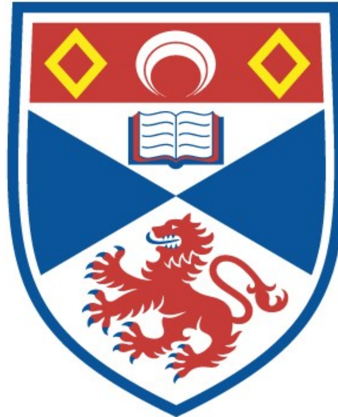
A DOMAIN-DRIVEN METHOD FOR CREATING SELF-ADAPTIVE APPLICATION ARCHITECTURE

Jin Huang

A Thesis Submitted for the Degree of MPhil
at the
University of St Andrews



2017

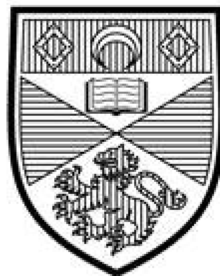
Full metadata for this item is available in
St Andrews Research Repository
at:
http://research-repository.st-andrews.ac.uk/

Please use this identifier to cite or link to this item:
http://hdl.handle.net/10023/15644

This item is protected by original copyright

A Domain-Driven Method for Creating Self-Adaptive Application Architecture

Jin Huang



This thesis is submitted in fulfilment for the degree of MPhil

at the

University of St Andrews

27.03.2017

1. **Candidate's declarations:**

I, Jin Huang, hereby certify that this thesis, which is approximately 22000 words in length, has been written by me, and that it is the record of work carried out by me, or principally by myself in collaboration with others as acknowledged, and that it has not been submitted in any previous application for a higher degree.

I was admitted as a research student in [09, 2011] and as a candidate for the degree of MPhil in [03, 2017]; the higher study for which this is a record was carried out in the University of St Andrews between [2011] and [2017].

(If you received assistance in writing from anyone other than your supervisor/s):

I, Jin Huang, received assistance in the writing of this thesis in respect of [language, grammar, spelling or syntax], which was provided by Janie Brooks

Date …… signature of candidate ………

2. **Supervisor's declaration:**

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of ……… in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date …… signature of supervisor ………

3. **Permission for publication:** (*to be signed by both candidate and supervisor*)

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that my thesis will be electronically accessible for personal or research use unless exempt by award of an embargo as requested below, and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis. I have obtained any third-party copyright permissions that may be required in order to allow such access and migration, or have requested the appropriate embargo below.

The following is an agreed request by candidate and supervisor regarding the publication of this thesis:

PRINTED COPY

a) No embargo on print copy
b) Embargo on all or part of print copy for a period of … years (maximum five) on the following ground(s):
    - Publication would be commercially damaging to the researcher, or to the supervisor, or the University
    - Publication would preclude future publication
    - Publication would be in breach of laws or ethics
c) Permanent or longer term embargo on all or part of print copy for a period of … years (the request will be referred to the Pro-Provost and permission will be granted only in exceptional circumstances).

Supporting statement for printed embargo request if greater than 2 years:

ELECTRONIC COPY

a) No embargo on electronic copy
b) Embargo on all or part of electronic copy for a period of … years (maximum five) on the following ground(s):
    - Publication would be commercially damaging to the researcher, or to the supervisor, or the University
    - Publication would preclude future publication
    - Publication would be in breach of law or ethics
c) Permanent or longer term embargo on all or part of electronic copy for a period of … years (the request will be referred to the Pro-Provost and permission will be granted only in exceptional circumstances).

Supporting statement for electronic embargo request if greater than 2 years:

ABSTRACT AND TITLE EMBARGOES

An embargo on the full text copy of your thesis in the electronic and printed formats will be granted automatically in the first instance. This embargo includes the abstract and title except that the title will be used in the graduation booklet.

If you have selected an embargo option indicate below if you wish to allow the thesis abstract and/or title to be published. If you do not complete the section below the title and abstract will remain embargoed along with the text of the thesis.

a) I agree to the title and abstract being published YES/NO

b) I require an embargo on abstract YES/NO

c) I require an embargo on title YES/NO

Date …… signature of candidate …… signature of supervisor ………

Please note initial embargos can be requested for a maximum of five years. An embargo on a thesis submitted to the Faculty of Science or Medicine is rarely granted for more than two years in the first instance, without good justification. The Library will not lift an embargo before confirming with the student and supervisor that they do not intend to request a continuation. In the absence of an agreed response from both student and supervisor, the Head of School will be consulted. Please note that the total period of an embargo, including any continuation, is not expected to exceed ten years.
Where part of a thesis is to be embargoed, please specify the part and the reason.

Acknowledgement

I would like to offer great thanks to my supervisor Dr. Dharini Balasubramaniam who gave me a chance to do research under her valuable guidance. I really appreciate that she is always being patient with me, even when I was suffering healthy and mental problems. I would like to show my special thanks to Ms. Janie Brooks for her great help in my English study.

I would like to thank my wife Fang Zhang who always gives me her greatest support in these months. It is great that I can meet you in last year. I would also like to thank my family specially my parents (Jingchang Huang and Guiqin Ma). Finally, I would like to thank the University and the School for accepting me as a research student. The time of being in St Andrews is always valuable and memorable for me. Thank you very much!

Abstract

Following the increasing complexity of modern software systems, software engineers have introduced self-adaptation techniques from the field of control theory into software development. However, it is sill difficult to construct self-adaptive software systems. By understanding the importance of software architecture, this dissertation concerns the issues of how to design a domain-specific self-adaptive software application architecture in a principled way. Specifically, there is still lacking of method for helping software engineers generate software architecture which is consistent with the domain knowledge. To achieve the research goal, this dissertation has: 1) investigated the existing definitions about software architecture; 2) proposed a framework of understanding self-adaptive software application architecture via appropriate architectural patterns; 3) proposed a novel high-order language, and the tools, to specify domain-specific uncertainty; 4) proposed an improved version of Grasp, and the tools, so that users can describe the dynamism of a self-adaptive application; 5) proposed a novel architectural pattern by selecting architectural patterns in a principled way; 6) evaluate this work by applying these methods to a business project.

Table of Contents

Abstract....1
1 Introduction....3
1.1 Issues of uncertainty....3
1.2 Importance of software architecture....3
1.3 Principled software architecture design....4
1.4 Self-adaptation....4
1.5 Motivations....5
1.6 Dissertation organisation....5
1.7 Summary....6
2 Related work....8
2.1 Software architecture and patterns....8
2.2 Software architecture design methods....11
2.3 Software architecture description methods....11
2.4 Uncertainty....12
2.5 Self-adaptive software system design methods....13
3 Proposed approach....14
4 A framework for understanding software architecture in an uncertain context....16
4.1 A self-consistent terminology....16
4.2 Properties of interest....18
4.3 Architectural patterns for self-adaptive applications....19
4.4 Summary....21
5 The role-behaviour-based modelling language....22
5.1 Design principles....23
5.2 Conceptual model....24
5.3 Grammar....26
5.4 A simple event calculus....28
5.5 Consistency checking....30
5.5.1 Using process calculus....30
5.5.2 Using transitions....30
5.5.3 Generating architectural outline with business constraints....31
5.6 Completeness checking....32
5.7 Summary....32
6 Grasp+:an enhanced version of Grasp....32
6.1 Grasp....33
6.2 Problems....34
6.3 The role-action-based conceptual model....34
6.4 Grammar....35
6.5 Case-Study: An intelligent data-providing system....37
6.6 Summary....39
7 Case study – an intelligent multiple strategies trading system....41
7.1 Domain requirements....42
7.1.1 stakeholders....42
7.1.2 Functional requirements....43
7.1.3 Non-functional requirements....43
7.2 Domain model....44
7.3 Architecture descirption....46
7.4 Proposed implementation....50
7.5 Evaluation....51
7.6 Summary....53
8 Conclusion and future work....54
References....54

1 Introduction

Both the growing complexity of current software systems and context requirements necessitates the research in the field of self-adaptive software systems [1]. However, to construct a self-adaptive application is still not easy, especially in a complex environment, e.g. a mobile environment. Further, architects now are facing the issues arising from uncertainties that are sourced from requirements [2], e.g. handling human errors, behaving intelligently, collaborating among multiple platforms in a dynamic environment, self-adapting. In this situation, one big problem is about constructing self-adaptive applications which have a desired behaviour that is consistent with their domain requirements. As a consequence, our research focus on the work on how best to make architectural decisions from domain requirements for providing desired properties to self-adaptive applications.

1.1 Issues of uncertainty

Nowadays, domain requirements are becoming increasingly complex, and thereby the design of a modern software application system is becoming dramatically more complicated and software engineering research is facing new challenges. One of the challenges is uncertainty. The sources of uncertainty are various, e.g. "unknownness" [3] and changing requirements. Traditionally, software engineering research concerns how best to resolve issues of a domain by modelling the certainty aspects of this domain. To successfully construct a software system, ideally, software engineers should first have a complete understanding of the design context and foresee any issues. But, in practice, software engineers may not be able to understand and foresee everything. That is one of the reasons why software application systems fail: people believe that it is possible to have a complete and clear understanding of their projects, but in reality, it is not true [2]. The world is changing and uncertain, so software engineers need new methods to ensure their design survive in this world. In this situation, to help construct applications that can deal with uncertainty, software architecture research faces new challenges: What uncertainty issues exist at architectural level? how best model the uncertainty at architectural level? How are context uncertainty and architectural decisions related? how best to make architectural decisions to handle uncertainty.

1.2 Importance of software architecture

Although there is still no widely accepted definition of software architecture, software architecture has been widely recognised as an important blueprint for guiding the construction of software systems, just like architecture guides the construction of the building. We can simply treat a software architecture as a set of architectural decisions at the design stage of a software system. A software architecture usually holds a set of abstract structures represented in diversified views, and each structure constrains and guides one aspect of the implementation of a software system, and more importantly, each structure can be used to reason about a set of properties of a software system [4]. Many researchers have summarised the importance of software

architecture in terms of their experience and work, and to emphasise the significance of our work, we have the main (not all) advantages of having a software architecture listed below [4] [5] [6] [7]:

- A well-documented architecture improves communication among stakeholders;
- An architecture works as an abstract guidance of how a software system can address context concerns;
- An architecture can be used as a basis for estimating costs;
- An architecture helps partition a software system into parts and thereby different groups of people can work cooperatively and productively together to solve a more complex problem;
- An architecture can be used as a basis for performing dependency and consistency analysis;
- An architecture can be used as a basis for reasoning about this system's properties and behaviour at design stage; and
- An architecture can be used as a basis for managing and reasoning about changes.

1.3 Principled software architecture design

The design of software architecture is principled. In other words, a software architecture is created following a set of constraints from its context. The context of a software system has multiple aspects due to the existence of the stakeholders of this system, e.g. technical and business aspects. For example, before making architectural decisions, architects should have a clear understanding of what the business purposes of this project are, what technologies can be used to complete this project, how long the project's lifetime is, and what properties desired of this system are, etc.. A well-designed architecture can be reused and applied in order to obtain the same properties in the future. A software architectural pattern (or styles) is a package of architectural decisions that have been proven in practice. It is more abstract than an architecture, and a software architecture is an instance of a software architectural pattern [5]. As Fielding illustrated in his work [8], architects can successfully create their own architectures by making their architecture as an instance of one or more architectural patterns.

1.4 Self-adaptation

Self-adaptation has been introduced as a resolution to the issues of the increasing complexity of software systems at runtime in a dynamic context [9]. To put it simply, a self-adaptive software system is a type of system that is capable of adapting to changes in the context of this system at runtime. Typically, a self-adaptation process is modelled as a MAPE loop [10] (Monitoring, Analysing, Planning, and Executing plans), which is a close-feedback-loop. By having a self-adaptation process integrated into it, a software system may provide more quality properties, such as the capability of being versatile, flexible, resilient,

dependable, energy-efficient, recoverable, customisable, configurable, and self-optimising [1]. However, self-adaptation is not a silver bullet for solving all complexity issues, and by introducing a self-adaptation process, a system may have more uncertainty introduced as well, e.g. uncertain plans [11]. In addition to this, architects may need to consider the impact of using self-adaptation on a software system, e.g. resource limitation, performance impact. A software system may contain more than one self-adaptation processes, and the need to guide carefully the design necessitates considering the coordination mechanism of coordinating these processes at architectural level.

1.5 Motivations

This dissertation explores a junction on the frontiers of three research disciplines in software engineering: software architecture, self-adaptation and system uncertainty processing. Software architecture research has long been concerned with the architectural decisions concerning certain aspects of software systems, but has rarely been able to evaluate objectively the impact of various design choices on uncertain system behaviour. In addition to this, researchers in this field do not concern much on the relationship between business constraints and architectural decisions. Self-adaptation research mainly focuses on how best to model a self-adaptation process, how an uncertainty can be solved by introducing a self-adaptation process, and how best to construct a self-adaptive software system in a given context, often ignoring the fact that introducing self-adaptation can bring new uncertainties and there is still a lack of the guidance on how best to build a self-adaptive software system when considering both the business constraints and architectural uncertainty. On the other hand, the community of architectural uncertainty research often pay attention to the methods of modelling and describing uncertainty and the methodologies of reasoning about possible system behaviour at architectural level, rather than the methods concerning how best to resolve architectural uncertainties by making appropriate architectural decisions which are consistent with business constraints.

As a consequence, my work is motivated by the desire to understand and evaluate the architectural design of a self-adaptive application system through principled use of business and architectural constraints, thereby obtaining the desired properties of an architecture in an uncertain context. In short, the output of this work is mainly about a method by which software engineers can design their applications to behave following domain constraints even in complex environments. Furthermore, if we understand architectural style as a set of reusable architectural constraints with a given name, which are used to solve a collection of general issues in a specific context, we can expect that for self-adaptive application systems, an architectural style may have certain aspects supporting the systems' runtime dynamic behaviour.

1.6 Research methodology

This dissertation focuses on the design of a method to best help software engineers construct self-adaptive application systems that work in an uncertain context. Due to the vital role of software architecture in the process of software engineering, we are concerned with the issues of the architecture design of a self-

adaptive application system, and by doing research in this way, we can also limit our research scope. Considering the required properties of a self-adaptive application, the context of making architectural decisions for a self-adaptive application is uncertain. Further, integrating a self-adaptation process with an application inherently brings uncertainties. In addition to this, a software system is developed to satisfy business purposes, and it has been shown by Peled [40] that two different process patterns can be identified as being consistent or not when considering them in different business contexts. Moreover, according to Evans [27], the design of a software system should satisfy the constraints extracted from the domain model that represents the specific context in which this system is designed.

As a consequence, to carry out our research, several issues need to be solved:

- how best to understand the design of software architecture in an uncertain context;
- how a set of business constraints, especially the ones from the uncertain aspects of a business context, can help create a software architecture in a principled way;
- how best to describe a self-adaptive application architecture, including description of the dynamic aspects of an architecture;
- how to create an architecture in a principled way for guiding the construction of an application, especially a self-adaptive one; and
- how to evaluate our research output.

To solve the above issues, we investigate the existing definition of software architecture, and then propose a framework for understanding self-adaptive application architecture via suitable architectural patterns, including a self-consistent terminology for software architecture in uncertain contexts. Further, we propose a novel language for representing business constraints, and by using a method based on CCS (Calculus of Communicating Systems), we can outline a software architecture constrained by a model in our language by arranging the architectural elements of this architecture in a certain structure that is consistent with this model. By providing an ADL based on Grasp, we obtain a suitable way to describe self-adaptive application architecture. Further, we provide an architectural pattern with desired properties, which is constructed in a principled way by applying proven architectural patterns, to help create architecture to satisfy the same context constraints. Finally, we evaluate our methods by applying them to an ongoing business project. In practice, by checking the system log, if an implementation created following our methods can behave as same as the design, then we can define that our research achieve our goal.

1.7 Dissertation organisation

Chapter 2 investigates the existing work related to software architecture, including definitions, description and design methods, and the existing methodologies of resolving architectural uncertainties. The methods of designing self-adaptive software systems at architectural level are also surveyed.

Chapter 3 defines a framework for understanding the software architecture in an uncertain context via architectural patterns that support self-adaptive behaviour, and reveal what uncertainties existing in it. A set of architectural patterns used for supporting self-adaptive behaviour are surveyed and classified based on the architectural properties of these patterns when applying them to a self-adaptive application architecture, especially in an uncertain context, e.g. decentralised environments. This classification is used to identify a set of architectural constraints that could be used to improve the architecture of self-adaptive and decentralised collaboration applications.

Chapter 4 presents and elaborates the design of a novel language used for modelling business constraints. These core concepts of this language are role, event, behaviour and processes. Through using this language, a business constraint can be described as an event causality chain created by the temporally ordered behaviour of a specific group of independent roles. With a business constraint in this model, we can perform completeness checking by having an event-reaction graph. To provide a potential capability of perform automatic checking, we also provide a simple formal framework for this language by introducing CCS [12] (Calculus of Communicating Systems).

Chapter 5 presents and elaborates the design of an ADL (Architecture Description Language) on the basis of Grasp [13]. To describe a self-adaptive application architecture, an ADL should be capable of being used to describe dynamic structures. The core concept of this enhanced Grasp is role. A role is an abstract element that provides one service or a set of cohesive services. Through using this language, a structure is re-defined as sequential fragments happening in various given scenarios following a temporal order.

Chapter 6 presents and elaborates the design of the Independent Reaction Chain (IRC) architectural style for self-adaptive and decentralised collaboration applications. IRC provides a set of architectural constraints that, when applied as a whole, emphasises scalability of node interactions, capability of collaboration without a central controller, generality of communication, capability of coping with unexceptional behaviour and independent working capability of nodes to maintain consistency in a work flow and reduce the uncertainty brought by its context.

A case-study is illustrated in chapter 7. I have evaluated the result of the above methods by applying it to the design of the architecture of a futures-trading system. This system aims to trade the futures of the commodities, e.g. iron ore, coke and steel, in Shanghai Futures Exchange online. To make a profit, multiple strategies are running in different nodes at the same time. Each strategy runs independently, but multiple strategies can collaborate with each other by exchanging trading signals. Further, in this domain, to place an order, several roles have to work together in a given work flow according to the domain context, e.g. internal and external regulations.

1.8 Summary

In summary, this dissertation makes the following contributions to software architecture research within the field of Computer Science:

- A framework for understanding software architecture from a new perspective, in which an architecture is treated as a guide of dealing with uncertainty, and includes a self-consistent set of terminology for describing software architecture with uncertainty;
- A novel language used for modelling business constraints for deriving the domain-specific outline of an architecture;
- A novel architectural style for guiding the construction of software systems in an uncertain context; and
- Application and evaluation of the above architectural style in the design and deployment of the architecture for a futures-trading system.

2 Related work

This chapter examines the background for this dissertation by investigating and discussing the relevant concepts of software architecture, including architectural patterns; the way in which an architectural pattern is used to guide the design at architectural level; the existing architecture description methods; the way in which a software architecture is derived from its context constraints; the uncertainty issues that need to be considered to design a software application; the challenges brought by these uncertainty issues at architectural level; the definition of self-adaptation; the way in which self-adaptation processes are used to solve uncertainty, especially for the applications in decentralised environments; and the existing architectural patterns or styles supporting self-adaptation with specific properties.

2.1 Software architecture and patterns

The design of software architecture is an important stage of software development process. But, regarding the definition of software architecture, there is still no common agreement. An early definition from Perry and Wolf [6] represented software architecture as a model that has 3 members {elements, form, rationales} by an analogy with building's architecture. According this model, an architecture is composed by a collection of elements that are constrained by a form. The so-called rationales are reasons for making architectural decisions, e.g. adopting a certain element or form. The most of existing definitions of software architecture is built on this model. Though there is still no a commonly accepted definition, there are some widely recognised characteristics of software architecture:

A software architecture is an abstraction that can be used to reason about the behaviour of a software system. An architecture is an abstraction of a software system, and therefore it only contains the selected set of information that is necessary for reasoning about the system properties [4]. An architecture not only constrains the structure of a system, but also the behaviour of this system[4]. The behaviour specified in an architecture is achieved by the collaboration of the architectural elements described in this architecture [14]. Design decisions made at architectural level are the most essential ones and are expensive to change [15]. An architecture does not only concern high-level design decision issues [8];it spans across multiple levels of granularity [16]. That is, at a lower-level, each element of an architecture may have its own architecture that specifies and constrains the structure and behaviour of this element. However, not all design at every level, e.g. colour of a button, is architectural design, since this kind of design is not sufficient for reasoning about a software system's behaviour.

