml version="1.0" encoding="UTF-8" ?>
<composites domain="OnLineTravelAgency">
     <composite name="BookFlight">
        <Interaction>
            <callingParty name="user"/>
            <calledParty name="FlightSelectionProcess">
                <operation name="getAvailableFlights"/>
            </calledParty>
        </Interaction>
        <Interaction>
            <callingParty name="user"/>
            <calledParty name="FlightSelectionProcess">
                <operation name="reserveSeatOnFlight"/>
            </calledParty>
        </Interaction>
        <Interaction>
            <callingParty name="FlightSelectionProcess"/>
            <calledParty name="PaymentCardValidator">
                <operation name="verifyPaymentCard"/>
            </calledParty>
        </Interaction>
    </composite>
    <composite name="MakeHotelReservation">
        <Interaction>
            <callingParty name="user"/>
            <calledParty name="HotelSelectionProcess">
                <operation name="getAvailableHotels"/>
            </calledParty>
        </Interaction>
        <Interaction>
            <callingParty name="user"/>
            <calledParty name="HotelSelectionProcess">
                <operation name="reserveRoom"/>
            </calledParty>
        </Interaction>
        <Interaction>
            <callingParty name="HotelSelectionProcess"/>
            <calledParty name="PaymentCardValidator">
                <operation name="verifyPaymentCard"/>
            </calledParty>
        </Interaction>
    </composite>
</composites>
```

Figure E. 2: Service Composites for Online Travel Agency domain.

The Pet store case study

```
<? xml version="1.0" encoding="UTF-8"?>
<Domain name="PetStore">
       <Service name="SalesPoint" id="1">
         <Interface>
             <operation name="browsePetInventory" returnedVar="void"
returnType="void">
                 <params>
```

```
                    <param name="na" type="na"/>
                </params>
              </operation>
               <operation name="buyPet" returnedVar="ok"
returnType="boolean">
                <params>
                      <param name="petName" type="String"/>
                       <param name="noOfPets" type="integer"/>
                </params>
               </operation>
        </Interface>
      </Service>
       <Service name="PaymentCardValidation" id="2">
         <Interface>
             <operation name="verifyCard" returnedVar="verified"
returnType="boolean">
                <params>
                      <param name="cardNo" type="String"/>
                       <param name="securityNo" type="String"/>
                </params>
              </operation>
         </Interface>
      </Service>
 </Domain>
```

Figure E.3: The Intention Model for Pet Store domain.

```
<?xml version="1.0" encoding="UTF-8" ?>
<composites domain="PetStore">
  <composite name="SellPetsToCustomer">
        <Interaction>
            <callingParty name="user"/>
            <calledParty name="SalePoint">
                <operation name="browsePetInventory"/>
            </calledParty>
        </Interaction>
        <Interaction>
            <callingParty name="user"/>
            <calledParty name="SalePoint">
                <operation name="buyPet"/>
            </calledParty>
        </Interaction>
        <Interaction>
            <callingParty name="SalePoint"/>
            <calledParty name="PaymentCardValidator">
                <operation name="verifyPaymentCard"/>
            </calledParty>
        </Interaction>
    </composite>
    <composite name="OrderPetFromSupplier">
        <Interaction>
            <callingParty name="InventoryChecking"/>
            <calledParty name="PetInventoryManager">
                <operation name="checkInventory"/>
            </calledParty>
        </Interaction>
        <Interaction>
```

```
            <callingParty name="InventoryChecking"/>
            <calledParty name="PetSupplier">
                <operation name="getAvailablePets"/>
            </calledParty>
        </Interaction>
        <Interaction>
            <callingParty name="InventoryChecking"/>
            <calledParty name="PetSupplier">
                <operation name="buyPet"/>
            </calledParty>
        </Interaction>
    </composite>
</composites>
```

Figure E.4: Service composites for Pet Store domain.

The Intelligent Door case study

```
<?xml version="1.0" encoding="UTF-8" ?>
<Domain name="IntelligentDoor">
       <Service name="AccessInterface" id="1">
         <Interface>
             <operation name="acceptUserPassword" returnedVar="message"
returnType="Text">
                <params>
                      <param name="password" type="Text"/>
                </params>
              </operation>
         </Interface>
      </Service>
      <Service name="OfficeDoor" id="2">
         <Interface>
             <operation name="open" returnedVar="void"
returnType="void"/>
             <operation name="close" returnedVar="void"
returnType="void"/>
         </Interface>
      </Service>
           <Service name="thermometer" id="1">
         <Interface>
             <operation name="getTemperature" returnedVar="temperature"
returnType="float"/>
         </Interface>
      </Service>
   </Domain>
```

Figure E.5: The Intention Model for IntelligentOffice domain.

