Developing a Global Observer Programming Model for Large-Scale Networks of Autonomic Systems

David John Lamb

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

School of Computing & Mathematical Sciences
Liverpool John Moores University
Liverpool, United Kingdom

June 2009

Abstract

Computing and software intensive systems are now an inextricable part of modern work, life and entertainment fabric. This consequently has increased our reliance on their dependable operation. While much is known regarding software engineering practices of dependable software systems; the extreme scale, complexity and dynamics of modern software has pushed conventional software engineering tools and techniques to their acceptable limits. Consequently, over the last decade, this has accelerated research into non-conventional methods, many of which are inspired by social and/or biological systems model. Exemplar of which are the DARPA-funded Self-Regenerative-Systems (SRS) programme, and Autonomic Computing, where a closed-loop feedback control model is essential to delivering the advocated cognitive immunity and self-management capabilities.

While much research work has been conducted on various aspects of SRS and autonomy, they are typically based on the assumptions that the structural model (organisation) of managed elements is static and exhaustive monitoring and feedback is computationally scalable. In addition, existing federated approaches to distributed computation and control, such as Multi-Agent-Systems fail to satisfactorily address how global control may be enacted upon the whole system and how an individual component may take on specified monitoring duties - although methods of interaction between federated individuals is well understood. Equally, organic-inspired computing looks to deal with event scale and complexity largely from a mining perspective, with observation concerns deferred to a suitably selective abstraction known as the "observation model". However, computing and mathematical science research, along with other fields has developed problem-specific approaches to help manage complexity; abstraction-based approaches can simplify structural organisation allowing the underlying meaning to be better understood. Statistical and graph-based approaches can both provide identifying features along with selectively reducing the size of a modelled structure by selecting specific areas that conform to certain topological criteria.

This research studies the engineering concerns relating to observation of large-scale networks of autonomic systems. It examines methods that can be used to manage scale and generalises and formalises them within a software engineering approach; guiding the development of an automated adaptive observation subsystem – the Global Observer Model. This approach uses a model-based representation of the observed system, represented by appropriately attached modelled elements; *adapters* between the underlying system and the observation subsystem. The concepts of Signature and Technique definitions describe large-scale or complex system characteristics and target selection techniques respectively. Collections of these objects are then utilised throughout the framework along with decision and deployment logic (collectively referred to as the Observer Behaviour Definition – an ECA-like observational control) to provide a runtime-adaptable observation overlay. The evaluation of this research is provided by demonstrations of the observation framework; firstly in experimental form for assessment of the Signature and Technique approach, and then by application to the Email Exploration Tool (EET), a forensic investigation utility.

Acknowledgements

Firstly, my utmost gratitude to Prof. Taleb-Bendiab for all his advice, support, encouragement and friendship throughout this research and writing up; in addition to facilitating funding for the PhD. In this regard, I must extend thanks to Prof. Merabti, Dr. Dhiya Al-Jumeily and the School of Computing and Mathematical Sciences.

Secondly, I would like to thank all the members of academic and technical staff, along with researchers for their friendship and support, with special mentions to: Dr. Martin Randles for his discussions and advice, particularly in the late stages of writing up, Dr. Thomas Berry for talking me into the PhD, Dr. John Haggerty for the opportunity to work on the EET project, and Dr. Denis Reilly for all his assistance in HND teaching. Mengjie Yu, Mike Baskett, David Llewellyn-Jones, Ian Rhead, Chris Wren and Oliver Drew all supplied endless good humour, chats and at times kept me sane. The administrative staff deserve many thanks for the help received in navigating the university's organisation; particularly Trish Waterson and Lucy Tweedle.

Thirdly, I would like to thank my family; particularly my Mum and Dad – Christine and Bill - without whom none of this would have been possible. Much appreciation to my Mum for all she did during this write up. Liz and Ste (my siblings) both provided opportunities to escape the write up when it was needed. Thanks also to Uncle David for donating his old laptop; I do not know how I managed without one.
It was my late Dad's interests in radio and electronics that ultimately led me towards computing as a career and academic interest. *This work is dedicated to his memory.*

Finally, I must thank Sarah, the love of my life for her support, understanding, patience and encouragement. Without her, I simply would not have completed this thesis. Her parents, Ken and Frances also allowed me to take over the dining table as a workstation at weekends, for which I am most grateful.

Table of Contents

Table of Figures

Table of Equations and Tables

Chapter 1 – Introduction

Software systems, by nature of their changing development styles, their usage patterns and ever-developing functionality, are increasing in both complexity (functional and organisational), and their operational scale (widely distributed systems-of-systems, where each component is in itself a major software system, and giant monolith-type systems). As each independent yet interlinked component is tweaked, tuned, or redesigned to fulfil a new requirement, the overall makeup of the system is affected, albeit subtly; the net effect that the software system as a whole gradually evolves – such that its original design and requirements no longer adequately describe its behaviour, organisation, or possibly even its intentions [1].

In turn, the complexity involved in the design, development and maintenance of such software systems is ever increasing. This problem is compounded by new generations of systems that go some way to automating this evolutionary process. Although this may not – yet – be as advanced as adjustments made by software engineers and administrators – software is capable of reconfiguration; for efficiency, fault tolerance, or some other system requirement. This can range from automated adjustment of user parameters or other configuration item(s), to effectively constructing component-based systems or employing alternate services/components; examples in [2, 3].

Hence, it is clear that given increasing system complexity; software that can manage many of its own operations is an extremely attractive proposition. However, although engineering models and techniques exist that are suited to evolutionary development, they are based on the *iterative* concept of co-ordinated dialogue between end-user/analyst/engineer – governed analyse → design → implement → reanalyse [4]. Following an engineering framework provides reassurance and guarantees, clear architecture, and design; facilitating application of post requirements engineering formal specification to provide proofs of key software behaviour. However, this approach seems to contrast with the near-organic, continual self-managing process by which a system may adjust itself; its configuration deviating further from original design models and specifications.

1.1 Resilient Self-Managing Software via Observation

In order to introduce the area of self-managing software and its relationship to observation, it is useful to consider the computer science interpretation of Cognitive Immunity (CI). The notion of CI was first introduced in a DARPA-funded Self-Regenerative System (SRS) programme in 2004 [5], which outlines four functional objectives, including CI. The latter was defined as "...introspection on the system's operation to understand the state, and reasoning about that state to recognise problem areas or errant behaviour. Further reasoning is then undertaken in order to determine solutions that will restore the desired system functionality."

The CI proposals in the SRS programme ranged from high-level architectural approaches for component management to facilitate failure recovery and component-trust quantification, through to plans for the specification of learning and repairing software systems. Given the aim of *this* research for a generalised software development approach part-facilitating self-managing systems, the SRS-related work on developing an software architecture approach [6] was of great interest. Further research by the same authors [7] showed that the approach aimed primarily to protect systems from unauthorised actions, sourced either internally or externally. However, of particular interest was that monitoring of the underlying system was undertaken by way of specialised instrumentation attachment to system processes. These approaches relied on a *pre-engineered* decomposition of important system tasks, identifying related processes and attaching the required monitors.

Whilst the CI concepts, architecture models and associated techniques were reported to be beneficial for the design of software-intensive systems, it has only been applied to small-scale systems including autonomic systems. However, this thesis aims to explore the applicability of CI to large-scale networks of autonomic systems. As such, the research must look to address the considerations brought about by the effects of scaling and complexity on these system-level instruments and observers in particular.

1.2 Motivation: Observation[1] in Complex Software Systems

In the interests of clarity, the motivating factors influencing this research will be broken down into three main categories, further detailed in the following sub-sections. The first is a brief description of the problem domain's characteristics – that of increasing software complexity and scale. The second borrows theory from traditional Software Engineering and Autonomic Computing to describe desirable properties of developed software, and why they are so. Thirdly the research motivation is concluded with an overview of existing software engineering techniques and how they are lacking in these areas; further narrowing the intended focus of this research.

1.2.1 Software System Complexity

As introduced in earlier sections, "complexity" is one of the characterising features of the Software Engineering (SE) domain under investigation. Perhaps the simplest definition is to consider complex software to be: where the operation is not well specified, and its behaviour is nonlinear (i.e. not easily understood by examining the behaviour or the specification of the component parts).

Often, this complexity is a product of the sheer scale of a system. It is quite possible to encounter a complex system composed of many very simple individual components. Taking the example of an ant colony [8], the important behavioural characteristics of each type of ant can be observed and recorded, though this does not provide the observer with an overview of the colony's (i.e. the entire system's) operation. This creates an incomplete design model situation. An observer can know each component's behavioural rules; yet understand none of the system's operation. If system operation is not well understood, the dangers of unforeseen and apparently unpredictable effects are undesirable effects. An observer could (due to an unintentionally-blinkered viewpoint) "tune-up" some component's operation, seemingly improving the local situation, while having catastrophically adjusted the

[1] Throughout this thesis, the term Observer is used in relation to a software system. Unless identified as otherwise, it is referring to a sub-system or component that is responsible for noting changes in the state of another component or system and acting accordingly.

behaviour of another dependent component. In other words, it is possible that the individual components of a system have conflicting concerns and in order for the whole system to operate satisfactorily, each contributing component must operate in a less-than (locally) optimum fashion.

These (and other related) characteristics further examined in Chapter 2 complicate the engineering process; successful software development relies on clear requirements and a complete design model. A software engineer may be working on a component in a large and complex system, whose behaviour is governed by the interaction of various individual software components. While they each may have well-specified design plans, behavioural models, coding APIs, and unit test results; this does not mean that the software engineer will have a clear and complete overall system design model. Any system-wide software created with an incomplete understanding will not necessarily operate in the best interests of the system.

Leading on from this, the next section introduces a set of properties and characteristics desirable in these software systems, which provide a conceptual guide to how some of these issues of complexity may be overcome.

1.2.2 Autonomic Software

Whereas the previous section concentrated on some of the issues surrounding software systems complexity, this section brings in some of the concepts from related work that looks to delegate the management of system complexity to the software system itself.

The first, and possibly most well known of these conceptual models is that of Autonomic Computing (AC), first discussed in 2001 by Paul Horn of IBM [9]. AC is aimed primarily at distributed software systems, and aims to tackle the complexity associated with the immense interconnectivity and management of these software systems. Kephart and Chess, also of IBM [10], pointed out that while there is a great deal of power in the ubiquitous nature of computing devices and their common standards for interconnectivity; directly managing the resulting system architecture and its dependent components is too complex for an engineer (or team of engineers) to contemplate.

The Autonomic Computing ideal is inspired by the autonomic nervous system of mammals, and suggests that software systems should self-manage without any conscious (i.e. engineer/operator) intervention. This led the way for a whole host of self-*something* buzzwords (collectively termed self-star / self-* [11]), above and beyond the initial "Self-CHOP" characteristics proposed by the IBM initiative, which were Self-Configuring, Healing, Optimising, and Protecting. The notion of the Self-Healing and Configuring characteristics is a particularly attractive proposition as a method toward equipping complex software systems with a degree of Cognitive Immunity. The next section aims to give an overview of how traditional engineering models are ill equipped to deal with these notions, and outline the difficulties of adopting the Autonomic Computing approach in a traditional software engineering scenario.

1.2.3 Lack of Engineering Support and Model

This section gives a brief overview of some traditional software engineering methods and how they are lacking or inappropriate when it comes to the development of support software for large-scale and complex systems. Software engineering is a discipline concerned largely with the development and support of an engineering-style approach and related processes to aid in the building of software systems.

While there is some dispute [12, 13] about the formal use of the "engineering" term, various UK-based engineering councils and guilds are recognising that their membership schemes can extend to Software Engineers. Regardless of the (non) clique surrounding the engineering-derivation, there is little doubt that many Software Engineers also work within a sub-discipline of Computer Science, concerned specifically with the formalisation, modelling, development and use of engineering concepts to the various aspects in a software system's lifecycle, from analysing through to testing [14]. As such, software engineers look to formalise, standardise; produce and use methods that lay down best practice to software development problems.

Analysis and Formalisation of Fixed Requirements

Traditional "waterfall" SE techniques rely on a firm set of requirements, the ability to formalise those requirements for future use in verification and validation, and a development cycle that refines and designs a software system based on the all-important system requirements. While this cycle adheres to traditional engineering techniques, it is inflexible to requirements changes throughout the development, let alone *after* software deployment. As such, changes during development combined with cost and other constraints can lead to patchwork-type modifications during the development, and ever-changing requirements, often known as requirements, scope or feature creep [15].

Evolution of the traditional waterfall approach led to techniques more flexible to change during development, such as iterative models [16], various examples of which are found in agile development [17]. These model types retain a linear backbone, but *assume* that requirements will change during development; as it is natural that requirements become more fully understood and refined [18]. As such, development cycles operate in short iterations; relying on rapid analysis, design and prototyping, which provide (with end-user support) feedback and information to the next development iteration. However, even iterative development models rely on a requirements (re)acquisition, or refinement phase, followed by a redevelopment phase. These development approaches do not take into account the architecture, design, maintenance and testing of software that can evolve and alter its own configuration during runtime [19].

This does not render existing SE models (nor an engineering-based approach) irrelevant for large, complex and evolving (i.e. dynamic) systems. However, it is clear that the translation from analysis/requirements phases to implementation may not be as rigidly connected to the implementation and testing as in a static system. The next subsection will look at SE modelling and architectural approaches that attempt to cater for runtime dynamism.

Runtime Adjustment and Requirements

The previous subsection gave a quick overview of software development for fixed requirements within a well-understood problem domain. This section will give a similar review of engineering approaches that permit runtime adjustment or some degree of dynamism in developed software.

Firstly, policy, rule-based, and other related software systems go some way to creating a layer of indirection between "system requirements" and "implementation concerns", allowing for a flexible runtime environment that can operate in accordance with a user-defined set of boundaries. Runtime inspection of system state is used to calculate suitable behaviour based on (typically design-time specified) goals/rules, enabling adaptive behaviour. At a small scale, rule-based software is capable of sophisticated reasoning and associated behaviour; adaptive to some extent to its environment against a set of goals [20], and approaching a large scale via component composition and re-use [21]. Equally, such work has been extended (via, for example, Agent-based systems with its specified dialogues [22]) to provide a degree of autonomous control over multiple elements within distributed systems, where manual per-component management would be too complex and/or costly [23]. Agent system architectures rely on global system policy taking the form of a basic knowledge-oriented goal approach [24] and are no doubt extremely useful as a form of self-management; whereby system configuration is best expressed as a set of global policies or rules. This approach allows elements to tailor their response appropriately to deal with local conditions whilst retaining some form of global system control.

However, these types of system architecture do not easily support overseeing observation, engineered co-operation, controlled configuration and optimisation [25]. The components' autonomy enforces behaviour based on assessment of local situations and system-wide goals or constraints; they effectively operate as independent components with a degree of governance via a system goal/rule-set. This architecture makes a good model for distribution of computation loading; each component or agent is responsible for managing its own domain, controlled by system rules. However, this creates difficulties in terms of adopting a responsive, scalable approach for observing and adapting (e.g. tuning) component rules based on system-wide overseen observations. System-wide observation and feedback controlled self-configuration and optimisation is hindered by the relative independence of each component and variability of design.

With such a large number of widely-distributed components, identifying areas in which to monitor the system and obtain feedback is not an easy task; with selection influenced by the organisation of components, the current system state and a multitude of system-specific concerns. While there are techniques that are applicable for engineers to model a large monitoring and feedback system, there is limited engineering support to guide the overall design and structure of such systems. In summary, a Software Engineering approach that combined tested engineering practice and could integrate research ideas and approaches associated with Cognitive Immunity would help in the development of self-managing/configuring systems.

There are still limitations in applying current SE approaches to this problem domain, as detailed above, and there will be issues in applying new (to the field of software engineering) techniques associated with large-scale and complex systems, which will be outlined in the next section.

1.3 Challenges

As discussed in the previous section, many of the motivating factors for this research project are indeed research challenges in their own right. However, as outlined below, this thesis focuses on a smaller subset, namely:

- Scale – The challenges associated with modelling a system whereby an exhaustive model is desirable, but the system size presents difficulty in exhaustive modelling. The proposed approach suggests that a suitable abstraction must be found; furthermore, the observation subsystem should be responsible for its automatic selection and management.
- Complexity – can manifest itself in many different ways; primarily that overall system behaviour and structure is more than the sum of its parts [26]. Observation must look to *effectively* reduce complexity, concentrating on relevant areas. Emergence, discussed in more detail in Sections 2.2.3 and 2.2.4, presents further complications, both structurally and behaviourally.

- Evolution – Many software systems may change – or evolve – during their operation. When this change occurs at runtime, this presents a problem to those subsystems or components wishing to monitor, or even model them.
- Formalising the approach – Given the nature of complex systems, it is likely that an approach that works well in one domain is not necessarily transferable to another. As such, the final significant research challenge is the specification and generalisation of any methods devised to manage the previous challenges.

1.4 Approach

Building on established methods for software design [14, 17, 27], Autonomic Computing [9, 28, 29], and graph theory [30-32]; this research examines how to extend existing autonomic software design methods to equip next generation software systems with scalable observer capability for Cognitive Immunity in large-scale networks of autonomic systems. This encompasses a range of concerns, including:

- Identification of issues that complicate the management of large-scale/complex software systems – preliminary literature/practice review.
- Collection and evaluation of techniques used for system complexity management – literature/practice review.
- Collection of relevant current software engineering systems practice – literature/practice review.
- Investigation of how the above techniques can be integrated to make a software engineering process – the application of the research.
- Investigation into large-scale/complex software applications and their common characteristics - Use this to generalise the findings to increase applicability.
- Evaluation of final methods using a case study.

This research examines some of the significant concerns in the development of self-managing complex software systems. It is embarked on with an understanding that there is significant research already undertaken in the areas of goal-driven, rule-based software. Equally, there is a developing understanding of methods that can be used to manage and simplify certain complex structures. The research therefore looks to formalise a development framework for defining, detecting and applying these practices at runtime to create an adaptive observation subsystem capable of managing large scale and complex systems.

As such, this necessitates the following contributory aims, drawn from the tasks outlined earlier:

- Collection of current software engineering observer design and practice
- Detail the conclusions drawn from evaluation; suggested refined approaches

The remainder of this section will look to provide further detail on those tasks that involve collection of current practice and research, and provide an overview of the scope of this research.

1.4.1 Objectives

In order to conclude the purpose of the project approach; the aim is that it should assist software engineering of large and complex systems, and as such, the main objectives of this research project are as follows:

- Identification of significant issues that complicate the observation management of software (and indeed software engineering as a whole) of a large scale and with complex system structures.
- Investigation into methods that can integrate complexity management and software engineering's approach to observation.
- Specification of software engineering and programming model for designing observers within large scale and complex systems
- Evaluation of methods; case studies and real world applications

1.4.2 Scope

This work has two distinct lines of research, each a large and well explored field in its own right. In this section the focus points in each area will be identified, in addition to linking the two areas together to explain the overall research perspective.

The first of these areas is the study of complexity, with its many sub-fields, many of which feature heavily in Chapter 2 and Chapter 3. The second somewhat distinct areas of study are Computer Science, Software Engineering, their connected disciplines and relevance to complex systems. The other chapters look to tie the two subjects together with the application of complexity management techniques in a software domain.

Complexity is a wide and well-studied field, with significant contributions from areas such as natural sciences (e.g. [33, 34]), business and information systems study (e.g. [35, 36]), and particularly importantly for this work, mathematics and graph theory [37-39]. Each of these fields looks to provide a method of understanding or even managing complexity; be it in the form of a simplistic statistical measurement, a system modelling approach, or simply a variety of observations describing one system from a variety of perspectives.

The relevance to this research is concerned with extracting a desired (and therefore simpler) subset of data from the complex system, or to influence the operation of the system in a controlled and limited manner, managing otherwise unpredictable global actions. This may range from the use of an algorithm to derive a measure from a system, or a modelling approach that divides a complex system into several subsystems with constraints on their interactions and responsibilities. As such, in this work, it is the mathematical and graph-based complexity-management approaches that are of most direct value, as mathematical algorithms are, after all, the most directly translatable to lines of code.

It would be short-sighted to assume that Computer Science is entirely distinct from the study of complexity; there is necessarily much overlap as the target systems have many common features [40]. However, it is the differing focus of computer science used as the separation criterion in this thesis. CS focuses on the study of applying many different techniques within computer systems, whereby complexity management techniques are "only" one such set of techniques. More specifically, the thesis is focused particularly on Software Engineering as a sub discipline; with particular note to the specification and design guidance it provides for the observation of system components and complete systems.

As discussed briefly in Section 1.2.3, SE as a whole specialises in techniques used in the development of software systems; at its broadest, from overall project management through to the maintenance of previously developed software. Given the types of software under investigation, the thesis design is influenced by the subset of software engineering models that acknowledge the evolutionary nature of complex software, both in design and at runtime. This considers traditional engineering-rooted approaches and designs [14, 21], along with iterative and evolutionary engineering models [17, 41, 42], and those that support runtime dynamism, such as dynamic composition approaches [21]. This wide scope is considered in order to try to examine the areas in which the complexity management techniques are relevant to system observers; through from design-time analysis to runtime adaptation.

The thesis will look to include relevant background information from both these areas of research as and where required, and will particularly focus on techniques in the mathematical management of complexity, in order to better understand the way in which it can be applied.

1.5 Main Contributions

As outlined previously, the thesis documents a body of work aimed towards the development and refinement of a software engineering process to support the design and implementation of observation subsystems intended to monitor large-scale, self-organising and complex system structures. To this end, this thesis presents a number of novel contributions to the field including:

- Collation of research relevant to the problem of modelling and observation of large-scale and complex structures, including a preliminary investigation into existing metrication of defining characteristics for large scale and complex structures, along with the related software-specific concerns.
- Definition of a scale-free detection metric, based on existing work by Cohen and colleagues in Acquaintance Immunisation [43].
- Specification of a Global Observer Programming Model and associated software engineering support. The programming model specifies the key programming concerns along with implementation guidelines for a global observation system for large-scale, complex and dynamic software systems and datasets.
- Specification of a high-level software engineering framework for this programming model; comprising architectural overviews, generalised software designs, and implementation examples for key components such as the adapter interface with the observed system.
- Definition of a runtime-adaptable Observation Behaviour language in XML, providing a runtime-evaluable specification of the connections between and concerns of the various observation subsystem components. This, along with design and implementation guidance is intended to assist with the development of runtime-inspection and adaptation "plug-ins" and components for the observation framework.
- Development of a prototype email and social network visualisation tool. While the primary research value of this tool is centred in Computer Forensics, it has provided a useful evaluation and a contributory reference implementation for the global observer programming model.

1.6 Structure

This chapter has given a brief introduction to the research work; its inspiration and where it fits into the fields of computer science, software engineering and research related to complexity management. A detailed breakdown of the format of this research work follows:

Chapter 1 introduces the thesis, giving firstly a brief overview of the motivation for the work and an outline view of the challenges involved. The research approach and relevance is detailed, before summarising the main contributions.

Chapter 2 provides background information and a literature-review-style overview of the elements of complex systems that have guided this research. The first half of the chapter gives an overview of large-scale and complex "features" and a brief review of management techniques; the second half conducts a review of both established and state of the art SE and/or general management techniques for the "systems-of-systems" that govern this research, covering topics such as ULS, Autonomic and Organic Computing.

Chapter 3 discusses some of the problems and potential solutions as regards observation within a complex model. This chapter continues the viewpoint-dependent complexity theme and begins to examine the creation of abstract models, discussing differences in model creation between complete-design functional decomposition and assembling a design "bottom up" from individual components. This complication leads towards graph theory as a method by which complex structures can be represented as graphs and simplified by suitable measures. The chapter concludes with a discussion of scale-free connectivity, its frequent occurrence in complex systems; finishing with examinations of how the connectivity can be detected (including the author's metric), and how the connectivity's strengths and weaknesses can be exploited, concentrating on Cohen's Acquaintance Immunisation.

Chapter 4 begins to address the aims of this research by examining the notion of a complex and large scale observer model, discussing the OO Observer pattern as a starting point. The approach taken is to look at an architectural specification that would permit an observer to operate on large-scale and/or complex systems, and to detail an approach that would manage some of the significant concerns associated with large scale and complex systems. Out of the requirements presented in the chapter, the research concentrates on the management of scale and complexity; adopting a model-based observation view of the system. The architecture proposes that system characterisation/identification, reasoning/planning/determination, and observer deployment should all form significant components within the system. The chapter concludes with a brief description of the Structural Observation Framework, and its requirements.

Chapter 5 refines the previous architectural specification, and presents software designs for the significant components within the structural observer system. This involves a detailed consideration of the Signature and Techniques, key building blocks towards the goal of Typed Observation, and how they will interact with the large scale and complex systems, represented in the observer model as simply a large structure, composed of Modelled Elements.

Chapter 6 completes the design of the structural observer system by an assessment of generalised observation policy specification, the manner in which signatures and techniques can be associated, and how the system-level observer units can be deployed about the system. The chapter is brought to a close with a summary of the presented designs, plus a formalised view of execution detail for some of the system's main processes.

Chapter 7 further develops the designs presented previously, and examines the concerns regarding adapting the framework's behaviour at runtime. This involved a detailed consideration of the designs thus far, along with identification of the areas of the system that made up the basic runtime process. The chapter then proposes the Observer Behaviour Definition, which is based on the reduction of key observation processes to an ECA-type specification.

Chapter 8 opens with a discussion of the considerations required to translate the OBD specifications between their evaluative-type objects and XML representation, considering implementation issues regarding exposure of observer functionality and data. It concludes with an OBDXML specification, which takes the form of an XML schema definition and then presents some use-case discussions surrounding the XML strings.

Chapter 9 provides an evaluation of the proposed observation framework, assessing and demonstrating the effectiveness of the signature matching mechanism. The approach taken is to try and evaluate and validate the components of the approach; progressing onto a case study which applied the model within a software system developed by the author.

Chapter 10 concludes the thesis, outlining the research approach and explaining how it developed into the contributions of this work. This chapter closes by outlining some of the areas of research that are considered suitable further work.

Chapter 2 – Autonomic Software Control

As discussed in Sections 1.2.1 and 1.3, software complexity continues to be the subject of many different research areas, including biological, social, business/information systems, along with mathematics and computing, with considerable interest overlap between fields. In regard of software complexity, complex systems are largely characterised by their many-component composition and non-linearity; therefore most easily understood at an architectural and design level via abstract descriptions [44].