A software architecture is designed or created in given contexts. All architectural decisions should always be made within some contexts [8]. "Context" is explained as "the interrelated conditions in which something exists or occurs" by Merriam-Webster [17]. To clarify the concept, we define a context of an architecture as a certain set of constraints from a specific stakeholder, which are interrelated to the architectural decisions. Context constraints are described in the form of requirements. Fielding [8] suggested the contexts to be

considered at architectural level, such as functional, behavioural, and social requirements. An architect has to evaluate the priority of constraints of a project at the very beginning of this project in order to make appropriate architectural decisions [4].

A software architecture has a set of elements constrained by specified structures. An architecture comprises of a set of structures, and an architectural element is the basic unit of an architecture that is viewed from a certain structure [4]. A structure is a certain arrangement of architectural elements [6], and architectural structures are important to reasoning about the properties of an architecture [4]. Each architectural element specifies a system element's behaviour. In general, Perry and Wolf [6] categorised architectural elements into 3 types, which are processing, data and connecting elements, from a component-connector viewpoint. In detail, the processing elements are the components performing computation and transforming data; the data elements are the components providing data used in a software system; the connecting elements are connectors that are used to glue the other elements together and characterise the properties of the interactions among the other elements.

A module is an architectural element in a module-based structure. Bass et al. [4] defined a module as a static partition of a software system from a module-based viewpoint, and in this kind of structure, a module has been assigned a set of functional responsibilities. A module-based structure makes a software system flexible by allowing architects to reassembled and replaced a module independently [18]. Clements et al. [7] treated a module as an important unit of a software system, and as a set of codes or data that hides the changing information and provides an interface to access this module's functionalities and data.

A component is an architectural element in a component-connector-based structure. An architectural component is a conceptual unit [8] with a clear boundary [4]. Particularly, a component is an architectural element of a software architecture if we partition this architecture into elements from the component-and-connector perspective. Perry and Wolf [6] defined architectural components as the elements which are responsible for transforming data elements. Bass et al. [4] defined architectural components as a set of architectural elements which can behave at runtime. Solms [16] claimed that architectural components are responsible for resolving technical issues. Gorton [19] defined components as a set of related architectural elements, and each of which elements has been assigned responsibilities, and at architectural level, a component is described as a black box that only has externally visible properties. In addition to this, Gorton [19] also pointed out that each component has a specific role in an application, and it is important to specify this and the collaboration among these roles at architectural level.

In a component-connector-based structure, a connector specifies a relationship. Connectors describe the interactions among components [4]. For example, protocols are a type of connectors. An architectural connector characterises a relationship between two components. Perry and Wolf [6] provided an intuitive description of connectors: the glue holding the elements of an architecture together. Shaw and Clements [20] defined a connector as an intermediate mechanism of interactions among components, such as

communication, coordination, or cooperation. Clements et al. [7] also describe connectors as a depiction of component interactions. In short, a connector can be characterised as an abstract mechanism for mediating interactions among components.

Data is a specific kind of architectural elements. In practice, a datum is generated or consumed, sent from, or received by a processing element via connecting elements [8]. A datum can be in any form, including files, byte streams, string-based messages, objects, etc. Fielding [8] argued that it is important to consider data elements in some cases since the selection of data elements may affect the properties of a software system, especially for network-based software systems. Buschmann et al. [21] introduced their work by modelling components through encapsulating core data and functionalities together;

A software architecture can provide a set of properties. Properties are emergence of software systems [22]. All the properties of an architecture are induced by the context constraints of this architecture [8]. There are two types of properties: functional properties and non-functional properties. Functional properties are the functionalities required to fulfil context functional requirements, and non-functional properties are the desired context qualities of the software system being designed. Non-functional properties are mainly derived from the constraints from technical, business, and application domains [19]. Examples of quality properties are: reliability, scalability, performance and availability. Fielding [8] claimed that the goal of architectural design is to create an architecture with a set of properties that can lead to a superset of the system requirements. This claim is not precise. The main reason is that a superset of the system requirements may lead to undesired system behaviour.

An architectural structure is presented in a view. An architectural view is a representation of a structure [4]. It is used as a presentation of a certain architectural structure for system stakeholders. Like a structure, an architectural view is written from various perspectives to illustrate different aspects of an architecture. It is important to document an architecture in different views from various perspectives since it is not easy to describe an architecture in a one-dimensional fashion [7]. An example is the 4+1 architectural view model designed by Kruchten [23]. The model presents an architecture by using 4+1 views: the development view describes system components; the logical view describes relationships; the physical view describes how components are deployed physically; the process view describes the runtime behaviour of a system; and the scenarios describe use-cases for testing the design presented in the other 4 views.

An architectural pattern is a reusable solution to a set of general problems and is more abstract than software architecture. Architectural styles, also known as architectural patterns, are reusable packages of architectural decisions that have been proven to be good ones in a given context [4]. Garlan and Shaw [24] claimed that "An architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined". Gorton [19] defined architectural style as a set of successfully applied structures facilitating certain kinds of component communication. Architectural styles can help architects to specify the core structure of an application and

achieve a particular set of properties [21]. An architectural style is more abstract and general than an architecture [6]. Some examples of existing architectural styles or patterns are layered architecture style, event-driven architecture style and micro-kernel architecture style.

2.2 Software architecture design methods

Software architecture design research focuses on how best to decompose a software system into parts and how best to arrange these parts in order to satisfy context requirements [25]. Software architecture design is a part of the whole process of software engineering. Albin [25] summarised the traditional design process of software architecture as 4 activities: modelling problems; identifying architectural elements and relationships; evaluating the design; and refactoring the architecture of being designed. Fielding [8] proposed that the best way to design a software architecture is to identify the design context, including stakeholders; confirm the properties required in the design context; make design choices by applying appropriate architectural patterns; and finally implement the adopted architectural decisions. By applying architectural patterns, architects finally find a way to share the architecture knowledge with developers for building a good software system [26]. Microsoft [5] listed the key design principles of software architecture: separating functionality at appropriate boundaries; and making each element responsible for only a specific group of functionalities; and making each element has minimum knowledge of the others; and not allowing two elements have the same functionality; and minimising the upfront design.

A challenge to software architecture design is to do architecting for an application required in a very complex domain. Since a domain may be very complex, even uncertain, architects may not possible to predict all potential properties of the application of the software system being designed. Failure of many software applications is caused by domain complexity rather than technical complexity [27]. For managing the domain complexity during software design, Evans [27] presented a set of principles and a pattern language for guiding mapping business models to the design of software systems. In addition to this, both Evans [27] and Vernon [28] positively encouraged domain experts to take part in software architecture design. As a consequence, both domain experts and architects need new methods and tools to share their knowledge with each other.

Another challenge to software architecture design is to do architecting for self-adaptive software systems [26]. Early software architecture only contains static information and concerns how best to decompose a software system into parts. But, considering the fact that a self-adaptive software system can behave dynamically at runtime in order to adapt itself to its changing operational environment, the software architecture of a self-adaptive software system should contain not only static information but also dynamic information. Further, the visible behaviour of a self-adaptive software system should be consistent with the architectural constraints.

2.3 Software architecture description methods

Software architectural description activity can be treated as a process of recording or modelling a software architecture in various forms, e.g. text, diagrams, formalisms. As W.Ellis et al. [29] pointed out, “an architectural description is a model — document, product or other artifact — to communicate and record a system’s architecture.”. Broadly speaking, when talking about software architectural description methods, we mean the methods used for documenting, representing and specifying an architecture [6]. A well documented software architecture works as a communication among stakeholders, and is essential to produce a software system with desired properties that can be predicated [7]. In detail, with an architectural description, we can: record architectural decisions at desired granularity and abstract level; ignore useless details; represent architectural structures from various stakeholders’ viewpoints; and perform certain kinds of analysis [6].

In a narrow sense, software architectural description methods mainly refer to architecture description languages, abbreviated as ADL. An ADL is a language used for describing and representing a software architecture in an informal or formal way. An ADL typically describes and specifies a software architecture’s structures (may also include the rationales behind a structure, e.g. Grasp [13]) from a given perspective. For example, the foundation of Acme [30] is an ontology comprising seven types of architectural concepts: components, connectors, systems, ports, representations, and rep-maps. An ADL may also be built on a formal framework in order to provide certain features, e.g. Wright [31] is built on Communicating Sequential Processes, and so the architectural models in Wright can be used to perform both port compatibility and system consistency checking. ADLs can also be used to describe large-scale systems. For example, Luckham [32]proposed Rapide for supporting component-based development of large-scale distributed systems, which is built on an event-based execution model introduced by Kenney [33].

Considering the fact that traditional ADLs provide limited features for describing dynamic structures, e.g. self-adaptive software systems in a decentralised environment, researchers in this field have begun to focus on the methods for specifying the dynamic behaviour of a software system. An example is π-ADL [34], which is built on π-calculus [35]. π-ADL provides a capability for describing dynamic, especially mobile, structure by using conditional statements and a certain set of keywords. However, the actual description ability of π-ADL is limited. Since it is still designed for describing certainty rather than uncertainty. That is, architects still need to have a clear understanding of their projects to elaborate every possible change of structures, otherwise, it would cause potential problems. Further, even architects know everything about their systems, it would require huge effort to elaborate all aspects of a large-scale software systems by a lot of conditional statements.

2.4 Uncertainty

Due the increasing complexity of modern software systems, there may be various uncertainties need to be solved at architecture design stage. For example, for constructing an automatic driving system, architects need to concern the uncertainties over the collaboration among various automotive parts, such as engine,

wheel, throttle, brake, and sensors, since the context of driving a car is changing. The main reason for the uncertainties is that software engineers cannot predict everything since their limited knowledge [3]. In detail, Garlan [2] listed the possible sources of system uncertainty: human error; intelligent system behaviour; mobility; rapid runtime evolution; complex automatic controlling behaviour. This situation is against the foundation of traditional requirement and software engineering methods, which assumes that designers are capable of having a clear understanding of their projects [36].

To solve these uncertainties, a software system may need to be more dependable, flexible, recoverable, or configurable [1], and hereby architects should carefully compose architectural elements following the context constraints to achieve overall goals. Self-adaptation is one of the solutions to uncertainty. A self-adaptive software system has one or more self-adaptation processes and the system can manage uncertainty at runtime by adapting its behaviour in terms of the analysis of its observation.

2.5 Self-adaptive software system design methods

For managing complexity and uncertainty at runtime, self-adaptation mechanisms have been introduced to software engineering in recent years [37]. Normally, a self-adaptation process is modelled as a close-loop named as MAPE (Monitoring, Analysing, Planning and Executing plans) loop in general [10]. A self-adaptive software system is a kind of software system developed with one or more self-adaptation techniques. Self-adaptive software systems can behave to achieve their goals by adapting themselves to changes at runtime. The properties of a self-adaptive software system are named as self-* properties [37]. Due to the increasing complexity of software system design, more and more researchers have joined the research in the field of self-adaptation. But, it is still not easy to design and develop self-adaptive software systems, especially in a complex environment, as Lemos et al. [1] pointed out, the issues relating to the design and development of self-adaptive software systems are:

- how best to capture, describe and represent domain uncertainty, including the domain-specific methods for managing the uncertainty, for making appropriate design decisions;
- how best to design a self-adaptation mechanism;
- how best to make architectural decisions on achieving the desired properties by making trade-offs among every potential choice;
- how best to fill the gap between the design and the implementation of self-adaptive software systems;
- how best to coordinate the collaboration self-adaptation mechanisms and processes.

In practice, to engineer a self-adaptive software system, there are three problems need to be considered [38]:

- performing adaptation in an off-line or on-line way. That is, whether a system is adapted externally or internally;
- conceptualising the processes maintained in self-adaptive software systems;

- processing the uncertainty caused by the introduction of self-adaptation processes.

For example, Garlan et al. [39] presented a framework named as Rainbow. This framework supports self-adaptation in an off-line way. In detail, Rainbow has 3 layers: from top to bottom, there are architectural layer, translation layer, and system layer. The system layer maintains an abstract system model at runtime, which is an ADL-based model of a target system, and this layer also has modules can monitor and adapt the target system. On the other hand, the architecture layer maintains a self-adaptation mechanism on the basis of a MAPE loop, so that it can make architectural decisions at runtime by analysing the information it receives. Thus, by communicating through the translation layer, the architecture layer can make decisions by comparing the system model with the target system and adapting the target system by sending execution orders to the system layer. But, a problem not being mentioned in this work is how Rainbow makes architectural trade-offs. Garlan et al. also did not discuss the actual performance of their framework: Considering it may need significant resources to maintain a complex abstract model at runtime and make architectural decisions at runtime based on the checking on this model, how can they ensure the target system will behave properly within the desired time? And how is the situation if the resource of the target system is limited?

To help engineer self-adaptive software systems, researchers in this field mainly concern the methods for modelling and analysing self-adaptation behaviour at architectural level, designing self-adaptation mechanism, and coding self-adaptive software systems. However, one problem being ignored normally is about how best to make architectural decisions in a given context via architectural patterns. That is, architects need practical guidelines for creating architectures for self-adaptive software systems within a given set of general constraints. To solve this problem, the following parts of this dissertation introduces a method for creating domain-specific self-adaptive application architectures by choosing appropraite architectural patterns.

3 A framework for understanding software architecture in an uncertain context

"*The thirty spokes unite in the one nave; but it is on the empty space (for the axle), that the use of the wheel depends. Clay is fashioned into vessels; but it is on their empty hollowness, that their use depends. The door and windows are cut out (from the walls) to form an apartment; but it is on the empty space (within), that its use depends. Therefore, what has a (positive) existence serves for profitable adaptation, and what has not that for (actual) usefulness.*" by Lao Tzu, Tao Te Ching

The understanding of software architecture improves as time goes on. The traditional definitions of software architecture are based on an assumption that it is possible to have a clear understanding of a project. But, in reality, this is not true, and a software architecture may be created within uncertain contexts. For example, a self-adaptive software system is designed to solve uncertainty at runtime by adapting its behaviour when facing changes from its contexts. Considering this, the architecture of a self-adaptive application may have dynamic aspects. To study how best to design architecture for self-adaptive application systems, we would first like to have a solid understanding of software architecture with respect to self-adaptation. In detail, we would like to solve the following problems through an examination of existing definitions related to software architecture:

- how to define software architecture, especially application software architecture;
- what the factors are that should be considered during architecture design;
- how best to design an architecture for self-adaptive application.

To solve the above issues, in this chapter, we provide a self-consistent terminology for software architecture with respect to dynamic behaviour. We also investigate the difference between software architecture and application software architecture. Finally, we discuss desired properties of self-adaptive applications and investigate existing architectural patterns which could be applied to provide these properties.

3.1 A self-consistent terminology

Based on the existing definition of software architecture and self-adaptation introduced in the previous chapter, we can thus present an understanding of self-adaptive application architecture from our viewpoint. To achieve this, we first need to clarify several issues regarding our research scope first:

- The issues of designing architecture for self-adaptive application is our main concern in this dissertation. An application here means a kind of software systems that is designed to satisfy a group of business purposes, e.g. perform a group of business tasks. That is, an application is designed to satsify end-users' requirements. A self-adaptive application is an application that can adapt itself at runtime according to its observation of its execution context. To limit the research scope, our work

focuses on the issues of developing self-adaptive applications. An application is a software system, but not all software systems are applications. Therefore, a self-adaptive application is normally domain-specific;

- Business context is the only design context that we focus on in this dissertation. According to the above discussion, a context actually is a set of constraints from a certain stakeholder. According to Bass et al. [4], the design of a software architecture occurs within multiple contexts. However, in this dissertation, we only focus on the impact of business context on the architectural design.

We can then provide our understanding of self-adaptive application architecture based on the existing definition introduced in the previous chapter as below:

- A self-adaptive application architecture is an abstract design that supports both business and software processes and deals with uncertainty with suitable self-adaptation properties. Application architecture bridges domain requirements and software implementations [5]. A domain-specific software system's architecture is restricted by the domain specific constraints [27]. According to Lemos et al. [1], it is important to comprehend fully the business and software processes and identify the related uncertainty issues before engineering a self-adaptive software system in order to take appropriate actions; further, for self-adaptive software systems, the self-adaptation (also known as self-*) properties are the emergence required for achieving certain quality requirements at runtime [37].
- An architectural element of a self-adaptive application architecture has a certain life-time at the operational stage of this application, and an architectural element of a self-adaptive application architecture may be structured in a dynamic way following one or more self-adaptation processes. The main design concern of a self-adaptive software system is how to structure an architectural element in terms of self-adaptation processes [40]. To design self-adaptation processes at architectural level, it is necessary to specify the architectural adaptation processes by elaborating what architectural element needs to be changed in what way, at what time and for what reasons [37];
- The business context of a self-adaptive application architecture may change over time. The business constraints which need to be considered at architectural level may change over time, and more importantly, the business purposes of a self-adaptive application may change over time too [4]. Further, due to the complexity of a domain, the understanding of this domain may improve over time. In addition to this, due to the development of technology, user requirements may change over time as well;
- The desired degree of the quality properties, also known as non-functional properties, provided by a self-adaptive application architecture may change over time. For example, in a mobile environment, the resource availability may change over time [2], and to solve this kind of uncertainty, it is

important for architects to evaluate the situation, so that they can balance all quality properties and achieve a better trade-off in different contexts [38];

- The architectural patterns adopted by a self-adaptive application architecture may change over time. An architectural pattern is a reusable solution to a certain group of issues in a given context. When the context changes, a self-adaptive application architecture needs to adopt different architectural patterns;
- An architectural view of a self-adaptive application architecture may need to specify the dynamism of this architecture. An architectural view is a representation of a given architectural structure [8]. The structures of a self-adaptive application architecture may have dynamic aspects. Therefore, to describe a self-adaptive application architecture, it is important for architects to specify at architectural level the dynamism of the application which is being designed by using a certain type of view.

3.2 Properties of interest

This section introduces the key properties of a self-adaptive application, including both functional and non-functional properties. This section does not aim to provide a comprehensive list of the properties, but only the essential properties that are important for constructing a self-adaptive application. Further, we do not intended to talk about self-* properties here, since our concern is the issue of how best to make architectural decisions to develop self-adaptive applications, rather than the issue of what properties a self-adaptive application provides.

According to the MAPE model [10], the basic functional properties of a self-adaptive application should be:

- the capability of performing business tasks;
- the capability of monitoring and capturing the changes which happened in the environment of this application at runtime. In detail, a self-adaptive application should be able to identify the changes, collect the changes, transfer the existence of the collected changes to the other architectural elements that are waiting for this information. To provide these functionalities, this system should also have its own way to represent a change;
- the capability of analysing and making architectural decisions based on the collected information at runtime. The application should provide functionalities to understand the collected changes at runtime;
- the capability of performing adaptation on the application. The application should be able to manage its own elements, e.g. components and connectors, at runtime;
- the capability of communicating among elements;

According to the understanding of self-adaptation, self-adaptive applications need to make architectural decisions in their operational phase. As a consequence, self-adaptive applications may need to have a flexible and extensible structure which can be evolved at runtime. Further, the self-adaptation processes within a self-adaptive application should be available and reliable, or event fault-tolerant; otherwise, the failure of a self-adaptation process may lead to system failure or unexpected behaviour. Additionally, the execution of a self-adaptation process should not cause unexpected performance issues. In detail, the possible non-functional properties of a self-adaptive application should be:

- flexibility. A self-adaptive application should be flexible so that it can respond to uncertainty by changing its behaviour and structures;
- scalability. A self-adaptive application should be scalable so that its behaviour or structures can be changed efficiently at runtime;
- changeability. It is easy to change or replace an architectural element at runtime;
- reliability. The self-adaptation processes within a self-adaptive application should be sufficiently reliable;
- availability. The self-adaptation processes within a self-adaptive application should be sufficiently available;
- performance. The execution of a self-adaptation process of a self-adaptive application should not negatively impact on the performance of the execution of domain-specific business processes;
- maintainability. It is easy to maintain the self-adaptive application since it has an understandable structure and the architectural elements of this application are loosely coupled.