```
<?xml version="1.0" encoding="UTF-8" ?>

<composites domain="IntelligentOffice">

   <composite name=" EnforceAutorisedAccess ">
        <Interaction>
            <callingParty name="user"/>
            <calledParty name="AccessInterface">
                <operation name="acceptUserPassword"/>
            </calledParty>
        </Interaction>
```

```
        <Interaction>
            <callingParty name="AccessInterface"/>
            <calledParty name="PasswordVerifier">
                <operation name="verifyPassword"/>
            </calledParty>
        </Interaction>
        <Interaction>
            <callingParty name="AccessInterface"/>
            <calledParty name="OfficeDoor">
                <operation name="setDoorStatus"/>
            </calledParty>
        </Interaction>
    </composite>
    <composite name=" ControlOfficeTemperature ">
        <Interaction>
.....................................................
    </composite>
</composites>
```

Figure E.6: Service composites for Intelligent Office domain.

The Salt World case study

```
/*****************************
*Developer: Yousef Abuseta** *
*University: LJMU            *
*****************************/

import java.util.*;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService()
public class SaltGrainContainer {
    private String color;

   @WebMethod()
    public void setColor( String color){
    This.color = color;

      //Implementation code goes here...
    }
    @WebMethod()
    public String getColor(){

      //Implementation code goes here...

    return this.color;
    }
  }
```

Figure E.7: Generated Java web service for SaltGrainContainer service.

```
/*****************************
*Developer: Yousef Abuseta** *
*University: LJMU            *
*****************************/

import java.util.*;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService()
public class SaltConcentrationCalculator {

    @WebMethod()
    public float calculateSaltConcentartion (){

    //Implementation code goes here...

      return Concentration;
    }
  }
```

Figure E.8: Generated Java web service for SaltConcentrationCalculator service.

```
/*****************************
*Developer: Yousef Abuseta** *
*University: LJMU            *
*****************************/
import java.util.*;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService()
public class CarrierStateManager {
    private String state;

    @WebMethod()
    public void setCarrierState (String state){
    this.state = state;

    //Implementation code goes here...

    }
    @WebMethod()
    public String getCarrierState (){

    //Implementation code goes here...

     return state;

   }
}
```

Figure E.9: Generated Java web service for CarrierStateManager service.

```
/****************************
*Developer: Yousef Abuseta** *
*University: LJMU            *
****************************/


import java.util.*;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService()
public class SaltGrainCarrier {

    @WebMethod()
    public void startGrainPicking (){

    //Implementation code goes here...

    }

    @WebMethod()
    public void startGrainDropping (){

    //Implementation code goes here...


   }
}
```

Figure E.10: Generated Java web service for SaltGrainCarrier service.