From a software management perspective, complexity of system design, structure or behaviour directly affects complexity involved in monitoring the system. Furthermore, monitoring a large-scale system with limited observation resources places efficiency constraints on the observers. As such, when monitoring large or complex systems; it is likely that complete observation will be unrealistically expensive, leading to a necessary selective reduction in observation targets [45]. Systems that undergo non-specified evolution or any form of runtime change place greater complexity on this selective reduction process.

The first part of this chapter will aim to explore the characteristics of complex software systems, and to identify specific observation and modelling challenges that arise from their complexity [2]. The latter half of this chapter will detail a review of some existing software approaches to system complexity; particularly those with a relevance to the issue of monitoring, management and observation.

[2] This chapter gives an overview of Complex Software Systems within the scope of this thesis, including their characterising features – and their challenges and potential areas for exploitation. As a whole, it is intended to give a sufficient, though by no means exhaustive background in the manifestations of complexity that characterise the systems under investigation.

2.1 Introduction

Goldenfeld and colleagues argued that a seemingly simple system can exhibit a very complex internal behaviour and/or structural properties, and that a system's perceived complexity is very much dependent on the observer's viewpoint [46]. The often-cited example is that an outside observer would see a tornado as a reasonably easily-described (even structured) flow of wind, though it would seem very much more complex (possibly even random) to a fly caught up inside. Admittedly, there is a great deal of simplification in this theory – the "viewpoint selection" method is described only as the correct descriptive level is "determined by the nature of the underlying problem."

This complements the idea that a complex system can be greatly simplified by functional abstraction [44]: defining and naming a system part or characteristic, and outlining only those behaviour elements that are relevant to the user. This black-box approach emphasises that while the inner workings of this bounded element are unimportant; the relevant external behaviour and interaction methods are of utmost importance. Hierarchical levels of a system's organisation are ideally comprised of appropriate functional abstractions of lower-level behaviour. As such, successful system simplification as a complexity management technique relies on finding the appropriate viewpoint or abstraction.

Equally, functional abstraction and viewpoint-dependent complexity applies to many software engineering aspects. Thus, in order to manage successfully a software system via observation, the observers must be able to cope with the managed systems' complexity in a manner that permits sufficient observation within available resource constraints.

Hence, as exemplified by Randles and colleagues [47], where system action histories were used as tools to manage the observation complexity, software could exploit this notion of viewpoint complexity such that the observer's viewpoint only examines a limited *and relevant* subset of the software "world" – creating something resembling an observation model. In a rather more generalised observation model, the difficulty is expected to arise in finding a suitably abstract viewpoint that:

- Effectively reduces the complexity of the system by including only the required observation facets/viewpoints.
- Ensures it is not so limited as to ignore potential areas of effect, such as those mistakenly left outside the observation targets, due to miscalculation, or incomplete/incorrect system information.
- Is not unnecessarily reliant on static structures; such that it cannot manage a changing system

Allowing such observation requires rigorous techniques to define, deploy and operate observation logic at a system level. Hence, the first half of this chapter examines precisely that; collation and review of characteristics that may allow a suitably generic modelling approach towards complex systems and their characteristics. Alternative self-regulation methods will be explored for unit management. This implies self-contained regulation, rather than that of a centralised controller or observer, and concepts and approaches will be examined in the later half of this chapter.

2.2 Features of Complexity

Software systems are varied, and their variable nature suggests that attempting to definitively specify a Generic Complex Software System would be of little merit. Accordingly, this section aims to give an overview of some common features, and therefore challenges and areas for exploitation present in software systems, along with many other complex systems. Thus, in specifying the features, the rationale is to indicate the significant properties associated with complex systems, and the ways in which they could be managed.

2.2.1 Large Scale: Huge Datasets and Monolith Software

As identified in Section 1.3, large system scale presents a significant challenge to monitoring and observing complex software systems in accordance with a model-based approach. In many software systems, apparent system complexity is brought about mostly by the number of system components and their many varied interconnections. Whilst an individual component may be adequately described by traditional techniques, the quantity of components alone could render it impractical to construct and, importantly, interpret an exhaustive design model.

It can be difficult to precisely quantify this characteristic in software; it could be argued that almost any software system exhibits large-scale properties if sufficiently functionally-decomposed. Equally, many software systems are capable of manipulating large data models, such as those built around large database systems (for which mature research and practice exists [48-50]). However, these are not necessarily *complex* systems in their own right - in this context of management and observation. Typically, these data sets and their component support are well understood in the target domain; as such, there are often domain-specific techniques such as appropriate indexing structures (e.g. [51]), exploration and search optimisations that can be used to reduce the "scale-only" complexity present in a large quantity of data.

Additionally, remaining in the context of software, scale is apparent in system organisation; where functionality relies on many different components. In many cases, these components provide one or more services for the system, thus distributing required computation about many different processing units. As with large data sets, in the case of monitoring and observing such systems, their (potential) component composition may necessitate a reduction of scale-related complexity. Very large-scale processing systems may make use of a huge number of hosts and processing components but may still make use of relatively well-understood or well-specified domain-specific techniques to manage the scale. For example, the hugely-parallel *SETI@home* loosely co-ordinates tasks and results across a great number of computational processes to facilitate "complex" processing; facilitated through both the massive nature of the system goal, and the well-specified manner in which it can be sub-divided [52]. Generally, when the system organisation is sufficiently simple or well understood, scale-only problems associated may be sufficiently reduced and represented as sub-models, created by domain-specific techniques, or monitored by sampling or statistical techniques of the whole system model [53].

However, in the complex system, scale is rarely present as an isolated property. While solely monolithic software systems do exist, they are often developed within domain-specific optimisations and tailored abstractions, into which monitoring information can be encapsulated. This research aims to specify a software engineering and programmatic model for observing large-scale complex systems. The nature of system complexity indicates that when a system exhibits large-scale properties, it is likely to feature other distinct characteristics and modelling challenges. The next section will define complex system structure by examining another property which describes characteristics of *the components* that are used to make up the large scale systems.

2.2.2 Component Independence and Systems-of-Systems

As alluded to in the previous section, and as identified in work related to Ultra-Large-Scale Systems [54], another key characteristic of a large-scale *complex system*, (rather than just a *very-large-scale standalone*) is that each of its components are independent. Systems fitting the "system-of-systems" definition are made up of components that fit one or both of the following definitions. Independent components are usable in a great many different situations outside of their current scope (*operationally-independent*), and/or do not rely on and are not wholly controlled by the utilising system (*managerially-independent*).

The case of system composition of *managerial-independent* components leads to a likelihood of each component operating to different (i.e. localised) optimal criteria than the system as a whole. This may, for example, indicate that the system cannot place equal reliance on a 3rd-party-managed vs. a system-managed component. In the case of a set of components utilised only by a single system, in the interests of simplicity, it is tempting to largely dismiss this concern and take the approach that each component should be "tuned" to the global system optimum. However, even then, the case of *operational independence* must be considered - if a component can be described as "independent" by the criteria above, it should be considered a system in its own right, therefore operating to its own component-specific criteria, using other services, and *potentially* providing services to other systems. In a multi-use situation, the component's own priorities may be subtly different to dependent system *x*, which in turn may be different to system *y* and so on. Furthermore, other users of the component may alter in numbers and needs during system operation, placing new external constraints on the quality of service delivered. While mechanisms for establishing performance measures and service contracts (e.g. [55]) can agree required levels of service, allowing the component to trade its services while fulfilling its requirements; from a user system's perspective, it may still be important to monitor and deliberate on usage statistics and make decisions on planned redundancy or configuration changes.

The next section examines system properties that often exist in conjunction with large scale, component-independent systems though are not generally apparent from a static system description.

2.2.3 Emergence of System Behaviour

Emergent system behaviour often occurs in large scale systems-of-systems. Emergence, though distinct from scale, is often associated because it describes behavioural and structural properties that occur only due to the *collective* organisational contribution of many components [56]. As such, emergent properties are those that are generally difficult to present as an enumeration of system states before system operation commences. Furthermore, they are difficult (or in cases, impossible) to calculate from the individual components' behaviour, as they are not the result of a single component's actions.

Emergence is popularly used to describe several naturally-occurring phenomena observed in the real and computing worlds. Barabasi, Albert et al observed that the frequency with which pages making up the World Wide Web on the Internet linked to other pages is described by a power law [57]; following the rich-get-richer model, rather than a normal degree distribution that would be expected were the network of pages built randomly. However, there is no centralised control of page links, with the possible exception of super-linking large search engines; the organisation simply reflects the developed social interest between pages. Flocking [58], Swarming [59] and Schooling [60] are all naturally-observed patterns of animal and insect behaviour that produce observed global co-ordination (i.e. the group appears to be acting as a whole), yet the individual animals or insects are following very simply-described social behavioural patterns and rules. Other complex social interactions, such as those associated with Ant Colonies (e.g. waste and disease management, defence of "home") arise from certain situations and a single ant's reaction and simple chemical stimuli forming a communication method between many ants and resulting complex behaviour [61].

It is therefore important to differentiate between scale and emergence. While they often occur together, large scale in isolation creates modelling and design challenges that complicate the understanding of overall system behaviour only due to the number of component interactions and resulting possibilities. As discussed previously, large-scale-only systems may be sufficiently "sub-modelled" by sampling. Therefore, in a large scale non-emergent system, behaviour monitoring and modelling may be theoretically determined by enumeration and reduction of potential environmental states, followed by calculation on each component's behaviour thus forming a model-based description of overall behaviour and resulting states.

However, emergence creates an "anti-reductionist" situation whereby system behaviour is not deterministically calculable from individual component behaviour [1]. As such, from a design viewpoint, it could be argued that the management of both large scale and emergent systems are best addressed via abstraction-type design or operation models. However, with emergent systems in particular, this point needs further clarification: an observer wishing to monitor a system exhibiting emergent behaviour must adapt to changing system conditions, and must be capable of obtaining system information from some level of "global" observation of the system model; rather than a purely reductionist and calculative model. Selective observation may need to alter selected targets to reflect changes in importance and relevance as new system organisations emerge. They may be characterised by structural or behavioural changes, though the work in this thesis will concentrate on monitoring structural change.

As such, systems providing overviews and/or management for emergent systems must have a reactive element in order that they can correctly adapt to emergent structural changes in the system, along with correctly understanding emergent trends in system behaviour [62]. Equally, if an observer is expected to provide feedback to the system, it must be appreciated that the observer's actions may alter the emergent behaviour of the system. While this could steer the system in a desirable direction, if the observer is unable to calculate completely the predicted results of its actions, it could prevent it from reaching a desirable stable/equilibrium state. This necessitates some form of analysis and prediction routine within the observation model [63]. In an attempt to avoid the complexities associated with this particular issue, some strategies [64] for managing emergent designs involves a "hold-off" approach - whereby the system is allowed to establish its structure and behaviour – deferring monitoring action / adjustments that can best serve the behaviour until it is established.

However, whichever approach is adopted, in order to ensure viability as a software management technique, the system must still be *sufficiently* monitored such that measures to establish emerging and steady states are defined - and can be evaluated, as in the example by the author [65]. In summary, emergence in both behaviour and structure is observed in many large systems, both within large computer software systems and in many other fields. It is a key factor in the definition of a complex system. Additionally, emergence brings several unique challenges in how management and observer systems must co-operate with the emergent system:

- Management with minimal design-based modelling of a system, which potentially frequently changes in both structure and behaviour. Observation may therefore need to enact observation techniques in response to system behavioural or structural change indicators.
- Calculation of future states is made difficult, and pre-emptive management actions (i.e. without historical track-record) are rendered unpredictable. Equally, the system's operation in a reactive-only manner may lead to constant unnecessary tweaking actions that upset emergent equilibrium. This requires a solution whereby the observer must be able to analyse, experiment and predict where necessary, leading to the explore/exploit dilemma.
- Unpredictable effects of external actions may cause the system to change state in an unpredictable manner with no observable cause.

Emergence is not confined to behaviour; an important type of complex emergence is that of self-organisation, in which the structural properties of a system are created and altered as the system is operating. The next subsection examines the management and observation concerns that arise regarding self-organisation and system topology.

2.2.4 Self-Organisation & Emergent Topologies

A key characteristic relating to the modelling and therefore observation of complex systems is closely linked to both scale and the emergence of new system properties. This issue is the structural organisation of the system.

Emergent structural organisation was noted by researchers who determined that in human social networks, despite relative organisation and regularity in local networks, the emergent topology of a "global" network demonstrated random characteristics – particularly small world tendencies [66]. The relevance of the wider topological characteristics will be discussed further in Chapter 3; the remainder of this section will concentrate on structural emergence within software systems.

Returning to the discussion of large datasets and monolithic systems from Section 2.2.1, the datasets or host organisation is largely determined at design time. Although organisations may undergo minor adaptations at runtime, these occur within well-defined boundaries (such as the expansion of data tree structures) and relate to the system's management of the scale of its dataset, operational host pooling and collection or both [52, 67]. However, with complex systems, particularly referring to those as described in the lead up to this section, such as loosely-designed service-based, system-of-system architectures, the organisation of the entire system is liable to evolve at runtime. Additionally, given simple constructional guides, organisations may emerge and re-emerge at runtime. Early related research work looked to address the problem of software evolution and its effect on formally-specified architecture; firstly by high-level description language approaches, including Architecture Description Languages (ADLs) such as *Darwin* [68] that define system structure in terms of component provisions and requirements, and allow the expression of runtime modification. A later development looked to develop the concept of connector-based ADL and associate it with a runtime-modifiable model. This model permits simple changes and model constraints preventing the system's structure entering unwanted states. Finally, compound "transactional" changes allow the system to pass through unwanted states en-route to a valid state [69].

The use of Architecture-level runtime-accessible and modifiable description languages (or related ideas and techniques) provides a useful insight into the manner in which runtime models can be maintained at Implementation-level, along with some of the concerns surrounding a structural system model. However, while these approaches separate the architectural and behavioural concerns, the latter integrates constraints as *purely* architectural concerns, thus limiting the potential for architectural/behavioural crossover in a large system-of-system architecture. Equally, while provision was made for code-level implementation detail, this is generally constrained to a particular architecture-supporting software framework (e.g. [70]), and as such, programming models are necessarily framework-specific and target architectural concerns in isolation. Developments of this work have retained the architectural-only focus, whilst opening up the detail in which the architectural model's changes and evolutions are described [71].

The Architectural model-based approaches provide a useful method of describing the system's structure, reacting to, and to some degree controlling alterations within the structure. However, they do not examine the possibility that the structure of the system's architecture may itself become too complex to manage; instead relying on the architectural abstraction and constraints being sufficient. As such, for observation model purposes, it is considered useful to further consider aspects of the notion of viewpoint-dependent complexity, and how that can be applied even at a structural system level; exploiting structural characteristics wherever possible. As such, the remainder of this chapter will further examine the modelling and management of *system* complexity in software. While established and recent software approaches to the features identified have been discussed throughout Section 2.2, the discussion centred on relevant *characteristics* of complexity. The next section will examine some software management and engineering approaches that look to manage complexity in software, along with their relevance to software observation.

2.3 Software Engineering and Complex Systems

The previous section gave an overview of the features associated with software system complexity along with brief notes on how those features may affect system observation and the required models. This section will provide a review of software engineering architectures, design approaches, and related fields that combine toward the design, maintenance, implementation and refactoring of complex software systems. The section is intended to flow from abstract to concrete; such that concepts and analysis techniques will appear first, leading on to design and implementation techniques towards the end. In brief, the subsections will examine the following approaches and fields of research and practice:

- Ultra-Large-Scale, Internet Scale and Systems of Systems [45, 72, 73]
- Organic Computing [74]
- Cognitive Immunity and Autonomic Computing [5, 10]
- Multi-Agent-System-based Complexity Management [22]
- Domain-specific approaches to complexity (e.g. [75])

2.3.1 Systems of Systems, Internet and Ultra-Large Scale

This section aims to give an overview of the system types in which this research is applicable, along with some examples of software approaches to the problem. In order to best situate this problem, it is important to look at *Monolithic* software. The terminology "monolith" has several connotations, even in software. In small-scale software, it can mean a very tightly coupled, many-featured software package, or can even make for negative commentary when describing software with poor cohesion and high coupling, indicating the software has no well defined design modularity [76]. However, in the domain of large-scale and complex systems, it is understood to have a secondary meaning. Monolithic software systems can still be large, complex entities that exhibit distributed processing and has many different data sets. However, monolithic complex software generally has less apparent evidence of the software's architecture within its implementation. Typically, it refers to software where component-based design is avoided entirely or *well abstracted* from runtime concerns. This does not specifically imply design with poor modularity, but software operating as a single apparent process and concealing architectural concerns from the implementation's operation [77]. While these monolithic large scale systems exert direct control over all aspects of their operation and as such present a global knowledge of system operation, they tend to have a relatively fragile architecture [78]; for example, there is no easily-specifiable mechanism of redundancy or an ability to exchange system components as they become overloaded or fail.

As such, the development of large-scale and complex software that may previously have become monolithic software has shifted to component or service-based architecture, whereby software is composed of a variety of components, often entirely independent [77] (e.g. [79]). There are two key advantages to this service or component-oriented software architecture; one is the ease of reusability and therefore the effects on development efficiency, while the second is the potential benefit of dynamic composition – components or services can potentially be re-sourced as required due to failure or new adaptations; allowing a truly flexible configuration [73]. In order to facilitate this design scenario, components must have well-defined responsibilities and a well-defined interface for communication with other components. In order to take advantage of the configuration flexibility, key components must have the capability to source their dependencies and make decisions on runtime reallocation. Utilised components are considered as black-box interfaces to processing functionality, a service, or an item of data, fitting component independence characteristics as discussed in Section 2.2.2; resulting in a *system of systems*. Along with the potential benefits of late or dynamic composition, there is the added complexity of self-organising and emergent properties; a system is composed of many different components, each with their own composition priorities and rules.

Current methods for dealing with this system-of-systems complexity include Federated Multi-Agent behaviour (see Section 2.3.4), in which system management and monitoring responsibility is located at appropriate subsystem "agents". Somewhat conversely, though with similar priorities; SE-based research has called for methods to monitor the need for system change [80], such that it can be engineered and controlled, rather than *entirely* autonomic and emergent.

Alongside the development of systems-of-systems design, two other significant system descriptions have emerged that relate to this work. Often, large and complex systems will exhibit characteristics of several of these definitions, so it is useful to set out a definition of some of the significant characteristics identified by these two related lines of research:

- Internet-Scale Systems (ISS) – are designated as classes of systems that can operate over networks such as the Internet, and those that are intended for deployment at a scale comparable with an Internet audience. As such, ISS-related research is concerned with issues particularly relating to the interoperability and the management of large quantities of component data – early ISS research work discusses the partitioning of Internet data that does not readily lend itself to such partitioning [45]. More recent work by active ISS authors has tended towards addressing the problem of formalising scalability in a variety of domains (e.g. [81]), and as such, research interest in managing very-large-scale systems has transferred to ULS:

- Ultra-Large-Scale Systems (ULS) – are described by Carnegie-Mellon's research team as very-large-scale software systems that will make use of resources at an Internet-scale, and will serve such large populations and diverse functionality that they cannot simply reach a natural life-cycle end, are discontinued and re-deployed; they *must* evolve [72]. The authors of related work identify a significant monitoring issue; systems will be so large that a complete specification will be impossible – therefore preventing runtime validation – so monitoring and management will take the form of assurance rather than assertions [82].

The next subsection will discuss related research and application in Organic Computing, examining some of the biologically-inspired methods and their place in management of large scale and the complexity associated with the many-tier architecture associated with systems-of-systems.

2.3.2 Organic Computing

Organic Computing (OC) describes software system development that aims to allow software to achieve a set of properties, many of which are in common with IBM's Autonomic Computing initiative, as briefly discussed in Section 1.2.2. AC will be discussed in greater detail in the next section. However, OC subtly differs from AC; concentrating specifically on biologically and organically-inspired solutions to these problems [74]. As such, and of particular relevance to this research, a significant focus is the study of self-organisation and emergence in systems [83]. One OC-based line of research [84] involves the definition of a generic observation architecture that uses a traditional sensor and actuation set of interfaces to represent goings on in the observed system. Observations are analysed on a time-series basis using a variety of appropriate techniques, and control feedback is decided on, altering the structure of the system if necessary and itself facilitating emergent change in the system – to optimise system characteristics such as performance or reliability. However, this work defers specific concerns regarding the magnitude of observation – in terms of observed units – to the System under Observation and Control (SuOC), which is intuitively expected to include the appropriate system data. The referenced work concludes that observational complexity within the SuOC could be managed by a series of agents (Section 2.3.4), or observations may be exhaustive and dealt with on a mining or machine learning basis.

While these represent potentially valid approaches to the problem of data-scale, the author reasons that there would be equal, if not greater value in developing an observation model that could intuitively select appropriate observation targets from an observed system on system-specific criteria to make best use of available resources. In order to manage some of the emergent aspects of complexity, the model should allow the entry and removal of targets based on system change; filtering out irrelevant data at the instrumentation level, rather than the processing stage. There are several other related approaches to this problem to be considered; they will be examined over the next few sections. The next subsection discusses the original inspiration work – the DARPA-cited Cognitive Immunity, along with focussing on the engineering aims and technical detail of the IBM Autonomic Computing programme.

2.3.3 Cognitive Immunity & Autonomic Computing

The topics of Cognitive Immunity and Autonomic Computing are briefly introduced in Section 1.2.2, and their root concepts - if not necessarily the proposed approaches, have influenced the aims and motivation of this research enormously. This section aims to discuss these two schools of thought and how the author feels the overlap of the two is useful, and how they have been interpreted in order to guide this research.

The DARPA Cognitive Immunity notion, as discussed in Section 1.2.2, fell primarily within the umbrella of the Self Healing Systems from the Autonomic Computing initiative. Research in Self-Managing Systems in general is multi-disciplinary; it explores other areas of study that have not traditionally been the domain of the computer scientist, with fields ranging from the study of behaviour of animals and people in social science and natural systems [37], and the study of antibody and antigen behaviour within medicine [85], along with other biologically-inspired areas, as per Organic Computing (previous section). It has also resulted in several different types of research project and resulting application. Driving factors have been found particularly in military [5], state administration [86], and several areas of industry and state have recognised the potential value of a self-managing system. Therefore, the remainder of this section will look in more detail at some of the other aims of Autonomic Computing. It will identify the outstanding research problems and how this research relates to IBM's existing work. As introduced in Section 1.2.2, Autonomic Computing started as an IBM initiative intended to investigate software that could autonomously self-regulate, in much the same way as the central nervous system of mammals. A significant aim of self-management in Autonomic Computing is to reduce the complexity overhead associated with the set up and configuration of complex and highly distributed systems.

The original IBM Autonomic Computing proposal includes four main desirable characteristics of an autonomic software system: Self-Configuration, Self-Healing, Self-Optimising and Self-Protecting [10]. IBM's proposals include an "Autonomic Manager" and "Managed Element" architecture, which defers autonomic-type responsibilities for the Managed Element (and its users) to the Manager. While this provides a wrapper approach in which to integrate older subsystems into a fully autonomic system, the Autonomic Computing initiative does not provide a complete engineering solution and programming model to developers wishing to create autonomic software systems out of legacy software, and it is perhaps unfair to expect it to do so. It is instead a collection of research, open and proprietary technologies that can help to facilitate the development of software within the abstract AC architecture.

One such research area given high priority by IBM [87] is the study of interoperability. Interoperability is a key concern with any system-of-system architecture; facilitation of the co-operation of distinct system components in the *autonomic element* model. It is also referred to in terms of the technical challenges involved in its implementation; standardisation of element log-files, interpretation of logs, and models to allow autonomic elements to expose their behaviour, workload and structure – along with mapping abstractions to the individual component settings.

However, it is important to differentiate between IBM's research vision and current technical position on the matter. At the time of writing (Summer 2009), the IBM Autonomic Computing Toolkit (current version (3) October 2006) consists of the following key elements [88]:

- Common Base Event (CBE) definition – a (template for an) event definition comprised of the reporting component, the affected component, and the new situation. This is intended to describe system changes that may occur, which would affect system operation. This structure and its support effectively forms the basis of IBM's method to resolve the issue of reporting and logging interoperability [89].
- CBE support – consists of log-file conversion routines for data extraction from legacy logs, along with tracing support - plug-ins for a small collection of Java IDEs assist the inspection of CBE and log-file-based system event data.
- Autonomic Management Engine (AME) specification – the system's AME is the environment in which model decision algorithms are executed. CBEs are transmitted from managed elements to the engine (manager) via a *Touchpoint* interface, responsible for managing RMI between remote hosts. The IBM toolkit contains a reference implementation AME, called TAME, which is capable of processing basic resource model reasoning. AC resource models define the way in which the AME attaches to the resource's CBE event model, the analysis that should occur and "autonomic" responses [90].

Relating the toolkit to the earlier description, the Autonomic Computing Manager is the domain-specific code that utilises the AME functionality to attach to system units and deliberate and react accordingly. Managers (AMEs in the toolkit) are attached to a number of managed elements via their *touchpoint* interface. Scaling is architecturally managed as Managers are a subtype of Managed Element; therefore a Manager can manage a set of Managers. However, the hierarchical arrangement must be determined according to domain-specific design criteria. Equally, it is not clear how this model would adapt to emergent system structures as discussed in Section 2.2.4.

The IBM architectural model (thus far) has significant strengths in terms of interoperability, integration of existing systems, along with new research development (e.g. [91]). However, the toolkit is in essence, with the exception of the XML-specified CBEs (an IBM-specific implementation of the OASIS Web Services Distributed Management (WSDM) Web Event Format), a proprietary framework with loose architecture, rather than an open programming model.

A well-defined and partially-implemented framework with proprietary database and server support reduces the development workload related to AC integration within a business system; in common with any large reuse of 3rd party closed-source restrictive-licence software, there are certain undesirable issues:

- Licensing costs – IBM indicate that in order to develop *for-release* software, licensing must be obtained for which a charge may apply.
- Level of flexibility – Adopting a third-party framework can introduce difficulties in flexibility. If the third-party framework is written to be used in a variety of different ways, adopting it can be cumbersome and future behaviour changes in an API can introduce problems in upgrades. Alternatively, if the framework was created with specific constraints on functionality, extending this functionality on a closed source base can prove difficult, if not impossible.

In summary, the IBM AC Model is made up of Managers (toolkit AMEs) that are responsible for a set of Managed Resources, and intercommunication is facilitated by CBEs. Managers are effectively Observers, Deliberators, and include provision for control feedback. The IBM AC Toolkit provides a comprehensive set of methods for processing log-files in order to generate CBE-type descriptions from legacy applications; however, is licensed in its current form only for testing and evaluation purposes. Therefore, if a pattern design, or open source implementation framework were to be developed that can achieve significant aspects of Self-Management as present in the IBM model, along with better addressing the issue of scaling and evolution; this would represent a clear step forward in terms of complex software observation and management. The next subsection will examine a programming methodology that appears better suited to the problems of scaling and evolution in system structure.