3.3 Architectural patterns for self-adaptive applications

The key issue which needs to be solved at architecture design stage is: how best to make architectural decisions which support both business and self-adaptation processes. To answer this problem, existing architectural patterns that can be used to support the activities of the MAPE loop should be evaluated and discussed. However, we are not going to list and discuss all patterns that could be used for supporting the development of self-adaptive applications here, since the number of existing patterns is huge, and our word count is limited. Therefore, in this section, we select and discuss several classical patterns that are well-known. We believe that it would be easier for software architects to understand the design process if we use common patterns rather than specialised patterns here. We introduce the following patterns by referring to the existing studies [21] [4] [8] [19] [25].

Layers pattern. A system can be separated into several layers and each layer has its own responsibility; each layer can only communicate with its neighbour layers through a limited number of interfaces. In this way,

domain-specific properties and self-adaptation properties can be separated into two layers, e.g. domain layer and self-adaptation layer, and the domain layer works on top of the self-adaptation layer. The non-functional properties provided by layer pattern are: maintainability and changeability. However, the failure of one layer can cause serious problems since one layer depends on the functionalities provided by the other layer; and it is very important for architects to decide the appropriate number of layers, since the performance may also be decreased if there are too many layers.

Microkernel pattern. By using this pattern, the architecture of an application contains 5 kinds of architectural elements: a micro-kernel; internal servers; external servers; adapters; clients. In practice, the micro-kernel encapsulates the lower-level system functionalities, e.g. interprocess communicating and file processing, as the foundation infrastructure of an application; internal servers provide various extended and complex services; external servers are used to request that appropriate services are executed according to the interpretation of the requests it has received from the clients. By being structured with this pattern, an application can behave adaptively and provides required services according to the various types of requests that have been received by this application. The non-functional properties provided by the Microkernel pattern are: flexibility, scalability, availability, reliability, maintainability, and changeability. However, it is complex to implement and may have performance issues. Further, since the Microkernel pattern is normally implemented with the Layers pattern, it may have the same disadvantages of the Layers pattern.

Reflection pattern. This pattern is used to provide large scale changeability. By using this pattern, the architecture of an application will be separated to two different layers. One layer includes the data elements that represent the states, properties, and configuration of this application; another layer includes the logic that can access and modify the data elements reflecting this application. These two layers normally communicate with each other through a specifically designed protocol. The applications created with this pattern inherently support runtime modification of these applications. However, it is not generally easy to implement this pattern, and in some situations, it may be impossible to do so. Further, a dependable mechanism is required to protect the applications from incorrectly modification.

Blackboard pattern. This pattern has been created to solve complex and uncertain problems. This pattern includes 3 main elements: blackboard, knowledge sources and controllers. The blackboard is responsible for managing a set of data structures in the solution space of a problem; the knowledge sources are special architectural elements that provides properties to solve a set of issues, which is the subset of the problem; the controllers monitor the blackboard, and make decisions on the next step according to the observable information collected from the blackboard. By using this pattern, a self-adaptive application can make decisions on uncertainty. Therefore, the Blackboard pattern provides availability and reliability. Further, this pattern also provides changeability and maintainability. However, this pattern does not ensure that a self-adaptive application created with this pattern can behave correctly. Further, the performance is not good since the knowledge sources cannot be executed in parallel. Finally, it is difficult to initialise the Blackboard pattern.

Shared Repository pattern. This pattern can be treated as a variation of the Blackboard pattern. The core element of this pattern is a central repository that can be shared with other elements. The repository can work as only a data warehouse, or a data warehouse with a synchronous mechanism, or a data warehouse with a notification mechanism. Self-adaptation processes can be implemented on the basis of this pattern, so that each module that is designed to perform a certain MAPE activity can collaborate through reading information from or writing information to this shared repository. The implementation complexity of this pattern is controllable. Further, this pattern also provides flexibility, scalability, changeability and maintainability. However, the performance of self-adaptive applications created with this pattern may be negatively affected if the access to the shared repository is synchronous.

Publisher-Subscriber pattern. The main elements of this pattern are a publisher and a set of subscribers. In detail, a subscriber can subscribe to certain information maintained by a publisher, and if there is any change to the information, the publisher will notify the subscribers who have subscribed to this information. In practice, this pattern can be used to implement the monitoring activity of a self-adaptation process. This pattern can be implemented in an asynchronous manner; therefore, the performance of self-adaptive applications created with this pattern is controllable. Further, this pattern provides flexibility, scalability, changeability, and maintainability. However, the reliability and availability of a self-adaptive application created with this pattern depend on the reliability and availability of the shared repository.

Even Channel pattern. This pattern is a variation of the Publisher-Subscriber pattern, but it allows multiple publishers rather than single publisher. It inserts an architectural element known as an event channel between a publisher and its subscribers. As a consequence, the event channel is the subscriber of the publisher, and it also the publisher of these subscribers. This pattern provides better maintainability, flexibility and scalability than the Publisher-Subscriber pattern.

Producer-Cache-Consumer pattern. This pattern is another variation of the Publisher-Subscriber pattern. The information provided by the producer is stored in the cache, and the consumer only needs to access the cache and consume the information stored in the cache.

Master-Slave pattern. This pattern is designed to decompose a complex task into several sub-tasks and solve this complex task by solving these sub-tasks. The core architectural elements are a master and slaves. The master element is responsible for coordinating and scheduling these slaves. By dispatching the same tasks to multiple slaves, this pattern can help an application be fault-tolerant, so that the application has better availability and reliability. Further, by allowing the slaves to perform sub-tasks in parallel, the self-adaptive applications created with this pattern can work in a parallel and concurrent manner, so that the performance of these applications can be greatly improved. Additionally, this pattern provides changeability, flexibility, and scalability. However, in some situations, it is not possible to use this pattern since a task may not be possible to subdivide. Further, it is difficult to implement this pattern.

3.4 Summary

To help understand self-adaptive application architecture, this chapter has discussed the concepts of software architecture with respect to self-adaptation, including a self-consistent terminology. Further, this chapter discusses the candidate properties of a self-adaptive application, including both functional and non-functional properties. Some existing architectural patterns that can be used to solve self-adaptation have also been listed and discussed, including the properties provided by these patterns.

4 The role-behaviour-based modelling language

As discussed above, business context constrains the design of a self-adaptive application architecture. The first step towards a successful architecture is to confirm the required domain-specific functionalities, which are the functional constraints in the business context, and hereby shape the functional outline of the architecture [22]. To develop a self-adaptive application in a complex domain, it is essential for architects to have a clear understanding of the uncertainty of the business context of this application, so that architects can make appropriate architectural decisions on the selection of the appropriate self-adaptation processes and the suitable architectural patterns for supporting these processes at the operational phase of this application.

A domain-specific model is an abstract representation of the business context of a software application. Therefore, to a certain extend, a software application architecture is constrained by a domain-specific model. Considering a software application architecture actually bridges domain-specific knowledge and implementation [5], it is also important for architects to ensure that a software application architecture, including architectural structures and properties, conforms the domain-specific model constraining it. In addition to this, considering a domain can be uncertain, and there may be some unknown aspects of this domain, it is also important for architects to ensure that the certainty is complete. That is, the designed behaviour of a software application in a changing context will not lead to an unexpected state, e.g. failure.

Further, for a software application, which is designed for performing certain business tasks, in a certain domain, the domain-specific knowledge is vital to the success of this application. But, for a very complex domain, it is hard for architects to have a clear understanding of this domain. Therefore, it can be very helpful if domain experts can join the architectural design [27]. To allow the participation of domain experts in the architectural design, an understandable method for describing domain knowledge that will affect architectural decisions should be available.

Given the above factors, it would not be suitable for domain experts to model domain-specific certainty with existing methods, such as UML(Unified Modelling Language) and formal methods. Since these methods may be too complex for domain experts. For example, the UML, which is widely used to model software systems, has more than 10 kinds of diagrams and various ambiguous graphical concepts and this tool is still evolving. Considering this, it may be difficult for domain experts to learn: there is an intuitive evidence, a brief guide [41] for using UML has more than 200 pages. Formal methods are much more difficult to learn, even for software architects. Further, these methods are all designed for describing certainty rather than uncertainty, and it would be very complicated for users to describe uncertainty information by using these tools. On the other hand, some more flexible tools, e.g. informal graphical tools, are lacking of support to the analysis of the models created by these tools.

To solve the above issues, in this chapter, we present a novel language for modelling business context, including business processes and domain-specific plans for dealing with uncertainty. By using this language,

domain experts can describe the domain-specific concepts and workflow in a concise way, and check the completeness of the models created in this language, and then architects can generate the functional outline of the architectures of their systems by adopting suitable architectural patterns on the basis of consistency checking.

4.1 Design principles

This chapter introduces the design of a modelling language for describing domain-specific business processes. As mentioned above, the design principles of this language are:

- simplicity. This language should have a concise grammar so that it is easy for domain experts who have limited software knowledge to understand and use. Further, this language should have limited concepts that are easy to be mapped to business concepts;
- expressiveness. This language should be capable of describing domain-specific certainty information, e.g. business constraints, business processes. Further, this language should be used to describe domain-specific uncertainty information and the corresponding reslolutions, e.g. domain specific changes and the domain-specific reactions to these changes;
- capability of managing complexity. This language should allow users to describe their models at various abstract levels, and provide concepts for users to reduce the complexity of their models efficiently;
- correctness. This language should have a formal foundation for users to understand the models in this language in a correct way. Further, a formal foundation is essential for users to perform formal checking, e.g. consistency checking, completeness checking, so that a domain-specific model in this language can provide a solid foundation for making architectural decisions at architecture design stage.

In practice, we introduce a role-behaviour-based method in which users can describe business constraints through specifying the behaviour of domain-specific roles. Further, we also introduce a high-order method for specifying various kinds of domain uncertainty with complex event patterns. In summary, by proposing this language, we have made the following contributions to the field of software engineering:

- providing a concise language that have limited concepts and easy for both domain experts and software architects to use;
- providing a method for users to describe not only the knowledge about domain-specific certainty, but also the knowledge about domain-specific uncertainty;
- provide a method for users to manage the complexity of their models by specifying certainty and uncertainty with roles, behaviour and events defined at various detail levels;

- providing a method for users to perform formal checking, e.g. consistency checking, completeness checking. As a consequence, domain experts can take part in the architecture design by constraining the functional topology with their domain-specific models.

4.2 Conceptual model

The following figure illustrates the proposed conceptual model of this language. We define a business domain as a set of business constraints. The core concept is role. A role has its own behaviour that is constrained by the roles context. A role can extends other roles and therefore behave like the others. A role extends the other roles is termed as the "child role", and the role being extended is termed as the "parent role". Two different roles do not have the same behaviour. A context comprises a set of events. From the outside of a role, the visible behaviour of a role is defined by a pair of events $<e_1, e_2>$, where e1 indicates the event triggering this behaviour; e2 indicates the impact of the behaviour. An event represents a change. An event can be complex or simple. A complex event is a set of simple events happening in a temporal order. An event is the consequence of an action. The invisible behaviour of a role indicates the responsibilities of this role. The invisible behaviour is defined as a set of actions being performed sequentially or parallelly. An action has a name, and it represents an invisible event sequence. A process indicates the collaboration of various roles. A process is denoted as a temporally ordered set of visible behaviour of the participants of this process.

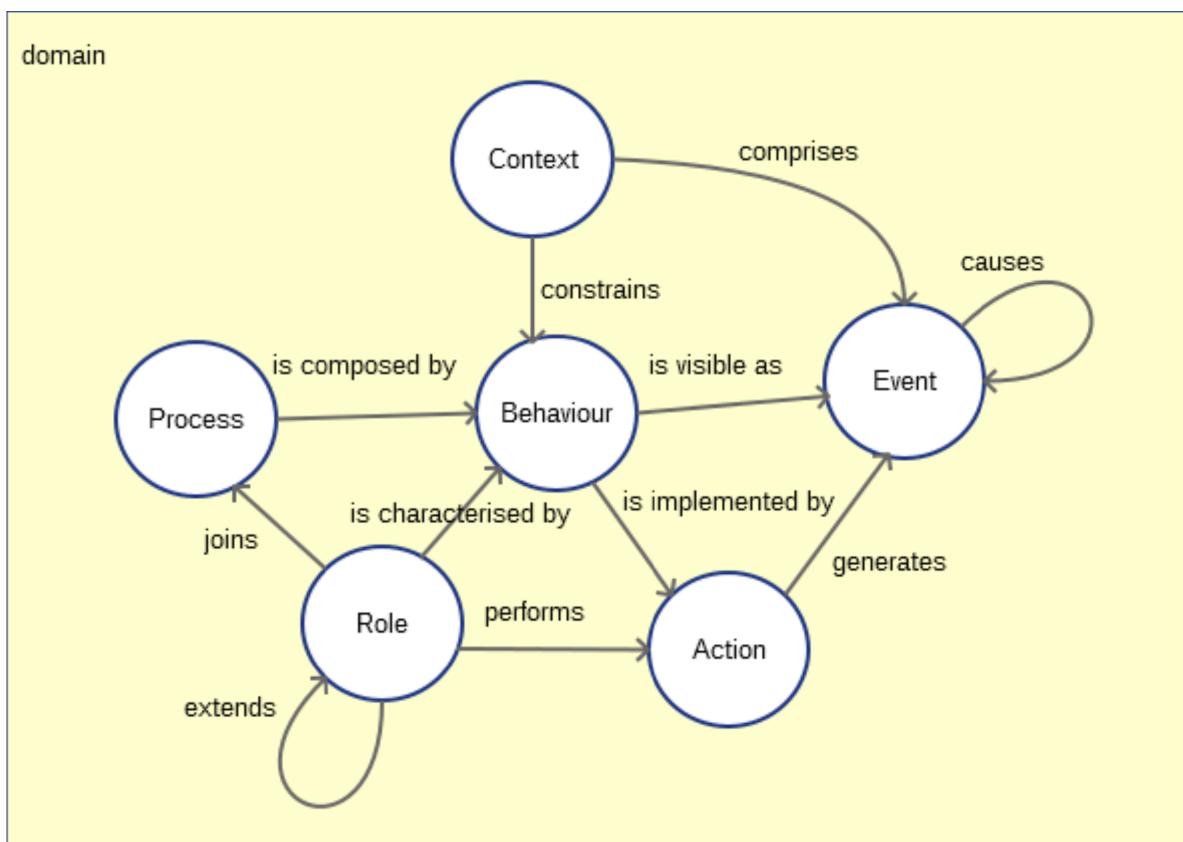


Figure 1: the conceptual model

This conceptual model refers to concepts of a well-designed ontology model [42] [43] [44]: The world is made of things. A thing can be simple or composite. A composite thing is a combination of simple things. A thing possesses a set of properties, and the properties of two things are always not the same. A class of things possesses a same group of properties which is a subset of the properties of each thing in this class. A property can be observable or unobservable. A thing is characterised only by its observable properties. At a given time, a property has a given value. The state of a thing is the values of all properties of a thing at a given time. The state space of a thing is all possible states of this thing. A constraint on the possible value of a property is called a state law. A state conforms to a state law is a lawful state. An event represents a change on the states of a group of things. The behaviour of a thing in a given period is denoted by an ordered set of states in this period. A thing X acts on another thing Y if Y's state has been affected by the present of thing X. If X acts on Y, and Y acts on X, then X interacts with Y. A system is a composite thing that cannot be decomposed into non-interacting simple things. An event happening on a thing is external only if this event is caused by a stimulus from the environment of this thing. Otherwise, an event is internal. Further, a thing will not behave arbitrarily. That is, without external stimuli, a thing will change its state only if there is a lawful transition from this state to another state. If there is such a lawful transition from state A to state B, then state A is unstable. A thing in an unstable state will change to a stable state. A thing can be changed by external stimuli since the value of at least one property of this thing can be changed by external events.

Based on the above ontology, we can clarify our conceptual model in this way: a role specifies the lawful behaviour of a class of things in its lifetime; invisible behaviour represents the internal behaviour of a thing, which is unobservable from the outside; visible behaviour represents the observable behaviour of a thing from outside; A thing acting as a role will behave following the behaviour restrictions imposed by this role; a simple event represents a change only and a complex event also represents a change between states, which can be split into an ordered series of simple events. An action in this model represents the changes, which are imposed by a role, on the environment of the things acting as this role, or the changes on the things acting as other roles; A business process represents an ordered set of lawful transitions. With the above conceptual model and ontological explanation, we can therefore provide a formal foundation for this language. Based on this, the rest of this section will introduces how we solve the other issues mentioned above.

The first design concern of the design of this language is to reduce the complexity of domain process models in this language. One reason for the increasing complexity of a business model is the need for describing complex collaboration among business entities, e.g. temporal order and potential conditional choices at every time point. To solve this problem, this language is built on an assumption that a business role behaves independently, a certain role does not have knowledge about the other roles. Thus, in a collaboration, one role only needs to respond to the changes happening on itself. As a consequence, on the basis of the concepts of roles and behaviour, users can describe a complex collaboration in an implicit way: a collaboration can be extracted from the interrelated behaviour of roles. Due to the inherently temporal attribute of an event, there is no need to define the temporal order of interactions in a collaboration explicitly, since the temporal order

of a certain set of behaviour of roles is actually defined by the temporal order of the events related to the set of behaviour.

Further, by introducing the event concept, including simple and complex events, we can therefore describe the domain-specific uncertainty in a simple way at a desired level. Since an event inherently represents changes, and thus it is ideally to describe uncertainty with events. To deal with uncertainty, users can describe the role behaviour to uncertainty events. Additionally, by using various event patterns, users can describe uncertainty in a concise and sophisticated way. That is, users can manage the complexity of their models by deciding the abstract level of the description of the events.

By establishing the extending relationship among roles, and by deciding the abstract level of events, users can describe the knowledge of a domain at various granularity in a collaborative way. In this way, domain experts can cooperate together for describing complex domain-specific knowledge through decomposing the knowledge into various layers. For example, a manager can be responsible for describing overall information in management layer, and an employee can then elaborate the detailed information in technical layer. In detail, the syntax of this language is defined in the following section.

4.3 Grammar

The grammar of this language is defined in the EBNF (Extended Backus-Naur Form) [45] notation. A domain defines the scope of a model, the syntax is:

```
domain-definition: ‘domain’ domain-name=identifier '{'event-definition* role-definition* process-definition* '}'
```

In this way, a domain has:

- zero or more business events;
- zero or more business roles;
- zero or more business processes;

The syntax of event definition is:

```
event-definition: 'external'? ‘event’ event-name=identifier ('(' arg-list ')' ':' event-expression)?
arg-list: args+=identifier (',' args+=identifier)*;
```

An event can be explicitly defined as an external event by beginning with a keyword “external”. For example, a key has been pressed by a user. Optionally, an event can be defined as a complex event by specifying it as a high-order predicate explained by an event expression. Otherwise, we can define an event as a simple one by simply giving it a name. The syntax of event expression is defined as:

```
event-expression: simple-event
                | 'not' simple-event
                | simple-event 'and' event-expression
                | simple-event 'or' event-expression
```

```
                                   | simple-event 'while' event-expression
                                   | simple-event 'then' event-expression
                                   | simple-event 'after' event-expression
                                   | simple-event 'before' event-expression
                                   | simple-event 'during' event-expression
                                   | '(' event-expression ')'
simple-event: event-name = identifier
```

This syntax shows an event calculus that will be discussed later. In this definition, a simple event is denoted by an identifier-based name. A complex event is an event defined by an expression specified with a set of operators and other events.