```
public class Monitor implements Observer {
   private SaltConcentrationCalculator_proxy proxy;
   public Monitor (SaltConcentrationCalculator_proxy proxy) {
      this.proxy = proxy;
      proxy.register(this);
  }
   public void receiveStateChange (Observable subject) {
    if (subject == this.proxy)
       float concentration = this.proxy. calculateSaltConcentartion ();
        if (concentration < 0.97)
            //send HIGH_CONCENTRATION_EVENT to controller
  }
}
```

Figure E.11: The monitor service for the SaltConcentrationCalculator service.

REFERENCES

1. Mellor, S., Balcer, M., *Executable UML: A Foundation for Model-Driven Architecture*. 2002: Addison-Wesley Professional.
2. Mellor, S., Scott, K., Uhl, A., Weise, D., *MDA Distilled: Principles of Model Driven Architecture*. 2004: Addison-Wesley.
3. Guelfi, N., Ries, B., Sterges, P. *MEDAL: A CASE Tool Extension for Model-Driven Software Engineering.* in *The IEEE International Conference on Software-Science, Technology & Engineering*. 2003. Washington, DC, USA: IEEE Computer Society.
4. Mille, J., Mukerji, J., *MDA Guide Version 1.0.1*. 2003, OMG.
5. Gardner, T., Yusuf, L. *A closer look at model-driven development and other industry initiatives.* 2006 [cited 2007; Available from: http://www.ibm.com/developerworks/library/ar-mdd3/.
6. Czarnecki, K., Helsen, S. , *Feature-based survey of model transformation approaches.* IBM Systems Journal 2006. **45**(3): p. 621 - 645
7. Reiter, T., Retschitzegger, W., Kapsammer, E., *A Generator Framework for Domain Specific Model Transformation Languages*, in *The English International Conference on Enterprise Information Systems*. 2006, INSTICC Paphos, Cyprus p. 27-35.
8. Czarnecki, K., Helsen, S. *Classification of Model Transformation Approaches.* in *The OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*. 2003. California, USA: OOPSLA.
9. Boocock, P. *Jamda Model Compiler Framework.* 2003 [cited 2007; Available from: http://jamda.sourceforge.net/docs/index.html.
10. JMatrix. *JET Code Generator.* 2003 [cited 2008; Available from: http://jmatrix.net/content/jetgen_home.jsp.
11. Van Erode Boas, G., *The Fantastic, Unique, UML Tool for the Java Environment (FUUT-je)*. 2008.
12. Manyta. *Codagen Architect.* 2004 [cited 2007; Available from: http://www.manyeta.com/en/Technology/codagen_architect_v3.2.

13. AndroMDA. *AndroMDA.* 2003 [cited 2008; Available from: http://www.andromda.org.
14. ArcStyle. 2008 [cited 2008; Available from: http://www.interactive-objects.com/products/.
15. Hendry, B. *Compuware Corporation's Optimal J,* 2001 [cited 2007; Available from: http://java.sys-con.com/node/36300.
16. Parr, T. *Enforcing Strict Model View Separation in Template Engines.* in *The 13th international conference on World Wide Web* 2004. NewYork, USA: ACM.
17. Naccarato, G. *Template-Based Code Generation with Apache Velocity.* 2004 [cited 2008; Available from: http://www.onjava.com/pub/a/onjava/2004/05/05/cg-vel1.html.
18. Giese, H., Waqner, R., *Incremental Model Synchronization with Triple Graph Grammars,* in *Model Driven Engineering Languages and Systems.* 2006, Springer Berlin / Heidelberg. p. 543-557.
19. OASIS, *Reference Model for Service Oriented Architecture 1.0.* 2006, OASIS.
20. Mahmoud, Q. *Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI).* 2005 [cited 2006; Available from: http://java.sun.com/developer/technicalArticles/WebServices/soa/.
21. Erl, T., *Service-Oriented Architecture: Concepts, Technology, and Designl.* 2005: Prentice Hall.
22. Actional. *Service-oriented Architecture (SOA)* 2007 [cited 2008; Available from: http://www.actional.com/soa/.
23. Cao, J., Wang, J., Zhang, S., Li, M., *A dynamically reconfigurable system based on workflow and service agents.* Engineering Applications of Artificial Intelligence, 2004. **17**(7): p. 771-782.
24. Arthur, J. *Autonomic SOA Web Services - Achieving Fully Business-Conscious IT Systems.* 2005 [cited 2006; Available from: http://soa.sys-con.com/node/136205.