2.3.4 The Multi-Agent-System Approach

The term *multi-agent systems* refers to a software programming methodology whereby the software is designed to operate via a set of *software (intelligent) agents* [22]. Without delving into a full explanation of agent-based software design, this section will give a brief overview of the key points, concentrating particularly on the potential relevance to complex system modelling, management and observation.

Agents operate as independent software components that communicate, negotiate and co-operate where appropriate with other agents via a common language, such as the FIPA Agent Communication Language (ACL) [92]. A popular model for *Intelligent* Software Agents is the Belief-Desire-Intention (BDI) approach [93], along with various extensions [94, 95], in which the following design paradigms are adopted:

- The local state of the environment – as observed by the agent (e.g. state of managed components) – is represented by a series of statements, known as the agent's *beliefs*.
- The *required* local state – is represented by another series of logical statements, known as the agent's *goals*.
- The agent's *intentions* are the internally-deliberated set of actions – the plan – that will bring about the agent's *goals*, given the current *beliefs*.

Multi-agent-based systems are generally considered appropriate for the design of complex software/systems-of-systems management for the following reasons:

- The methodology is a decentralised and distributed architecture, and importantly, employs a bottom-up design strategy, thus allowing appropriate agent organisations to emerge for a particular problem or system state [96].
- Individual software agents should be designed as autonomous – that is, they can operate with *minimal* direction, and are expected to adapt – according to observation of their local environment in order to develop *plans* to correctly achieve their *goals*.
- Interoperability between agent units is "guaranteed" providing agents all subscribe to the same communication specification (e.g. FIPA's ACL), although remembering agents represent autonomic units, they are free to refuse to perform requested actions; an agent communication is not an instruction.

Inter-agent control is usually set up through advert and contract messages; thus permitting agents to request and provide certain functionality.

- Increased scaling is generally managed in the same way as the hierarchical manager model in IBM's AC model. Agents responsible for large parts of the system may manage their responsibilities via delegated control of several other agents, each managing considerably smaller subsystems.

In theory, therefore, intelligent multi-agent system design provides a method by which complex and large-scale systems can be managed and partially modelled and observed in a fashion similar to that in which their own organisation forms and evolves. Required global control can be propagated throughout the system via messaging, or can be specified as global goals to which all agents subscribe. While the model is naturally distributed, self-managing and tailored towards scalable systems, it is not without its disadvantages as an underlying model for observation. The intended flexible deployment structure well suits the concept of a hierarchical observer set monitoring evolving large-scale systems, and the notion of messaging also fits the traditional event exchange of information in observers. As such, this research work will look to take a similar stance on division of observational load, along with a flexible structural organisation.

However, the agent model does not set out to specify a generic method for distribution of agent groups and delegation of responsibility between them. This adaptive behaviour instead relies on either appropriate delegation specification in each agent unit, or automatic emergent organisation by appropriate interaction rules within the agent "colony". While this may prove adequate while the system is behaving in a manner anticipated at design time, this does not extend to the provision of new organisational features; notwithstanding the deployment of new agent types.

In summary, the author considers the agent model has much to offer large scale and complex software system observation and modelling. IBM's CBE model illustrates that observed components can be wrapped into a standard event-generation form; an abstraction pattern equally applicable to agent models. However, much work is to be done in terms of specification of the self-organisation and management techniques that allow the controlling model of observation delegation to adapt yet behave predictably, within relevant constraints in a variety of situations. The next subsection will provide an overview of domain-specific approaches for management of scale and complexity to determine lessons that can be learned and potential generalisations.

2.3.5 Domain-specific Monitoring and Management in Complex Systems

The previous subsections gave an overview of some current significant architectural thinking concerning the application of software engineering to self-managing systems. Wherever applicable, the role of the observer was highlighted to show the separation or indeed integration of observation within general system concerns and elements. This subsection will identify and discuss recent research by others, showing domain-specific examples of the adoption of models similar to those discussed, either partially or in their entirety. This section will include elements from others' detailed designs or implementations for various requirements of complex system or complex element monitoring and management, and draw conclusions on the implications for a complex, adaptive observation framework.

Controlled Self-Organisation via Observer/Controller Collaboration

This work cites its primary inspiration as Organic Computing and aims to apply generic observer/controller architecture to a simple problem. This is based on work already discussed briefly in Sections 2.2.3 [63] and 2.3.2 [74], which jointly suggest the notions of an Observer architecture in Organic Computing, and that of *controlled* self-organisation; harnessing the flexibility of self-organisation, yet bounding it such that systems can still be engineered. The work in question [75] explores how a series of cars in opposing directions can cross an intersection efficiently, concentrating on how a form of observation and co-operation can improve the situation vs. a simple sensing approach whereby cars operate entirely independently and selfishly. As such, the referenced paper aims to demonstrate the differences in localised collaboration vs. central, high-level control. It shows how in simple cases, localised collaboration is outperformed by an abstract view, while in high-complexity cases (which are simulated by a nondeterministic environment, with greater outside influence) the localised collaboration actually outperforms the centralised controller. The authors of the referenced paper conclude this is an indication that in a dynamic and complex system, observation-based control must be as dynamic as the system it is observing.

In certain cases, higher-level observational control will represent the best action, while in others a localised and potentially collaborative approach may suffice or even outperform the potential high-level response. The referenced paper demonstrated a convenience abstraction for centralised control – given the simulation environment; all the system elements were known and indeed well specified. As such, the author considers there is research value in determining a generic model that will facilitate an abstract-level controller that can adapt its lower levels to the system it is observing as it undergoes change.

Adaptive Monitoring via Reflective Proxy/Proxies

The adaptive monitoring work considered [97] is situated in the context of evolving software systems, and acknowledges that continuous monitoring is a necessity for dynamic systems, in which the software requirements cannot be entirely encapsulated in the design-based code. As such, the referenced paper introduces a novel need to instrument software elements for monitors that would be taken for granted in terms of assertion or simple hard-code in static systems, and presents the basis of a reflective framework to help support this requirement. The reflective technique makes use of a series of reflective proxies in Java [98] to allow both: monitoring of elements' behaviour that has not been deliberately exposed, and to allow the level of monitoring to vary according to either element-specific criteria, or even external influences. Given this work, the author considers that a globally-adaptive observer framework should make provision for specifying monitoring-levels (be it event-filtering, or proxy-based adjustment) to observed elements that can support this adjustment.

Existing ULS and Distributed Monitoring Systems

In order to clarify requirements for observer frameworks that monitor very-large-scale and complex systems, it is helpful to look at an existing widely-deployed approach to monitoring on an ultra-large-scale, and an attempt to further generalise this approach into a wider-usable approach. The discussed system is Ganglia [99], which is used primarily as a monitoring system in cluster and grid-type computing systems.

"Ganglia" uses multi-cast transmission (i.e. broadcast) techniques of monitored data to keep track of nodes appearance and disappearance within a cluster, and to distribute the data; assuming high bandwidth and availability of connections within a cluster. Several clusters are monitored through a tree-based aggregation of data from different cluster nodes; approaches which are both known to scale well and can carry and distribute the required data sufficiently. "Ganglia" fulfils several of the requirements of a complex monitoring system: it scales well and automatically handles the dynamism within the system – i.e. the arrival and departure of nodes. However, it is monitoring systems that are subtly different to the "systems-of-systems" anticipated; clusters are largely homogeneous, and have fairly well specified elements of "dynamism", rather than the previously-discussed *emergence.*

As such, later research work introduced a paper to promote the development of these and connected techniques to create a generic design for adaptive monitoring in ultra-large-scale systems [100]. This set out to outline requirements for an ULS monitoring design pattern; specifying concerns such as inter-element messaging, attachment and removal of sensors. The author recognises these are valid concerns, yet also that there is extensive research into concerns such as instrumentation and the distribution and aggregation of information. As such, this research aims to specify the observation model in terms of a generic framework that manages the scale and complexity of the underlying system, providing either a series of distributed observer units, or an adaptive architecture to provide a single high-level abstraction that connects via a series of hierarchical, adaptive layers to the element instrumentation.

2.4 Requirements: Large Scale and Complex Systems Observation Model

The chapter thus gave an overview of features present in complex software along with some current software engineering thinking and approaches towards the architecture, design and implementation of complex systems. Each section has focussed on features/approaches that affect, or can help with the management of complex systems from a monitoring and/or observation perspective.

What has emerged is a general consensus that as systems reach a certain level of scale and/or complexity, traditionally-established SE methods of exhaustive specification of behaviour and design (i.e. the monolith software approach) become impractical, if not impossible. As such, more appropriate methods of engineering software have emerged; composition-type approaches integrate the architectural concerns of scaling, distribution and redundancy in the design and therefore implementation methods, permitting dynamism and flexibility at runtime. However, traditional engineering methods for operational verification and validation via observation and monitoring are not appropriate as: the system is not likely to be *completely* specified in advance, and the complexity present in the system prohibits accurate, exhaustive observation.

This section will reiterate some of the identified significant challenges, and propose requirements for complex observers that can help manage the issues identified in this chapter; highlighting areas in which existing models are inspirational or deficient. The requirements, shown in Table 1, will describe the basic features the author reasons must be present in a generic observer programming model; as such, wherever possible, they are stated in a methodology-agnostic terminology.

Having set out the four basic requirements for the observation model, it is clear that there will be other implied requirements relating to the specification of an observer system; for instance, the observers must have a way to report their domain-specific findings, propagate them appropriately, and to examine the current observation model in order to direct feedback to the appropriate system element. Many of these observation-specific system requirements will be explored during the specification of the observation system; starting in Chapter 4.

However, this section has collected the complexity-specific requirements, based on likely characteristics and inspiration and shortcomings of related work.

	Requirement	Additional Detail
Scale	1. Observers must be capable of managing this scale by appropriate reduction or delegation of target selection - given any constraints under which the system may be operating.	Scale creates issues of computational complexity; suggesting a need for some form of automated and appropriate scope reduction in the observation target set. Federated agent behaviour shows that delegation of responsibility can prove a useful management technique, providing sufficient resources can be made available and the agents can each deal with the manner in which the large-scale is presented. Equally, Autonomic and Organic approaches show that exhaustive data monitoring could theoretically be processed by mining or other appropriately selective techniques at runtime.
Systems-of-Systems	2. Observers must be able to determine relevant observation targets by resolution of their design-time observation requirements *alongside* examination (or other characterisation) of their observed environment; design-time observation instruction may be incomplete.	Component Independence creates issues of incompatible sensing techniques, competing concerns, along with a potential for isolated component failure – information that must be propagated to a level that can make alternative plans. Systems of systems may have an architectural or design brief that does not adequately describe its runtime state to interested observers. This is highlighted in both Component/Service-based and Multi-agent systems – both of which facilitate dynamic composition; one via service contracts and the other via request and advert messages. As such, final composition details may only be known precisely at runtime. For example a component-based architectural design may give the required service connectors at design time, but the precise serving components may only be known at runtime when the system is assembled.
Emergence	3. Observers must be able to instrument the system in order to determine relevant configuration change and to update their observation targets appropriately if required; by re-characterisation (see Req. 2), or incremental change. 4. Given the dynamic nature of complex systems, the observation system must support either localised feedback or propagation of relevant observations to permit higher level control; providing an appropriate response to differing situational complexity.	Continuing the previous point, emergence of behaviour and structure complicates the manner in which systems can be modelled, and therefore observed. This means that new components may appear, existing components may be removed, along with existing components' roles – or observational importance – altering or being altered during system execution. Again, a software solution could lie in the use of federated agent-type behaviour, whereby components each manage their own domain, with hierarchical control providing levels of abstraction and propagation of appropriate control messages. Addition or removal of components and changing of roles is managed at the appropriate agent(s), and inter-agent communication keeps each "managed domain" updated. Equally, organic-type approaches look to identify newcomers or change in much the same way as biology may use danger signals [85]. While this seems a useful abstraction, biologically-inspired computing is not a one-size-fits-all approach. The model must still be applied at the code-level, and a suitable mechanism for describing the system's current state (i.e. "self") must be determined. Equally, work on controlled self-organisation shows that while localised-collaboration can outperform centralised control in situations exhibiting very high complexity, the reverse is true in situations with lower inherent complexity.

Table 1: Requirements for Observation Model

2.5 Summary

This chapter provided background information to allow the reader to familiarise themselves with the elements of complex systems that direct this research. The first half of the chapter gave an overview of the significant features in large scale and complex systems along with some relevant management techniques, while the second half of the chapter reviewed some established and state-of-the-art Software Engineering and Management techniques related to Large-Scale and Complex Software Systems, extending to those termed as Systems of Systems.

Of particular interest are the concepts of Cognitive Immunity and Self-Management in its many guises (this work retains the IBM Autonomic Computing definition), whereby software is expected to take an active role in its own configuration and dealing with external aspects outside the initial development scope of software. Additionally, the overview describes several Autonomic Computing-like methodologies outlining key requirements anticipated of complex software in the future.

It is apparent that while various techniques have been researched and practised to manage some of the issues here, there is not a generic and coherent approach to engineered and controlled observation of large-scale systems. As such, the final section investigated requirements for an observation system that can operate within these system types, as perceived by the author. The next chapter will examine a key requirement for observation; how to go about modelling the systems that need observation and formalising the relevant considerations.

Chapter 3 – Graph Theoretical Modelling

The previous chapters discussed motivations and challenges for this research, along with an outline of the state of the art relevant to software engineers, in terms of equipping large-scale software systems with Cognitive Immunity and Self-Management characteristics. This chapter argues the importance of mechanisms to facilitate an *adequate description* of a managed or observed system; taking into account that such a system is likely to be too large and complex to model using existing *exhaustive* modelling techniques[3].

Thus, this chapter looks at methods for modelling complex system, how they assist in overcoming the challenges reiterated above, along with those identified in Chapter 2, and how this can help to manage the system's operation.

3.1 Modelling and Abstraction

A continuing research theme regarding complexity management is the concept of a correctly-selected viewpoint. Translating this idea to software systems, an observer or monitoring viewpoint is represented by its target model of the system, its current scope in terms of modelled elements, along with messages or events it receives from its targets. This section looks at the concerns when designing a suitable abstract model, along with some of the options available. The aim of any abstract model is to create a simplified representation of the desired system, removing unnecessary detail. The model should adequately describe the important system characteristics, (potentially functional and structural), but can afford to discard individual component detail; particularly if it is irrelevant or of minor consequence at the point of study.

One way of interpreting a model is as an expression of a *design overlay*, where each element in the model maps to one or more elements of the real system. Elements in

[3] Additionally, a significant way in which the system's organisation may demonstrate its complexity is by firstly emerging into an initial state – potentially of some stability – and then evolving through different input/environmental factors. As such, the adopted modelling technique must be capable of both reducing the complexity and scale present in the real underlying model, while remaining flexible enough to adapt to system changes.

the underlying system are aggregated where necessary and represented by a new composite model element, or even omitted completely in order to reduce the size of the resulting model. If this abstraction approach is repeated and extended such that it is multilayered and hierarchical, each layer of the model can represent the real system in decreasing layers of granularity – each layer up a further abstraction.

For example, to take an entirely subjective model of a simple vehicle according to its functional composition:-

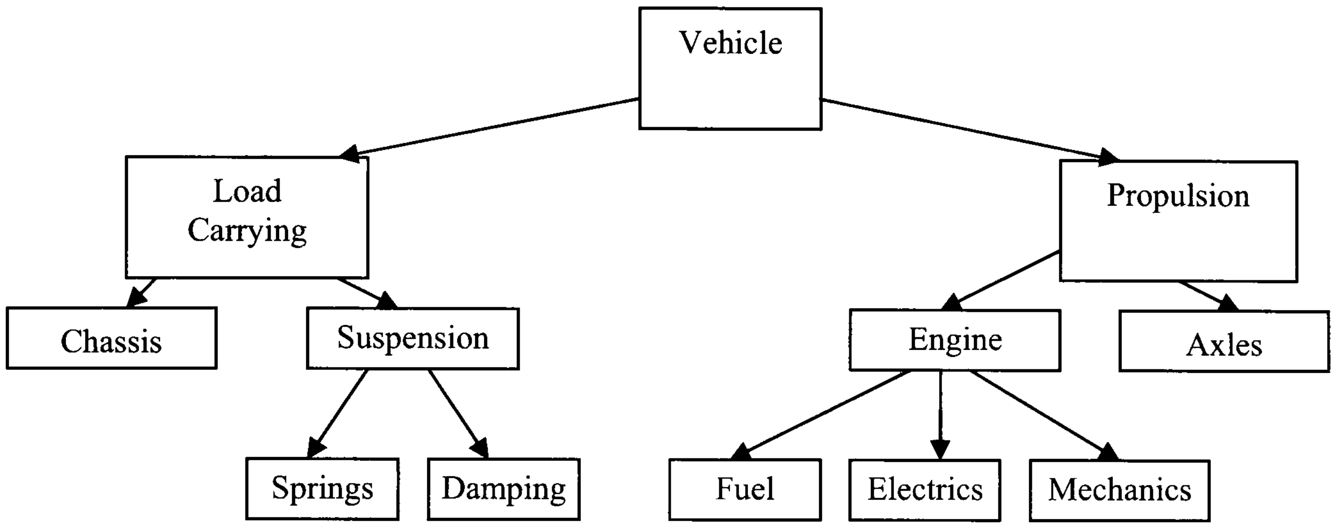


Figure 1: Example Decomposition of Vehicle Model

The decomposition, albeit incomplete, has been carried out entirely subjectively, based on elements of functionality most obvious to the designer. It was also carried out based on previous definitions of a well-known and design-static system. However, decomposition of even such a trivial example shows that there are modelling issues apparent. The first issue is that the result of decomposition has both *structural* and *functional* characteristics. As such, the use of this model is largely dependent on who interprets it; there are likely to be disagreements about both terminology and scope. For instance, a mechanical engineer working on final drive-train components may only consider it their domain to study the Axles sub-component. Should Springs (presently within Suspension) also be placed under the Axles element? If the designer wanted to model the vehicle's onboard computer and its engine control system, where would the Fuel Injection system reside in this model – under Fuel (as it *clearly* belongs), or within Electrics – is it not an electrical subsystem? The change in scope and use of the model has potentially changed the chosen categorisation method.

In addition, if this simple vehicle model was taken and used by another system engineer, could they make use of it easily? Hence, in order for any model to be useful to others, there needs to be a standardisation or explanation of terms – and in the case of all but the most trivial of models, a taxonomy; effectively a model of a model. Additionally, even in the case of relatively simple models, the requirements of the model user must be understood; abstraction must be undertaken with a relevant scope.

Top-down and Bottom-up models – As mentioned, this example model has been built with a complete knowledge of the system it is modelling. The model was created by starting with a full description of the system, and decomposing it into several functional elements – in this case, selected arbitrarily by the author! Each new element was then further decomposed into sub-elements, and so on; each more specialised and detailed. This approach has the advantage that the resulting model has a hierarchical nature, and therefore represents the system at a variety of levels of granularity. This allows for the model to provide greater detail at the lower levels, yet avoids overcomplicating the system overview.

However, as discussed in Chapter 2, the nature of complex systems means that creating a top-down model of a system is not an approach that lends itself to generalisation, due in part to the following factors:

- Requirement for complete system knowledge vs. availability of incomplete and changeable system information.
- Description and ontological representation of the system.
- The system representation must be static – even if only at the point it is modelled.

Alternatively, systems can be modelled by the study of individual components and their behaviour; eventually generalising functional or structural descriptions in an upwards direction - a bottom-up design. However, while bottom-up design brings with it some advantages, it is not without problems:

- Shortcomings of reductionist approach when applied to emergent systems (i.e. overall system behaviour may not be understood by studying minute components).
- Requirements, as per top-down design, for a static model – along with the required deliberation to (in this case) create an abstract model from specifics.

However, this work is not focused on building models of relatively well-understood and static (in terms of system functionality and organisation) systems. Systems under consideration demonstrate apparently random structures and behaviours, and are expected to alter these features during their operation. They are, as discussed above, large-scale, such that the model must act as a plan for attachment of observation. As such, despite complications; the value of creating a *representative model* should not be overlooked as impractical.

The next section will examine a method that can be used to model structures and relationships between system elements; permitting a mathematical approach to the problem of "viewpoint-dependent" complexity.

3.2 Use of Graph Theory

Graph Theory is a mathematical field of study for modelling relationships between different objects. It is particularly useful when modelling systems as an abstraction of their organisational graph – or topology. This section will look at some available graph theory techniques and their uses in managing and modelling large-scale systems. A brief overview of some important Graph Theory terminology is provided in this section to avoid the reader having to follow references [39, 101] to gain a basic understanding of the terms used:

- Edges, Arcs, and Links – these terms are all used to refer to the connections between the vertices in a graph. If an Edge is connects vertex A to B, but not B to A, it is said to be *directed.* Directed Edges are also known as Arcs. Edges that are not directed are termed *undirected.*
- Vertex, Vertices (pl.) or Nodes – are synonymous in terms of graph theory and represent the elements in a graph. The set of vertices that are directly connected to a vertex are known as its neighbours.
- Degree – the degree of a vertex is its number of neighbours.
- Hop Count – a hop refers to an intermediate vertex encountered on a route traversed between any pair of vertices. The hop count is therefore the number of intermediate vertices encountered on a specific path.

Graphs are often classified by their topology or structure, and some of these classifications are used to describe complex structures in this thesis. Therefore, a brief explanation of some common topological classes follows:

- Random: The simplest and most correct definition of a random graph is one that has been created through a random process [102]. However, throughout this thesis, the term *random graph* is used to indicate a graph where a given pair of vertices is connected according to a probability, referred to as p.
- Regular: A graph is regular if each vertex has an equal degree. Subtypes and strongly-regular graphs that place additional constraints on neighbourhoods, such as the *lattice* are often used to illustrate properties and transitions between phases of connectivity.
- Complete Graphs and Cliques: A graph or sub-graph is said to be complete or a clique, if every vertex is connected to every other vertex; i.e. for every pair of vertices there exists an edge between them. Disconnected graphs are the opposite; for every pair of vertices, no edge connects them.
- Small World: A graph is said to have small world properties if any two vertices are likely to have a short path between them, and that cliques occur throughout the network more frequently than they would in a randomly-constructed graph. The latter property is described as a high clustering coefficient [103]. More information on this class can be found in Section 3.3.2 and a detailed description of clustering co-efficient in Section 3.3.3.
- Scale-Free: Many graphs that occur in complex systems are said to have scale-free properties. Scale-Free graphs have a power law degree distribution [104], and share the short path characteristic with Small World graphs. A discussion of Scale-Free connectivity follows in Section 3.3; to which further description, including that regarding the power law is deferred.

3.2.1 Modelling Software with Graphs

In order to use graph theory as a modelling tool, it is first necessary to represent the system as a graph structure. At its simplest, this involves representing system elements as vertices, and their connections, be they physical connections, logical links (such as a network connection, a dependency or another architectural connector) as edges. While this creates a useful abstraction, it is not in itself a design solution – only a modelling approach. In order to be useful, a designer must determine how to apply a graph-type model to the system and how to interpret it; for example which system elements should be modelled as nodes and edges between nodes.

However, if a graph model *can* be applied, graph theory techniques can be used to help to model and manage the system's scale and complexity. Graph-based techniques have already been applied to several areas of software specification, particularly architecture design to provide formalisation and or connectivity description. A pertinent example uses a graph-theoretical grammar-based approach [105] to describe software architecture in terms of the components/agents and their connectors (interactions and dependencies). This allowed basic components to be specified in a recognisable formalism, and therefore proofs to be constructed to ensure the specified software design is compliant with its architecture. This work was extended by [106] to provide a constraint-based formalism by which transitions and basic dynamism in the architecture could be included from a proof-based perspective.

A recent review and consolidation of a variety of similar approaches towards architectural dynamism in graph-based grammars can be found in [107], which helps determine the applicability of the different proposals in a variety of domain problems. Additionally, software change and maintenance management – from a low-level design and coding perspective has been examined in [108], which examined its applicability to software refactoring. This considered the representation of programming code in graph-form, then the specification of valid code alongside that of refactoring transformations; thus allowing a programmer, potentially a system, to determine whether a given refactoring operation would produce valid transformations.

These approaches demonstrate that both graph-based formalisms and general graph theory is useful in describing software design, both at a high abstract level and even at a code-based object level. However, this work is concerned particularly with the application of graph theory as an adaptor to a software component map, and (rather than validation or proofs) its assessment, via graph theory techniques to determine characteristics of the software's organisation and/or behaviour.

3.2.2 Exploiting the structure

Therefore, in order to take the graph abstraction further - where the scale of the system is such that it cannot possibly be exhaustively modelled, any management system must operate on an abstract model that adequately describes the system. With large scale systems, a significant challenge is representing the large amounts of data with enough detail that the representative model is meaningful. In such cases, it can be beneficial to inspect the overall structure, in order to determine the limited set of elements that should be represented in the model.

This approach will be termed as "exploiting" the structure or topology; using prior (perhaps even domain-specific design) knowledge of noted structural characteristics in order to indicate subsets of the topology that merit particular investigation. However, given the complexity of system structure involved, the method cannot rely on explicitly-specified knowledge of the actual "runtime" structure encountered. As such, this implies that both the system model and the way in which the characteristics are specified must be either generic or adaptive such that they can be applied usefully in a variety of similar, though not identical situations.

In summary, a key element of this research is to simplify elements of complex systems using graph theory to identify and abstract characteristics within their complex structure. The rest of this chapter will look to illustrate the discussion above as related to exploiting structures in a complex system. While the remaining sections will concentrate on a single topological class; the intention of the derived approach described in later chapters is that it can be applied to systems other than those exhibiting scale-free connectivity.