The syntax of a role is shown as below:

```
role-definition: ‘role’ role-name-list +=identifier '{' behaviour = reaction* '}'
reaction: internal-transitions+= event-logical-expression*
      | 'on' ('context''::')? stimuli = event-logical-expression  '{' invisible-behaviour+ = event-logical-
expression* (':' result = event-logical-expression )?  '}'
event-logical-expression: simple-event
                                         |
                                         | identifier logic-operator event-logical-expression
                                         | '(' event-logical-expression ')'
logic-operator: '&' | '|'
```

A role is characterised by its behaviour in different contexts. According to the conceptual model, the behaviour of a role specifies a transition between states of the things acting as this role, and thus we can describe behaviour with events. In the syntax, we describe a behaviour as a set of reactions to certain external stimuli, and each reaction is denoted by a tuple <c, a, r>, where c is a set of external events, a is a simple or complex event and it indicates the invisible behaviour of this reaction, and r indicates the result of this reaction, which is also a set of events. For example, the following statements define a role and its behaviour, and these statements mean: analyser is the name of a role; this role should behave when external event AnalysisRequest happens and cause event TradingSignalFound; In detail, the behaviour of role analyser is achieved by a set of events, which represents internal lawful transitions.

```
role analyser{
      on AnalysisRequest{
            prepare_strategy then calculate_strategy: TradingSignalFound
      }
}
```

The grammar of a business process is specified as below:

```
process-definition: ‘process’ process-name=ID '{' participant-name-list = participant* interactions =
transition-ordered-set * '}'
participant: ‘role’ name = identifier
transition-ordered-set: transition-logical-expression (‘->’
```

```
transition-
ordered-set)?
transition-logical-expression: transition (logic-operator transition-logical-expression)?
transition: role-name = identifier <(trigger = event-logical-expression? ',' result = event-logical-
expression?)?>
```

Actually, the collaboration among a set of roles can be extracted from the behaviour of these roles. But, we still decide to clarify business processes, since it is still essential for users to clarify the scope of a certain business process. According to this syntax, a business process has its own participants that are roles. For representing the interactions among roles in a simple way, we denote an interaction by an ordered set of transitions. Each transition is described by a pair <trigger, result>, where trigger is a set of events, which triggers the transition, and result is a set of events, which is the result of performing this transition. Trigger or result can be empty, which means that there is no external stimuli nor external impact.

4.4 A simple event calculus

o solve the complexity of a domain process model, another important concern is the size. Traditionally, researchers tend to use first-order conditional statements, e.g. if-else, to describe all aspects of the dynamisms of a business process. As a consequence, assuming that it is possible to know everything about the business processes of a very complex domain, the size of a business process model can be dramatical huge. To solve this issue, we prefer a higher-order way. In this section, we propose a simple calculus for defining complex events. The basic definitions of this calculus is shown below:

event = *e* | **not** event | event **and** event | event **or** event | event **than** event | event **while** event

| event **before** event | event **after** event | event **during** event

An event represents a change in a business domain. An event has a name, and can be complex. A complex event is a set of event happening in a temporal order by being connected with a set of temporal operators that is shown below:

- **not**. Indicating the unoccurrence of an event;
- **and**. Indicating both events should happen (not have to be in parallel);
- **or**. Indicating at least one of two events should happen;
- **then**. Indicating two events happen in a causal way;
- **while**. Indicating two events happen in parallel;
- **after**. Indicating an event happens after the occurrence of another event;
- **before**. Indicating an event happens before the occurrence of another event;
- **during**. Indicating the occurrence of an event overlaps the occurrence of another event;

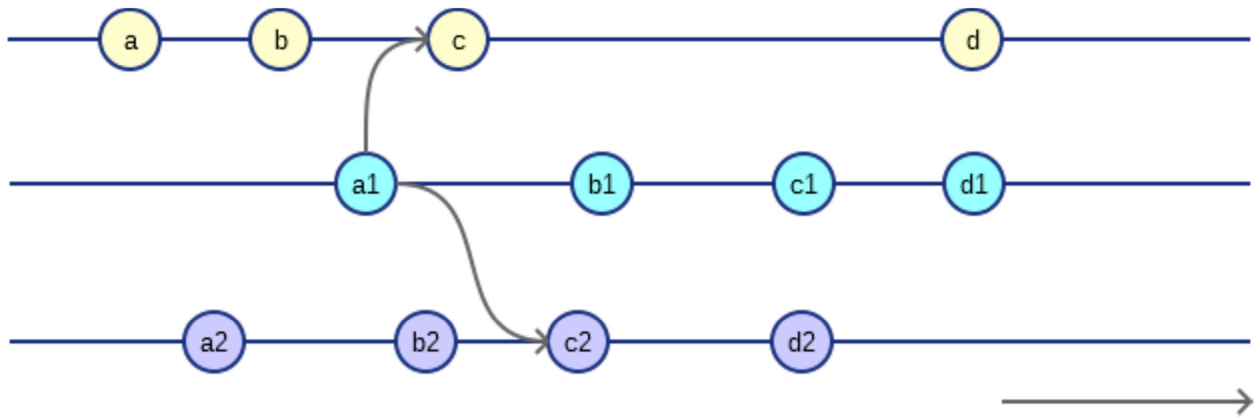


Figure 2: an example of events happening in temporal orders

The above figure illustrates an example of the events happening in temporal orders. For example, we can say a before b, or a2 after a, according to the time line; we can say c and c2, since the occurrence of both c and c2 are being expected after a1, and if the occurrence of c and c2 is uncertain, we can say c or c2; we can say d while d1 since these two events happen in parallel; we should say c1 and (not b2) in some cases since b2 cannot happen after c1.

In practice, the event patterns can be implemented in various way. For example, we can implement the unoccurrence of an event as "not sensing the occurrence of an event within 5 secionds"; we can also implement the pattern "event A while event B" as "the time difference between the occurrence of event A and event B is no more than 3 seconds". We have to admit that we may not list all possible event patterns here, but, in this dissertation, we concern the issues of how our method can be used to describe uncertainty from a new perspective.

By having these operators, we can thus define a complex event by using high-order predicates as follow:

```
some_person_break_into_my_house(
        my_house_is_lighting_up,
        i_am_not_in_home,
        shadow_shaking_on_window): i_am_not_in_home
                                        while my_house_is_lighting_up
                                        while shadow_shaking_on_window
```

In this way, the size of a model in this language can be controllable. Further, based on the calculus, the domain models in this language can be used to perform consistency checking for testing whether the design and implementation of software systems being created on these models are desired or not.

4.5 Consistency checking

As mentioned above, a business model constrains the business-specific aspects of a software architecture, e.g. business functionalities. By checking the behaviour consistency between a business process model and

the design or implementation of a software system, both domain experts and software engineers can be more confident about their projects. There are two ways for checking the consistency:

4.5.1 Using process calculus

By adopting an event-based mechanism, it is inherently convenient for us to provide a formal foundation for our language by using some formal language, such as process calculus. As an example, we can adopt CCS (Clculus of Communicating Systems) [12]. The reasons of adopting CCS may be:

- CCS is the seminal work in the filed of process modelling and it has been cited by many work as a research foundation;
- CCS has a concise grammar and limited concepts;
- By using CCS, there is a potential choice for us to transfer our language to other improved process calculus, e.g. π-calculus [35].
- It is easy for us to map the concepts of our language to the concepts of CCS.

To apply CCS, we can first map the concepts of this language to the existing methods of CCS. For example: a business event represents the execution of a CCS action; a business role is a CCS agent; a business process is a CCS process; for changes happening in the context of a role, we denotes these changes as the result of a set of invisible CCS actions triggered by a set of invisible CCS events; the sequence of CCS actions can be represented by a complex event.

4.5.2 Using transitions

The method introduced in this section is based on the following assumption, which is also the assumption on which the process calculus is built [46]:

For a give external stimuli, if two things always behave in the same way, then we can say these two things are identical.

Therefore, to check the consistency between two things, we can compare all observed transitions of these two things, which are triggered by the same external stimuli. As a consequence, we can then compare the event chains of both things, which start from the same external events and include visible event only. For example, the following two systems can be treated as similar:

system A: $(not\,\alpha) \rightarrow \beta \rightarrow (\gamma\, while\, \sigma)$

system B: $(not\ a) \rightarrow b \rightarrow (c\ while\ d)$

If the semantics of the events in the above systems can be understood in the same way, e.g. event α semantically equals event a, then we can say that system A can semantically replace system B, and vice versa. B is termed as the semantically replaceable structure of A.

4.5.3 Generating architectural outline with business constraints

By utilising the above method, we can propose a novel method for generating architectural outline with domain-specific models. This problem can be understood from another viewpoint: how to find two similar structures. To solve this issue, at first, we need to define a set of meta actions, for example:

- manage. A unidirectional push-action regarding producing or recycling, e.g. creation of a thing, deletion of a thing;
- communicate. A bidirectional action regarding coordinating, ;
- get. A unidirectional pull-action regarding requesting and receiving;
- send. A unidirectional push-action regarding responding, e.g. data transmission, replying;

We can then specify the relationship between two things by using these actions. Further, based on the conceptual model introduced above, we can then specify a structural relationship between thing A and thing B as a transition from a set of internal events of A to a set of external events of B, which defines a direct impact of A on B. A structural relationship is denoted as a pair a(r1, r2)<e1, e2>, where a is the name of this transition that happens across role r1 and r2, e1 is the internel event set of A, and e2 is the external event set of B. Additionally, if we carefully assign semantics to a such structural relationship with the meta action set defined above, we can then find similar structures.

For example, assuming in a given financial trading domain, there are two roles: trader and analyser. A trader needs to retrieve analysis result from an analyser. As a consequence, we can model this process as:

```
process Request_Analysis_Result{
        role trader
        role analyser

        trader<, request_analysis> -> analyser<receive_request, >
}
```

By a clear understanding of this process, we can then specify this process as a relationship defined as:

get(trader, analyser)<request_analysis, receive_request>

Further, if we can find an architectural pattern that supports the similar action, we can simply generate an architectural outline by initialising the architectural style with domain-specific information. For example, assuming we have a candidate architectural pattern, which defines a structural relationship like:

get(client, server)<request, response>

Thus we can adopt this architectural pattern as a candidate architectural solution.

4.6 Completeness checking

We define the completeness as the capability of being complete so that the emergence of certainty and uncertainty information will not lead to unexpected behaviour. For the models in our language, the completeness means that any certainty and uncertainty information being represented by events are

associated with one and only one lawful transition. That is, in a complete model in the language designed in the above section, for any event, there is always a role will take care of it, and no role or behaviour is useless.

To check the completeness of a model, we propose to use a graphical method defined as follow:

- cycle. A cycle represents a role;
- directed link. An incoming link represents an external stimulus and an outgoing link represents an external impact of one role's behaviour.

Therefore, in practice, the completeness of a model means that an outgoing link always points to a cycle.

4.7 Summary

As a conclusion, this chapter introduces a role-behaviour-based language for modelling domain-specific business process, including a simple high-order event calculus for improving the capability of describing uncertainty information with a controllable complexity. The concepts of this language are built on a proven ontological model so that it can help users to perform formal checking, e.g. consistency checking and completeness checking. In addition to this, this chapter also proposes a method for generating architectural outline from a domain specific model.

By providing the above benefits, we have establish a solid foundation for our future research. To design a domain-specific self-adaptive application architecture, we need to have an architecture description language that can describe dynamism aspects of an architecture, which will be introduced in next chapter.

5 Grasp+:an enhanced version of Grasp

Having a language of modelling domain-specific business processes, an architecture description method is required to describe the architecture of a self-adaptive application, which is constrained by the domain-specific elements introduced in the previous chapter. In order to describe the a self-adaptive application architecture, this method should be able to:

- allow users to describe not only static aspects, e.g. topology of architectural elements, but also dynamic aspects of a software architecture, e.g. runtime creation of a component;
- allow users to describe what uncertainty exists and how the uncertainty can be handled at architecture design stage.

Specifically, to describe an architecture created with the domain-specific models in the language introduced in the previous chapter, the following principles should also be considered:

- It is important for architects to specify the information about how the architecture is constrained by domain-specific elements;

- This method should be consistent with the modelling language presented in the previous chapter.

Considering these issues, we would like to design a new language, Grasp+, to describe self-adaptive application architecture. To simplify this work, the language should be built on an existing study that has been proven, and this language should also be extensible. Therefore, Grasp [13] is adopted by this work, since:

- Grasp has been proven to describe static aspects of an architecture;
- Grasp allows users to specify rationales that can include requirement information;
- Grasp has a concise grammar;
- Grasp is extensible.

As a consequence, this chapter outlines Grasp first, and then discusses the problems of Grasp. Further, this chapter presents how our work improves the original Grasp through introducing a role-based meta-model. Finally, we illustrate this language through a simple case study.

5.1 Grasp

Grasp is a textual and general purpose ADL based on the intuitive architecture model introduced by Perry and Wolf [6]. The core concepts of Grasp are architectural elements and rationale. As Perry and Wolf [6] pointed out, a rationale represents the motivations behind architectural decisions. In Grasp, a rationale is a set of reasons described in natural language. A rationale is associated with an architectural element with a keyword "because". Rationale A can extend Rationale B and A can inherit the reasons included in B. According to the design of Grasp, a rationale can also be a quality attribute that includes a set of quality properties. From this viewpoint, it would be simple for us to connect both business constraints with architectural elements by specifying business constraints with rationales. The conceptual model of Grasp is shown as bellow.

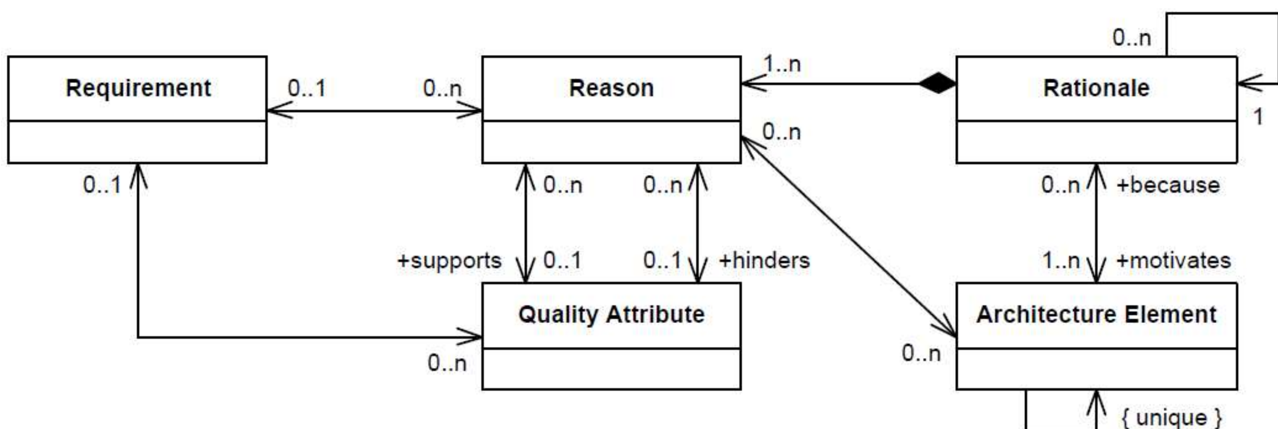


Figure 3: the conceptual model of original Grasp, cited from [13]

In detail, an architecture element in Grasp can be a component or connector.

5.2 Problems

To describe the domain-specific architecture of a self-adaptive application, the problems need to be solved are:

- inability to describe dynamic presence of architectural elements;
- inability to describe how to respond to uncertainty;
- inability to describe the temporal order of the collaboration among architectural elements;
- inability to describe how domain models constrain architectural decisions;
- the lack of a formal foundation.

5.3 The role-action-based conceptual model

To solve the above issues, this section introduces a role-action-based conceptual model on which the new language is built. Having investigated the existing studies in the field of architecture description methods, especially architecture description languages, we found that the concept of roles is used to define the collaborators at each end of a connector. That is, roles are characterised by connectors, rather than the other way round. However, it may be inappropriate since roles should not be defined with connectors. Let us consider the following question: why can we call a component as client and why can we call another component as server? Because these two components are linked by a certain kind of connectors? Obviously not. We define the roles of architectural elements on the basis of their responsibilities. Further, in a self-adaptive application architecture, an architectural element can perform different roles in different situations only because it have different responsibilities in different situations. Therefore, being specific, the assumption of our work is:

In a self-adaptive application architecture, the architectural elements are arranged dynamically due to the changes of the runtime responsibilities. At a given time, for a self-adaptive application, an architectural element is changed due to this application's understanding about this element's responsibilities at this time.

Thus, we can define our conceptual model based on the conceptual model of the Grasp as follows:

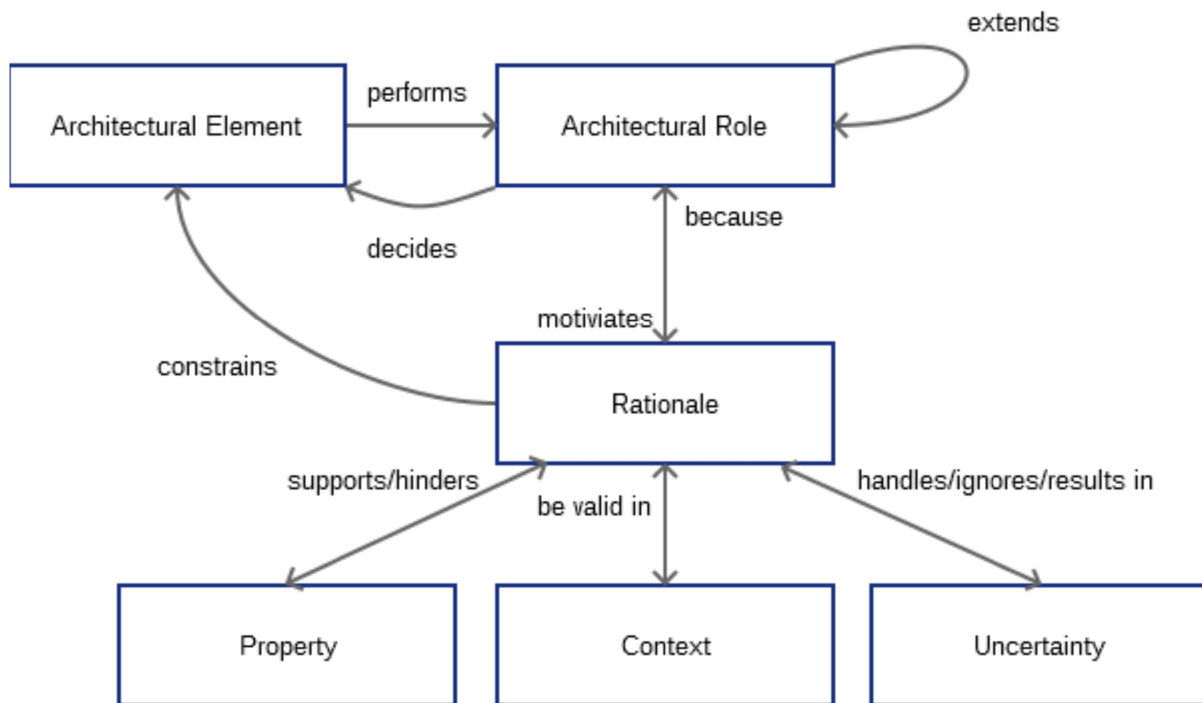


Figure 4: the updated conceptual model

According to this conceptual model, rationale includes not only functional properties and non-functional properties, but also contexts and uncertainty. For example, the architectural elements are structured following the Blackboard pattern due to uncertainty in the solutions to a complex problem. Further, rationales can be only valid in a certain context. In the updated conceptual model, rationales do not motivate architectural elements, but rather architectural roles. From this viewpoint, architectural roles can be more abstract than architectural elements. An architectural role can extend its responsibilities by inheriting another architectural roles, so that this language provides a certain extensibility. Further, according to this conceptual model, rationales can also include uncertainty. That is, rationales constrain the presence of architectural elements, e.g. a rationale is the trigger to the creation or removal of an architectural element.