25. Singhal, A., Winograd, T., Scarfone, K., *Guide to Secure Web Services.* 2007, National Institute of Standards and Technology, U.S. Department of Commernce: Gaithersburg, MD, USA.
26. Monday, P., *Web Service Patterns: Java Edition.* 2003: Apress.
27. Gunzer, H., *Introduction to Web services.* 2002, Borland Software Corporation: CA, USA.

28. Gottschalk, K., Graham, S., Kreger, H., Snell, J., *Introduction to Web services architecture.* IBM Systems Journal, 2002. **41**(2): p. 170-177.

29. Huebscher, M., McCann, J., *A survey of autonomic computing—degrees, models, and applications.* ACM Comput.Surv, 2008. **3**(40): p. 1-28.

30. Tesauro, G., Chess, D., Walsh, W., *A Multi-Agent Systems Approach to Autonomic Computing,* in *the Third International Joint Conference on Autonomous Agents and Multiagent Systems.* 2004, IEEE Computer Society New York p. 464-471.

31. IBM, *Automatic problem determination: A first step toward self-healing computer systems.* 2003, IBM.

32. IBM, *Autonomic Computing.* 2005, IBM.

33. Parashar, M., Hariri, S., *Autonomic Computing: Concepts, Infrastructure & Applications.* 2006: CRC Press.

34. Bantz, D., Bisdikian, C., Challener, D., Karidis, J., Mastrianni, S., *Autonomic personal computing.* IBM Systems Journal, 2003. **1**(42): p. 165–176.

35. Russel, S., Norvig, P., *Artificial Intelligence: A Modern Approach.* 2003: Prentice Hall.

36. Kephart, J., Chess, D., *The Vision of Autonomic Computing.* Computer 2003. **36**(1): p. 41- 50.

37. Damianou, N., Dulay, N., Lupu, E., Sloman, M., *Ponder: A language for specifying security and management policies for distributed systems.* 2000, Imperial College London: London.

38. Lymberopoulos, L., Lupu, E., Sloman, M., *An adaptive policy-based framework for network services management.* Journal of Network and Systems Management, 2003. **11**(3): p. 277-303.

39. Lobo, J., Bhatia, R., Naqvi, S. *A policy description language.* in *The sixteenth national conference on Artificial intelligence.* 1999. CA, USA.

40. Agarwala, S., Chen, Y., Milojicic, D., Schwan, K. *QMON: QoS- and Utilityaware monitoring in enterprise systems.* in *The 3rd IEEE International Conference on Autonomic computing (ICAC).* 2006. Dublin, Ireland.

41. Batra, V., Bhattacharya, J., Chauhan, H., Gupta, A., Mohania, M., Sharma, U. *Policy driven data administration.* in *the Third International Workshop on Policies for Distributed Systems and Networks.* 2002: IEEE.

42. Lutfiyya, H., Molenkamp, G., Katchabaw, M., Bauer, M. *Issues in managing soft qos requirements in distributed systems using a policy-based framework.* in *In POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks.* 2001. London, UK: Springer-Verlag.

43. Ponnappan, A., Yang, L., Pillai, R., Braun, P. *A policy based qos management system for the intserv/diffserv based internet.* in *The Third International Workshop on Policies for Distributed Systems and Networks.* 2002.

44. Melcher, B., Mitchell, *Towards an autonomic framework: Self-configuring network services and developing autonomic applications.* Intel Technology Journal, 2004. **8**(4): p. 279-290.

45. Kaiser, G., Parekh, J., Gross, P., Valetto, G. *Kinesthetics eXtreme: an external infrastructure for monitoring distributed legacy systems.* in *The Autonomic Computing Workshop at the Fifth Annual International Workshop on Active Middleware Services (AMS).* 2003.

46. Parekh, J., Kaiser, G., Gross, P., Valetto, G., *Retrofitting autonomic capabilities onto legacy systems.* 2003, Columbia University: NewYork, USA.

47. Bigus, J., Schlosnagle, D., Pilgrim, J., Diao, Y., *ABLE: A toolkit for building multiagent autonomic systems.* IBM Systems Journal, 2002. **3**(41): p. 350–371.

48. Gracanin, D., Bohner, S., Hinchey, M. *Toward a Model-Driven Architecture for Autonomic Systems.* in *The 11th IEEE International Conference and Workshop on the Engineering of Computer Based System.* 2004. Washington, DC, USA IEEE Computer Society

49. Bulter, J., Barrett, S. *Providing for change through adaptive object models and autonomous computing technique.* in *The 2006 International Conference on Autonomic and Autonomous Systems.* 2006. Dublin: IEEE Computer Society.