3.3 Modelling Scale-Free Connectivity

As discussed in previous sections, challenges associated with modelling complex systems include the scale, and evolutionary/emergent nature of the system's structure. This section will examine a model that goes some way to explaining the topological emergence found in many complex and naturally occurring systems. The subsections look at some defining features in the scale-free model, and how measuring these can be used to help detect the topology's occurrence or emergence. Additionally, some of the measures can help identify key structural features that are of great importance when modelling a large and complex system in a simplified and compact manner. Firstly, an introduction to and discussion of existing research in scale-free systems:

Scale-free connectivity describes a particular type of graph structure that is found to occur in many complex systems. It is most easily identified by its power-law degree distribution [104], shown in Equation 1:

$$P(d) = cd^{-\lambda} \text{ for } d=m, \ldots\ldots M$$

Equation 1: Power Law Degree Distribution

In this equation, c is a normalisation factor, and d is selected from the range $m \ldots M$ (the range of possible node degrees). It indicates the probability of a particular degree occurring in a scale-free graph, subject to correction factors. The power law degree distribution indicates that there are many nodes with a relatively low degree, but crucially, a small number with a very high degree. These significant few are represented in what is termed the tail of the distribution, which is shown to the right of the graph; an example of which is shown Figure 2. As mentioned, scale-free connectivity is known to occur in various natural systems, and has been suggested as a method to describe the connectivity of the World Wide Web [57], according to Web Pages' links to one another.

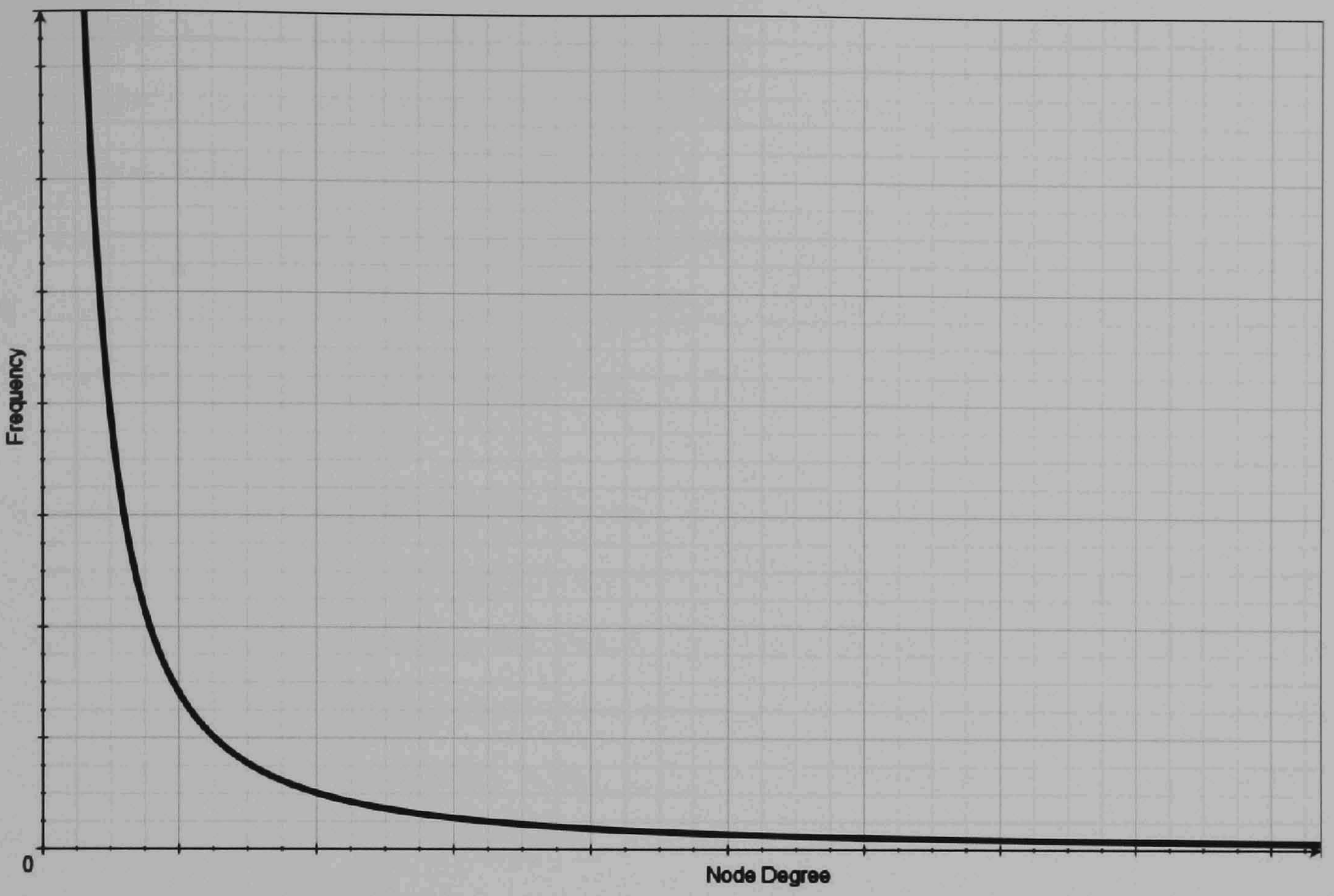


Figure 2: Example Power Law Degree Distribution

Therefore, considering the power-law properties of scale-free connectivity when applied to a network such as this, there are some important observations that have been made: As there are likely to be a large number of low-degree nodes, the random selection of a node is likely to return a low-degree node. Translating the graph to a representation of some critical system, the failure or removal of a random node is unlikely to significantly disturb the overall connectivity of the network. As such, when graphs exhibit scale-free connectivity, they are said to be resistant to random failure and attack. However, there are likely to be a very small number of high-degree nodes. Returning to the WWW example, consider pages such as search engines – that necessarily link to a large number of other pages. In this type of model, these high-degree nodes (termed "hubs") are considered to be particularly important. The removal or failure of a hub is likely to significantly affect or possibly even destroy the connectivity throughout a scale-free graph. Therefore, scale-free graphs are said to be susceptible to targeted attacks.

Returning to the need to model complex systems; considering a large and complex system that happens to exhibit scale-free connectivity, it provides a hint for modelling. If modelling the entire system is too costly, and the representation must be simplified; hub nodes are significant. Before selecting the adoption of scale-free connectivity as a modelling tool for complex systems, there are two significant issues to be considered:

- Do large and complex systems tend to exhibit scale-free connectivity?
- How to discern the occurrence of scale-free connectivity, and to identify important characteristics, including that of "hub" nodes?

In order to try to address the likelihood of complex and large systems exhibiting scale-free connectivity, it is worth looking at how scale-free graphs come into being. Much research work discusses the manner in which SF graphs are constructed or the way in which they emerge from network graphs of other topologies. A common introduction involves a network being created (or growing) by way of "preferential attachment" [109]. In this description, new nodes are connected to an existing node with a probability based on the existing node's degree. Put simply, high degree nodes are more likely to gain the new node as a neighbour than lower degree nodes. This *rich-get-richer* model leads to high-degree nodes being more attractive to new nodes, which in turn leads to the very highest degree nodes maintaining their status as *hubs*.

Given that SF connectivity networks are observed in a variety of naturally-occurring structures (particularly those arising from social-type interactions), it is logical to extend that some conclusions drawn on these other complex systems could potentially be applied to, and exploited within software systems. In fact, recent research has applied knowledge of the scale-free connectivity model to examine Java class libraries and theorise on the results of this analysis. Additionally, the authors of that work [110] produced a software tool to process a static *package* of Java classes and determine its Component Dependency Network – effectively a graph describing the reliance (indicated with the `import` keyword) between a given *package's* classes and those that reside in other packages. In almost all cases, the investigated libraries demonstrated scale-free tendencies. The research discussed the possibility that such graphs could demonstrate the quality and efficiency of a chosen software system's code re-use – providing sufficient sampling was undertaken.

The work was extended recently [111] and it discussed some of the software conclusions that arise from studying a system's code-dependency networks. Of particular interest to this work was the discussion on the application to distributed complex software systems. As well as outlining the difficulties in determining the component / code / service graph for such a system – as discussed in this work – the notion was put forward that complex software systems may follow rules determined via observation for other large complex systems. This is best described as a large scale-free connectivity graph gradually emerging from what starts out as a random – or even complete network of objects. The authors of the referenced work suggest that when considered hierarchically, a complex software system would exhibit the differing stages of scale free evolution. At the high-level abstract, it would appear to be a well-connected, potentially complete network, while at lower levels, components would demonstrate one or more of the scale-free connectivity characteristics.

The author of this thesis believes it is this theoretical approach – describing reasonably predictable system characteristics that emerge from fairly unpredictable systems – that will assist software engineers in developing overlay observation and control frameworks, of which this work is intended to provide software design guidance. As such, to begin tackling the problem of identifying and exploiting the scale-free topology in a *changing* system, methods – both effective and efficient – to measure scale-free properties must be examined and assessed. The following subsections look at a variety of graph theory measures, and related research in scale-free connectivity; assessing their applicability to a modelling approach. The measures will address one or both of the following points:

- Check for presence of scale-free connectivity
- Locate points of interest in a scale-free topology

3.3.1 Hub Connection Density

When considering scale-free connectivity, it is important to consider the role of "hubs" in the connectivity graph. Hubs are nodes that have a high degree of connectivity – i.e. they connect with many other nodes. A defining property of scale-free connectivity is the manner in which hub nodes tend to connect to other hub nodes. A simple indicator of the proportion of interconnected hubs can be calculated, based heavily on a similar approach from [104], by the following algorithm:

$$\frac{\sum_{(i,j)\in C} d_i d_j}{N}$$

Equation 2: Simplified Hub Connection Density Algorithm

N is the measured size of the system in number of nodes, d_i is the degree of node *i* and C is the set of connections between nodes *i* and *j*. In short - hub connection density may be used as an indicator of scale-free connectivity: high values are produced when hubs connect to other hubs. This hub-to-hub connection pattern can be considered as a system backbone, and as such, of special interest when wishing to exploit the structure of a system. The spinal cord arrangement can be considered a tool by which the "important" (considering hubs as important points) areas of a structure are highlighted. Highlighting important areas may provide a suitable method for structure simplification – even if just ignoring the "unimportant" structural elements.

Additionally, in [112] it was shown that a similar combined algorithm could be used to demonstrate phase transitions from a regular lattice, through small world and random networks, to a scale-free network. At the circular lattice stage, the measure is easily calculated as it is based on the maximum degree of the nodes – if the maximum node degree is *c*, the hub density will be c^3, irrespective of network size.

As neighbour nodes break the lattice structure by reconnecting to more distant nodes [113], the measure predictably decreases; eventually reaching that of a random network. If scale-free connectivity emerges, then the measure once again increases. While the effects of this transition are on one hand a statement of the obvious, the referenced paper demonstrates a potential use of this measure as a detection tool for an emerging scale-free connectivity – and therefore the possible emergence of a "complex" system structure.

3.3.2 Mean Shortest Path

Bearing in mind the high proportion of interconnected hubs in a scale-free topology; this property leads to the network having *small world* characteristics [103]. Small world graphs are characterised by any two nodes being only a few hops apart, despite a direct connection not existing between them. A *hop* in this instance is defined as having encountered an intermediate node when planning a route between the two chosen nodes. When determining whether a graph demonstrates small world characteristics, it is useful to be able to calculate the mean shortest path:

$$S = \frac{1}{N}\sum_{i}^{N} S_{i,j}$$

Equation 3: Mean Shortest Path

$S_{i,j}$ is the shortest path between the two nodes i and j, and appropriately, S indicates the Mean Shortest Path (MSP) for the network. Networks with small-world properties will have relatively low MSPs when compared to similarly-sized regular networks. However, random graphs also tend to have low MSP values – when compared to any given regular graph [101]. Calculating the many routes between many node-pairs to determine the shortest path may very well be computationally expensive. As such, while a useful measure of small-world tendencies, there is limited value in the structural graph's MSP as both a reliable and timely indicator of scale-free connectivity.

3.3.3 Clustering Coefficient

Watts and Strogatz produced another algorithm [113] to measure the small world properties of a particular graph. This measure is the Clustering Coefficient and is shown below:

$$C = \frac{1}{N}\sum_{i}^{N} C_i \quad \text{where} \quad C_i = \frac{2|\{e_{jk}\}|}{d_i(d_i - 1)}$$

Equation 4: Clustering Coefficient Measure

As can be seen, the Clustering Coefficient for the graph is a mean measure of the coefficient for each node. C_i is the coefficient for a given node, *i*, and is made up based on the number of edges of the "i" node's neighbours:

- An edge between nodes *j* and *k* (taken to be the node neighbours) is denoted by e_{jk}
- The degree of node *i* is denoted by d_i

In summary, the measure is a proportion of connections within a node's neighbourhood (directly connected nodes) from the number of *potential* connections. The Clustering Coefficient is effectively a measure of how well a node's neighbourhood is interconnected. Watts and Strogatz found (as discussed earlier) both random and small-world graphs demonstrated low MSPs; yet that small world graphs produced relatively high clustering coefficients.

3.3.4 Acquaintance Nomination

While the algorithms discussed in the previous sections provide various methods of calculating properties that can be used to identify scale-free graphs, these methods are not without issue. In particular, the Mean Shortest Path algorithm suffers from calculation complexity in as much as routing calculations are required to determine the shortest path between numbers of nodes in the graph. Additionally, the other measures are good indications of small world properties – in as much as most nodes are a small number of hops away from a randomly-chosen other node. However, as discussed in Section 3.2, the intended use of graph theory is to simplify the complex structures and to exploit any given available features.

In terms of scale-free connectivity, this means finding the high degree hub nodes which form the discussed topological backbone. In short, an indication is required that confirms 1) high degree "hub" nodes exist; and 2) there are relatively few of them.

Related work on the subject of Acquaintance Immunisation [43] provides an interesting lead. Acquaintance Immunisation involves selecting a small subset of nodes to "immunise" while ensuring good coverage of a graph. Good coverage is taken to mean that the selected nodes will be either directly or 1-hop connected to the vast majority of nodes. Interested readers can refer to the included reference [43] for full details and an assessment of the algorithm but in short, Cohen et al select the "immunised" nodes in the following manner:

1. For a given network size *n*, select a random set of nodes, (herein termed the "Interrogated nodes"), of size *pn*, where *p* is a probability between 0 and 1.
2. For each interrogated node, randomly select a connected neighbour, adding it to the set of immunised nodes.

In scale-free graph types, the immunised nodes are considered important; statistically, due to the graph's topology, they are likely to be the well-connected hub nodes. It is the author's opinion that this represents a valuable approach for selecting observation points as it requires little global knowledge and low computation cost in selecting the immunised set. Additionally, the author has extended this work to produce a reasonably simple metric, titled the "Acquaintance Nomination" measure. The Acquaintance Nomination measure indicates the suitability of a given graph to the hub abstraction/simplification method. It follows the same steps as Cohen et al's work, but the calculations are based on the size of the sets produced by the algorithm.

Examining the algorithm, Acquaintance Immunisation must produce a set of nodes, sized *a*, where *a* lies between 0 and *pn*. Near these extremes would *give an indication* of graphs that are either totally disconnected or cliques, respectively. Therefore, the number of immunised nodes can be normalised by the extremes, such that the measure lies between 0 and 1:

$$aMetric = \frac{|immunisedNodes|}{|network| * p}$$

Equation 5: Normalised Immunised Set Size

Thus, *aMetric* is the size of the immunised node set (as selected by Acquaintance Immunisation) divided by product of the network size and probability *p*.

Scale-free networks produce low (though greater than 0) *aMetric* values, as the immunised set is likely to contain relatively few nodes; the important "hubs" as described above. This is best explained by considering the implications of scale-free connectivity:

1. Scale-free connectivity suggests that the randomly selected nodes (the *interrogated set*) are likely to be low-degree nodes

2. The low-degree nodes are likely to be connected to a high-degree node (a hub), thus the acquaintances are likely to be hub nodes common to several of the low degree nodes. Therefore, the *immunised node* set will contain a small number of nodes.

However, random and regular networks tend to produce values nearer to 1 as the nominated set is likely to contain many different nodes. In these cases, the graph connectivity does not bias the selection of nodes towards a set of common "hub" nodes. Simulation results published in [114] validate the use of the proposed *aMetric* as a positive identification of scale-free networks making them suitable for this type of observation.

3.4 Summary

This chapter has set out to give a brief background in terms of modelling systems; particularly those concerned with the complexity and scale of a system. Referencing back to the challenges identified in Chapter 2, this chapter has examined existing research that can be used to simplify and abstract a large scale system's structural design and effectively address some of those challenges.

The examples discussed in this chapter centre on one type of structural organisation: the scale-free topology. This is because of the high occurrence of scale-free or similar topological characteristics in the organisation of many large-scale systems. Additionally, thanks to Acquaintance Immunisation and the author's metric development (Section 3.3.4), it provides a convenient way to scale-down a large system's structure while retaining the key points of the organisation.

However, it is worth reiterating that while the remainder of the work will concentrate on the use of these scale-free examples, the proposed techniques (and the framework) are not limited to this one topological example.

Chapter 4 – Large-Scale Observer Design Pattern

Building on the previous sections, this chapter discusses the conceptual and architectural concerns for the design of a large-scale, complex software observer[4]. This observation is required in order to facilitate the behaviour associated with Cognitive Immunity, and how to implement this support. The discussion will focus on three main points, namely:

- How to deal with scale, complexity and evolution – Previous sections (with particular reference to Chapter 3) have examined the challenges associated with complexity and methods of managing these issues. This section will examine the programmatic techniques developed during this research.
- How to identify the structure to be observed – As introduced in Chapter 2, dealing with and simplifying a complex system is very much a technique of finding the appropriate viewpoint or abstraction. In order to do that, the software observer would need to be able to recognise the structure it is dealing with. Previous sections (particularly Section 3.3) examined the types of structure known to emerge in complex systems, along with models that can generate such structures. This section will investigate how these structures can be recognised quickly, and how this recognition can be encoded within software.
- How to specify the behaviour model for a large-scale observer system – This builds on the previous sections to discuss the various elements of the observer model and how it can be built into software.

[4] Software engineering is a mature computer science field that has established practice for developing observer frameworks (example in [27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable OO Software*: Addison-Wesley, 1995.). There is much research in rule-based reasoning systems to provide cognitive function to developed software. As such, the major research concern is how to best specify, deploy, co-ordinate and manage observer components within a large-scale and complex system.

4.1 Applying the Observer Pattern

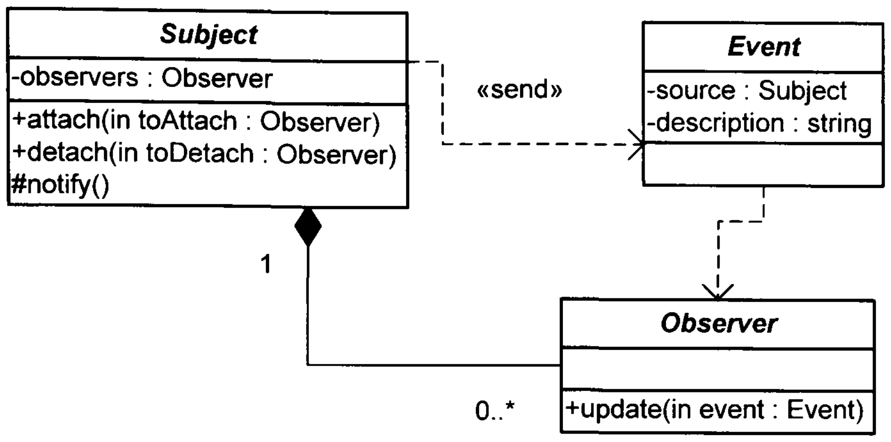


Figure 3: OO Observer Class Requirements (UML)

The Observer is a well-known software design pattern [27], the basics of which are illustrated in the UML class diagram, shown above in Figure 3; *Observers* register with *Subjects* to receive *Events* when they are changed.

The Observer pattern has some shortcomings that are apparent when applied to large-scale networks of autonomic software systems. They are outlined below along with a brief explanation of the author's interpretation of the problem. This is not a specific criticism of the pattern, rather a brief identification of areas in which the pattern either lacks relevant detail or is unsuited without greater detail to the problem:

- **Event Description** – event descriptions in the Observer pattern are typically hard-coded; object instances are parameterised with sufficient detail. However, in a dynamic system, this may not be sufficient: components may generate events that require new event types or extensible types, and observers may need to recognise novel events, or aggregate existing types to produce a new compound event.
- **Number of Subjects** – a typical implementation of the Observer pattern may use a single observer handling a whole set of particular user interface or data change events from many components or data items (subjects). The observer pattern facilitates the design and usage of multiple observers, each with specific responsibilities and scope. Additionally, in traditional observation implementations, the set of subjects is either fixed (e.g. user interface components), or the mechanism by which it changes is domain specific and well understood. However, with large-scale and complex systems, the organisational

scope and domain responsibilities are not necessarily clear at design time. Conversely, a single Observer component handling events from the entire system will prove computationally impractical, both in terms of event handling and maintenance of observer scope (i.e. the monitored subjects).

- **Despatch Mechanisms** – events are despatched in the Observer pattern by an interface-enforced method implementation and call. However, in a widely-distributed system-of-systems arrangement, this mechanism is likely to be unsuited. Some of the related concerns are present in certain small-scale problems that demonstrate limited complexity – such as in multi-threaded user interface complexity. One such concern that arrives with the multi-threading of event despatch is that of incorrectly-timed, or inaccurate and outdated events.
- **Event Feedback** – All but the most trivial Observers are active, rather than passive – they are responsible for some sort of action in response to certain events. However, the Observer model does not specify a mechanism by which response actions or environmental feedback should be provided.

4.2 Requirements for a "Complex" Observer Pattern

This section provides a concise summary of the requirements that were identified in the previous subsection. These are necessitated when applying the Observer pattern to complex or large-scale systems:

- **Reduction of Observer Complexity** – where applicable the system should operate the observers as a *relatively* simple overlay on the complex system. This may include varying the **scope** of an observer's interest as the system configuration changes or **filtering** or discarding particular event types as they cease to be relevant. Complexity reduction techniques may be domain specific, and certain domains may lend themselves more readily to this than others. The proposed method of reducing complexity in a generic and/or adaptable manner is discussed further in Section 4.3.
- **Externalisation of event descriptions** – the observer pattern may operate in systems that exhibit unique behaviour (such as runtime-emergent states). As such, the descriptions contained within events (which are traditionally hard-coded, with

limited runtime parameterisation) must be sufficiently flexible to describe events that only arise at runtime. This involves examining methods of externalising (in an interpretable form, such as in XML) both the event description, and information necessary to interpret the event description.

- **Suitability of event despatch mechanism** – as discussed previously, the complexity of systems creates additional concerns for event despatch and processing, outside the specification of the Observer pattern. Consideration must be given to the possibility that the scale of event processing may lead to situations where the current event being processed has arrived before events that logically precede it, or events that cancel or alter its impact.
- **Feedback mechanism** – as discussed in the previous section, the issue of how to provide feedback to the environment is complicated by both the scale and particularly the non-deterministic nature of the environment types under consideration. The author considers that the study of event consequences with regard to this problem is a research project in its own right. However, it is important that the complex observer model makes sufficient consideration of the interface by which control will be exerted over the environment, or by which event reporting can be consolidated for further reasoning and deferred feedback.

4.3 The Observer Conceptual Model

The previous section has identified the main concerns relevant to the extension of the observer pattern to support complex and large scale systems. A key requirement was that of **Reducing Observer Complexity**, which forms the main topic of this section.

As described in Section 2.2, complexity in software systems manifests itself in several ways. Not all of the features identified in those Sections are necessarily expected to be present in each "complex" software system; however two features have significant implications at a design level, and as such, it is logical to examine these first.

Section 2.2.1 described how large-scale systems create modelling problems that can be overcome in a number of different ways. Techniques for modelling large-scale systems each have their advantages and disadvantages; while a given technique may be perfectly suited to a vast number of situations, it may be completely inappropriate for a specific type of system. In order that a useful, reusable engineering framework may be created, it must be able to cope with these differing scenarios, and flexible enough to make appropriate use of the different techniques available to it. Continuing the theme of this research; when dealing with large and complex systems, one of the key research aims is the **exploitation of** system structure such that the required observation is simple – relative to the original. This aims to reduce the computational cost of observation, and the key software components must effectively address:

- Structural identification
- Selection or creation of a suitable observation overlay
- Deployment of the overlay on the system.

Sections 3.2 and 3.3 gave some examples of how graph theory and statistical techniques can be used to mathematically simplify a scale-free structure in order to reduce the number of structural elements that need to be managed – without losing important system parts. However, the developed approach should not be tied to the scale-free example; a large structure of any type should be effectively reducible to a set of specifiable metrics and key exploits.

By selecting the right set of measures or metrics, a topology description (a structural type) can be encapsulated in a type signature. This signature can be used on first deployment and during model changes to help determine which modelling strategies are applicable to the structure being observed. Once the observation strategy has been selected or created based on information from the previous steps, it must be appropriately deployed. The basic stages in the approach are shown in Figure 4:

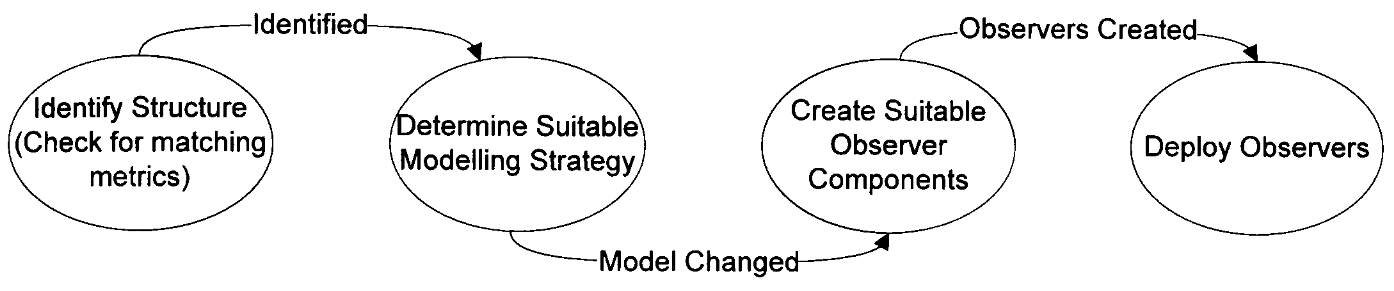

Identified
Observers Created
Identify Structure (Check for matching metrics)
Determine Suitable Modelling Strategy
Create Suitable Observer Components
Deploy Observers
Model Changed

Figure 4: Overview of Observation Processes

The following sections will examine the basic concepts involved in each of these stages, some of the sub-processes omitted in this simplification, and limitations associated with the linear nature of this process.

4.3.1 Identifying the observed structure

Signature: the identification process makes use of the aforementioned signature methodology. In this framework, a *signature's* purpose is to specify one or more features of a system's structure, and allows a system to check whether a particular collection of modelled system elements have these features. As such, a basic signature would indicate, when queried, via a Boolean return value whether the specified features are present. The internal workings of a signature are a matter for individual software developers; though it is suggested that features are identified by a combination of simple metric or algorithmic results. Further discussion surrounding the design and implementation of a structural signature specifying Scale-Free characteristics can be found in Section 5.2.