5.4 Grammar

Since Grasp+ is built on Grasp, therefore, one design principle is that the grammar of Grasp+ should not be very different from Grasp. The core concept of Grasp+ is role. To allow users to specify their own roles, the original grammar of Grasp has been updated as:

```
start: architecture_definition;
architecture_definition: annotation* kw='architecture' name=identifier '{' architecture_inner_statement*
system_statement architecture_inner_statement* '}';
architecture_inner_statement
: requirement_statement
        | quality_attribute_statement
        | role_statement
        | template_statement
        | rationale_statement
```

```
          ;
role_statement: annotation* kw='role' name=identifier ('{' action_statement* '}')? because_opt? ';'?;
action_statement: annotation* kw='action' name=identifier '(' parameters? ')' ('{' pattern_action* '}')?
because_opt?';'?;
pattern_action: pattern_action_keywords architectural_element ('in' life=parameters)? because_opt? ';';
pattern_action_keywords: 'create'|'remove';
architectural_element: type='link'? subject=identifier ('to' provider=identifier)? ('via' connector =
identifier)?;
```

As shown in this grammar, the definition of an architecture now includes a set of role statements. Each role statement specifies a role. A role is characterised by a set of actions. Each action refers to a certain architectural task: create or remove a link or component. The presence of roles and their actions are motivated by rationales. Further, now a template can perform one or more roles.

```
template_statement
          : annotation* kw='template' name= identifier (':' maxinst=integer_literal)? '(' parameters? ')'
act_as_opt? extends_opt? because_opt? '{' statement* '}'
          ;
act_as_opt: kw='act_as' roles+= identifier (',' roles+=identifier)*;
```

The following grammar is used to define a system. Users can specify rationales as the triggers to an architectural change by using keyword ‘on’. Therefore, by treating rationales as external events and treating architectural roles’ actions as lawful transitions, we partially provide a formal foundation by referring to the work introduced in the previous chapter.

```
system_statement
          : annotation* kw='system' name=identifier because_opt? '{' statement* '}'
          ;
statement
          : layer_statement
          | context_statement
          | component_statement
          | connector_statement
          | link_statement
          | check_statement
          | property_statement
          | action_statement
          | behaviour_statement
          ;

context_statement
          : annotation* kw='scenario' name=identifier ';'
component_statement
          : annotation* kw='component' name=identifier '=' base=identifier '(' arguments? ')' because_opt? ';'
          ;
link_statement
```

```
            : annotation* kw='link' name=identifier? link_consumer 'to' link_provider because_opt? ('{'
link_inner* '}' | ';')
            ;
layer_statement
            : annotation* kw='layer' name=identifier layer_over? because_opt? '{' statement* '}'
            ;
behaviour_statement
            : annotation* member_expression life_cycle_opt? because_opt? ('{' event_action_rule* '}')?
            ;
event_action_rule
            : 'on' event_name_list += identifier (, event_name_list += identifier)* member_expression
because_opt?';'
            ;
life_cycle_opt: kw='in' life_cycles+= identifier (',' life_cycles+=identifier);
member_expression
            : member_part ('.' member_part)*
            ;
member_part
            : name=identifier '(' member_args? ')'
            | name=identifier
            ;
parameters
            : parms+=identifier (',' parms+=identifier)*;
```

To illustrate how to describe the dynamic aspects of an architecture, the next section illustrates the application of this language in a case study.

5.5 Case-Study: An intelligent data-providing system

Imaging there is an intelligent data-providing system that can provide data according to its clients' requests. To improve the system's performance, the system works in an asynchronous way. That is, it can process incoming events asynchronously by adopting an asynchronous mechanism. Further, to manage the data transferring throughput in the local network environment, the clients and server of this system communicate with each other in non-reliable protocol, e.g. UDP; However, in the Internet environment, the communication mechanism works on reliable protocol, e.g. TCP. In some extreme environments, this system needs to establish a channel between its clients and server to ensure the data transfer, in case of the data consistency and safety problems; if there is a dramatically increasing incoming requests, this system will activate a connection pool to cache the incoming requests in a F-I-F-O way.

To elaborate the use of Grasp+, this section will assume that all requirements have been elicited properly and the architecture is also carefully designed according to these requirements. Based on this assumption, the related architecture is shown below:

```
architecture Smart_Data_Serv_Sys{
            rationale critical_failure(){
```

```
            reason #'Core functional part does not work';
        }
        role protocol;
        role server{
            action cache_access(conn_pool){
                create link server to conn_pool;
            }
            action asyn_accept_conn(asyn_io){
                create link server to asyn_io;
            }
            action process_request(data_processor){
                create link server to data_processor;
            }
        }
        role client{
            action access(server, protocol){
                create link client to server via protocol;
            }

            action access(channel, protocol){
                create link client to channel via protocol;
            }
        }

        role asyn_io{
            action notify(server){
                create link asyn_io to server;
            }
        }
        role data_processor;
        role channel{
            action secure_access(server, protocol){
                create link channel to server on protocol;
            }
        }
        role conn_pool;
        role monitor{
            action notify(actuator){
                create link monitor to actuator;
            }
        }
        role actuator{
            action subscribe(monitor){
                create link actuator to monitor;
            }
            action activate(subject, lifecycle){
                create subject in lifecycle;
                create link actuator to subject in lifecycle;
            }
            action deactivate(subject, lifecycle){
                remove link actuator to subject in lifecycle;
                remove subject in lifecycle;
            }
        }
        role subject;
        template NetworkComponent() act_as client, server{}
```

```
template NonReliableProtocolConnector() act_as protocol,subject{}
template ReliableProtocolConnector() act_as protocol,subject{}
template AsynProcessorComponent() act_as asyn_io{}
template ChannelConnector() act_as conn_pool,subject{}
template CoreComponent() act_as data_processor, actuator{}
template ConnectionPoolComponent() act_as conn_pool,subject{}
template SensorComponent() act_as monitor{}

system DynamicModel{
        scenario normal_scen;
        scenario tcp_conn_scen;
        scenario channel_scen;
        scenario conn_pool_scen;

        layer CoreLayer over NetworkLayer in normal_scen{
                component core_c = CoreComponent();
                component asyn_io_c = AsynProcessorComponent();
                component sensor_c = SensorComponent();
                component conn_pool_c = ConnectionPoolComponent();

                create core_c in normal_scen;
                create asyn_io_c in normal_scen;
                create sensor_c in normal_scen;

                core_c.activate(conn_pool_c, conn_pool_scen) in conn_pool_scen{
                        on failure system.exit();
                }

                core_c.subscribe(sensor_c) in normal_scen{
                        on failure system.exit();
                }
                sensor_c.monitor(core_c) in nromal_scen{
                        on failure system.exit();
                }

        }

        layer NetworkLayer in normal_scen {
                component client_c = ClientComponent();
                component server_c = ServerComponent();
                component channel_c = ChannelConnector();

                connector udp_p = NonReliableProtocolConnector();
                connector tcp_p = ReliableProtocolConnector();

                create client_c in normal_scen;
                create server_c in normal_scen;

                client_c.access(server_c, udp_p) in normal_scen;
                client_c.access(server_c, tcp_p) in tcp_conn_scen;

                client_c.access(channel_c, tcp_p) in channel_scen;
                channel_c.access(server_c, tcp_p) in channel_scen;
        }

        CoreLayer.core_c.activate(NetworkLayer.channel_c, channel_scen) in channel_scen{
```

```
				on failure system.exit because critical_failure();
			}

			NetworkLayer.server.asyn_accept_conn(CoreLayer.asyn_io_c) in normal_scen{
				on failure system.exit because critical_failure();
			}
			NetworkLayer.server.cache_access(CoreLayer.conn_pool_c) in conn_pool_scen{
				on failure system.exit because critical_failure();
			}
		}
}
```

5.6 Summary

This chapter introduces a novel architecture description language, Grasp+, which is built on the basis of Grasp. By updating the conceptual model with architectural roles and uncertainty, Grasp+ allows users to describe the dynamic aspects of a self-adaptive application, and specify how to respond to changes. Further, by mapping the concepts of rationale and roles' behaviour to the concepts of external events and transitions, we have gone some way towards providing a formal foundation to Grasp+. However, due to the word limit, we are unable to discuss this further in this dissertation. This chapter also illustrates how to apply this language through a case study.

6 An architectural pattern for self-adaptive application

Architectural patterns are reusable solutions to a certain set of design problems, which provide a specific group of non-properties. By choosing and applying the architectural patterns that provide various non-properties, one can design his or her architecture and ensure the quality of the applications of being designed in a principled way. To elaborate how architectural patterns can be used to design self-adaptive application architecture and provide a reusable solution to develop self-adaptive applications with a group of desired non-functional properties, this chapter introduces a new architectural pattern created by applying various existing architectural patterns. This architectural pattern is expected to provide enough flexibility, scalability, changeability, maintainability, controllable performance, reliability and availability. The remaining content of this section will introduce a possible path for software engineers to structure their self-adaptive applications for supporting the self-adaptation processes within these applications.

6.1 The black box

This work starts from a black box. That is, a self-adaptive application is treated as a black box. This black box can always perform domain-specific tasks with desired properties. This design can be understood in this way: this self-adaptive application is well functional, however, its elements are tightly coupled, and thus it has poor maintainability and reuse-ability. That is, this application can only be used in a very limited situation, and it depends magic to work well.

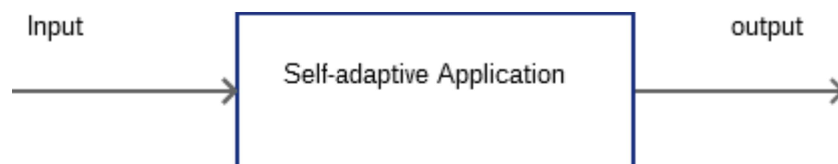


Figure 5: the black box of a self-adaptive application

6.2 The Client-Server pattern

This work is on the top of the above draft. The Client-Server pattern is applied here for clarifying the responsibility of the self-adaptation process within this application. This self-adaptation process is created to solve domain-specific uncertainty by evolving system services or the business processes directly. Therefore, the self-adaptation module that maintains the self-adaptation process can be seen as a client that needs to access the domain-specific module that maintains the business-specific processes and system services. Now we have a more clear structure as shown below. The self-adaptation module works as a client, and it keeps accessing its subject by using proper techniques, e.g., ping request, and the subject works as a server which provides required information to its client, the self-adaptation module. But, we still need to update this structure for the self-adaptation module to retrieve the information about the domain-specific module in a more scalable and flexible manner.

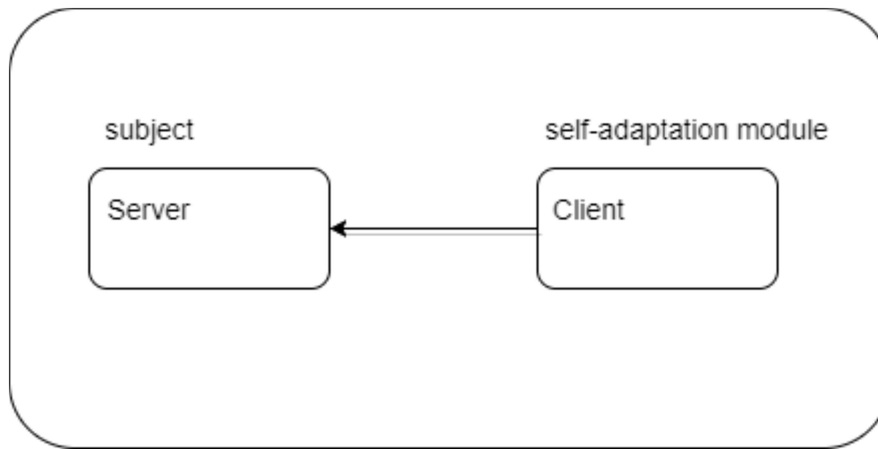


Figure 6: The overall style after applying the client-server pattern

6.3 The Event-Channel pattern

Now, have the above work done, a particular concern can be considered: how to monitor a module of a system in a more scalable and flexible way. To achieve this, the Event-Channel pattern can be further applied to this application. In detail, the self-adaptation module can behave as the subscriber to an event channel, and the domain-specific module can behave as the publisher who is responsible for notifying the changes about itself to the event channel. By applying this pattern, now this application has a clear structure for the self-adaptation module to monitor the domain-specific module. Therefore, this application now has better changeability, maintainability, flexibility, and scalability. After applying this pattern, the overall style is shown below:

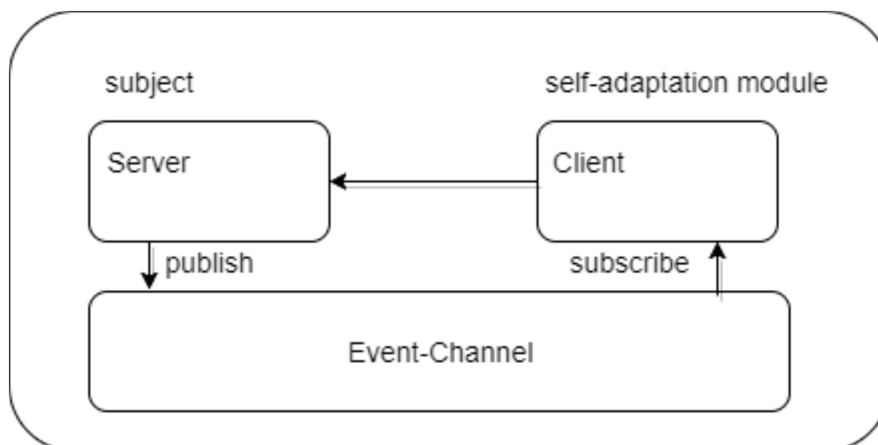


Figure 7: The overall style after applying the event-channel pattern

6.4 The Layer pattern

At this step, to make the structure more clear, software engineers may like to separate different modules into different groups. By applying the Layer pattern, the application is structured into 2 layers: the business layer on the top; and the event channel is in the bottom layer. In detail, the business layer contains the domain-specific module, and the self-adaptation module. The self-adaptation module adapts the domain-specific module according to its observation. As discussed before, by using this pattern, this self-adaptive application now has even better maintainability, scalability and flexibility. Any modification happens in one layer would not affect the other one as long as the behaviour of this layer keeps unchanged. But, since the collaboration between the domain-specific module and the self-adaptation module is built on the event channel, therefore, if there is any failure in the bottom layer, this system may not work properly.

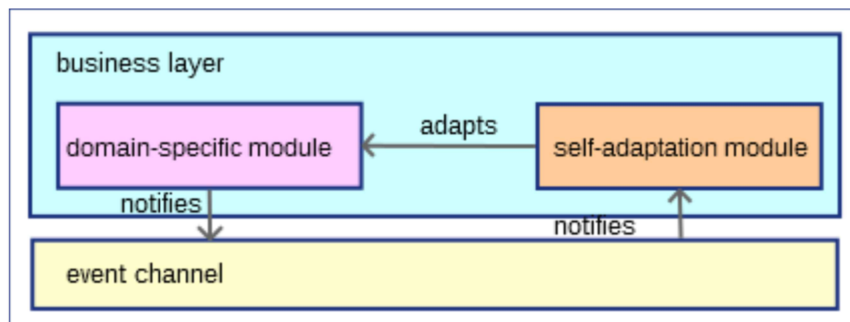


Figure 8: the self-adaptive application architecture with layer pattern

6.5 The Microkernel pattern

To improve the reliability and availability of this application, now the Microkernel pattern is applied into the domain-specific module of this application. By using this pattern, the domain-specific module now has multiple internal servers that can perform domain-specific tasks in parallel, so that the application can have an improved performance. Through replicating a task and deploy the replications on different internal severs, this application can have a better reliability and availability. In detail, the external servers running in parallel will accept the adaptation instructions sent from the self-adaptation module, and then this external servers will send these instructions to the relevant internal servers.

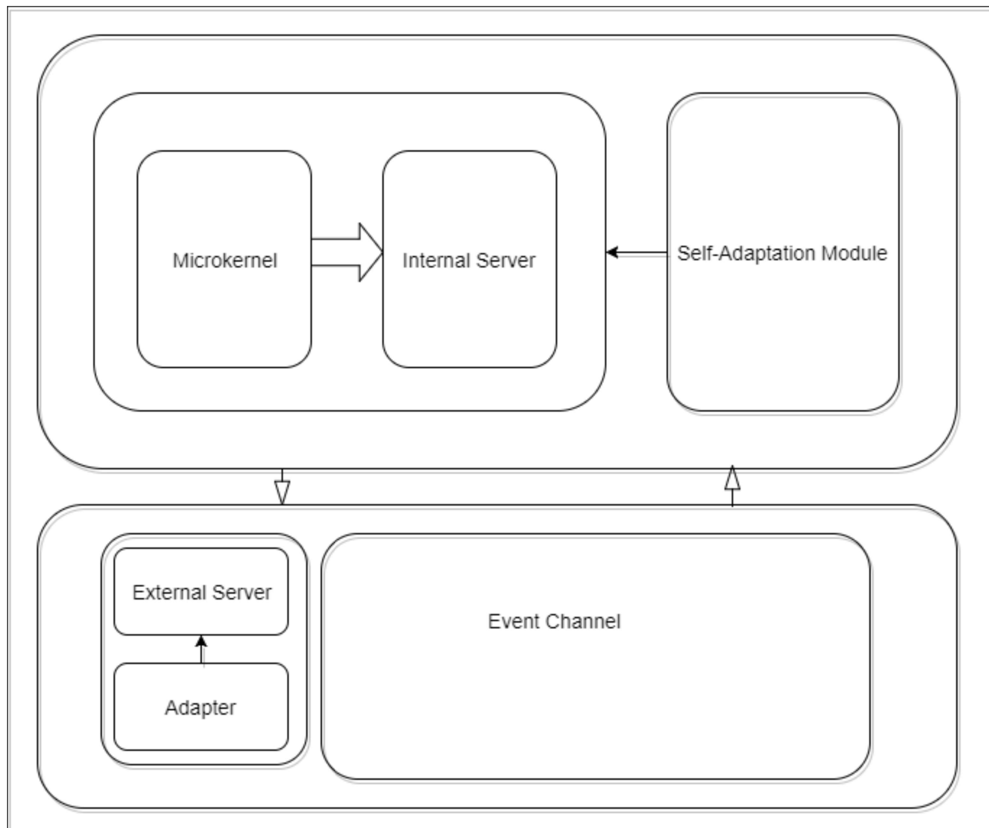


Figure 9: The overall style after applying the microkernel pattern

6.6 The Proactor pattern

The performance issues can be solved by using the Proactor pattern [47]. By refining all elements with this pattern, this application can process events in an asynchronous way, so that the performance of this application can be improved. Further, it also provides great flexibility. But, the complexity of this application will be increased as well. Therefore, this pattern should be used only when the performance is very important.

6.7 The Reactive Messaging pattern

The Reactive Messaging pattern is built on the Actor model [48]. In this pattern, the basic architectural element is actor. An actor does not need to have the knowledge about the other actors. Therefore, an actor only needs to behave according to its observation of its environments. The actors interact with each other via messaging mechanisms. By using this pattern, the flexibility, scalability, availability, reliability and maintainability will be greatly improved. According to the discussion in the previous chapter, the complexity of collaboration can be decreased by specifying individual’s behaviour only. Considering this, we can finally create our pattern as shown below:

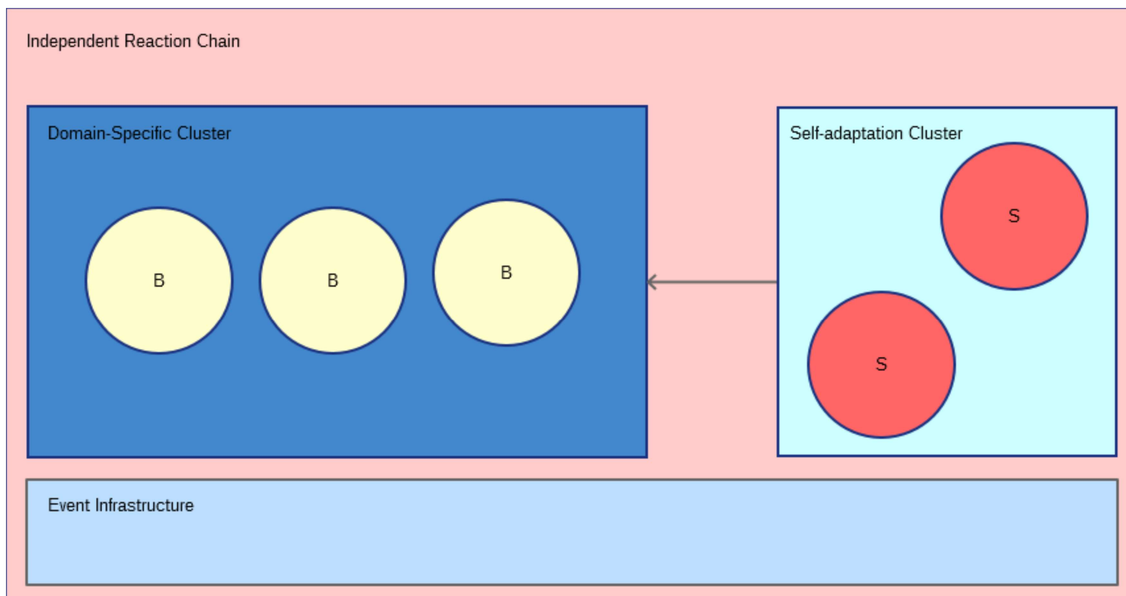


Figure 10: The Independent Raction Chain

In this pattern, each domain-specific module, being denoted by a "B" cycle, and each self-adaptation module, being denoted by a "S" cycle, are both designed as actors. Each actor in this pattern can asynchronously process events by using the Proactor pattern. Each actor will be structured with the Microkernal pattern, and therefore a complex task can be split into multiple parallel sub-tasks in a certain role, and a sub-task can be replicated within an actor. The responsibility among these actors are coordinated by an Event Channel that has been improved with synchronous mechanism. As a consequence, this pattern can provide flexibility, scalability, availability, reliability and changeability. However, due to the adoption of the Proactor pattern, the maintainability can be decreased. Further, since the synchronous collaborations are coordinated by a single architectural element, it may cause performance problems.