50. Pena, J., Hinchey, M., Sterritt, R., Cortes, A., Resinas, M. *A Model-Driven Architecture Approach for Modelling, Specifying and Deploying Policies in Autonomous and Autonomic Systems.* in *The 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC'06).* 2006: IEEE Computer Society.

51. Laws, A., Taleb-Bendiab, A., Wade, S., Reilly, D., *From Wetware to Software: a Cybernetic Perspective of Self-adaptive Software*, in *Self-Adaptive Software: Applications.* 2003, Springer verlog, p. 341-357.

52. Bustard, D., Sterritt, R., Taleb-Bendiab, A., Laws, A., Randles, M., Keenan, F. *Towards a systemic approach to autonomic systems engineering.* in *The 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS).* 2005. Newtownabbey, UK.

53. Beer, S., *Diagnosing the System for Organizations.* 1985: John Weiley & Sons.

54. Beer, S., *Brain of the Firm.* 1981: John Wiley & Sons.

55. Checkland, P., *Soft Systems Methodology in Action.* 1999, Chichester: John Wiley & Sons, Ltd.

56. Checkland, P., *Soft Systems Methodology: A Thirty Year Retrospective.* Systems Research and Behavioral Science, 2000. **17**(S1): p. S11-S58.

57. Taleb-Bendiab, A., Bustard, D., Sterritt, R., Laws, A., Keenan, F., *Model-based self-managing systems engineering,* in *The Sixteenth International Workshop on Database and Expert Systems Applications,* 2005.

58. Badr, N., *An Investigation into Autonomic Middleware Control Services to Support Distributed Self-Adaptive Software.* PhD thesis, *School of Computing and Mathematical Sciences.* 2003, Liverpool John Moores University: Liverpool.

59. Pereira, E., *Impromptu: Software Framework for Self-Healing Middleware Services.* PhD thesis, *School of Computing and Mathematical Sciences.* 2005, Liverpool John Moores University: Liverpool.

60. Omar, W., *Self-Management Middleware Services For Autonomic Grid Computing.* PhD thesis, *School of Computing and Mathematical Sciences.* 2003, Liverpool John Moores University: Liverpool.

61. Reilly, D., *A Dynamic Middleware-based Instrumentation Framework to Assist the Understanding of Distributed Applications.* PhD thesis, *School of Computing and Mathematical Sciences.* 2006, Liverpool John Moores University: Liverpool.

62. Herring, C., *The intelligent control paradigm for adaptable and adaptive architecture,* in *Information Technology and Electrical Engineering.* PhD thesis, 2002, University of Queensland, Brisbane, Australia.

63. Lapouchnian, A., Liaskos, S., Mylopoulos, J., Yu, Y. *Towards Requirements-Driven Autonomic Systems Design.* in *The 2005 workshop on Design and evolution of autonomic application software.* 2005: ACM Press.

64. Walker, R., Baniassad, E., Murphy, G., *An Initial Assessment of Aspect-oriented Programming*, in *the 21st International Conference on Software Engineering*. 1999, IEEE: Los Angeles, CA, USA.

65. Dantas, A., Borba, P. *Adaptability Aspects: An Architectural Pattern for Structuring Adaptive Applications with Aspects*. in *Third Latin American Conference on Pattern Languages of Programming*. 2003: The SugarloafPLoP 2003 Conference.

66. Yang, Z., Cheng, B., Stirewalt, R., Sowell, J., Sadjadi, S., Mckinley, P. *An Aspect-Oriented Approach to Dynamic Adaptation*. in *The first workshop on Self-healing systems* 2002. New York, NY, USA: ACM Press.

67. Greenwood, P., Blair, L., *A Framework for Policy Driven Auto-Adaptive Systems using Dynamic Framed Aspects*, in *Transactions on Aspect-Oriented Software Development II*. 2006, Springer Berlin, p. 30-65.

68. Duzan, G., Loyall, J., Schantz, R., Shapiro, R., Zinky, J. *Building Adaptive Distributed Applications with Middleware and Aspects*. in *The 3rd international conference on Aspect-oriented software development*. 2004. Lancaster, UK: ACM.

69. Falcarin, P., Alonso, G. *Software Architecture Evolution Through Dynamic AOP*. in *The First European Workshop on Software Architecture*. 2004. St Andrews, Scotland.

70. Kuster, J., Sendall, S., *Comparing Two Model Transformation*. 2005, IBM Zurich Research Laboratory: Zurich.