This signature methodology forms the basis of the identification component within the observation framework proposed and evaluated in this research – as it will enable the specification of key structural characteristics that can be exploited by an observer.

In [115], the author demonstrated a basic implementation of a signature-based topology identification and protection system. Structural change along with a small pool of signatures were put to use as a "trigger" for a simple scheme of system protection; when the system detected (by signature matching) that it was operating on a certain topology, it would deploy a predetermined type of protection strategy known to be effective on that topology type. A basic illustration of the relevant processes and data requirements are shown in Figure 5.

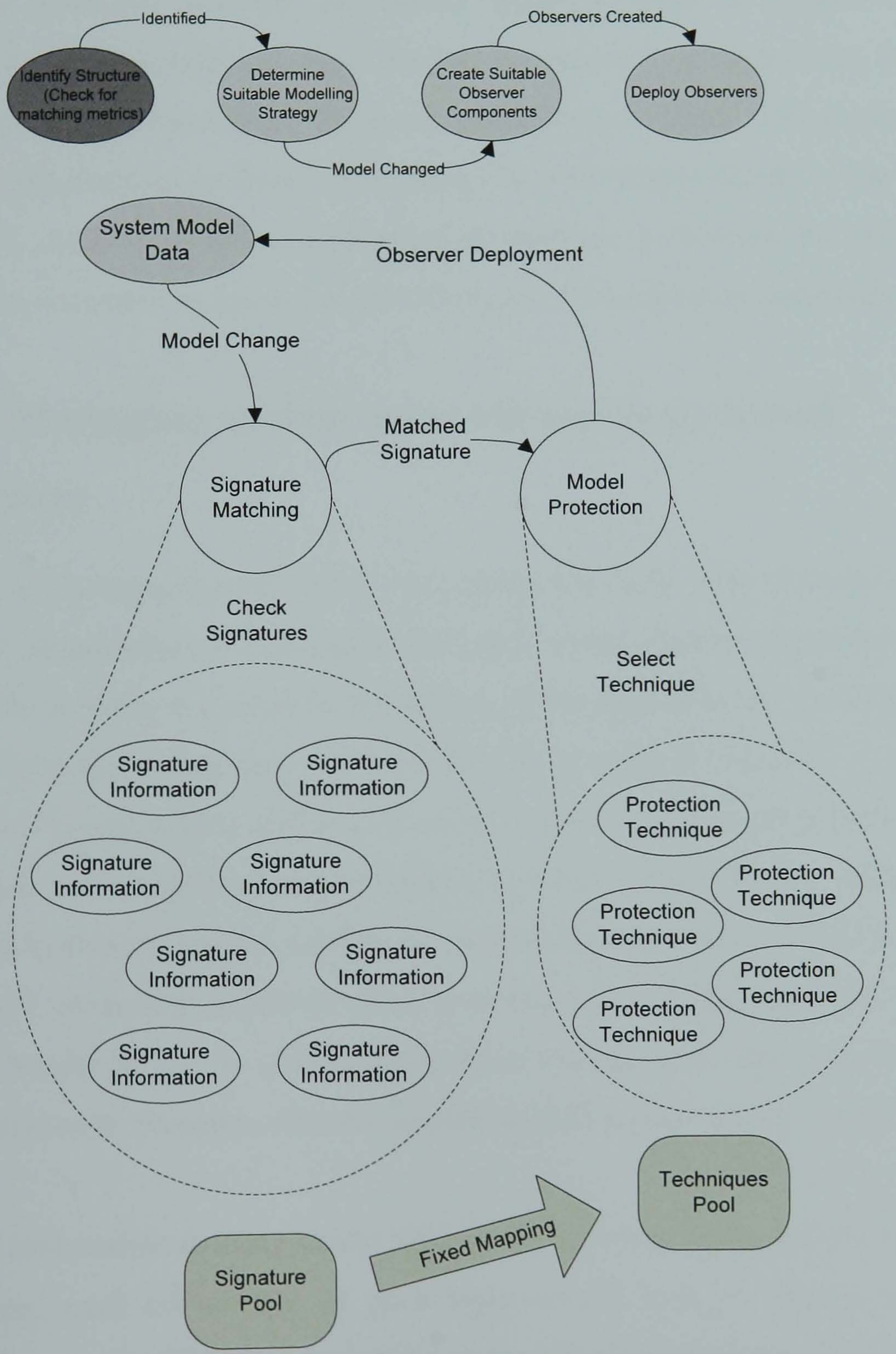


Figure 5: Signature-Selected Observation Techniques

The protection strategy in the referenced paper takes the form of an observation overlay, specifying where observation resources should be concentrated within the system for best effect, according to the selected protection technique. The limited experiment made the assumption that signatures would specify only topological characteristics that could be measured using simple graph theory metrics.

A signature matching was attempted whenever the underlying system structure changed, requiring exhaustive deployment of simple monitors at each system element. The simple monitors notified the parent observer when a system component connected to, or disconnected from another component. A connection represented communication or dependency between components. More information on the measures employed can be found in Section 3.3; with further details in the associated paper [115]. Additionally, the evaluation of methods in Section 9.2 develops this simulated environment to assess the effectiveness of the signature approach.

4.3.2 Managing system scale via topology-based architecture

While the initial experimental work was concerned only with determining certain topological characteristics associated with the whole system organisation; it was proposed that in order to extend the usefulness of the approach, the hierarchical nature of the observation subsystem could be further developed. Equally, a hierarchical breakdown of responsibility and area of effect is attractive as it can provide a solution to the issues of scalability, providing that overlay-level observer units could be constrained to monitor only a certain set of structural elements. While a single root overlay-level component would in effect hold responsibility for the entire observation system, it would do so via child overlay-level components, each responsible for a subset of structural elements, with this model applied recursively as required.

Any given observation overlay model should take its own structure from the system it is modelling; each component of each layer would look to exploit the relevant topological features of the observed (sub) system; itself a key aim of this research. As discussed in the earlier vehicle example of Section 3.1, abstraction via the hierarchical *systems-of-systems* approach can be a useful simplification tool. The level of abstraction applied at the top of the hierarchy allows an overview control of many elements, while the level of detail at the bottom of the hierarchy should permit precise control of a single element. Levels in between provide the relevant translation stages between abstract concepts – such as goals – and the technical detail required for individual components.

The potential benefits of the hierarchical organisation almost make it too tempting to assume that this approach can simply be applied to the previously-discussed experimental work without modification. However, without regurgitating all of the referenced section, it is worth revisiting the distinction between bottom-up and top-down system design. So far, it has been convenient for modelling purposes to assume that the newly-proposed hierarchical overlay creates a top-down system overlay, neatly encompassing related and relevant system structures in equally-sized subsystems. In this case, organising observation hierarchically would be a simple case of adopting a similar hierarchical structure to the system's own model.

However, previous chapters have gone to some length to discuss the features and challenges that make complex systems difficult to model. One such challenge is the tendency for structural features to *emerge* from a previously-unstructured system arrangement. Additionally, complex software construction through service and component composition can lead to a situation where traditional software modelling techniques do not give an accurate breakdown of the entire system.

Difficulties aside, it is apparent that extending the previously-discussed structural identification, such that the observed systems' models of organisation can be partitioned and further broken down into hierarchically-organised subsystems, is a useful addition. This functionality permits different observation components to deal with the individual parts of the system in a suitably-adapted manner, and removes an centralised control element from the observation overlay's architecture. However, supporting this approach brings several challenges of its own:

- How to partition the observed system's organisation model – which metrics can be used to determine areas of different topological identification? How can these be used to set the boundaries for different observers' scopes of interest?

- Computational intensity of the partitioning (and sub-partitioning) operations. The signature/trigger approach discussed above is able to operate on large systems because it is intended to require minimal computation. It is therefore aimed to be flexible as it can be repeated as required. As it is likely that constructing a suitably-partitioned overlay will be more computationally intensive, this will limit its flexibility when the underlying system undergoes change.
- Continual maintenance of hierarchical structure – in order to keep the observer scope correctly adapted, it will be necessary to maintain the arrangement of observers such that they adequately reflect the changing system structure. However, to avoid the problems of centralised control and excessive loading on a single group of units, the changing system structure and resulting change in observer deployment must be handled in a decentralised manner.

However, while the work referenced and described in the previous sections demonstrated the value of a signature-based approach, it still relies on a hard-coded link between a given signature and the relevant observation outcome. It is considered that this limitation is a significant obstacle to the partitioning discussed above, in addition to the flexibility of the entire approach. The following section will examine how this direct link can be removed such that a signature match simply informs the modelling and observation processes.

4.3.3 Determining a suitable modelling strategy

Technique: the eventual response to a particular signature match (which indicates a detected characteristic) is the creation or implementation of a suitable modelling strategy for the observers. This framework defines a *Technique* as a model-generating component, which is responsible for creating a new, reduced-complexity system model. As such, a basic Technique should select important system elements; importance defined by the type of technique. Further discussion along with an applied example of an Acquaintance Immunisation Technique for model generation can be found in Sections 5.3 and 5.3.1, respectively.

Once a particular structural type has been identified within a system via a set of metrics or characteristics, the next step is to determine the most suitable way to model this structure. The model should encompass the important areas of the structure while reducing complexity sufficiently to allow observer(s) to be deployed at key points. It is proposed that this would be achieved by the production of a cut-down model that contains only those structural elements that are deemed important. At the very least, that would require the signature data and access to the full system model.

In the initial simulation (see Section 9.2 for related simulation work), the decision regarding the production of the simplified model was entirely hard-coded based on the matched signature. For example, if a scale-free signature matched using an Acquaintance Nomination metric, then the Acquaintance Immunisation model strategy was the hard-coded response. If the network was shown to match the Random metric, then the Random monitoring strategy was deployed.

While the approach in this simulation in its own right was not without merit, it was certainly limiting. The limiting effects can be reduced by altering the way in which signatures are used. Instead of a signature acting as a simple condition for a triggered action, it provides a description of a certain set of characteristics, which may include vulnerabilities, exploits, or modelling techniques that are applicable to these characteristics. A signature match is then used *as part of* the simplified model creation process, rather than the final say. Model creation is deferred to another component, which reasons on the quality of match and whether any of the supplied techniques may produce a benefit, within the signature's specified parameters. If no signature matches satisfactorily, the system model could be referred for further processing – such as attempted partitioning, as mentioned in the previous section. Deferring the model generation in this way would also allow details of the resulting model to be affected by policies outside of an *identify-and-exploit* process.

A brief simplification of the process thus far is shown in Figure 6 and Figure 7, which show the basic component responsibilities, and a simple process overview respectively.

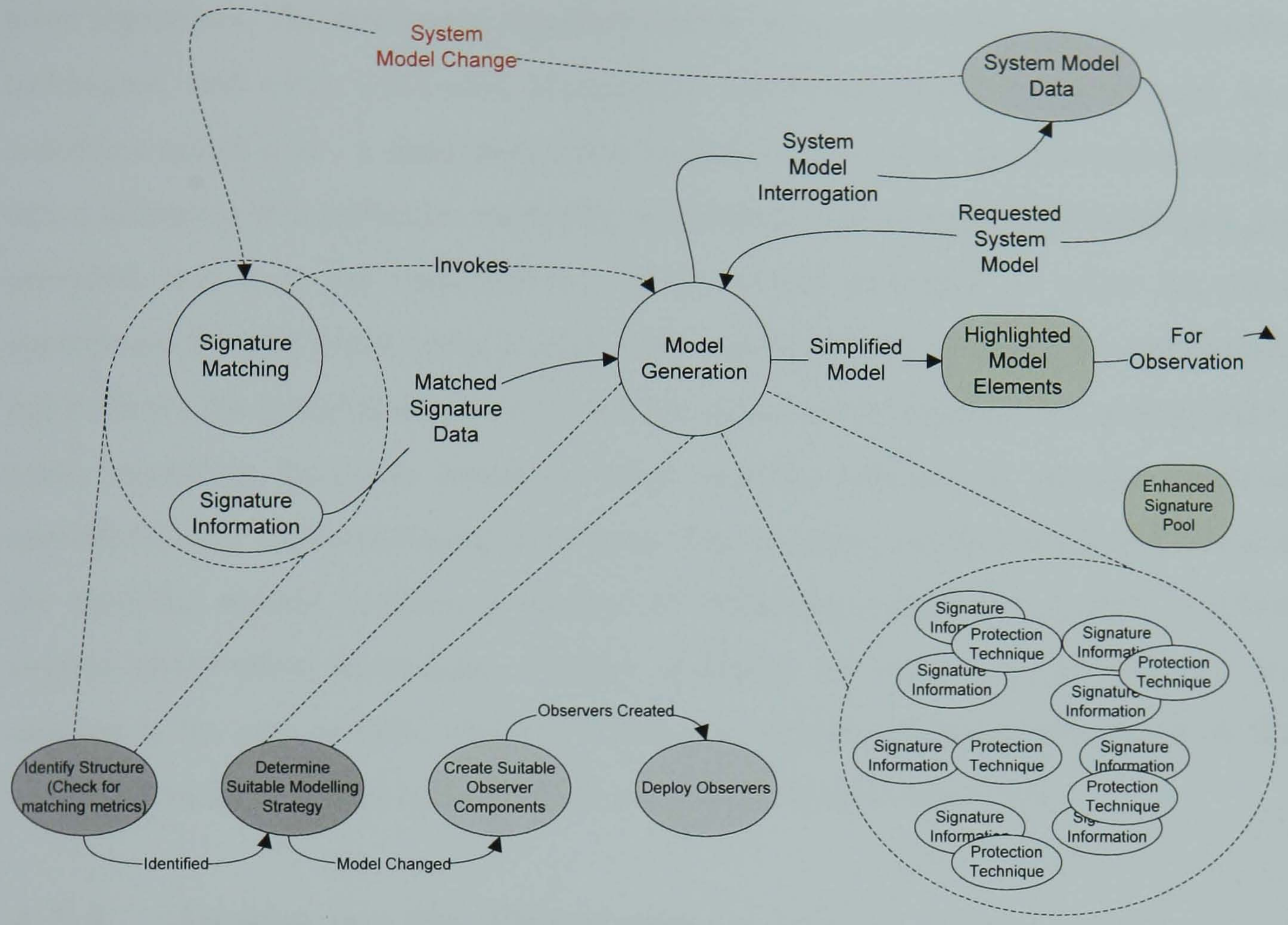


Figure 6: Required Model Generation Components

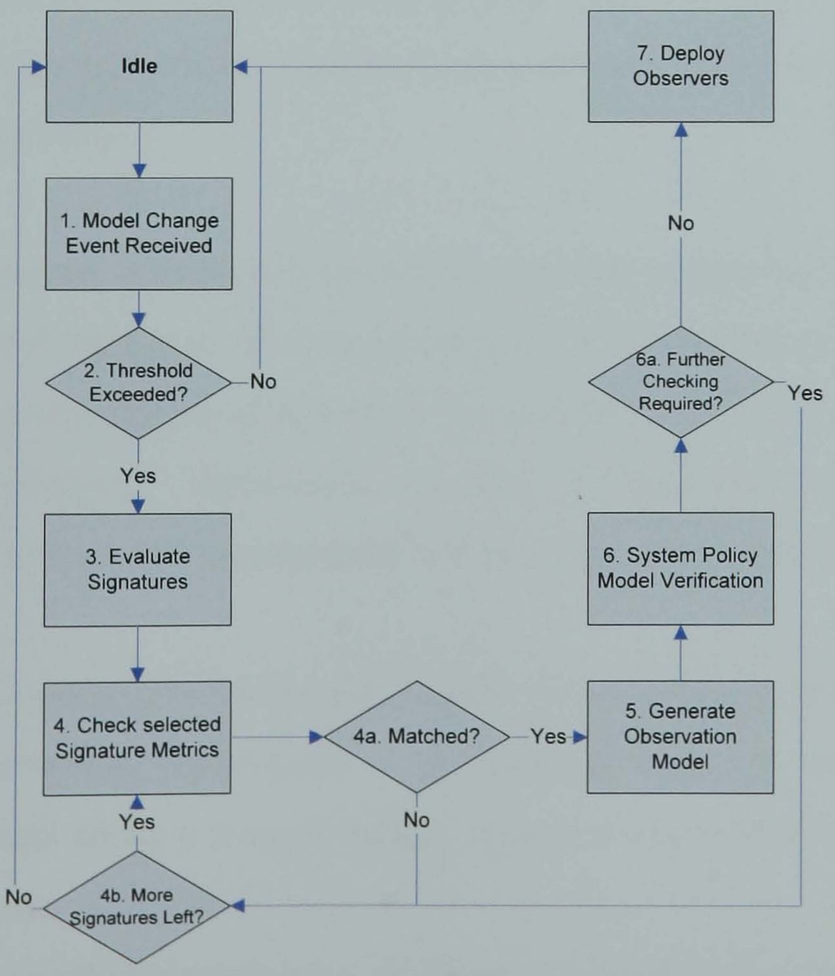

Idle
1. Model Change Event Received
2. Threshold Exceeded?
No
Yes
3. Evaluate Signatures
4. Check selected Signature Metrics
4a. Matched?
Yes
No
4b. More Signatures Left?
Yes
No
5. Generate Observation Model
6. System Policy Model Verification
6a. Further Checking Required?
Yes
No
7. Deploy Observers

Figure 7: Identification and Model Generation Process

In summary, the model generation process is still triggered by the matching of one or more signatures. The developed signature carries more information regarding suitable techniques, and when evaluated, a signature should be capable of specifying *how suitable* a match it is – a simplified utility measure may suffice. Rather than making a direct inference to a particular modelling technique, a set of applicable techniques are provided, and it is the responsibility of the model generator to select the most appropriate for any given input model. This may seem like unnecessary indirection, but it allows the model generator to check for other closely matched signatures, and to make modelling decisions based on other external policies, in addition to those specified in any signature-based techniques. For example, a policy could indicate that the modeller should produce a number of differently-simplified models to allow several observation techniques. Further examples of these external policies are detailed in the next section, which is concerned with the processes that occur on the simplified model in order to successfully deploy the system observers.

4.3.4 Deploying the Observers

The previous sections discussed identification and simplification of the system model. This section identifies the system-level processes that: are involved in taking this information, making appropriate considerations, and finally deploying or redeploying observers about the system.

A basic implementation may deploy an observer to each component identified within the simplified model. However, this would place all of the responsibility and decision-making regarding conservation of resources firmly within the model generation stage. Therefore, the proposed framework includes a separate component whose responsibility is resource and management concerns for observer deployment.

The observer deployment process should consider the following concerns:

- Existing observation deployment – to facilitate use on evolving systems, deployment must create a *transformation plan* to avoid removing observers only to redeploy them.
- Available observation resources – in order to deploy required observers, the deployment co-ordinator must ensure it has sufficient resources to do so. If not, it

may be able to select a further subset of the elements for monitoring, or refer a subset of the elements for monitoring at another unit.

- Other observation policies / rules – certain domains may require that particular elements are given a high observation priority, and in these cases, system-specific policies should override generated results.

A simplified architectural view of this process is shown in Figure 8, highlighting significant components and major data requirements. Note how in this diagram, the Observed Elements are shown as three separate entities, all linked to a single Observed Element Coordinator. This represents the manner in which observation responsibility may be delegated to another observer, according to resource data. In order to maintain system scalability, these observers are modelled as children, in effect coordinated by the parent observer. The actual implementation of this may vary depending on system requirements. For example, if another observer (known to the parent unit) has capacity, it could delegate to this unit. Alternatively, if the available resource data determines that the system has available resources (but the parent observer does not), it could request the creation of a new observer to manage a section of the subject set – effectively a new child observer.

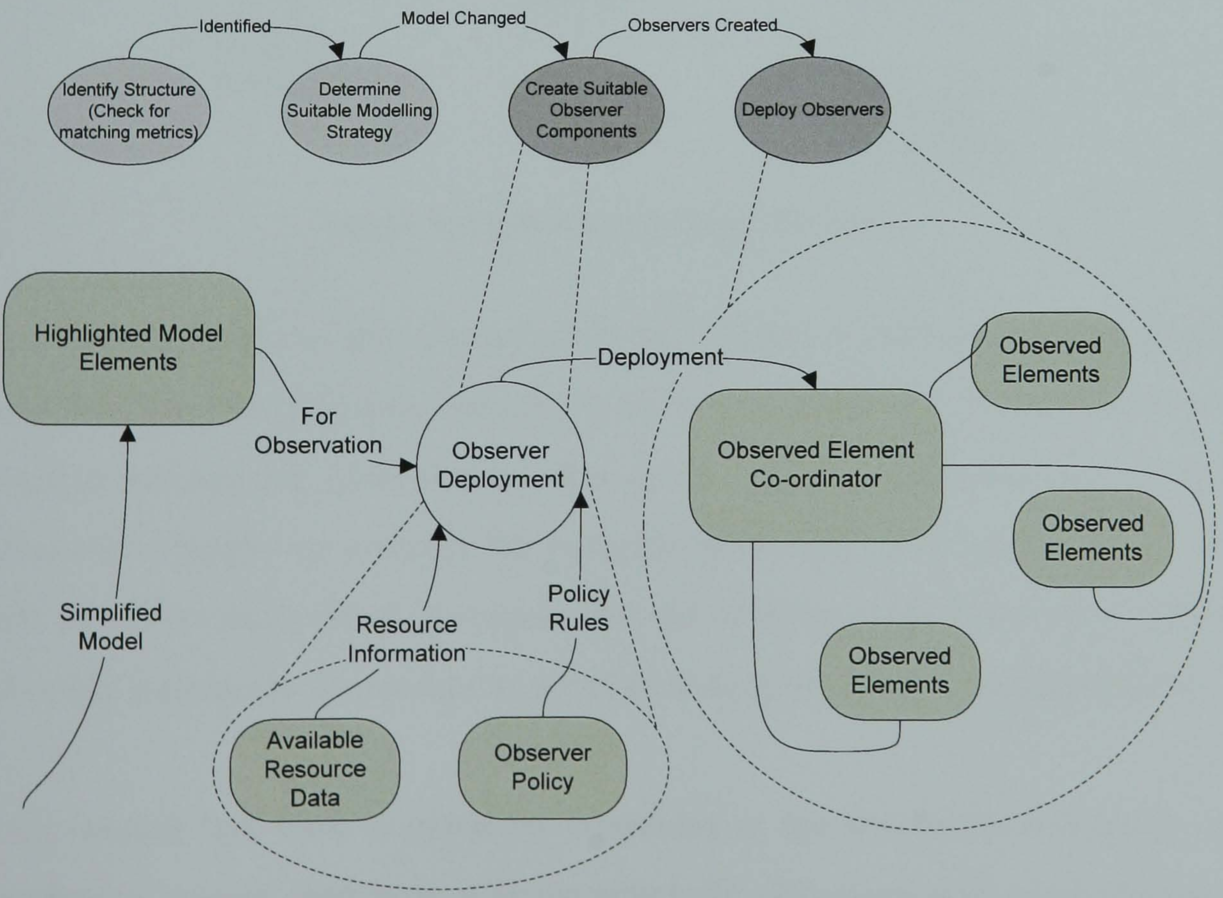


Figure 8: Observer Deployment Components

The observers are attached to targets in their simplified model, and then managed or overseen by their Observation Element Co-ordinator. The responsibility for creation, deployment, and removal of a given observer lies with its co-ordinator, which may remove and redeploy existing observers in accordance with available resource data. In order that the resulting deployment of observers is manageable and scalable, each level is expected to provide a filtration of events; observers collating, interpreting or otherwise processing the events before passing them up to the next level, potentially a level of system control, or a parent observer. A basic overview of the deployment process is shown in Figure 9:

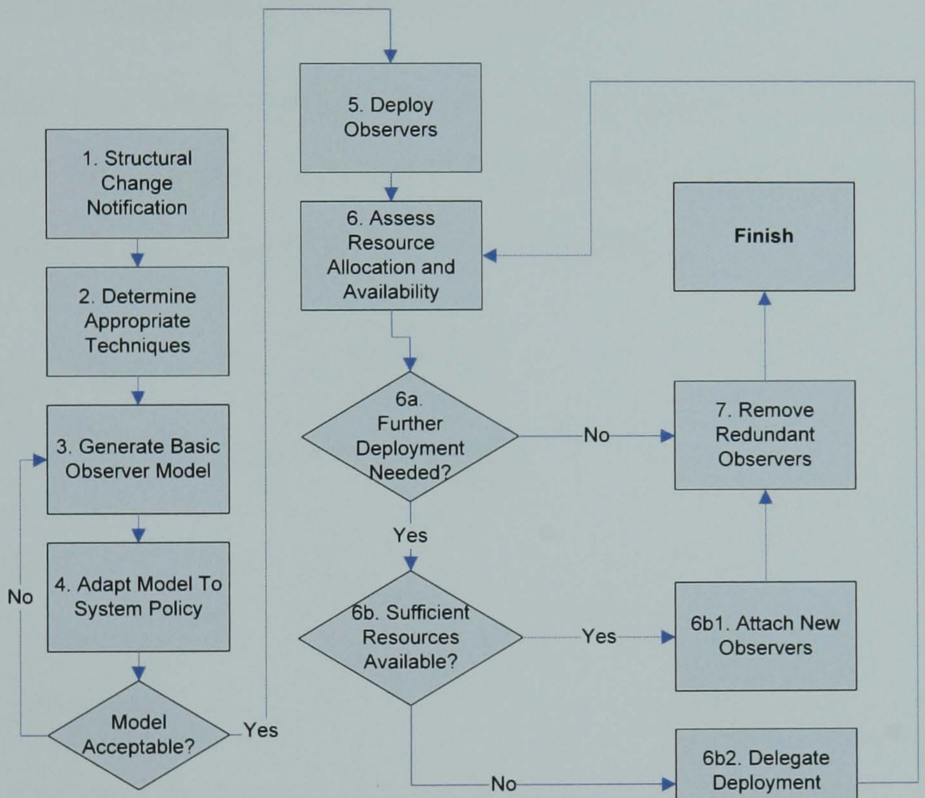

1. Structural Change Notification
2. Determine Appropriate Techniques
3. Generate Basic Observer Model
4. Adapt Model To System Policy
Model Acceptable?
No
Yes
5. Deploy Observers
6. Assess Resource Allocation and Availability
6a. Further Deployment Needed?
No
Yes
6b. Sufficient Resources Available?
Yes
No
Finish
7. Remove Redundant Observers
6b1. Attach New Observers
6b2. Delegate Deployment Responsibility

Figure 9: Deployment Process Overview

The previous subsections have identified the key points of information processing that fit in the Identify-Model-Deploy strategy proposed in Section 4.3, and how to use this approach to manage the scale present in a given system. Layers of units are used to progressively simplify or abstract the complexity present in a large system, while a division of observation effort is responsible for ensuring that the System Observer-level overlay represents the system as accurately as observation resources allow.