6.8 Summary

This chapter introduces a method for software engineers about how to get a novel architectural style that can be used to create the architecture for a domain-specific self-adaptive application. This architectural pattern is created by applying different existing architectural patterns step by step. A self-adaptive application created with this pattern is expected to have flexibility, scalability, availability, reliability and changeability. However, it is not able to achieve all desired properties by using one architectural pattern [49], and therefore architects have to achieve a good trade-off depends on the priorities of their requirements.

7 Case study – an intelligent multiple strategies trading system

Following the development of information techniques and financial trading markets, software techniques have been widely accepted in trading area. Not only researchers, but also organisations and personal investors are interested in the methods for developing a workable trading application. However, due to the complexity of finance, it is not easier for software engineers to understand all of it, e.g. stochastic process, pricing model, volatility, macro and micro fundamental analysis techniques. On the other hand, the required knowledge to build a software application with required qualities is also beyond the capability of trading masters.

Keeping consistency between trading logic and the application which is built on the top of it is very important. There is a painful lesson: Everbright Securities, a well-known financial organisation in China, used to have a high-performance trading system constructed for trading treasure futures had worked without problems for nearly 1 year, including several months test, and had made a lot of profit by executing a high-frequency trading strategy automatically. But, this company was warned by an exchange that the trading system mentioned above had made a lot of unusual transactions and thereby violated one of the exchange's regulations. After a complicated self-inspection, this company finally reported to the exchange with an explanation of the reason: The core thought of this trading system is to make profit by doing arbitrage trading in the form of buying nearby-month contracts and in contrast selling far-month contracts. To complete this kind of trading, this system always provides the best offers of the market by calculation and withdraws the unfilled order after a given period, e.g. 1 millisecond, automatically. Normally, these offers can always lead to deals. However, the market could be abnormal. Due to the decreasing liquidity of the market at that time, this trading system had to withdraw all offers because there was no matched counterparty. Therefore, this trading system had had withdrew 2,000 orders in 2 seconds, and finally caused this accident. Frankly, it is not easy to foresee this issue, since it requires the designer of this system has a through understanding of both trading and software, and even the regulations of finance industry.

The above example, from a different perspective, shows why the gap between domain knowledge and software engineering is critical to the success of a domain software application. In this chapter, we are going to demonstrate how our method works as a whole. This demonstration comes from a part of a concrete project in the financial area. Simply, this project is about building an application for a futures company so that they can have their strategies executed on exchanges and make profit automatically.

Before the begin, we would like to introduce some basic terms of financial trading first so that we can use these terms for convenient. Please note, these financial concepts can be understood from different views, and for convenient, we find the following definitions from an online financial knowledge centre [50].

- financial instrument. A financial instrument is a trade-able asset. Trade-able assets can be security, cash, contractual right, and ownership evidence;

- exchange. A marketplace in which people trade financial instruments;
- futures. A futures is a standardised contract obligating traders to buy or sell an asset, e.g. physical commodity or other financial instruments, in the future;
- long (position). The buying of a financial instrument and be expecting the value of this financial instrument will rise;
- short (position). The opposite of a long position;
- open interest. the total number of open futures contracts existing in a given day.

7.1 Domain requirements

A trading strategy normally includes a set of predefined trading rules. A trading rule specifies when to start or exit trading, when to long or short position an asset, how to manage capital, and etc.. In practice, strategy engineers need to submit their trading decisions to risk managers first. Risk managers are responsible of evaluating the trading decisions on the basis of the current open interest of the target assets. If the result of the evaluation shows that the current risk level is relatively safe, than risk managers will pass the trading decisions to traders. Otherwise, risk managers will notify strategy engineers the the received decisions have been refused. Traders are responsible of generating orders according to the received trading decisions, and submitting these orders to a certain counter. Only counters are authorised to trade with exchanges directly. When receives a group of orders from a certain trader, a counter will pass these orders to the related exchange. If these orders are valid, the exchange will try to match two unfilled orders and send the result back to the counter. Otherwise, the invalid orders will be returned back to the counter directly. The counter will send all responses from exchanges to traders. Strategy engineers may need to update their strategies according to the trading responses sent back from traders.

Specifically, two or more strategy engineers may collaborate together. In this case, the final trading decisions are made on the basis of all trading decisions made by each trading strategy. That is, the collaborating strategy engineers may decide to long position a futures only if all these strategy engineers independently decide to long position the same futures. Additionally, risk managers need to be able to handle any potential uncertain behaviour of a trading strategy. For example, unexpected failure of a trading strategy, unsuccessful orders, etc..

As a consequence, the aim of this project is to design and implement a software platform to automate the execution of trading strategies, risk managing, orders transferring, communicating among strategy engineers, traders, risk managers, counters and exchanges.

7.1.1 stakeholders

The stakeholders of this project are:

- strategy engineers. Strategy engineers concern the issues of executing his or her strategies, e.g. performance, availability, reliability, flexibility, scalability;

- risk managers. Risk managers concern whether a trading strategy being run in this system is available and reliable;
- traders. Traders concern whether this system can automatically perform their tasks and report the results to them;
- the company. For the company who purchases this system, it concerns the cost of developing and managing this system and how long it will take for developing this system;
- counter. A counter is normally maintained by an organisation authorised by the regulators. A counter concerns whether this system conforms to its open interfaces;
- exchanges. Exchanges concern whether this system conforms to their trading interfaces;
- regulator. Regulators concern whether this system behaves within the context permitted by law and regulates;
- developers. Developers concern whether need new knowledge to develop this system and how easy it will be to develop and maintain it.

7.1.2 Functional requirements

The required functional requirements of this system are:

- executing trading strategies. The capability of loading a trading strategy into the system at runtime, preparing the required resources, including capital, and executing this strategy;
- sending the buying or selling instructions with exchanges through the services provided by the exchanges;
- monitoring the executing trading strategies. The capability of monitoring the running trading strategies and recording the behaviour of each active strategies, e.g. profit and loss;
- managing the executing trading strategies. The capability of stopping a trading strategy at runtime;
- managing users. Different strategy analysers can log into or exit the system; and the system can manage the profile of each analyser;
- generating reports at demand.

7.1.3 Non-functional requirements

The required non-functional requirements of this system are:

- Availability. The whole system and each trading strategy running in this system should have a higher availability, otherwise, it can cause great loss;
- Reliability. The whole system and each trading strategy running in this system should also have a higher reliability;
- Performance. Performance is important for a trading strategy, since a trading opportunity is very fleeting and one step behind would cause loss;
- Concurrency. Different trading strategies need to run in parallel. Further, risk managing should also run in parallel with these trading strategies;

- Flexibility. The system should be flexible enough so that a trading strategy can be running or stopping at wish and multiple trading strategy can collaborate with each other;
- Scalability. A trading strategy can be added or removed from this system freely;
- Changeability. It would be easy to change the structure of this system for satisfying the evolving trading and risk managing requirements;
- Maintainability. It would be easy for developers to understand the structure of this system.

7.2 Domain model

Following our method, to delivery the required software application, we have to model the trading processes required to be implemented first. That is, we need to describe the interaction processes among domain roles with the role-behaviour-based modelling language introduced above. As discussed above, another main concern at this step is to check the completeness of the domain model. That is, software engineers need to have a clear understanding what events need to be solved and how a domain role will behave in certain environment, e.g. only perform actions when a given set of event happens.

The first role we are talking about here is strategy engineer. According to the requirements documented above, a strategy engineer designs strategies according to current market environments, making trading decisions according to one or more trading strategies by the observation on information of interest, communicating with risk managers to send trading decisions and receive responses, and communicating with traders to receive responses from exchanges so that this engineer can update these trading strategies. Further, multiple strategy engineers can collaborate with each other to make trading decisions.

According to above understanding of this role, we can therefore specify the events concerned by strategy engineers:

```
//a strategy engineers works once he or she receives working orders
external event STRATEGY_ORDERED
// a strategy engineer will update his or her strategy once market environment has been updated
external event MARKET_ENVIRONMENT_UPDATED
// a strategy engineer is responsible for sending trading decisions (strategies) to risk managers
// for evaluating risks
event TRADING_STRATEGY_PLANNED
// a strategy engineer takes proper actions once he or she receives any response from traders
event TRADER_FEEDBACK_UPDATED
// a strategy engineers may request collaboration from other strategy engineers
// whiling making trading decisions
event TRADING_COLLABORATION_REQUESTED
// a strategy engineer needs to revise his or her trading strategies
// once the risk manager reports any founded risk
event RISK_CLAIMED
event TRADING_STOPPED
```

There are 3 external events defined above. When we consider whether an event is external or not, depends on the source of this event. From the domain perspective, the events STRATEGY_ORDERED, MARKET_ENVIRONMENT_UPDATED are outside of the system: the event STRATEGY_ORDERED is caused by user (manager) requests, and the event MARKET_ENVIRONMENT_UPDATED happens on trading markets, and both users and markets are not the concerned domain roles of this trading system. We can hence specify the role strategy engineer as below:

```
role StrategyEngineer
{
        on STRATEGY_ORDERED
        {
                create_trading_strategy:
                                TRADING_STRATEGY_PLANNED
                                | TRADING_COLLABORATION_REQUESTED
        }
        on TRADING_COLLABORATION_REQUESTED
        {
                do_collaboration_work: TRADING_STRATEGY_PLANNED
        }
        on MARKET_ENVIRONMENT_UPDATED
        {
                update_trading_strategy: TRADING_STRATEGY_PLANNED
        }
        on TRADER_FEEDBACK_UPDATED
        {
                update_trading_strategy:
                                TRADING_STRATEGY_PLANNED
                                |TRADING_STOPPED
        }
        on RISK_CLAIMED
        {
                update_trading_strategy: TRADING_STRATEGY_PLANNED
        }
}
```

With the above statement block, we specify the role strategy engineer. This role will only behave when a given set of events happens. At this point, we have to clarify that it is domain experts' responsibility to double check the completeness of this model since software engineers may have no idea about the exact behaviour of any domain role. For example, in some situations, trading experts may need a strategy engineer to report if the event TRADING_STRATEGY_PLANNED does not happen as scheduled, and in this context, the domain expert who is responsible of modelling this role should have a complex event defined, which happens following a carefully defined pattern.

Further, from the above statements, we can see that the collaboration behaviour is defined in a relatively easier way. Of cause we can decompose the event TRADING_COLLABORATION_REQUESTED into more detailed events, e.g. a request to the signal generated by another trading strategy on a specific financial instrument, or a request to the calculation of a complex stochastic model. We can also detail the collaboration domain mechanism among strategy engineers. But this kind of detail can be implemented in various ways in

practice and does not provides us any more help on understanding the essential behaviour of the role strategy engineer, therefore we choose to ignore the detail and focus on the higher level logic. This is the power of events. That is, with the concept of simple and complex events, we can concentrate on our work at a desired level without trying our best to think of all combinations of details.

Following the above example, we can then define the second role: trader. According to the requirements, the role trader cares about the events related to do trading with counters with valid trading strategy. In detail, this role is responsible for generating orders according to approved trading strategies that the role trader receives from the role risk manager, and transferring the generated orders to the role counter. Further, the role trader needs to concern the trading exception if there is no any feedback from the role counter. The model of the role traders is:

```
event COUNTER_FEEDBACK_UPDATED
event TRADING_EXCEPTION_RAISED
event COUNTER_EXCEPTION:
        (not COUNTER_FEEDBACK_UPDATED) after TRADING_ORDER_PLANNED
event VALID_STRATEGY_GENERATED:
        STRATEGY_APPROVED after TRADING_STRATEGY_PLANNED

role Trader
{
        on VALID_STRATEGY_GENERATED{
                generate_orders_and_submit_to_exchanges: TRADING_ORDER_PLANNED
        }

        on COUNTER_FEEDBACK_UPDATED{
                evaluate_exchange_feedback:
                        TRADER_FEEDBACK_UPDATED | TRADING_EXCEPTION_RAISED
        }
}
```

From the above, we can see that the role trader behaves when two events happen. Further, the above code block shows how we use complex events to decrease the model size and simplify the modelling process. By having complex event defined, domain experts can compact complicated patterns by one event definition. Additionally, please notice that, modelling uncertainty with the concept events does not mean an event is uncertain. An event can be understood as a recognised change on a domain entity, which is meaningful to the domain experts. With this understanding and assumption, the output of a role should not be a full list of all possible consequences after performing an action, but the concerned and observable events.

Next role being modelled is risk manager. From the trading perspective, this role receives trading decisions from strategy engineers, evaluates these decisions and decides whether need to refuse these decisions or approve these decision by claiming any risk. Further, risk managers should be able to handle any unexpected situation according to a set of predefined risk managing rules. The following statements specify the role risk manager.

```
event STRATEGY_APPROVED
event TRADING_EXCEPTION_RAISED

role RiskManager
{
	on TRADING_STRATEGY_PLANNED{
		evaluated_strategy: RISK_CLAIMED | STRATEGY_APPROVED
	}
	on TRADING_EXCEPTION_RAISED{
		report_risk_and_try_to_recover: RISK_PROCESSED
	}
}
```

The role counter needs to authenticate the received orders and transfer the authorised orders to the role exchange.

```
event AUTH_ORDER_PLANNED
event EXCHANGE_FEEDBACK_UPDATED

role Counter
{
	on TRADING_ORDER_PLANNED{
		authenticate_and_forward_orders: AUTH_ORDER_PLANNED
	}

	on EXCHANGE_FEEDBACK_UPDATED{
		forward_exchange_feedback: COUNTER_FEEDBACK_UPDATED
	}
}
```

The role exchange checks whether an order received from the counter is valid, and tries to match two valid orders, and notifies the role counter about the result.

```
role Exchange
{
	on AUTH_ORDER_PLANNED{
		execute_order: EXCHANGE_FEEDBACK_UPDATED
	}
}
```

After specifying these roles, we need to elaborate business process and clarify the complementary events. A process can be treated as a clue about what are expected during a workflow in which the defined roles interact with each other. A process also specifies an expected temporal order among roles. By describing a process, domain experts can have a chance to double check whether there is any missing information which will lead to an unexpected stop. In the trading domain, there are 3 main processes: trading process, risk managing process and collaboratively analysing process. The following specification defines the trading process:

```
process TradingProcess
{
	role StrategyEngineer
	role Trader
	role RiskManager
	role Counter
```

```
        role Exchange

        StrategyEngineer<STRATEGY_ORDERED, TRADING_STRATEGY_PLANNED>
                -> RiskManager<TRADING_STRATEGY_PLANNED, STRATEGY_APPROVED>
                -> Trader<VALIED_STRATEGY_GENERATED, TRADING_ORDER_PLANNED>
                -> Counter<TRADING_ORDER_PLANNED, AUTH_ORDER_PLANNED>
                -> Exchange<AUTH_ORDER_PLANNED, EXCHANGE_FEEDBACK_UPDATED>
                -> Counter<EXCHANGE_FEEDBACK_UPDATED,
                                COUNTER_FEEDBACK_UPDATED>
                -> Trader<COUNTER_FEEDBACK_UPDATED, TRADER_FEEDBACK_UPDATED>
                -> StrategyEngineer<TRADER_FEEDBACK_UPDATED,
                                        TRADING_STRATEGY_PLANNED|TRADING_STOPPED>
}
```

There are two things should be noticed: 1) operator '->' indicates the consequence of a behaviour, rather than a temporal order between two behaviours; 2) A process actually describes a causality chain of the behaviours of the participants in this process, and any empty event means the end of a brand of this causality chain. It seems that there is a resemblance between the concept process and the sequential diagram of UML, but the underlying meaning is totally different. Since a sequential diagram indicates the action sequence among several classes, a process indicates a temporal order by specifying the transition between events without considering how these events are triggered. In short, the sequential diagram concerns detailed design, but the concept process concerns logic at a higher level. As a consequence, with the concept process, software engineers can design their systems in various way as long as their design ensure their system can output the expected events when there is a given set of events.

The second process the risk managing workflow:

```
process RiskManagingProcess
{
        role Trader
        role RiskManager
        role StrategyManager

        Trader<, TRADING_EXCEPTION_RAISED>
                -> RiskManager<TRADING_EXCEPTION_RAISED, RISK_PROCESSED>
        StrategyEngineer<, TRADING_STRATEGY_PLANNED>
                -> RiskManager<TRADING_STRATEGY_PLANNED, RISK_CLAIMED>
                -> StrategyEngineer<RISK_CLAIMED, TRADING_STRATEGY_PLANNED>
}
```

The above code block specifies two temporal orders in the process RiskManagingProcess. Please note, the blank triggers in the transitions denotes any trigger events, since only result events are cared about. That is, without clearly specifying the trigger events, we still can recognise such an event chain as particular domain process. The following statements show the collaborative analysing process:

```
process CollaborativeAnalysingProcess
{
        role StrategyEngineer
```

```
        StrategyEngineer<STRATEGY_ORDERED,
                            TRADING_COLLABORATION_REQUESTED>
                -> StrategyEngineer<TRADING_COLLABORATION_REQUESTED,
                                    TRADING_STRATEGY_PLANNED>
}
```

Now the domain modelling work is completed and we can start to design our application. Being precise, at next step, we need to design the application architecture by mapping domain elements into architectural elements. The following section demonstrate how we achieve this.

7.3 Architecture description

By having the domain model clearly defined, we can try to find a way for generating the desired architecture description on the top of this domain model. users uses the concept rationale to constrain the construction of an architecture. As discussed before, with Grasp+, users can constrain an architectural element's behaviour by specifying its role according to proper rationales, and we can believe two systems behave same if these two systems can always produce same output with a given input in a certain context. Thus, to build a proper application architecture with a given domain model in Grasp+, users first need to specify what architectural elements exist and how these elements interact with each other under the constraints imposed by a given set of architectural roles and rationales, which are transformed from the domain model.