71. Oldevik, J., *UML Model Transformation Tool: Overview and user guide documentation*. 2004.

72. OMG, *MOF 2.0 / XMI Mapping Specification, v2.1.1*. 2007, Object Management Group, Inc.

73. Akehurst, D., Boardbar, B. , Evans, M., Howells, W., McDonald-Maier, K., *SiTra: Simple Transformations in Java* in *The 9TH International Conference on Model Driven Engineering Languages and Systems*. 2006, ACM.

74. Bordbar, B., Howells, G., Evans, M., Staikopoulos, T. *Model Transformation from OWL-S to BPEL via SiTra*. in *The European Conference on Model Driven Architecture Foundations and Applications*. 2007.

75. Oldevik, J., Neple, T., Aagedal, J., *Model Abstraction versus Model to Text Transformation.* Technical Report- University of Kent at Canterbury Computing Laboratory, 2004: p. 188-193.

76. Barber, G. *Service Component Architecture* 2005 [cited 2008; Available from: http://www.osoa.org/display/Main/Service+Component+Architecture+Home.

77. Abuseta, Y., Taleb-Bendiab, A., *AutoTaSC: a Model Driven development Support for Autonomic Software.* Technical Report, DASEL Technical Report 2008/05/YA01, 2008, Liverpool John Moores University: Liverpool.

78. Abuseta, Y., Taleb-Bendiab, A., *Model Driven Development for Autonomic Systems.* The Systemics and Informatics World Network (SIWN), 2008. **4**: p. 178-182.

79. wikipedia. *QVT.* 2008 [cited 2008; Available from: http://en.wikipedia.org/wiki/QVT.

80. Singh, I., Stearns, B., Johnson, M. , *Designing Enterprise Applications with the J2EE(TM) Platform.* 2nd ed. 2002: Prentice Hall

81. Randles, M., Zhu, H., Taleb-Bendiab, A. *A Formal Approach to the Engineering of Emergence and its Recurrence.* in *2nd International Workshop on Engineering Emergence in Decentralised Autonomic Systems (EEDAS).* 2007. Florida, USA: IEEE.

82. Karaorman, M., Holzle, U., Bruno, J., *jContractor: A Reflective Java Library to Support Design by Contract* in *The Second International Conference on Meta-Level Architectures and Reflection.* 1999: Santa Barbara, CA, USA.

83. Meyer, B., *Object-Oriented Software Construction* 2nd edition ed. 2000: Prentice Hall.

84. Leavens, G., Cheon, Y., *Design by Contract with JML.* 2003.

85. Beugnard, A., Jezequel, J., Plouzeau, N., Watkins, D., *Making Components Contract Aware.* Computer IEEE, 1999. **32**(7): p. 38-45.

86. Arnout, K., Simon, R., *The .NET Contract Wizard: Adding Design by Contract to languages other than Eiffel.* 2001, USA.

87. Meyer, B., *Applying Design by contract.* IEEE Computer Society Press, 1992. **25**(10): p. 40-51.

88. Wamber, D., *Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces*, in *Industry Track at AOSD 2006*. 2006: Bonn, Germany.

89. Tran, H., Zdun, U., Dustdar, S., *View-Based Integration of Process-Driven SOA Models at Various Abstraction Levels*, in *First International Workshop, MBSDI 2008*. 2008, Springer-Verlag: Berlin, Germany.

90. Weske, M., *Business Process Management: Concepts, Languages, Architectures*. 2007: Springer.

91. Arkin, A., *Business Process Modeling Language*. 2002.

92. OASIS, *Web Services Business Process Execution Language Version 2.0*, 2007, OASIS.

93. Kruczynski, K., *Business Process Modelling in the context of SOA: An empiric study of the acceptance between EPC and BPMN*. 2008, Leipzig University of Applied Sciences, Germany.

94. Aalst, W., *Patterns and XPDL: A Critical Evaluation of the XML Process Definition Language* 2003, BPMcenter.org.

95. White, S., *Business Process Modeling Notation (BPMN)*. 2004, IBM Corporation.

96. wikipedia. *Business Process Modeling Notation*. 2008 [cited 2008; Available from: http://en.wikipedia.org/wiki/BPMN.

97. Kuchana, P., *Software Architecture Design Patterns in Java*. 2004: CRC Press LLC.

98. wikipedia. *Observer pattern*. 2008 [cited 2008; Available from: http://en.wikipedia.org/wiki/Observer_pattern.

99. Gamma, E., Helm, R., Johnson, R., Vlissides, J. , *Design patterns : elements of reusable object-oriented software*. 1995: Addison Wesley

100. BlackWasp. *Proxy Design Pattern*. 2006 [cited 2008; Available from: http://www.blackwasp.co.uk/Proxy.aspx.