The next section will look to place the observers in the context of a complex and/or large software system, and how it is expected the observers will communicate with and serve the rest of the system's elements.

4.4 Complex Observers in Context

The previous subsections looked at the main processes surrounding management of system scale through to deployment. This section will summarise the requirements, describing the main components of the extended observer framework. It is hoped that by the conclusion of this chapter, the reader will be able to visualise the architectural elements necessary for the proposed framework, along with the ways in which the framework could be overlaid on a complex system design.

4.4.1 The Complex Software System

Before going into any detail on the observers, it is appropriate to introduce briefly how they fit into the complex software system model. Figure 10 shows the way in which observers fit in the overall system control model.

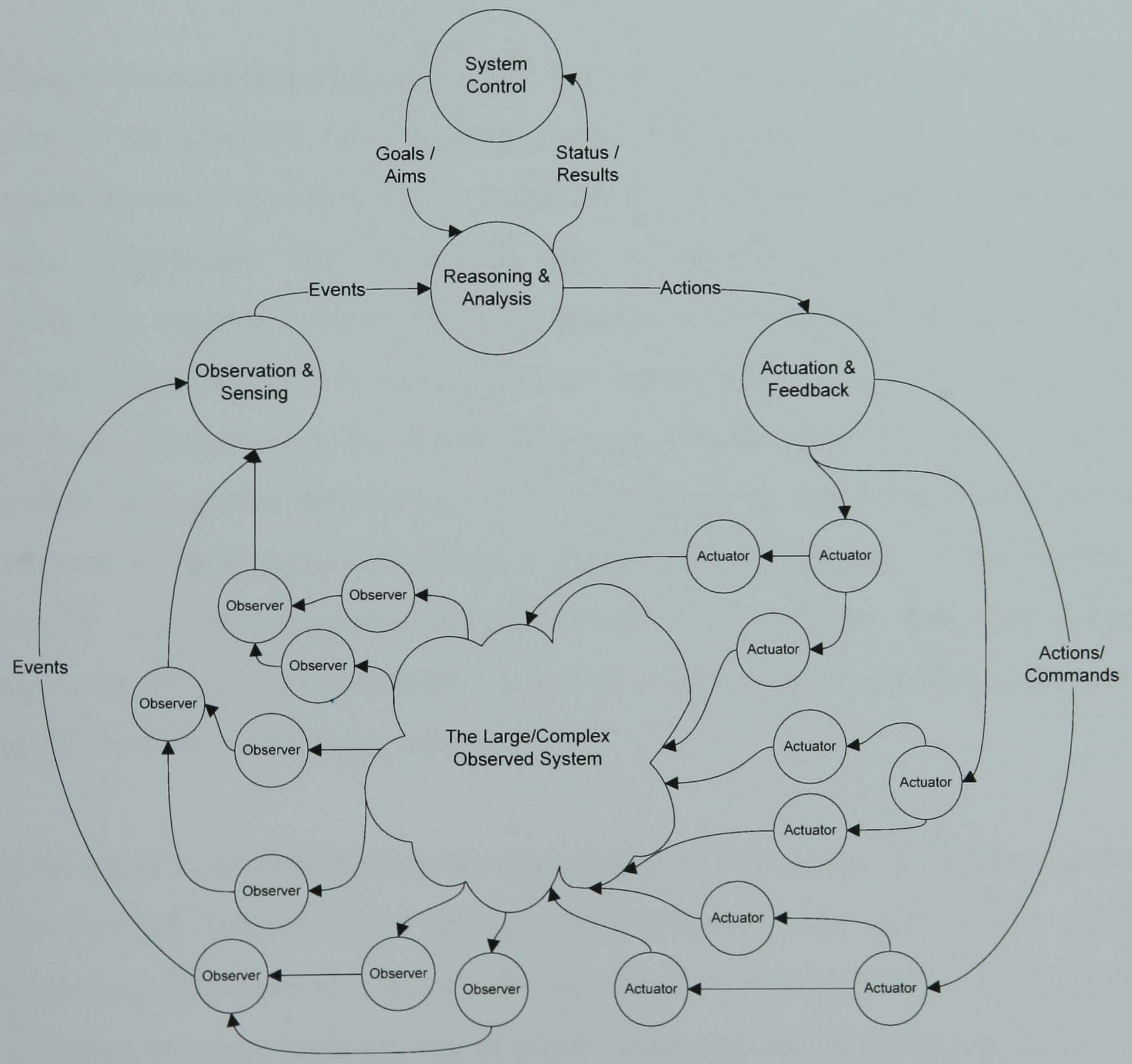


Figure 10: Controlled Complex System Architecture

This system is intended to represent a "generic" complex system, in as much as the interface with the actual system components is by way of:

- Observation – instrumented and attached in whichever way the system requires.
- Actions – performed upon the system via actions and commands specified in the correct manner by system-specific actuators.

The left side of the diagram represents the architecture of the observers with the right showing the control input to the system. Generally speaking, the top of the diagram represents the abstract, large area-of-effect (macro) actions, while the lower half represents the specialised, technical and component-specific (micro) actions – growing more specific for components nearer the complex system "object". The overall system architecture (along with the architecture of the observer subsystem which overlays it) takes themes and variations from the following:

Observer/Command/Action patterns – the model uses simplified interpretations of several of the common OO design patterns [27]. There is a clear separation of concerns between the components being observed (subjects) and the subsystem's monitor components (the observers), plus a standardised attachment interface. Additionally, applying a hierarchical arrangement means that a higher-level observer may have the task of monitoring several system-level observers and providing event→rule translation for the deliberation layer. Equally, there is a component-level separation between the deliberation of the system and the mechanism that enacts the chosen actions upon the system. While feedback to the system can take any number of forms, by specifying a generic feedback Command or Action, individual actuator components can implement specific Command wrappers, thus providing a translation layer for the components being altered.

Rule-based & Goal-driven systems – the generic system model shown above uses or implies several features found in rule-based systems, well supported in modern programming languages (e.g. [116]). Firstly, the system's objectives and constraints are specified at a high level by way of goals / aims that indicates the state the system should aim to regulate, or should work towards. Equally, system policy is used at all levels as an additional control mechanism specifying component-specific clarifications of goals, or other particular constraints. Secondly, the Reasoning and Analysis level is responsible for taking the observer's findings (also, at this level, assumed to be in the form of rules, or rather current "statements" in terms of rule variants) and translating them into suitable Actions that bring about the system's Goals, such as in the Agent methodology discussed in Section 2.3.4.

Autonomic systems and autonomic system control – there are many aspects of the system model inspired by or compliant with those common to autonomic systems as discussed in Chapter 2. The whole system is architecturally adaptive – that is, as the environment (in this case, the complex system being observed) changes, the observation overlay should adapt accordingly. Equally, it is expected to be aware; both self-aware (in terms of its goals), and aware of other systems that can impact its goals. Importantly, this level of monitoring / sensing is needed in order that the system can self-manage and take appropriate actions such that it achieves its goals while handling necessary administrative tasks. Finally, the system is modelled as a huge feedback/control loop, such as [117] – where sensing is performed within the observer components, reasoning components decide the appropriate feedback, (based on the system goals, including any necessary tuning criteria to try and correct undesired change), and closing the loop, the actuator components are instructed to perform actions or commands upon the system

The focus of this work is on the observation concerns of large, complex systems. To recap, software observers report their observations by way of events, which describe a change in system state at one a component. In the typical small-scale system model, these events arriving at an observer generally result in an action, via some simple conditional checks. Generally speaking, an observer has a specific responsibility – even something as simple as updating a user interface control to reflect a data change. In the complex system model in Figure 10, in accordance with the autonomic architecture discussed above, the link between observation and resulting actions is not as clearly defined or rigid. The observation findings are routed through an additional level of indirection before resulting in actions on the system. The observers send information via a hierarchy of observer control through to the reasoning and analysis component. The reasoning and analysis component is responsible for enacting the appropriate actions, based on the current system goals or aims – given the currently-observed status. In this generic complex system model, the method of abstraction in higher level observers is not specified. It simply implies that the events are passed from child observer up through parents and reach the reasoning or analysis parts of the system in order. As briefly discussed in Section 4.3.4, in order that the structural overlay of observers function as desired, each level of observation must perform some "filtering" of events, be it a simple collation, or some cognitive function on events before passing them up to the parent observer level.

The following section will explain how the proposed observers fit this high-level architectural view and manage scale and complexity, along with handling any structural evolution present in the system structure.

4.4.2 The Complex Observers

This research work concentrates on the way in which large-scale networks of observers can be attached to complex systems in order to provide suitably-simplified or abstract descriptive events to higher-level cognitive systems. There are similar concerns regarding feedback and actions, decided at cognitive levels and passed down to the system by way of an actuator system. However, the research focus is upon observation and description; analysis, returned feedback and actuation are largely outside the scope of this work.

The observer designs are split into two distinct architectures, representing the separation of concerns in the two parts. The first of these is the Structural Observation framework, proposed as a hierarchical overlay that is capable of monitoring the system components under observation; specifically with regard to component interconnection. The second part of the architecture is the System-level Observation framework. The Structural framework manages deployment and to some extent, the communication between components of the System-level framework. However, the System-level observer components are domain-specific, because of their close interaction with system components. As such, this research concentrates design effort on the Structural part of the framework, with specific requirements for the System-level observers noted as required.

4.4.3 Structural Observation Framework

The structural observation addresses the following requirements of the system, which were previously identified in Section 4.2: *Reduction of Observation Complexity* and *Variable Observer Scope.* The setup, observation and evolution processes of the structural observer are refined throughout Section 4.3. A simplified view of a set of structural observers is shown in Figure 11. The system's components or elements are shown as small dots at the system level, and each of the lowest layer of structural observers has a scope of interest, indicated by the dashed circle.

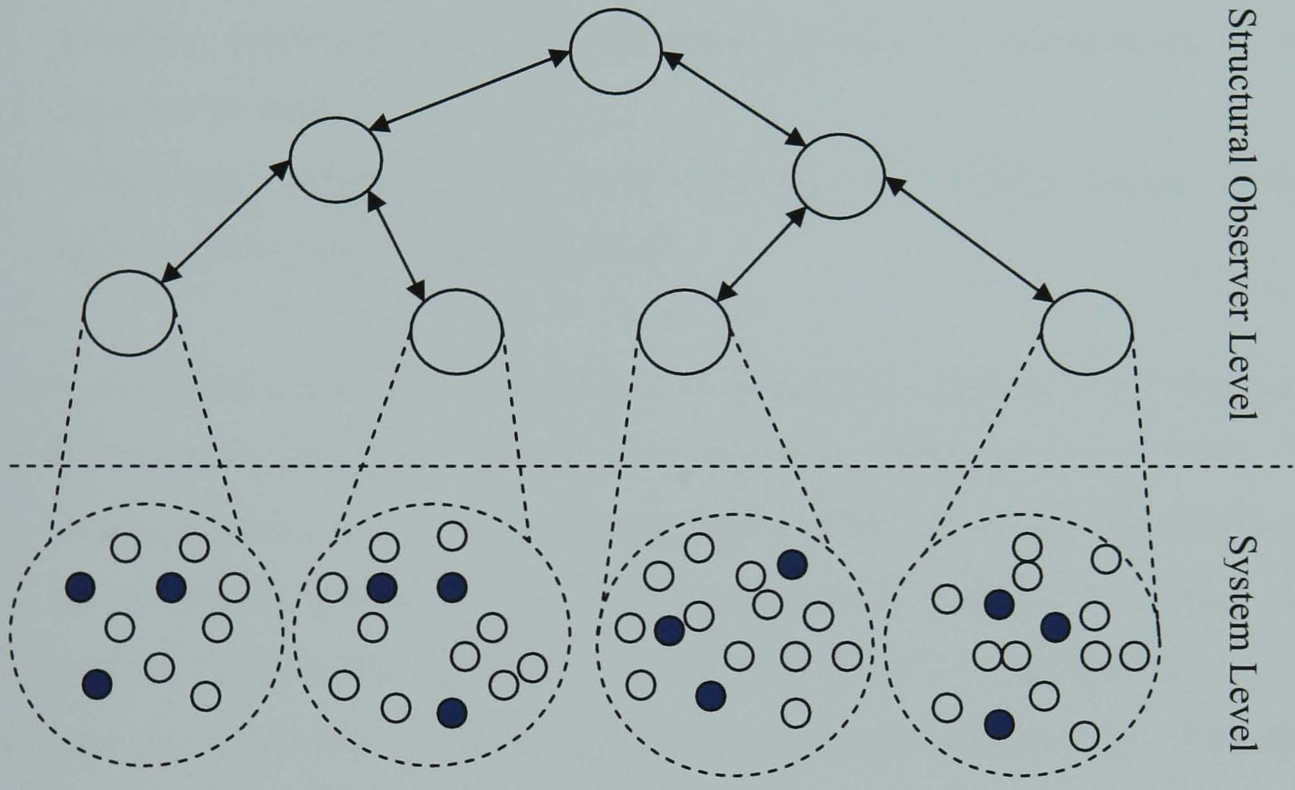


Figure 11: Hierarchical Organisation of Structural Observers and Scope of Interest

While the operation of an overlay-based and hierarchical deployment is a scalable architecture, it creates the issue that certain targets may need to be *interleaved* into more than one observer's scope at system level. This issue is less pertinent at the structural level, owing to the structure-sensitive signatures; as such, it will not receive detailed discussion in this research. However, if interleaving of several observers' scope (i.e. subject set) is required at the system-level, the adoption of an overlay-based architecture provides a solution: a single shared-scope subject may appear in the overlay as a collection of different subjects, depending on their subject requirements.

In deployment, the observers' scope of interest is tailored to the system configuration, adapting accordingly, as outlined in Sections 4.3.3 and 4.3.4 – as is the responsibility of the Structural framework. Components that are identified of particular interest are shown as filled dots, which fall into the second part of this architecture model - the System-level Observation. Each structural observer has the following responsibilities:

- Appropriate instrumentation within each system component to detect the following situations: Creation of a new component, removal of an existing component, and connection between two components.
- Modelling this data facilitating considerations on the observed structure.
- Sending and receiving a set of events that describe these structural changes.
- Creation, deployment and removal responsibilities for management of the structural overlay*
- Identifying components of particular interest that should receive further attention from other observation units*

The last two points (marked *) are items of responsibility deferred to the structural observer's reasoning module, composed of two elements; illustrated in Figure 12:

- Structural Identification – identify the type of structure being observed based on the details obtained from each instrumented component, monitor the structure for changes, and identify key areas for intensive observation.
- Resource Allocation – in conjunction with structural identification, determine the exact structure of the observation overlaid on the system by way of resource constraints specified as rules, and any observer-level policies.

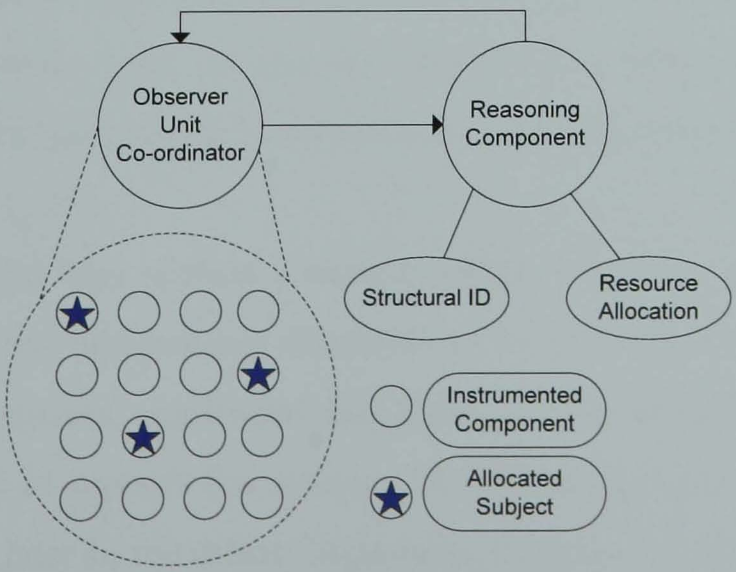


Figure 12: Architectural Overview of Structural Observer Unit

Figure 12 also shows a static snapshot of an isolated structural observer unit. In this simplification, the co-ordinator has allocated 3 subjects for intensive system-level observation. These subjects have been selected from the instrumented set of components, based on the appropriate structural identification process and in accordance with the unit's resource allocation rules.

This assumes the structural observer unit has sufficient resources available to it (defined in system policy, as discussed in Section 4.3.4). In the case of a very-large-scale enterprise, a single structural unit may have insufficient resources. Therefore, the structural unit may:

- Refuse observation responsibility and advertise for another unit to take over.
- Delegate observation responsibility for the coverage to another known observation unit. The other observation units are then effectively managed by this unit in a parent-child relationship; the parent sets policy to describe its expectations and requirements from the child.

In both cases direct instrumentation responsibility and structural management then passes to the new unit. However, the manager / parent unit retains responsibility for managing the fulfilment of system observer policy. Therefore, if the parent determines it can adequately fulfil the requirements without delegation, it can request the child units un-deploy their system-level observers and manage them directly. Given that the system must manage and interpret the system-level observers' reports, domain-specific design is still needed to determine the types of observation required. The required observation criteria and models for system-level observation need to be engineered, following an appropriate modelling strategy. While adaptive, the model is not intended to rewrite code on the fly, determining *behaviour* of system-level observers; rather the organisational, management and deployment characteristics.

In order to summarise this section a simple system overview diagram is shown in Figure 13, outlining the components discussed in Section 4.3. The diagram shows the main software components, significant data flows, and the repositories of information that will be required to support this system. The following items on the diagram will be further specified later in the thesis: Signatures (Section 5.2), Techniques and basic

Model Generation (Section 5.3), Policy and Model Generation (Section 6.1.2), and finally, the Deployment Process (Section 6.1.3).

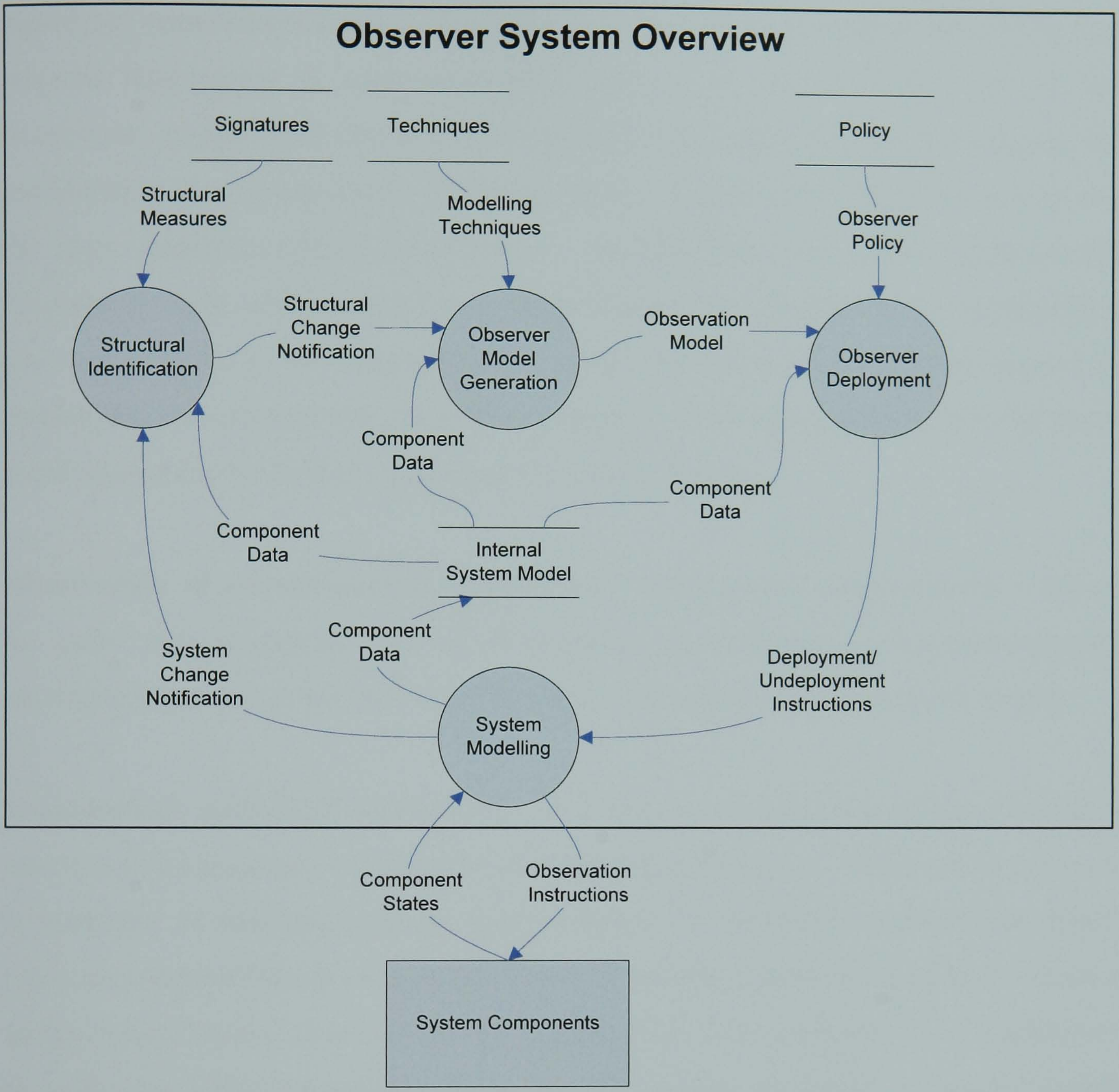


Figure 13: Structural Observation Overview

However, to complete this chapter, significant architectural concerns in the System-level framework are still to be fully defined. The next section will briefly examine the concerns at the System-level to provide the desired level of system observation.

4.4.4 System-level Observation Framework

References to the system-level observers and the system-level observation framework have been made throughout the previous section. This section aims to clarify their purpose, state their high-level requirements and produce design guidance aimed at engineers producing observation subsystems for deployment within this framework.

Structural observers are considered to be low-cost and effectively ubiquitous throughout the structure, yet system-level observers are domain specific, may require significant computational power and as such, require targeted deployment. While this targeted deployment is undertaken and managed by the structural part of the framework, system-level observers are responsible for observing and reporting on the individual system components. Referring back to Figure 12 in the previous section, the stars represent system components / elements that will receive system-level observation. This section describes a basic architecture comprising the expectations and requirements of the system-level observers. Given the Structural framework is responsible for efficient and targeted selection of observation targets, the high-level requirements for the System-level observers are as follows:

Observation of system-specific components / collection of measurements – this is the most domain-specific area of observation, encompassing component-specific instrumentation and suitable processing of this information into descriptive events.

Standardised method of deployment – The structural framework defined thus far is based on a homogenous arrangement of structural components. This does not mean it is incapable of handling systems with a variety of component composition types; rather that these differing component types are suitably abstracted in order to create a global system model. However, at the system-level, this simplification is unlikely to be sufficient. Different components within the system are likely to require unique methods of instrumentation and different business logic in order to handle and generate suitable events. Equally, the structural-level observers, with their abstract system viewpoint, require a standardised deployment interface. Without stepping into low-level design/implementation details, the deployment mechanism could make use of a factory-type model [27] to control creation of system-level observers. The factory would create a suitable observer based on the target component type.

Co-ordinating System-level Observation – The final issue regarding system-level observer structure is co-ordinating observers' reported events. Again, this process is largely domain-specific but is seen from two distinct architectural perspectives:

- **Complex** – The complex approach accepts that although the structural framework may have reduced the complexity and scale of the target set; it still has organisational aspects rendering it unsuitable for design-time observation strategies. This approach relies on the structural observation framework taking responsibility for managing the scale of the system, yet places responsibility on system-level observation to determine co-ordination strategies at runtime.

- **Simplified** – In the simplified approach, system-level observers monitor their targeted system components and either: act independently, are co-ordinated in a (near) traditional manner, or a combination of both models. This makes the assumption that the structural framework has reduced the scale and complexity of observed targets sufficiently for normal SE architectural and design patterns to apply. This approach is preferred; for the relative simplicity of organisation in the high-cost/system-level observation, and as it enforces a clear division of responsibility between structural and system-level components. Structural observation is responsible for management of scale and structure, whereas system-level deals with the application/system concerns.

Regardless of the system-level method used to handle collation and event processing, the structural framework must provide a suitable interface to coordinate the two distinct structural and system areas. The Mediator design pattern [27] can provide a solution such that event handling, collation or processing is provided as mediation, and individually-deployed observers refer any events that they cannot handle locally to a suitable processing mediator. However, this design pattern tends to centralise the handling protocol, which may make it unsuitable for a large and diverse system.

4.5 Summary

This chapter aims to give an architectural overview of how the findings of this research should be applied to complex systems observation design. It began by giving a brief overview of the existing Observer design pattern and then discussed how the extensions required would provide better support for the kinds of structures found in complex and large software systems. An examination of the aims followed; identifying the major processes to be addressed in the makeup of an adaptive and scalable observer framework. The chapter proposes a two-part architecture, divided into structural and system-level observation; the former managing the latter:

The **Structural Observation** elements are responsible for breaking the system down into manageable "chunks" of observation, and for exploiting topological features within those chunks, where possible. These responsibilities are broken down into three subsystems:

1. Structural Identification – identifying the type of structure / substructure on which the observation should operate, along with (a set of) techniques that can be applied to best exploit the structural type.
2. Modelling and Reasoning – creating a limited model of the structure, using the supplied exploitation techniques. This limited model represents the observation set, based on resource constraints, along with other general system observer policy.
3. Creation and Deployment of System Observation – deploying system-level observers on the system based on the observation model. In order to manage the magnitude of the observed target set, this may involve the delegation of parts of the observation to child observers.