A suggested set of automatic mapping rules is presented as below:

1. Any domain role or process is described as a rationale in Grasp+;
2. For each rationale created at step 1, define an architectural role on top of it;
3. For each architectural role, define a template that acts as this role;
4. In a system instance:
    1. initialise a template as component if this template is from a domain role;
    2. initialise a template as connector if this template is from a domain process;
    3. define a default scenario for modelling the normal architectural context;
    4. if any need, define other scenarios for modelling abnormal architectural contexts which are explicitly described, and make sure there is no overlap between any two scenarios;
    5. deploy architectural elements, and make sure that any two components should be linked (with a connector if applicable)

After applying these rules, the candidate architecture model is:

```
architecture TradingPlatform{
        rationale DR1(){
```

```
                reason #'domain-specific role StrategyEngineer';
                @trigger(STRATEGY_ORDERED)
                @result(TRADING_STRATEGY_PLANNED,
TRADING_COLLABORATION_REQUESTED)
                reason #'create_trading_strategy';
                @trigger(TRADING_COLLABORATION_REQUESTED)
                @result(TRADING_STRATEGY_PLANNED)
                reason #'do_collaboration_work';
                @trigger(MARKET_ENVIRONMENT_UPDATED)
                @result(TRADING_STRATEGY_PLANNED)
                reason #'update_trading_strategy'
                @trigger(TRADER_FEEDBACK_UPDATED)
                @result(TRADING_STRATEGY_PLANNED, TRADING_STOPPED)
                reason #'update_trading_strategy'
                @trigger(RISK_CLAIMED)
                @result(TRADING_STRATEGY_PLANNED)
                reason #'update_trading_strategy'
        }

        rationale DR2(){
                reason #'domain-specific role Trader';
                @trigger(VALIED_STRATEGY_GENERATED)
                @result(TRADING_ORDER_PLANNED)
                reason #'generate_orders_and_submit_to_exchanges';
                @trigger(COUNTER_FEEDBACK_UPDATED)
                @result(TRADER_FEEDBACK_UPDATED, TRADING_EXCEPTION_RAISED)
                reason #'evaluate_exchange_feedback';
        }

        rationale DR3(){
                reason #'domain-specific role RiskManager';
                @trigger(TRADING_STRATEGY_PLANNED)
                @result(RISK_CLAIMED, STRATEGY_APPROVED)
                reason #'evaluated_strategy';
                @trigger(TRADING_EXCEPTION_RAISED)
                @result(RISK_PROCESSED)
                reason #'report_risk_and_try_to_recover';
        }

        rationale DR4(){
                reason #'domain-specific role Counter';
                @trigger(TRADING_ORDER_PLANNED)
                @result(AUTH_ORDER_PLANNED)
                reason #'authenticate_and_forward_orders';
                @trigger(EXCHANGE_FEEDBACK_UPDATED)
                @result(COUNTER_FEEDBACK_UPDATED)
                reason #'forward_exchange_feedback';
        }

        rationale DR5(){
                reason #'domain-specific role Exchange';
                @trigger(AUTH_ORDER_PLANNED)
                @result(EXCHANGE_FEEDBACK_UPDATED)
                reason #'execute_order';
        }
```

```
rationale DP1(){
        reason # 'domain-specific process TradingProcess';
}

rationale DP2(){
        reason # 'domain-specific process RiskManagingProcess';
}

rationale DP3(){
        reason # 'domain-specific process CollaborativeAnalysingProcess';
}

role DomStrategyEngineer because DR1;
role DomTrader because DR2;
role DomRiskManager because DR3;
role DomCounter because DR4;
role DomExchange because DR5;
role DomTradingProcess because DP1;
role DomRiskManagingProcess because DP2;
role DomCollaborativeAnalysingProcess because DP3;

template StrategyEngineer() act_as DomStrategyEngineer{}
template Trader() act_as DomTrader{}
template RiskManager() act_as DomRiskManager{}
template Counter() act_as DomCounter{}
template Exchange() act_as DomExchange{}
template TradingProcess() act_as DomTradingProcess{}
template RiskManagingProcess() act_as DomRiskManagingProcess{}
template CollaborativeAnalysingProcess() act_as DomCollaborativeAnalysingProcess{}

system DomainFunctionalModel{
        // normal scenario:
        //      only expected events occur
        scenario s1;

        // architectural elements for domain functions
        component engineer = StrategyEngineer();
        component trader = Trader();
        component risk_manager = RiskManager();
        component counter = Counter();
        component exchange = Exchange();
        connector trading = TradingProcess();
        connector rmp =RiskManagingProcess();
        connector cap = CollaborativeAnalysingProcess();

        create engineer in s1;
        create trader in s1;
        create risk_manager in s1;
        create counter in s1;
        create exchange in s1;

        create link engineer to risk_manager via trading in s1;
        create link rmp to trader via trading in s1;
        create link trader to counter via trading in s1;
        create link trader to engineer via trading in s1;
        create link counter to exchange via trading in s1;
```

```
                create link exchange to counter via trading in s1;
                create link risk_manager to trader via rmp in s1;
                create link trader to risk_manager via rmp in s1;
                create link engineer to engineer via cap in s1;
        }
}
```

Now a functional outline of the architecture is presented as above. It provides software developers a clear guide about how to structure the system and what functions need to be implemented. All of the architectural operations should be established at scenario s1, which indicates the normal context (the situation in which only expected events occur) of this application. But this architecture requires to be refined of cause. One reason is that it is not possible to decide all architectural elements from a domain-specific model only. Further, domain-specific structures need to be refined via appropriate patterns so that ensure the required qualities [28]. For example, the Microkernel pattern introduced before can be applied here for coordinating domain-specific roles. According to this pattern, some new architectural roles need to be created: internal server, external server, adapter, client, and microkernel. Further, to support the domain-specific roles, it would be better to put the architectural role microkernel into a separated layer. In practice, the architectural elements created for implementing domain-specific roles can perform as internal server. Additionally, considering that the microkernel role is responsible for communicating, all existing mutual links can be redesigned as well.

Further, according to requirements, we may want to introduce a self-adaptation process for maintaining the reliability of trader component by re-activating (creating/removing) trader component at run-time. Therefore, we need to introduce more architectural roles for refactoring the existing architecture. As a consequence, the updated architecture is:

```
architecture Trading_Platform{
        ...
        rationale AR1(){
                reason #'pattern: microkernel';
        }
        role ArchExternalServer because AR1{
                action request_service(ArchMicroKernel){
                        create link ArchExternalServer to ArchMicroKernel;
                }
        }
        role ArchInternalServer because AR1;
        role ArchMicroKernel because AR1{
                action call(ArchInternalServer){
                        create link ArchMicroKernel to ArchInternalServer;
                }
        }

        rationale AR2(){
                reason #'self-adaptation: if trader dies, remove it and then create a new one';
                reason #'self-adaptation: report dies of trader';
        }
```

```
role ArchSubject because AR2;
role ArchMonitor because AR2{
        action monitor(ArchSubject){
                create link ArchMonitor to Subject;
        }
        action notify(ArchAdapter){
                create link ArchMonitor to ArchAdapter;
        }
}

role ArchAdapter because AR2{
        action adapt(ArchSubject){
                remove Subject;
                create Subject;
                create link ArchAdapter to Subject;
        }
}
...

// architectural elements for maintaining qualities
template Kernel() act_as ArchMicroKernel {}
template Coordinator() act_as ArchExternalServer{}
template Monitor() act_as ArchMonitor{}
template Adapter() act_as ArchAdapter{}

//domain related architectural elements
template StrategyEngineer() act_as DomStrategyEngineer, ArchInternalServer{}
template Trader() act_as DomTrader, ArchInternalServer{}
template RiskManager() act_as DomRiskManager, ArchInternalServer, ArchSubject{}
template Counter() act_as DomCounter, ArchInternalServer{}
template Exchange() act_as DomExchange, ArchInternalServer{}
template TradingProcess() act_as DomTradingProcess{}
template RiskManagingProcess() act_as DomRiskManagingProcess{}
template CollaborativeAnalysingProcess() act_as DomCollaborativeAnalysingProcess{}

system DomainFunctionalModel{
        scenario s1;
        // exceptional context
        //      a trader component dies
        scenario s2;

        layer BusinessLayer in s1{
                component engineer = StrategyEngineer();
                component trader = Trader();
                component risk_manager = RiskManager();
                component counter = Counter();
                component exchange = Exchange();
                connector trading = TradingProcess();
                connector rmp =RiskManagingProcess();
                connector cap = CollaborativeAnalysingProcess();

                component monitor = Monitor();
                component adapter = Adapter();

                create engineer in s1;
                create trader in s1;
```

```
                    create risk_manager in s1;
                    create counter in s1;
                    create exchange in s1;

                    create link engineer to risk_manager via trading in s1;
                    create link rmp to trader via trading in s1;
                    create link trader to counter via trading in s1;
                    create link trader to engineer via trading in s1;
                    create link counter to exchange via trading in s1;
                    create link exchange to counter via trading in s1;
                    create link risk_manager to trader via rmp in s1;
                    create link trader to risk_manager via rmp in s1;
                    create link engineer to engineer via cap in s1;

                    create monitor in s1;
                    create adapter in s1;

                    create link adapter to monitor in s1;

                    monitor.monitor(trader) in s1;
                    monitor.notify(adapter) in s1;
                    adapter.adapt(trader) in s2;
          }
          layer CoreLayer in s1{
                    component kernel = Kernel();
                    component coordinator = Coordinator();

                    create kernel in s1;
                    create coordinator in s1;

                    coordinator.request_service(kernel) in s1;
          }
          kernel.call(strategy_engineer) in s1;
          kernel.call(risk_managing) in s1;
          kernel.call(trader) in s1;
          kernel.call(counter) in s1;
          kernel.call(exchange) in s1;
     }
}
```

In the above code block, we have defined new architectural roles with actions. An action can be understood as an architectural function that defines a set of architectural operations which is required to be executed as a whole. Additionally, besides scenario s1, we have also defined scenario s2, which indicates a special situation: when a component trader dies. That is, any architectural operation defined in s2 should be dynamically established in this situation. Regarding the above architectural description, there are several points should be emphasised:

- An architectural description provides a high-level design on the configuration or structure of an application, it is not a detailed design which describes the behaviour of the components and connectors of an application. For example, the above architectural description specifies that the component monitor should establish a link between itself and the component trader. But it does not provide any technical detail about what the link is, how and when the link will be established, e.g. call a function when there is an incoming signal. Regarding self-adaptation processes, an

architectural description specifies a configuration or structure to support one or more particular self-adaptation processes with the desired qualities, but not descirbes an algorithm specifying how a self-adaptation loop happens logically;

- The existence of a context is determined by events. The technical detail of the context s1 and s2 can be decided at a lower level. For example, software engineers can describe the context s2 as a context in which a component trader dies, and more detail, a component trader can be labelled as died if it does not respond to any given input within 10 seconds;
- It is software engineers' responsibility to ensure the implementation of a given software architecture following the given architectural constraints. For example, in the above architectural description, the statement "adapter.adapt(trader) in s2" requires the application implementation can establish a link between the component adapter and the component trader dynamically only if it works in the context s2;

In short, an architecture defines a configuration or a structure of an application, and a set of configurations or structures in different environments for a dynamic application. A too detailed design leads to less flexibility and reuse-ability. Therefore, software engineers need to achieve their own trade-off.

There is an ongoing project on the compiler, which is built on Antlr [51] and shown as following figure, of the business process modelling language. As a notice, the content shown in the figure is based on an earlier version of the modelling language.

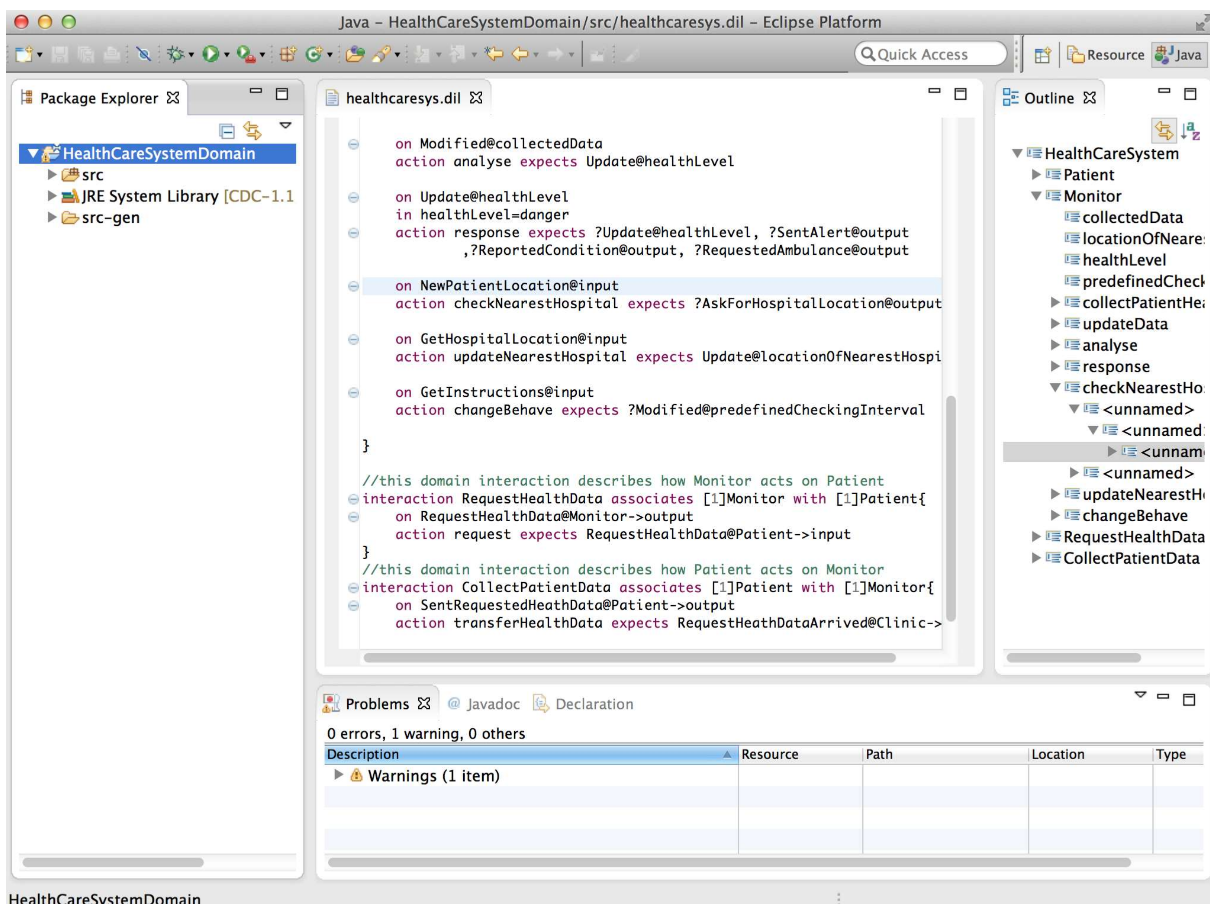


Figure 11: An earlier implementation of the editor for domain modelling

With a set of automatic mapping rules, we can also build a mechanism for generating an outline for the target architecture automatically. This aim can be achieved by using a set of string templates and replacing key

variables. The following codes show an earlier version of the string template which are used to produce architecture description from domain models:

```
template Template_<EntityName>() because domain.<EntityName>{
        [property <PropertyName> satisfies <ConstraintName>;]*


        provides Interface_<EntityName>{
                [action <ActionName> dependency <CollaboratorName>
because domain.<ActionName>;]*
        }
[requires  interface_<CollaboratorName> ]*
}


[link Template_<EntityName1>.Interface_CollaboratorName> to <Multiplicity>
Template_<EntityName2>.Interface_<EntityName2>]*


external? event <EventName>{
        [ {if external} Template_<EntityName>-><PropertyName> , => <new_value>
| {else}Template_<EntityName>-><ActionName>, => DONE]*
}


[{CollaboratorName == EntityName2} link <EntityName1>.Interface_<CollaboratorName> to
<Multiplicity><EntityName2>.Interface_<EntityName2>]*

[link Template_<EntityName1>.Interface_<CollaboratorName> to <Multiplicity> Template_<
CollaboratorName >.Interface_< CollaboratorName >{
        [on <EventName> <EntityName| CollaboratorName >.ActionName]*

}]*
```

7.4 Implementation

With the generated architecture, we can now try to implement the target application. More precisely, we need to structure this target application by generating a code skeleton. At this point, developers need to consider how to implement the structure defined by a given architecture by using proper coding method. A set of rules is required to help build implementation automatically. However, currently, there are many programming paradigms, e.g. procedure, object-oriented, aspect-oriented, functional programming, and each of them has its own advantages. As a consequence, this section does not provide a discussion about how to implement a given architecture in a better way, but a demonstration on generating a set of runnable code skeletons. To achieve this, from our own perspective, the transforming rules used in this section are shown below:

1. There is a common interface specifying the common behaviour of an architectural role if applicable;
2. A specific architectural role is implemented as a specific interface;
3. A template is implemented as a class implementing its architectural roles;
4. A component instance of a given template restricts the architectural behaviour of the class describing this template, including the references to other classes on templates;

5. A connector instance of a given template highlights the connection mechanism should be explicitly specified at architectural level;

6. Rationales are comments as reference to the implementation;

With the above rules, for example, the template StrategyEngineer can be implemented as below:

```
class ArchitecturalRole {
public:
	virtual void on(std::string& e) = 0;
};

class ArchInernalServer;

// domain-specific role StrategyEngineer
class DomStrategyEngineer
{
public:
	virtual void on_strategy_ordered() = 0;
	virtual void on_trading_collaboration_requested() = 0;
	virtual void on_market_environment_updated() = 0;
	virtual void on_trader_feedback_updated() = 0;
	virtual void on_risk_claimed() = 0;
};

class Trader;
class RiskManager;
class ConnectionManager;


class StrategyEngineer : public DomStrategyEngineer, public ArchInernalServer, public
ArchitecturalRole{
	ConnectionManager* _cm;
	RiskManager* _rm;
	std::vector<StrategyEngineer*> _collaborators;
	std::vector<Trader*> _traders;
public:
	inline StrategyEngineer() { init(); };
	~StrategyEngineer() {}
	virtual void init() { _setup_conn(); }
	virtual void set_risk_manager(RiskManager* rm) { }
	virtual void add_collaborator(StrategyEngineer* se) {}
	virtual void add_trader(Trader* t) {  }

	// Inherited via DomStrategyEngineer
	//@trigger(STRATEGY_ORDERED)
	virtual void on_strategy_ordered() { _create_trading_strategy(); };
	//@trigger(TRADING_COLLABORATION_REQUESTED)
	virtual void on_trading_collaboration_requested() override { _do_collaboration_work(); };
	//@trigger(MARKET_ENVIRONMENT_UPDATED)
	virtual void on_market_environment_updated() override { _update_trading_strategy };
	//@trigger(TRADER_FEEDBACK_UPDATED)
	virtual void on_trader_feedback_updated() override { _update_trading_strategy };
	//@trigger(RISK_CLAIMED)
	virtual void on_risk_claimed() override { _update_trading_strategy(); };
```

```
        // Inherited via ArchInernalServer
        virtual void serve() override{/*to-do*/}
        // Inherited via ArchitecturalRole
        virtual void on(std::string & e) override {/*to-do*/}
private:
        void _setup_conn() {}
        //@result(TRADING_STRATEGY_PLANNED,
        //TRADING_COLLABORATION_REQUESTED)
        void _create_trading_strategy() { }
        //@result(TRADING_STRATEGY_PLANNED)
        void _do_collaboration_work() {          }
        //@result(TRADING_STRATEGY_PLANNED)
        //@result(TRADING_STRATEGY_PLANNED, TRADING_STOPPED)
        //@result(TRADING_STRATEGY_PLANNED)
        void _update_trading_strategy() { }
};
```

The above codes may not be perfect or elegant. However, the core concern on the code generation is to make a structural reference that can be customised to a good product. Therefore, not the coding style, but the structures shown in these codes will be evaluated. Additionally, the existing compiler and code generator of Grasp+ are built on Antlr [51]and Xtend [52]. Due to the time limit, these existing tools have not been updated to the latest version .

Further, the following code block shows how to implement the adaptation mechanism described in the given architecture. From software engineering perspective, the following statements actually initialises the observation pattern [53]. With a careful design, the Monitor can capture changes in an application's context and the Adapter can behave accordingly. For example, a loop can be placed in the monitor member function, and try to access the ArchSubject every 1 or 2 seconds, and notifies the Adapter by calling proper member function of the Adapter (or a callback function if applicable) if the Monitor cannot receive any response from the ArchSubject. The following code follows the given architectural constraints: the Adapter will try to adapt its subject only if the Monitor has detected the change of the system context from s1 to s2.

```
class ArchSubject {};

class ArchMonitor {
public:
        virtual void monitor() = 0;
        virtual void notify() = 0;
};
class ArchAdapter {
public:
        virtual void adapt() = 0;
};

class Monitor: public ArchMonitor, public ArchitecturalRole {
        std::vector<ArchSubject*> _subjects;
        ArchAdapter* _adapter;
public:
        Monitor() = default;
        ~Monitor() = default;
```

```
        void add_subject(ArchSubject* s) {}
        virtual void monitor() {}
        //normal context to the context in which the trader dies
        virtual void notify() {for(int i=0;i<_subjects.size();i++){_adapter->adapt(_subjects.at(i));}}
};
class Adapter: public ArchAdapter, public ArchitecturalRole {
        std::vector<ArchMonitor*> _monitors;
public:
        void set_monitor(ArchMonitor* m){}
        virtual void adapt(ArchSubject* s) { this->_do_adaptation(); }
private:
        void _do_adaptation() {}
};
```

7.5 Evaluation

As saying above, an architecture can be implemented in various ways, and it is not possible to tell which type of the implementation is better. Since only code skeleton can be generated from the architecture description, therefore, to evaluate the functional properties of an implementation, we have to customise the existing code skeleton and fill some concrete logic, such as print statements, so that we can check whether the interactions among each object of the defined classes can work fine. Regarding the required non-functional properties, they can be ensured by the principled application of the existing architectural patterns as long as these patterns behave as their design [8]. To demonstrate the runtime behaviour of the generated code skeleton, we have built the required connection mechanism with the following class:

```
struct DomTradingProcess {/*to-do: domain trading process info*/ };
struct DomRiskManagingProcess {/*to-do: domain risk managing process info*/ };
struct DomCollaborativeAnalysingProcess {/*to-do: domain collaboration analysing process info*/ };
struct TradingProcess :public DomTradingProcess {/*to-do: arch info*/ };
struct RiskManagingProcess: public DomRiskManagingProcess {/*to-do: arch info*/ };
struct CollaborativeAnalysingProcess:public DomCollaborativeAnalysingProcess {/*to-do: arch info*/ };

class ConnectionManager {
public:
        template<typename T>
        void conn(std::string& e, T* dest, TradingProcess& tp) {dest->on(e);}
        template<typename T>
        void conn(std::string& e, T* dest, RiskManagingProcess& tp) { dest->on(e); }
        template<typename T>
        void conn(std::string& e, T* dest, CollaborativeAnalysingProcess& cap) { dest->on(e); }
};
```

By filling with proper and simple algorithm, we can make the bundle of code skeleton runnable, and the following figure shows the output.

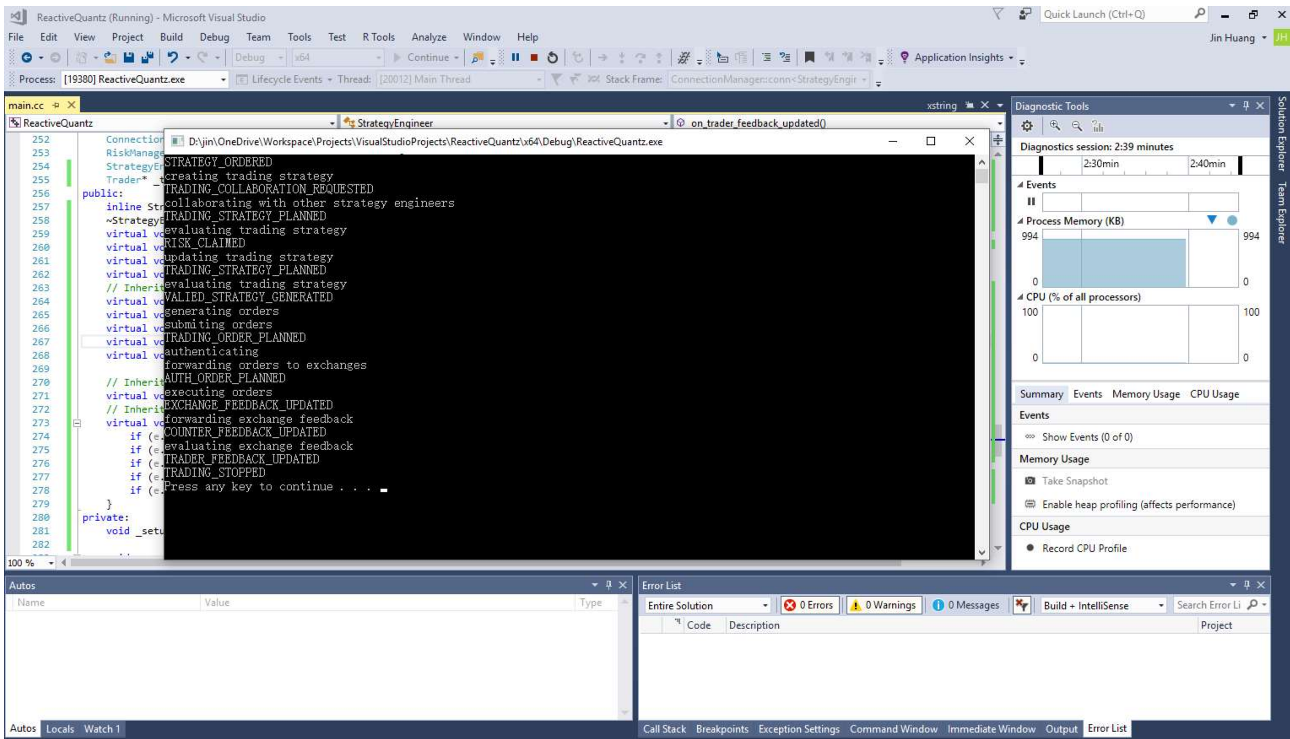


Figure 12: the result of functional testing on the TradingProcess

We can see from the above figure, by filling with concrete and proper algorithms, the generated code skeleton can work in the expected behaviour and output the events in a desired sequence which is consistent with the designed domain trading process. At this point, it is important to clarify that the code skeleton is not generated for making everything be arranged in a best way, but just for structuring the target application in a workable way. Therefore, from this perspective, we are providing an alternative method for software engineers to build their applications which are consistent with the domain knowledge. With this method, software engineers still need to find proper methods to complete their work, e.g. technically defining trigger and output events, context, component, and connectors, etc. with specific programming languages and tools.

Finally, since the existing tools have been constructed for an earlier work and have not been updated, currently, the whole framework cannot be effectively evaluated. The following figure shows how the whole framework, including tools, can be evaluated:

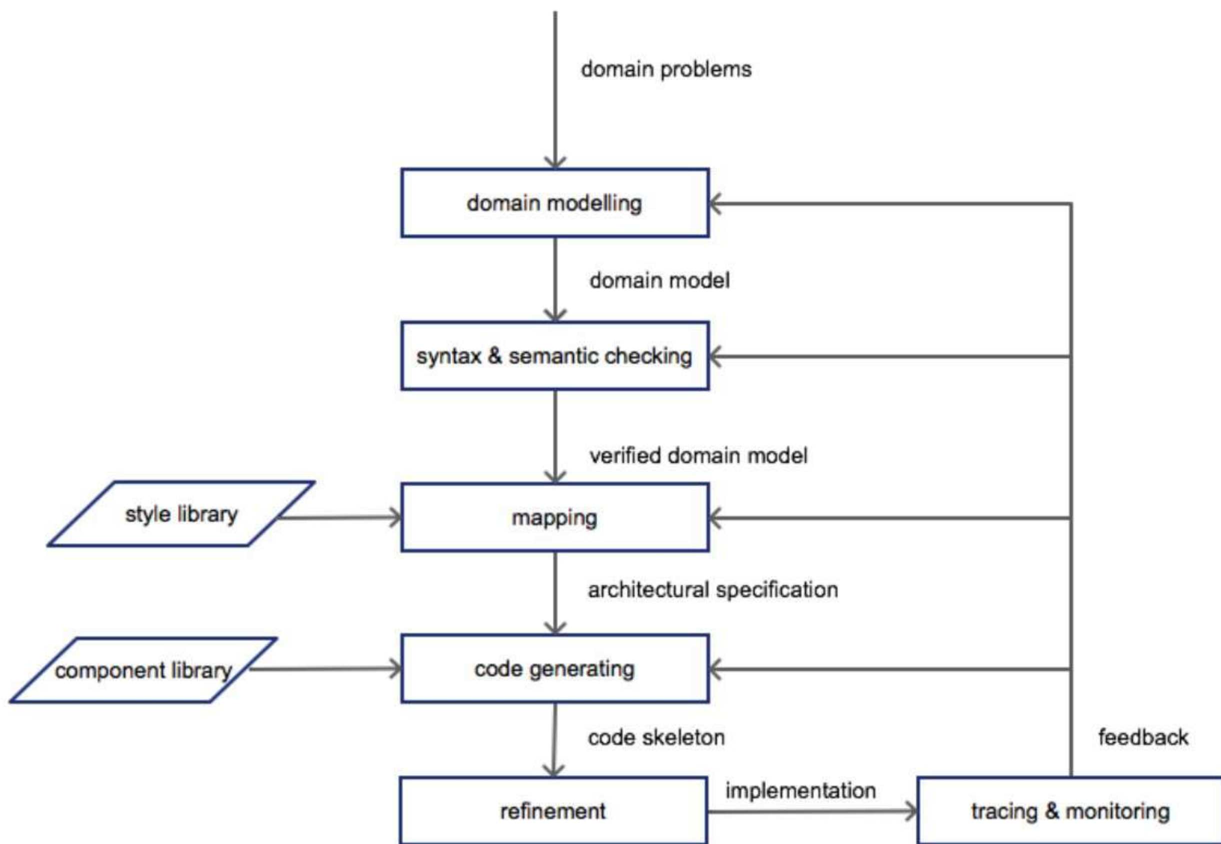


Figure 13: the evaluation method for the whole framework

In detail, the domain modelling method is used to generate domain models according to domain problems, and the verified domain models will be used to generate architectural description by matching and selecting appropriate architectural patterns. By adopting existing implementation from a component library, we can finally get an implementation that will behave as same as the domain design.

7.6 Summary

As a conclusion, this section illustrates how our methods designed in the previous chapters can be applied. Especially how can domain uncertainty be modelling in a specific language, and how can the models in our language can be transformed into architecture description, and into codes further. Since domain models and software architecture descriptions are all abstract representation, it is very difficult for researchers to generate detailed implementation. However, it is still possible to generate a code skeleton that conforms to a domain model and the relevant architecture design. As discussed before, the functional properties of a self-adaptive application can be evaluated by checking the consistency between the concrete event sequences and the designed event sequences.

8 Conclusion and future work

Following the increasing complexity of business domain and modern software systems, self-adaptation techniques have been introduced into the field of software engineering. With these techniques, a software system can be self-adaptive at its operational stage. That is, a self-adaptive software system can adapt itself to changes for achieving the system goals at runtime. However, it is still difficult to construct self-adaptive applications, especially the applications created for solving domain-specific uncertainty. Particularly, there is still lacking of a way for software engineers to develop their applications which can behave consistently with the domain knowledge. Without having this kind of problems solved, software engineers cannot answer the questions, like why designing in this way, how to keep the dynamic behaviour of an application consistent with the domain processes, and why we need self-adaptation technique for solving a given domain problem. Therefore, in this situation, new methods and tools for helping design of self-adaptive applications are required.

To solve this issue, this dissertation proposes a domain-driven method for creating self-adaptive application architecture, especially the methods and tools for describing domain-specific uncertainty and dynamic architecture. In detail, this dissertation has made the following contributions to the research in the field of computer science:

- provides a new understanding of software architecture with respect to self-adaptation via architectural styles that can be used to guide the design of self-adaptive applications, including a self-consistent terminology;
- provides a novel language that can be used to describe domain-specific processes with uncertainty, including the compiler, a generator for producing architecture description with the models in this language and the relevant transforming principles. This language also has a formal foundation, including a simple event calculus, which can be used to perform formal checking, e.g. consistency checking and completeness checking;
- provides an improved version of Grasp, which can be used to describe the dynamic aspects of an architecture. Further, this improved Grasp can be inherently used to specify how domain-specific knowledge and uncertainty constrain the architectural decisions. The compiler and a generator that can be used to generate code skeletons from the architecture descriptions in this language and the relevant code generating principles are also included;
- provides a novel architectural pattern for guiding the design of the self-adaptive application architecture. This pattern is created by applying existing architectural patterns.

However, due to the time limit, there are also some issues have not been solved or fully solved in our work: currently, there is still not a complete formal foundation for Grasp+;

- not all patterns have been evaluated and discussed;
- there is still no dependable method for designing architectural pattern;

- it is still not clear that how much a certain architectural pattern supports the design of a self-adaptive application is;
- currently there is still no automatic generators that can automatically produce a sound software architecture or a collection of codes composed in good style;
- there is still lacking of a framework for both domain experts and software engineers to construct self-adaptive applications in a simple, easy and dependable way;

Therefore, we would like to continue our research by trying to find solutions to the following issues:

- how best to design a self-adaptive application in an extremely complex environment. For example, considering an application that has uncertain number of self-adaptation processes;
- whether it is possible to produce quality implementation from a well-designed domain model automatically. The core difficulty is that models are more abstract than implementations, and a model may not provide the sufficient information at a desired granularity;
- whether it is possible to discover architectural patterns from domain knowledge. The success of software applications is based on proven architectural patterns. Considering the interaction patterns in human society are also proven, whether it is possible for an application to evolve itself with new patterns that are learnt from the domain knowledge automatically.

References

1. de Lemos R, Giese H, Müller HA, et al (2013) Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. Springer Berlin Heidelberg, pp 1–32
2. Garlan D (2010) Software engineering in an uncertain world. In: Proc. FSE/SDP Work. Futur. Softw. Eng. Res. - FoSER ’10. ACM Press, New York, New York, USA, p 125
3. Raccoon and Dog (2013) Unknownness. ACM SIGSOFT Softw Eng Notes 38:8. doi: 10.1145/2507288.2507318
4. Bass L, Clements P, Kazman R (2013) Software architecture in practice. Addison-Wesley
5. Microsoft Patterns & Practices Team (2009) Microsoft® Application Architecture Guide, 2nd. Microsoft
6. Perry DE, Wolf AL (1992) Foundations for the study of software architecture. ACM SIGSOFT Softw Eng Notes 17:40–52. doi: 10.1145/141874.141884
7. Clements P, Bachmann F, Bass L, et al (2011) Documenting software architectures : views and beyond. Addison-Wesley
8. Fielding, Thomas R (2000) Architectural styles and the design of network-based software architectures.
9. Murch R (2004) Autonomic computing. Prentice Hall Professional Technical Reference
10. Kephart JO, Chess DM (2003) The vision of autonomic computing. Computer (Long Beach Calif) 36:41–50. doi: 10.1109/MC.2003.1160055
11. Perez-Palacin D, Mirandola R (2014) Uncertainties in the modeling of self-adaptive systems. In: Proc. 5th ACM/SPEC Int. Conf. Perform. Eng. - ICPE ’14. ACM Press, New York, New York, USA, pp 3–14
12. Milner R (Robin) (1989) Communication and concurrency. Prentice Hall
13. de Silva L, Balasubramaniam D (2011) A Model for Specifying Rationale Using an Architecture Description Language. Springer Berlin Heidelberg, pp 319–327
14. Shaw M, Garlan D (1996) Software architecture : perspectives on an emerging discipline. Prentice Hall
15. Ran A, Nokia Research Center (2001) Fundamental concepts for practical software architecture. ACM SIGSOFT Softw Eng Notes 26:328. doi: 10.1145/503271.503269
16. Solms F (2012) What is software architecture? In: Proc. South African Inst. Comput. Sci. Inf. Technol. Conf. - SAICSIT ’12. ACM Press, New York, New York, USA, p 363
17. Merriam-Webster Definition of Context. In: Merriam-Webster. https://www.merriam-webster.com/dictionary/context. Accessed 26 Feb 2017

18. Parnas DL, Carnegie-Mellon Univ., Pittsburgh P (1972) On the criteria to be used in decomposing systems into modules. Commun ACM 15:1053–1058. doi: 10.1145/361598.361623

19. Gorton I (2011) Essential software architecture. Springer

20. Shaw M, Clements P A field guide to boxology: preliminary classification of architectural styles for software systems. In: Proc. Twenty-First Annu. Int. Comput. Softw. Appl. Conf. IEEE Comput. Soc, pp 6–13

21. Buschmann F, Meunier R, Rohnert H, et al (1996) Pattern-oriented software architecture : a system of patterns. Wiley

22. Crawley E, Cameron B, Selva D System architecture : strategy and product development for complex systems.

23. Kruchten PB, Kruchten PB (1995) The 4+1 view model of architecture. IEEE Softw 12:42--50.

24. Garlan D, Shaw M (1994) An Introduction to Software Architecture.

25. Albin ST (2003) The art of software architecture : design methods and techniques. Wiley

26. Garlan D, David (2014) Software architecture: a travelogue. In: Proc. Futur. Softw. Eng. - FOSE 2014. ACM Press, New York, New York, USA, pp 29–39

27. Evans E (2004) Domain-driven design : tackling complexity in the heart of software. Addison-Wesley

28. Vernon V (2012) Implementing domain-driven design. Addison-Wesley Professional

29. Ellis WJ, Rayford D, Hilliard RF, et al Toward a recommended practice for architectural description. In: Proc. ICECCS ’96 2nd IEEE Int. Conf. Eng. Complex Comput. Syst. (held jointly with 6th CSESAW 4th IEEE RTAW). IEEE Comput. Soc. Press, pp 408–413

30. Garlan D, Monroe R, Wile D (1997) Acme: an architecture description interchange language. Proc 1997 Conf Cent Adv Stud Collab Res 7.

31. Allen RJ, Allen RJ, Jackson D, Shaw M (1997) A Formal Approach to Software Architecture.

32. Luckham DC, Luckham DC (1996) Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. Princet. Univ.

33. Kenney J (1996) Executable Formal Models of Distributed Transaction Systems Based on Event Processing.

34. Oquendo F (2004) π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. ACM SIGSOFT Softw Eng Notes 29:1. doi: 10.1145/986710.986728

35. Milner R (Robin) (1999) Communicating and mobile systems : the π-calculus. Cambridge University Press

36. Esfahani N, Razavi K, Malek S (2012) Dealing with uncertainty in early software architecture. In: Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng. - FSE '12. ACM Press, New York, New York, USA, p 1

37. Salehie M, Tahvildari L (2009) Self-Adaptive Software: Landscape and Research Challenges. ACM Trans Auton Adapt Syst 4:1–42. doi: 10.1145/1516533.1516538

38. Andersson J, Baresi L, Bencomo N, et al (2013) Software Engineering Processes for Self-Adaptive Systems. Springer Berlin Heidelberg, pp 51–75

39. Shang-Wen Cheng, An-Cheng Huang, Garlan D, et al Rainbow: architecture-based self-adaptation with reusable infrastructure. In: Int. Conf. Auton. Comput. 2004. Proceedings. IEEE, pp 276–277

40. Brun Y, Brun Y, Desmarais R, et al A Design Space for Self-Adaptive Systems.

41. Fowler M (2004) UML distilled : a brief guide to the standard object modeling language. Addison-Wesley

42. Wand Y, Weber R (1991) A unified model of software and data decomposition. Proc twelfth Int Conf Inf Syst 101–110.

43. Wand Y, Weber R (1990) An ontological model of an information system. IEEE Trans Softw Eng 16:1282–1292. doi: 10.1109/32.60316

44. Wand Y, Weber R (1995) On the deep structure of information systems. Inf Syst J 5:203–223. doi: 10.1111/j.1365-2575.1995.tb00108.x

45. Wirth N (1996) Extended Backus-Naur Form (EBNF). Iso/Iec

46. Peled DA (2001) Software Reliability Methods. doi: 10.1007/978-1-4757-3540-6

47. Schmidt DC, Buschmann F, Henney K (2000) Pattern-oriented software architecture. Wiley

48. Vernon V Reactive messaging patterns with Actor model : application and integration patterns in Scala and Akka.

49. Brooks FP The mythical man-month : essays on software engineering.

50. Investopedia - Sharper Insight. Smarter Investing. http://www.investopedia.com/. Accessed 19 Mar 2017

51. ANTLR. http://www.antlr.org/. Accessed 28 Mar 2017

52. Xtend - Modernized Java. https://www.eclipse.org/xtend/. Accessed 28 Mar 2017

53. Gamma, E. (1995). *Design patterns : elements of reusable object-oriented software*. Addison-Wesley.