Conversely, **System-level Observation** elements are deployed and managed by the structural observation component. They are responsible for observation of real system events (i.e. those that are of interest to a general system observer, rather than one interested in structural exploitation) and have the following responsibilities:

- **Instrumentation** and component / unit attachment – this should follow traditional software engineering patterns for instrumentation and observation of components. System-level observers will attach to the subject in an appropriate manner and either:
    - Hold full control over their observation / response (i.e. independent observers); with the rest of the framework providing only guidance regarding where and when to be deployed.
    - Make observations and generate their own descriptive events that can be packaged and sent to higher-level observers for further co-ordination.
- **Standardised Deployment** – while this design section does not consider low-level design specifics of the system observers, it is important to specify the manner in which the structural observers will control the deployment of the system observers. This takes the form of a standardised deployment interface, parameterised for further domain-specific requirements. As discussed

previously, a factory pattern is a preference to allow a standard interface to deploy a variety of different types of observer according to the selected target components.

- **Observation Co-ordination** – In the event the system observation units cannot operate independently, or the design for the resulting observation system is insufficiently understood to be modelled with standard approaches at design time, there must be mediation/co-ordination between system observers.

As discussed throughout the chapter, the system-level part of the framework involves considerable domain-specific information; both in terms of the targets to be observed and the types of event to be collected, and generated for higher levels in the overlay. As such, the next chapter will discuss designs for the structural part of the framework. This will take the form of developing the identified design requirements, and continuing to follow the proposed architectures in order that a suitably-detailed design emerges. The design will describe the relevant detail required for an engineer to apply these techniques in a real software system.

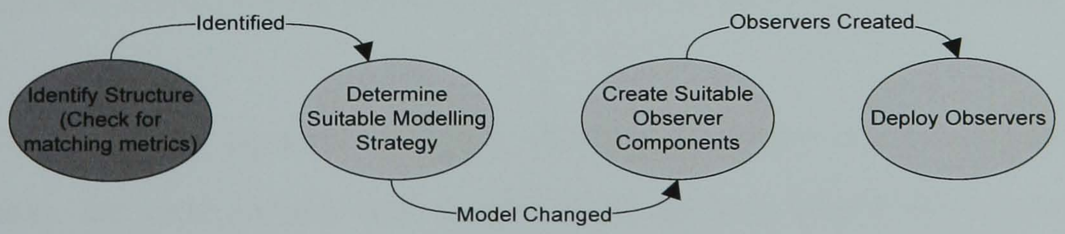
Chapter 5 – Specifying the Observer Programming Models

As described in Chapter 4, the observer framework specifies two support elements:

- **Structural Observation,** responsible for dividing the system into observable chunks, managing them, and exploiting features of those "chunks" for best observation effort.
- **System-level Observation**, deployed and essentially controlled by the Structural Observation elements, and responsible for observing the "real" system events and producing appropriate feedback. The feedback may be direct to the component or produced by generating or forwarding events that are passed to system-level observers operating at a higher level.

For the sake of conciseness, this Chapter only outlines the key aspects of the proposed framework and associated programming model. However, a more detailed specification along with relevant supporting information can be found in Appendix I. Throughout this chapter, the design process is explained in detail, and summarised by way of the recommended classes and interfaces in a UML class diagram form. The diagrams follow standard UML notation, and for the sake of brevity, obvious *accessor / mutator* methods for attribute access are typically omitted.

5.1 Specifying Structural Types


Identified
Observers Created
Identify Structure (Check for matching metrics)
Determine Suitable Modelling Strategy
Create Suitable Observer Components
Deploy Observers
Model Changed

The first step to exploitative observation is to identify the structure on which the observation must operate. The author's experimentation work on structural identification, as discussed in Section 4.3.1 [115] deployed observers about a hypothetical system based on the identified structure. This section will begin by identifying significant points from this work, before expanding to show how this can be generalised and therefore made more widely-usable.

The structural identification signature used a hard-coded metric-based evaluation on the structure under investigation. The signature was simply referenced where required, as demonstrated in Figure 14. The simple code sample outlines the use of a parameterised Acquaintance Nomination Signature (refer to Section 3.3.4 for a reminder of the specifics of the metric in this signature), which requires two data items in order to check for a match – the structure (*network* in the code) and the required match value. This is the desired value of "aMetric" (*matchValue* in the code), which is specified with a given +/- tolerance (+/- 0.1). The signature is checked for a match simply by calling its *checkMatch* method.

```
DoubleTolerance matchValue = new DoubleTolerance(0.1d, 0.1d);
StaticSignature acqSig = new AcquaintanceSig(network,
                                 matchValue);

Boolean ACQUAINTANCE_MATCH = acqSig.checkMatch();
```

Figure 14: Simple Hard-coded Signature

This code sample lifted from the experimentation work has some immediately apparent limitations, along with some assumptions being made:

- Signature calculation specifics are defined in code (the *AcquaintanceSig* class), with a single runtime-specifiable parameter – the desired aMetric value.
- The system structure is already specified, as a form of collection, in a variable called *network*.
- Signature evaluation is invoked by an explicit call to the *checkMatch* method.

Before examining the signature-specific limitations, requirements and designs that address them, the next section will examine the ways in which the system structure itself can be modelled for signature and observation needs.

5.1.1 Modelling the System Structure

In order to facilitate the structural signature match approach, the system structure must be modelled in a standardised way. In addition to the specification of a standard structural model, consideration must be given to the manner in which this model is kept up to date with the system. As described in Section 4.3.1, the signature approach relies on a set of simple monitors exhaustively deployed about the system, which:

- Keep the structural model up to date , and
- Enable "structural change" triggers to be fired, allowing signatures to be checked only when the structure changes

Structural signatures will assess this model when required, and must be capable of specifying their trigger mechanism in terms of the simple structural events fired by these monitors. The structural model must represent the necessary aspects of a complex system (bearing in mind the potential to extend signatures beyond structural characteristics, as mentioned earlier), while allowing a domain-specific specialisation to represent other *globally-important* characteristics. When considering the potential specialisation, it is important to remember that while the structural model may represent additional information, the system-level instrumentation required to represent it, along with the signature matching criteria, has computation cost. The structural model should have as minimal overhead on the underlying system as possible. Signature match criteria in particular should not reach a level of complexity where real-time operation would become impractical.

Therefore, at its most basic level, the structural model must represent every significant element in the observed system – i.e. everything that may require overlay observation. Equally, the model must represent the relationships between different elements. The experimental work briefly discussed in the previous chapter and examined in [118] suggested the structure could be representative of workstations in a networked setup. The same graph model could be used to indicate dependencies between software components, usage of various services, or connections between servers. The precise nature of the relationship between model elements will, to a certain extent, determine the nature of signatures checked, and the observation techniques reasoned on and then applied. This relationship information at each low-level component or *modelled element* will be implemented using domain-specific and technology-specific techniques.

It is envisaged that production-quality systems may need to make use of several distinct overlays, each having the inter-component relationships modelled in different ways, and serving a different type of observation requirement.

Therefore, the following model specification designs will be presented at an abstract level, and although concrete implementation examples will be included for illustration, they will demonstrate *a single* usage, not *the only available* usage. The graph model provides a simple abstraction for the types of relationship stated above – components modelled as vertices and their relationships as edges. As discussed in Section 3.2, treating the structure's relationship model as a graph allows the system to apply well-established graph theory techniques to simplify and examine the structure.

The designs for the structural model will take the following form: Firstly, the classes required to support the model will be introduced, along with a discussion on their implementation. This section will conclude by explaining how the structural model will self-manage, leading on to the specification of signatures to assess these models.

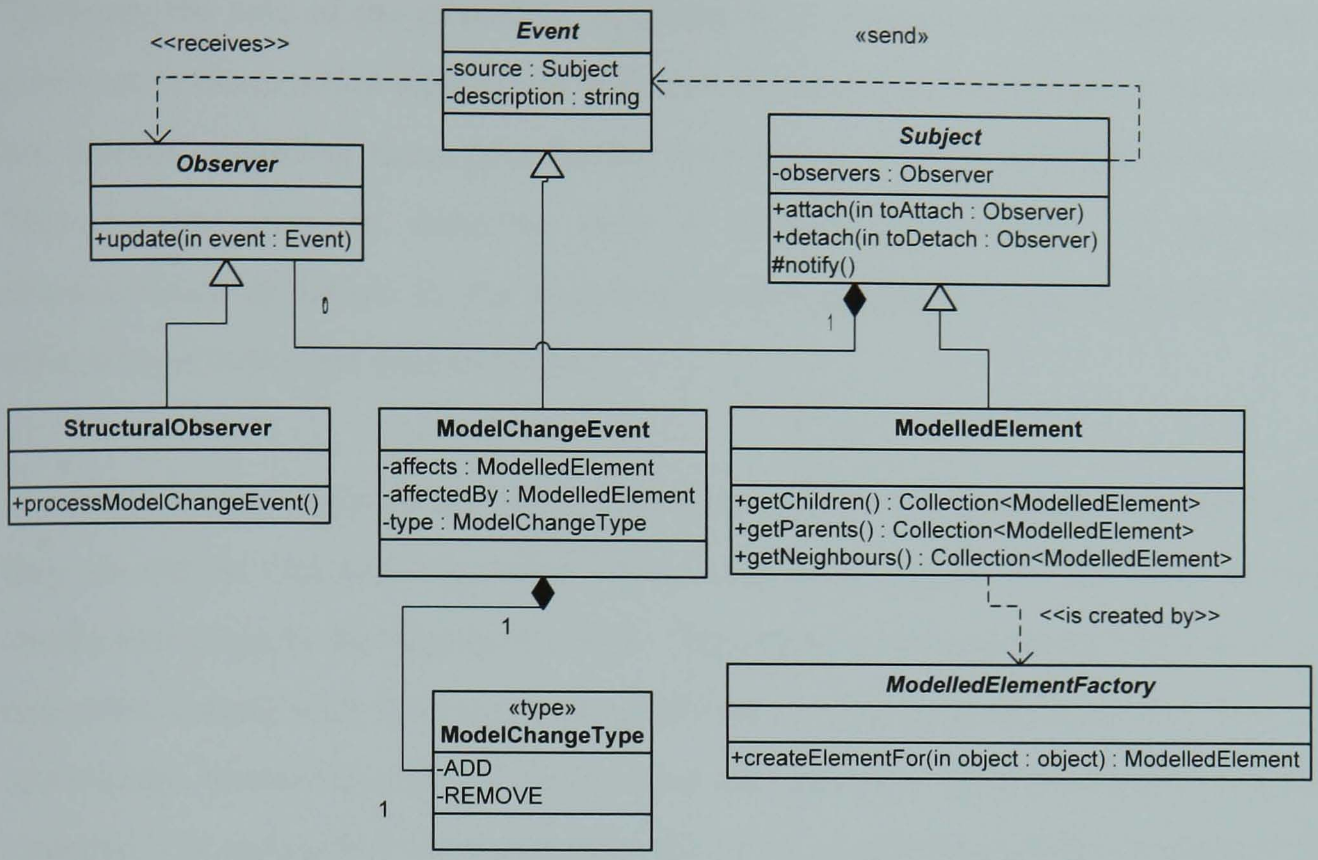
The structural model will be formed with appropriate instances of a single type, known as `ModelledElement`. In an implementation, this class may be extended as required to encompass any system-specific information for particular elements that need consideration within a signature. Instances of `ModelledElement` maintain a simple child and parent relationship with one another, allowing representation as a directed graph (directional edges represented as children and parents). Model information is monitored by `StructuralObservers`, which are attached to the real system elements, each represented by a `ModelledElement` which hold responsibility for keeping element-specific information up-to-date. Modelled Elements represent a Bridge or Adapter between the structural model and its individual components. As such, the implementation details of each element's relationship to its matching component are necessarily component specific; depending on, amongst other concerns, the way in which the subject component allows instrumentation of the required data.

However, regardless of the system-specific implementation concerns involved in the creation of a Modelled Element; in order to reduce complexities to the Structural Observer, a standardised method of Modelled Element creation must be made available. This method should:

- Manage the component-specifics for hooking into the events necessary for the Structural Observer.
- Create a Modelled Element with adequate descriptive content for the underlying component.
- Have a standard interface such that objects (i.e. Structural Observers) requesting a Modelled Element do not need to know the specifics of the target object.

Given these requirements, the design proposes the use of an Abstract Factory pattern [119]; an abstract `ModelledElementFactory` will be implemented as required for the various component types encountered in the model. The system-specific observer will use the Modelled Element Factory to create the appropriate Modelled Elements, containing any type-specific logic to this class. This allows the system to create Modelled Elements via the factory without needing to be aware of the runtime type of the underlying system elements.

The `StructuralObserver` expects to receive `ModelChangeEvents` from its subject elements, which describe the type of model change and the affected element(s). The basic types of change proposed are **add** (connect) and **remove** (disconnect), covering the events that govern the evolution of a system's structure. Therefore, the basic functionality expected of a `StructuralObserver` is to maintain the accuracy of any overlay model information based on observed `ModelledElements` – by responding to add/remove events. Additionally, the `StructuralObserver` will also provide the basis for the attachment of signature triggers, which will be explained in more detail in Section 5.2. The relationship between the low-level observation classes is shown in the following class diagram, Figure 15. For ease of reference and to assist those unfamiliar with the Observer design pattern, the conceptual link to the `Subject`, `Event` and `Observer` classes is also explicitly shown on this diagram.



<<receives>>
Event
-source : Subject
-description : string
«send»
Observer
+update(in event : Event)
Subject
-observers : Observer
+attach(in toAttach : Observer)
+detach(in toDetach : Observer)
#notify()
0
1
StructuralObserver
+processModelChangeEvent()
ModelChangeEvent
-affects : ModelledElement
-affectedBy : ModelledElement
-type : ModelChangeType
ModelledElement
+getChildren() : Collection<ModelledElement>
+getParents() : Collection<ModelledElement>
+getNeighbours() : Collection<ModelledElement>
<<is created by>>
1
«type»
ModelChangeType
-ADD
-REMOVE
1
ModelledElementFactory
+createElementFor(in object : object) : ModelledElement

Figure 15: UML Class Diagram showing Structural classes

When a Structural Observer is monitoring a system, the underlying system model (as represented by modelled elements) is kept up-to-date by way of the observer receiving add/remove events for newly connected or removed components; then creating or discarding the related modelled elements appropriately. However, when the observer is first deployed; it cannot rely on these events to build a complete system model. Therefore, the observer must be able to employ a method to allow it to build this model from a system on which it has just been deployed; detailed specification of this process is found in Section 6.2.2. Additionally, a worked example of the structural model can be found in Appendix I (Section I.I).

5.2 The Structural Signatures

With a standardised minimum level of modelling information provided by the structural observation subsystem, it is now possible to consider the requirements of the basic structural signature, and how it will obtain the necessary information from the system model. Equally, it is important to consider how the structural signature could be extended to support any model extensions that have been made at the **modelled element** level.

To recap, the role of the structural signature is to provide the association between a given set of observation techniques with a particular (structural) trend or characteristic set. Details regarding these observation techniques will be deferred to Section 5.3. This section aims to examine only a suitable specification of the relevant characteristics or trends in the structural model specified in the previous sections; effectively a structural type identifier.

In certain cases, it may be appropriate for the signature to specify the manner in which they should be checked; e.g. time interval between checks, or low-level modelling events that triggers the signature check. The typed structural model intends to allow observers to deal with systems with unfamiliar system-structure specifics. Due to the uncertainty, incompleteness of information and potential variety inherent in complex systems; it is unfeasible to try and cater for every possible structural or organisational formation. Instead, the signature approach is the basis for an abstract typed observation framework – specifying heuristics which are detected at runtime, thus adapting observation accordingly. In summary, a signature must fulfil the following requirements:

Type Specification: Have the capability to check (a set of) characteristics in the structural model and return a result indicating type compliance or otherwise. In the example discussed in Section 5.1 and Figure 14, this requirement is fulfilled by the Boolean *checkMatch* method.

Potential Specialisation: The signatures must be open to extension to allow additional domain-specific information to be included within a defined type.

Invalidation and Triggers: Allows the association of a trigger event (or set thereof) from the structural model that indicates a previously matched signature has become "invalid" and should be rechecked. As a signature encapsulates the structural metric data required for type specification, it is best suited to specify the structural alterations likely to lead to metric invalidation.

In essence, the definition of a signature is an example of a Template Method pattern [27] within the structural observer model. Implementations will define the algorithms, metrics and measures that provide the signature's characterisation, while the Template defines the interface to the signature. As such, the model of a generalised signature template within the framework must consider:

- How structural model interaction (querying or exploration) should be facilitated, and how the match criteria can be specified against this model.
- How the signature invalidation (i.e. the trigger to recheck the signature) can be specified in terms of a model change event, or a composite thereof – along with potential implications for the invalidation model.

The next section will briefly outline how a Scale-Free Signature fits the above model, thus examining the suitability of the proposed requirements.

5.2.1 A Scale-Free Signature

When considering the required methods for structural model examination, it is worth remembering that signatures may require rechecking at regular intervals. As such, low computational complexity is a desirable property. Therefore, signatures may make use of simplification measures; examples of which can be found within Chapter 3. Section 3.2 discusses scale-free systems and metrics providing a heuristic indication of scale-free topology. In order to develop the signature specification, the following applies the previously-defined criteria to a scale-free signature.

Type Specification: The Signature should specify the measures required for the Acquaintance Nomination metric, as discussed in Section 3.3.4; in order to give an indication of Scale-Free (SF) topology. Summarising the data requirements from this section indicates that the signature requires the following data from the model:

- The *size* of the model – i.e. number of elements.
- A set of elements, selected randomly from the *entire model*
- The directly-connected elements for a given element (its neighbours)

The model structure proposed in Section 5.1.1 can only fulfil the final requirement. The other requirements suggest a level of global data access that has not yet been specified. However, the Modelled Element Factory also proposed in Section 5.1.1 must maintain this information for object issuing and pooling; therefore it is logical to formalise this functionality and propose a data relationship between Signatures and their related Modelled Element Factory. Finally, the type indication may be provided in several forms. Referring back to the pseudo-code example in Figure 14, this signature provides a Boolean result; however, it could potentially provide a continuous result of bounded accuracy by providing a numeric indication; for example, a normalised distance from the required value.

Potential for Specialisation: While it is impossible to predict every potential specialisation; with a SF-indicative signature such as this, a possible specialisation for heterogeneous component-based systems may involve a filter on underlying Modelled Elements; only certain *types* of connections are included in the metric.

Invalidation Requirements: The SF indication provided by the signature is invalidated after a certain number of connection alterations have been made to the underlying model graph. As discussed in Section 3.3, a system's organisation can undergo connectivity transition phases, during which the topological features required to match this signature may emerge. Connectivity changes are quantified in terms of Model Change Events, as discussed in Section 5.1.1; as such, the signature must specify the magnitude or types of events that lead to signature invalidation.

5.2.2 Formalising the Requirements

In summary, the extended requirements for the structural signature's design model (and implications on the model design thus far) are as follows:

Type Specification: The system structure modelling must be extended to support some "global" (i.e. structural observer domain) information; with a complete collection of the modelled elements as a minimum requirement. The required global information is, as per previous designs collected in the Modelled Element Factory. However, in order to clearly separate the design concerns of the factory from these requirements, it is proposed to make this functionality available via the Structural Observer. Developers implementing the factory must maintain the element list in a scalable manner within the modelled element structure – such as placing the elements in a distributed shared space or hash table.

Potential Specialisation: while it is impossible to cater for every potential specialisation of a given signature, a signature definition should not make it difficult for developers to implement likely specialisations, based on the proposed model access and constraints on signature execution. The potential specialisation in the example case relied only on the specialised signature's ability to interrogate the runtime-type model object, rather than modelled element interface constraints.

Invalidation and Triggers of a signature's match status; a signature should define the criteria (i.e. sequence and/or magnitude of events) that render a previous match invalid, if possible. However, it is also clear that the framework needs to incorporate a standardised method by which interested parties can exercise control over the priority given to various concerns within a signature's triggering. This allows for an appropriate balance between timely signature accuracy and the computational cost of repeated signature matches. However, should this trigger describe invalidation criteria more complex than just the magnitude of structural change that must occur (i.e. a number of connect/disconnect operations), it is possible that capturing and processing these contributory events may involve greater overhead than simply rechecking the signature at periodic intervals. Therefore, invalidation should be used with care, and only to help regulate the amount of computational load present in a fast-evolving system. The next section will demonstrate how these refined requirements in terms of class and interface changes.

5.2.3 Modelling the Signature in Software

Previous sections divided the signature specification requirements into three areas aimed to refine the Template's requirements. However, examination also highlighted a number of further design requirements for the underlying structural model. These are presented below with a brief summary of each requirement's meaning, along with their contributory model requirements as discussed above:

Type specification (The model metrics or statistics that define a particular type and how they do so)

- Indication of type compliance – at the simplest level, a Boolean true/false, and potentially a continuous (normalised between 0-1) value indication, allowing further consideration on the quality of a type indication.

- Access to model via existing graph / structural model – in order to explore graph model and specific elements (Provision via Structural Observer, which accesses the elements in the appropriate Factory).
- Direct access to all model elements, without the need to "explore" (i.e. follow neighbour links) to gain a complete model overview, perform statistical calculations on the model, and to rapidly select subsets of elements according to certain criteria.
- A specification of compliance test – the signature must specify the criteria by which a collection of modelled elements are tested for compliance. In a typical implementation, this specification will take the form of an algorithm in the checking method. While this will fulfil the requirements of the signature, it should also be considered that other components may benefit from being able to inspect the specifics of the compliance test. Chapter 7 discusses a variety of issues surrounding externalisation.

Potential for specialisation (Required flexibility in a design to permit implemented signatures access to required data in specialised model elements)

Invalidation and Triggers (How a signature should specify events that lead to a potential change in signature state)

- A New event type - Invalidation Event - that can adequately describe the change that must occur within the model to "invalidate" the previous signature match value.
- Granularity of change - It is appreciated that the smallest of model change may slightly affect the continuous match value, necessitating possibly-needless repeated signature rechecking. Therefore, the invalidation event should include a granularity value, exposing a method of control over the magnitude of change required to invalidate the signature. However, even at minimum sensitivity; the invalidation event should specify the degree of model change that is required to alter the discrete match value.

Therefore, in breaking down these requirements into areas of responsibility, it is shown that there is interaction with the components from previous stages of design:-

Modelled Elements – they provide the description of the structure to be assessed and are accessed in two ways. The first is as a direct reference to the targeted modelled element, which allows access via exploration to the entire model, while the second is via the proposed "shared space" whereby all the elements within a model are available directly.

Model Change Events – they will provide the description for the invalidation mechanism previously discussed, by way of disconnection and connection events describing structural change, and potential "triggers" for reassessing the signature.

Structural Observer – the related observer will provide the signature with access to the following information:

- "Global" list of elements – by modifying the Modelled Element Factory so that it exposes the list of element objects that it has "issued", this allows the observer to make available all the elements it is monitoring
- Shared variable space – this is to allow a signature to place data for evaluation by other components within the structural observer's scope.

As there are significant new functional requirements, this necessitates new design components for the system model. These can be summarised as:-

Signature – the Template – implementations are responsible for: specifying the match criteria, the model to assess, and for assessing the match criteria and providing a result. This result should be available in both discrete and continuous forms. Once a signature has been assessed, in addition to the results, the signature should be responsible for specifying the potential invalidation trigger events. This is a separate area of responsibility in its own right and is described below.

Invalidation Handler – the invalidation handler should provide functionality (to be used by an interested party, such as an observer) to determine when a signature should be reassessed. Externally, it should encapsulate the model changes that potentially alter the signature's match status. It should provide a method of control to allow the balance between accuracy and overhead to be fine-tuned. It will be an event transformer; subscribing to change events from the model in order to determine when the signature is potentially invalidated, then generating its own event to inform interested parties when this invalidation occurs.

These new areas of responsibility are assigned appropriately between new classes, and shown (along with their significant interactions as external methods) with the related existing classes in the following class diagram, Figure 16.

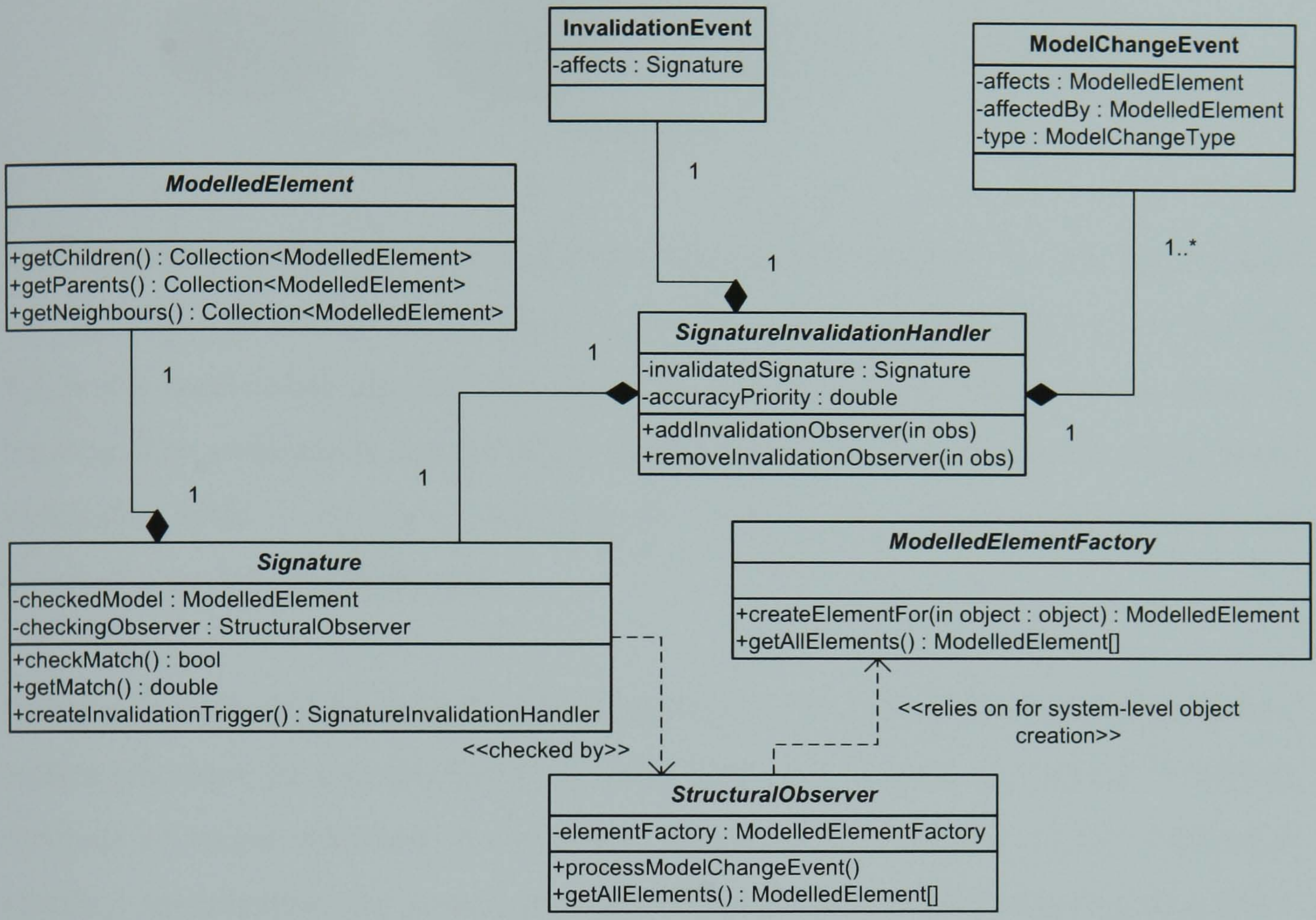
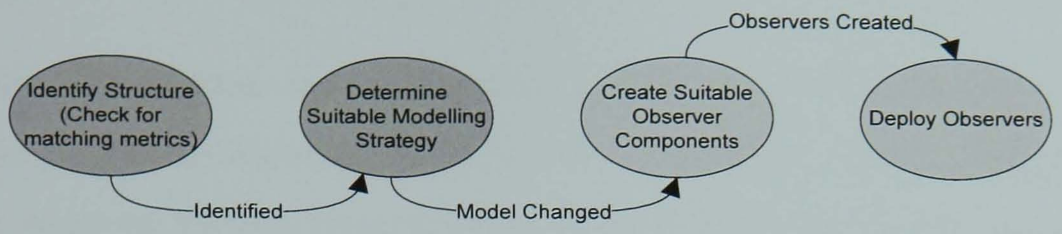

InvalidationEvent
-affects : Signature
ModelChangeEvent
-affects : ModelledElement
-affectedBy : ModelledElement
-type : ModelChangeType
ModelledElement
+getChildren() : Collection<ModelledElement>
+getParents() : Collection<ModelledElement>
+getNeighbours() : Collection<ModelledElement>
SignatureInvalidationHandler
-invalidatedSignature : Signature
-accuracyPriority : double
+addInvalidationObserver(in obs)
+removeInvalidationObserver(in obs)
Signature
-checkedModel : ModelledElement
-checkingObserver : StructuralObserver
+checkMatch() : bool
+getMatch() : double
+createInvalidationTrigger() : SignatureInvalidationHandler
ModelledElementFactory
+createElementFor(in object : object) : ModelledElement
+getAllElements() : ModelledElement[]
<<relies on for system-level object creation>>
<<checked by>>
StructuralObserver
-elementFactory : ModelledElementFactory
+processModelChangeEvent()
+getAllElements() : ModelledElement[]
1
1..*

Figure 16: Signature and Invalidation Handler - significant classes, relationships and methods

In this diagram; note how the existing classes `ModelledElementFactory` and `StructuralObserver` have gained methods to allow access to the collection of Modelled Elements that have been issued by a factory. As discussed in the previous section, this data coupling (*getAllElements*) is present as Mediation, avoiding direct coupling of a Signature object and a Modelled Element Factory implementation:

Section 5.2 has discussed the requirements of a structural signature and used an example to derive simple outline software designs. Before continuing to show how the signatures will be utilised by the structural observers, the next section will outline the design for the responses to the signature matches – the observation techniques.

5.3 Specifying Observation Techniques


Identify Structure (Check for matching metrics)
Identified
Determine Suitable Modelling Strategy
Model Changed
Create Suitable Observer Components
Observers Created
Deploy Observers

Referring back to the hard-coded example discussed in Section 5.1, the observation overlay deployed was decided during the code's design stage. As outlined in Section 4.3.3, this hard-coded link between signature and appropriate observation overlay is limiting. The architecture instead proposes the association of one or more observation techniques with a signature, deferring the decision on which technique to the observer's model generation unit.

In order to allow any given observation technique to be applied by a model generator, techniques must be specified in a standard form, customisable to specify different methods of target selection. As with the signature specification, which required a standard specification for system structure, observation technique specification must similarly make use of the structural model. Continuing the "typed observation" theme, while the framework proposes that observation techniques should still relate to one or more signatures, an observation technique should be capable of producing useful results when applied to systems that only *partly* conform to the signatures with which they are associated. In short, although techniques may produce poor results when applied to poor signature matches, they should still produce results.

An observation technique must identify a subset of target elements from the overall system. In order to do so, it may be beneficial if there is interaction – and potentially, a degree of data coupling – between the observation technique and the signature match that brought it about. Referring to Section 3.3.4, the acquaintance-based metric used to determine scale-free connectivity (as invoked by the signature) is similar to the acquaintance immunisation technique used to select the suggested target set. There is a benefit in allowing the observation technique to make use of the signature check method, and vice versa. Doing so can avoid the unnecessary duplication of metrics and increased maintenance, along with unnecessary repeated invocation of the algorithms – more computation to obtain the same results.

Additionally, observation techniques may need to prioritise elements within the resulting target set; as discussed in Section 4.3.4, observer deployment may need to reduce the target set further. Encapsulating data regarding the relative importance of a particular element within a technique's target set allows the deployment process to make better decisions regarding the actual target set (i.e. the observed elements).

As with signatures, discussed in Section 5.2, the technique demonstrates an effective use of the Template Method pattern, allowing the subclasses to implement the appropriate model simplification algorithms. To summarise, the Template specifying the observation technique should have the following main interface concerns:

1. To select a (reduced) **"suggested target" set** from the modelled system that can be passed to the deployment module to arrange the placing of observation units.
2. To **interact with the matched signature** that brought about the use of this technique. This interaction should facilitate:
    - Shared use of the "algorithm" specified within the signature (avoid algorithm duplication)
    - Shared use of the results of the signature's algorithm (avoid wasted computation)
3. Production of the target set should take the form of an **extension of the system model** to allow the included modelled elements to be specified with additional data – such as a degree of importance or priority for observation.

5.3.1 Acquaintance Immunisation Technique

Continuing the example of the scale-free model signature, this section examines the Acquaintance Immunisation technique for:

- Applicability as an observation technique,
- Fulfilment of previously listed requirements, and
- Generalisation to provide the basis for the observation technique specification.

As a brief reminder from Section 3.3.4 and its referenced publications, Acquaintance Immunisation [43] refers to a graph theory technique that is extremely effective in reducing the complexity in a scale-free graph. This brief overview will re-examine the previously identified objectives with particular consideration to the Acquaintance Immunisation Technique:

The suggested target set: is selected in accordance with the Acquaintance Immunisation technique, and is a collection of Modelled Elements to which the deployment unit can attach system-level observers.

Interaction with matched signature: as discussed in previous sections, there is significant re-use of algorithm and data between the Acquaintance Signature and Technique; thus a mechanism to permit this relationship is logical. However, the architectural design suggests a weakening of this relationship due to the potential many-to-many association between signatures and techniques. Therefore, to permit re-use without over-constraining the architectural relationship, three clear design responsibilities emerge: Algorithm, Signature and Technique. The relationship between them is shown in the UML class diagram, Figure 17.

Extension of the system model: Acquaintance Immunisation requires no additional metadata on the suggested target set; the algorithm does not specify importance between "immunised" nodes. However, it is conceivable that other modelling techniques may require methods to attach additional data to its suggested target set.

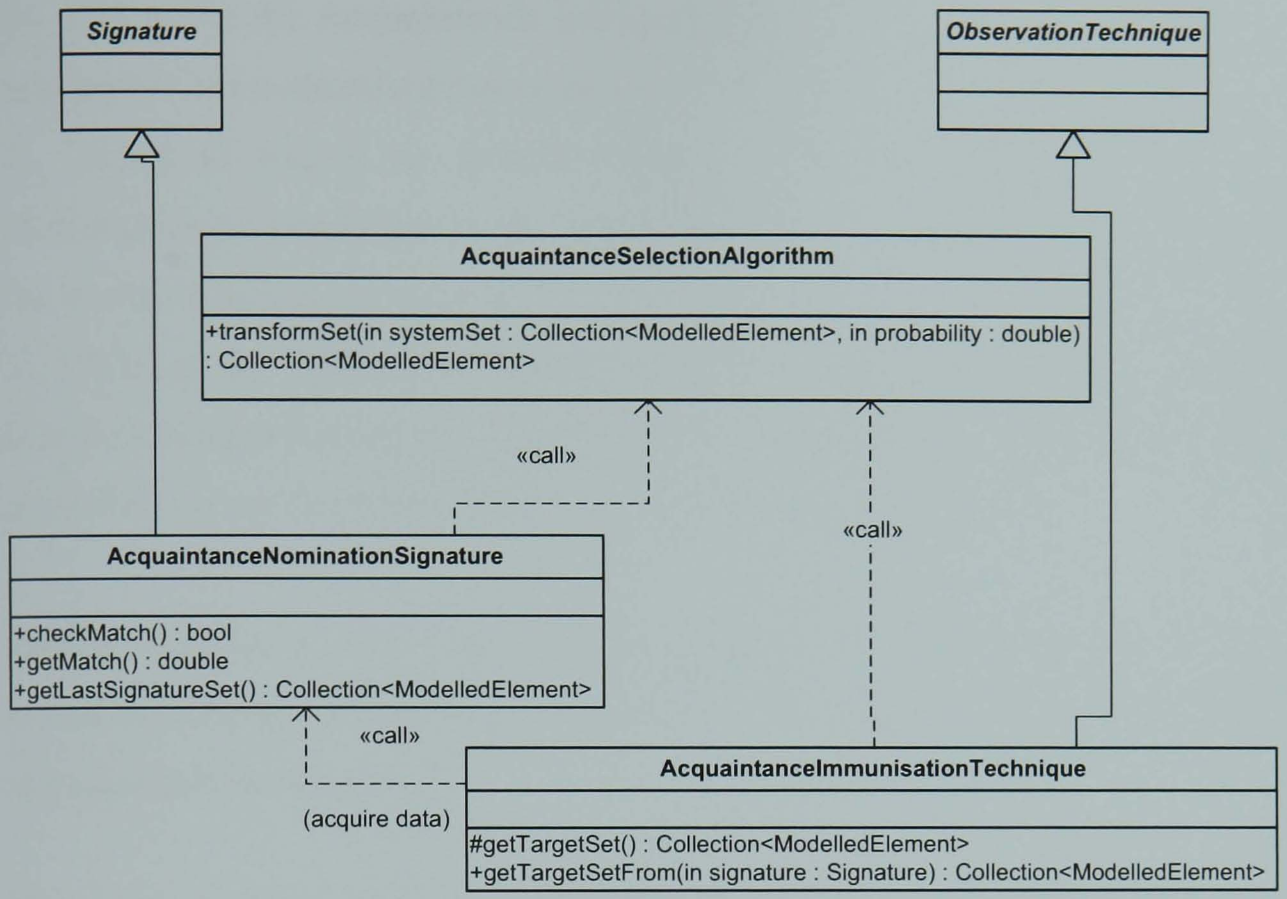

Signature
ObservationTechnique
AcquaintanceSelectionAlgorithm
+transformSet(in systemSet : Collection<ModelledElement>, in probability : double) : Collection<ModelledElement>
«call»
«call»
AcquaintanceNominationSignature
+checkMatch() : bool
+getMatch() : double
+getLastSignatureSet() : Collection<ModelledElement>
«call»
(acquire data)
AcquaintanceImmunisationTechnique
#getTargetSet() : Collection<ModelledElement>
+getTargetSetFrom(in signature : Signature) : Collection<ModelledElement>

Figure 17: Relationships between Acquaintance-based Signature, Technique and Algorithm

5.3.2 Modelling Generic Observation Techniques

Building on the findings of the previous section; this section will produce outline and generalised software designs, illustrated where necessary with UML class diagrams. In order to avoid duplicated diagramming, representations showing changes to existing classes will be limited to the changes or additions only.

Summarising previous proposals and design decisions, the place of Observation Techniques in the framework is as the first stage in an observation response to a particular structural signature match. They exist to:

- Provide or identify a set of target elements selected for observation from the entire system set of elements.
- Add any element-specific meta-data (such as the previously-mentioned example of observation priority) to particular modelled elements within the observation target set.
- Return the target set of modelled elements and meta-data to the observation co-ordinator or deployment unit in order that the recommended technique can be implemented as the blueprint for deploying system-level observers.

As outlined in the Acquaintance Immunisation examples, two significantly different approaches were identified that enable an Observation Technique to function:

1. Taking an output set from a related Signature, performing optional further examination or translation on the set of elements, and returning the translated set as the desired observation targets.
2. Utilising an Algorithm defined elsewhere and implementing a standardised interface or inheriting from a suitable abstract class to allow appropriate re-use of that algorithm (as per the Acquaintance "Selection" example)

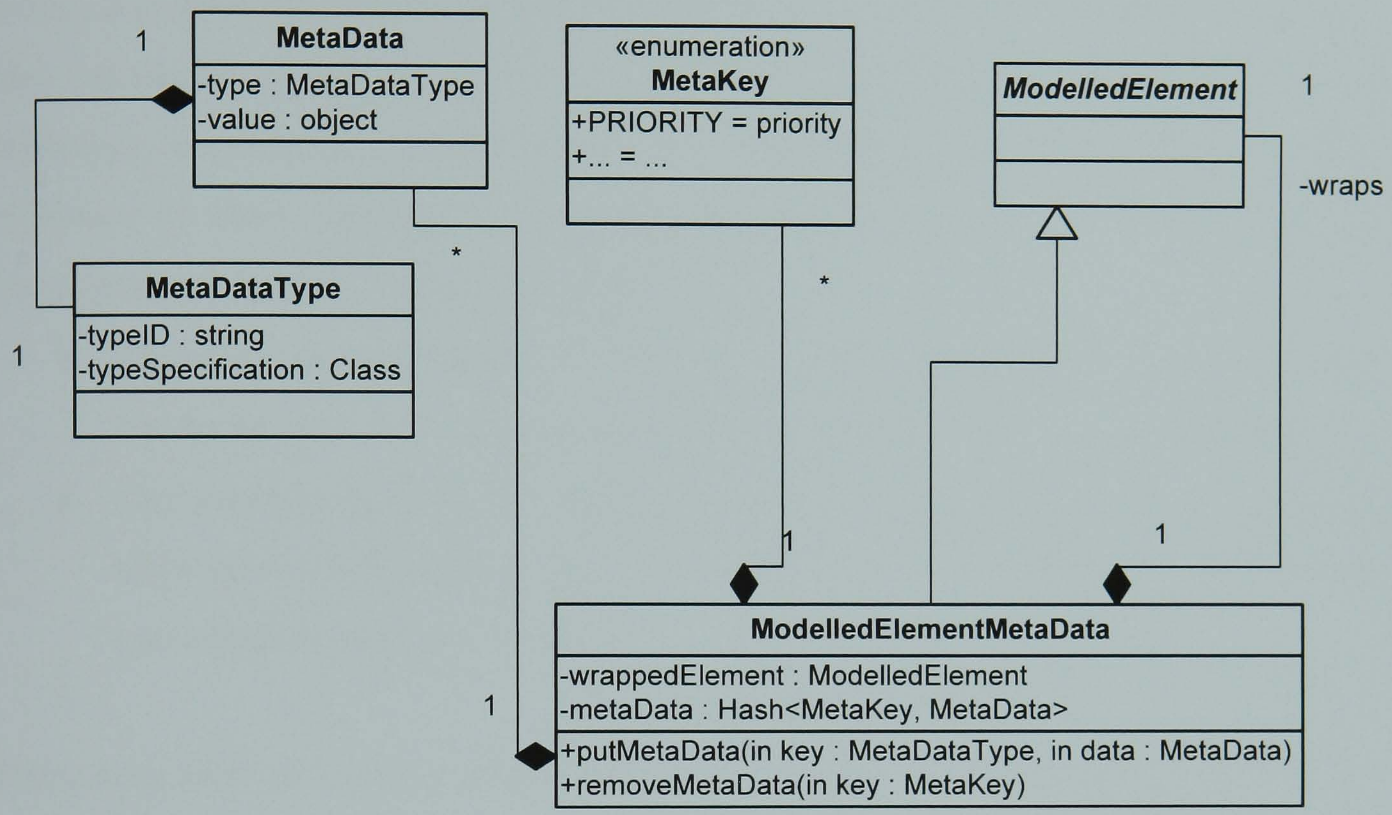
Before discussing the finer design decisions regarding algorithms and their interactions with techniques and signatures, the design for representing elements' metadata will be formalised.

Extending Modelled Element with Metadata

The element metadata concerns can be appropriately represented with a Decorator approach, which will contain any other relevant observation technique-added data and behaviour. Domain-specific metadata requirements will necessitate flexibility - in that metadata must not be constrained to a particular type – yet it must adequately describe itself to enable other units to use it.

The proposed design is based around a new type, `ModelledElementMetaData`, which is a special subtype of Modelled Element. It both inherits from Modelled Element (to allow it to appear in models without the need to further complicate the structural model) thus acts as both a Proxy and Adaptor, wrapping the "real" instances of Modelled Element. The associated data is represented by a Meta Data / Meta Key hash table structure, allowing Modelled Element to associate a number of pieces of metadata, each identified by a specific "key". In the example class diagram (Figure 18), a single key is defined – Priority – as discussed throughout the previous text. Domain-specific implementations could specify additional keys, allowing flexibility but ensuring that the metadata is identified in a standardised form across the framework.

Many OO languages allow runtime examination of an object's type (such as the `instanceof` keyword in Java), but values in the `MetaData` class are also type-specified by the `MetaDataType` class; another Adapter, allowing simplification of runtime types to a programmer-defined collection of acceptable metadata types.



MetaData
-type : MetaDataType
-value : object
«enumeration»
MetaKey
+PRIORITY = priority
+... = ...
ModelledElement
MetaDataType
-typeID : string
-typeSpecification : Class
ModelledElementMetaData
-wrappedElement : ModelledElement
-metaData : Hash<MetaKey, MetaData>
+putMetaData(in key : MetaDataType, in data : MetaData)
+removeMetaData(in key : MetaKey)
-wraps
1
1
*
*
1
1
1
1

Figure 18: Wrapper Modelled Element and supporting Meta Data representations

Algorithms, Signatures and Techniques

As discussed previously, there is a case for sharing both data and algorithmic logic between Signatures and Observation Techniques. Figure 17 showed an example of an overview class design that was applicable to the Acquaintance Immunisation Technique and Signature, and their shared algorithm. While the approach fulfilled the necessary requirements for that example, the specification was limiting as it relied on example-specific interactions between each of the main classes. In order to provide a design with wider application, and to assist developers in creating their own domain-specific designs, it is necessary to generalise these classes, and better specify the relationships between them. The rest of this section will discuss key design features of the example, generalise the feature and then demonstrate how it can be re-applied to the example in question.

The first feature concerns a signature's relevance to one or more techniques. The example showed how the Acquaintance Nomination signature defined a new method exposing the signature's "nominated set" of elements, allowing the Acquaintance Technique to interrogate it directly to determine if it needed to calculate the set itself (using a suitable algorithm), or whether it could simply use the set that the signature had calculated. However, this creates a strong relationship between signature and technique, something that the earlier sections had looked to remove to allow greater influence of other concerns on the mapping between signatures and techniques. As such, generalising this design must address:

- The associations (though not simply one-to-one mappings) between signatures and techniques (i.e. which techniques are applicable for certain signatures)
- The standard for transfer of data between a signature and one of its related techniques (remembering that a technique may be associated with more than one signature and vice versa)

Extracting both of these concerns into their own data-managing classes moves the responsibility for managing them away from Signature. Additionally, this allows a basic level of functionality to be implemented in these new classes that can be extended as required, rather than placing a dependency on any given signature. The new classes are shown in Figure 19. The new structure to manage associations between Signatures and Techniques is appropriately named as `SignatureTechniqueAssociator`. It is capable of storing associations between any number of signatures and techniques and returning a collection of techniques for a given signature and vice versa. A Façade method is added to Signature to allow a signature's related techniques to be determined from a specified association object. A similar approach is adopted for the matter of any Signature data that should be made available to other interested parties. A new class, `SignatureResults`, encapsulates a collection of Modelled Elements along with the Signature that produced it. It can therefore be extended to support other features independently of any signature; thus allowing new common types of result data to be created without having to duplicate definitions across different signatures. Accordingly, an abstract method has been added to the Signature class to get the result data from an instance of Signature.

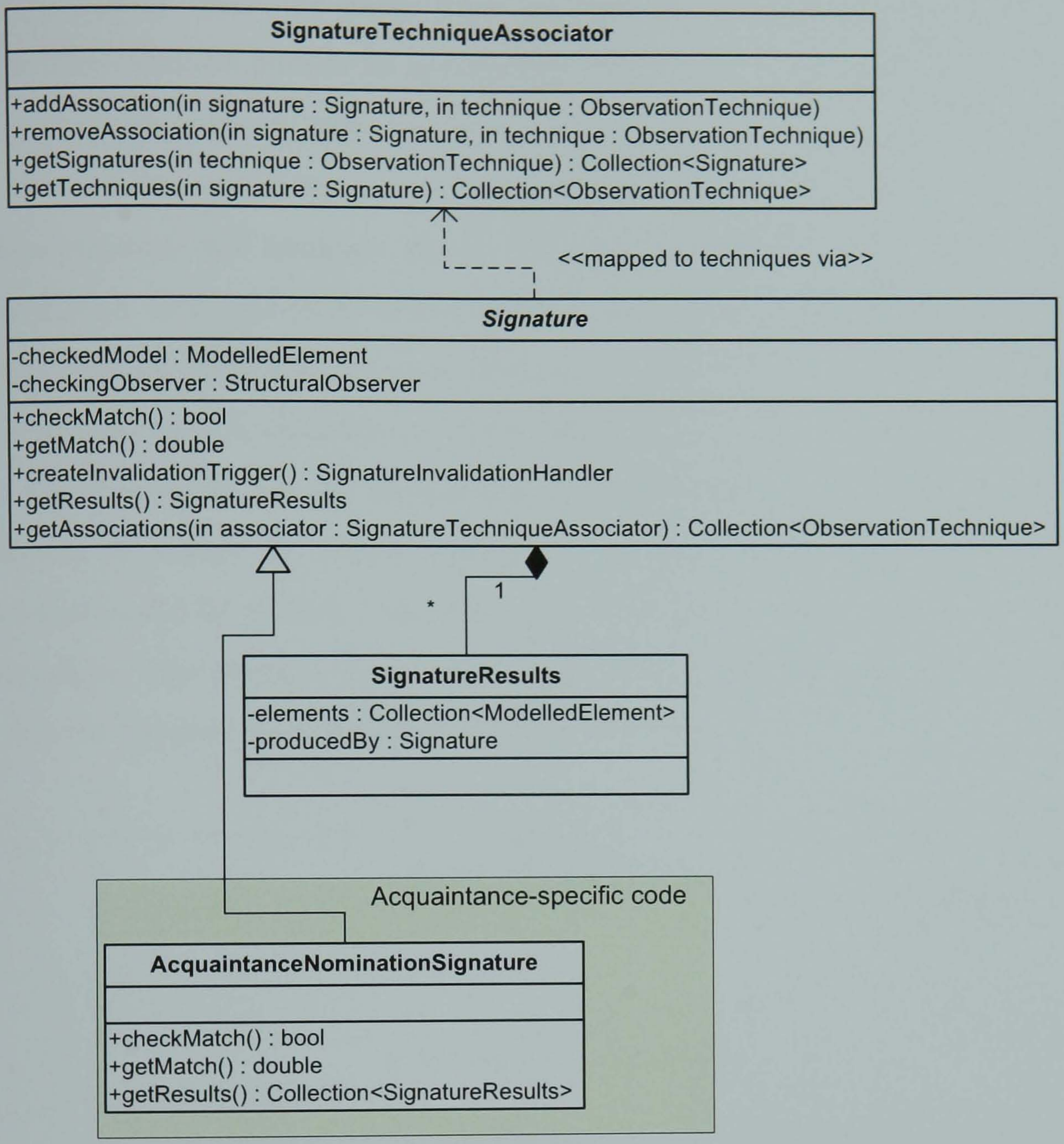

SignatureTechniqueAssociator
+addAssocation(in signature : Signature, in technique : ObservationTechnique)
+removeAssociation(in signature : Signature, in technique : ObservationTechnique)
+getSignatures(in technique : ObservationTechnique) : Collection<Signature>
+getTechniques(in signature : Signature) : Collection<ObservationTechnique>
<<mapped to techniques via>>
Signature
-checkedModel : ModelledElement
-checkingObserver : StructuralObserver
+checkMatch() : bool
+getMatch() : double
+createInvalidationTrigger() : SignatureInvalidationHandler
+getResults() : SignatureResults
+getAssociations(in associator : SignatureTechniqueAssociator) : Collection<ObservationTechnique>
*
1
SignatureResults
-elements : Collection<ModelledElement>
-producedBy : Signature
Acquaintance-specific code
AcquaintanceNominationSignature
+checkMatch() : bool
+getMatch() : double
+getResults() : Collection<SignatureResults>

Figure 19: Revised Generic Signature and support classes for Results and Associations along with Acquaintance Nomination-specific example

---

The second feature concerns the extraction of common algorithms into separate code shared by both signatures and techniques. The Acquaintance Immunisation example introduced a new class; `AcquaintanceSelectionAlgorithm` defining a method *`transformSet`*, requiring two parameters – the whole collection of Modelled Elements and the probability value determining the set size in acquaintance selection. This method encapsulated the algorithm required for Acquaintance-based selection and returned the collection of Modelled Elements selected through Acquaintance Immunisation. However, with other Signatures and Techniques, it is impossible to know what parameterisation they will require. Equally, including this parameterisation in the algorithm access method means that re-use of an algorithm on a general basis elsewhere is rendered impossible due to incompatible interfacing.

The solution takes the same form as used for Signatures and Techniques; the Template Method pattern to specify the interfacing of the algorithm – the method `transformSet` in a new `ModelledElementAlgorithm` interface. This takes a collection of `ModelledElement` objects as a parameter and returns another collection. Implementing this interface allows for common algorithms or element collection-translation functions to be extracted into another class, allowing reuse of the same algorithm by both a `Signature` and `ObservationTechnique` object. Any required parameterisation is included not in the call to the `transformSet` method, but instead in the constructor for the appropriate implementation of the algorithm interface, as shown in Figure 20. This separation approach allows simple Signatures and Techniques to be generic code units, specialised only by inclusion of an appropriate algorithm. The example shows how Acquaintance Selection Algorithm's constructor takes the required parameterisation (the probability).

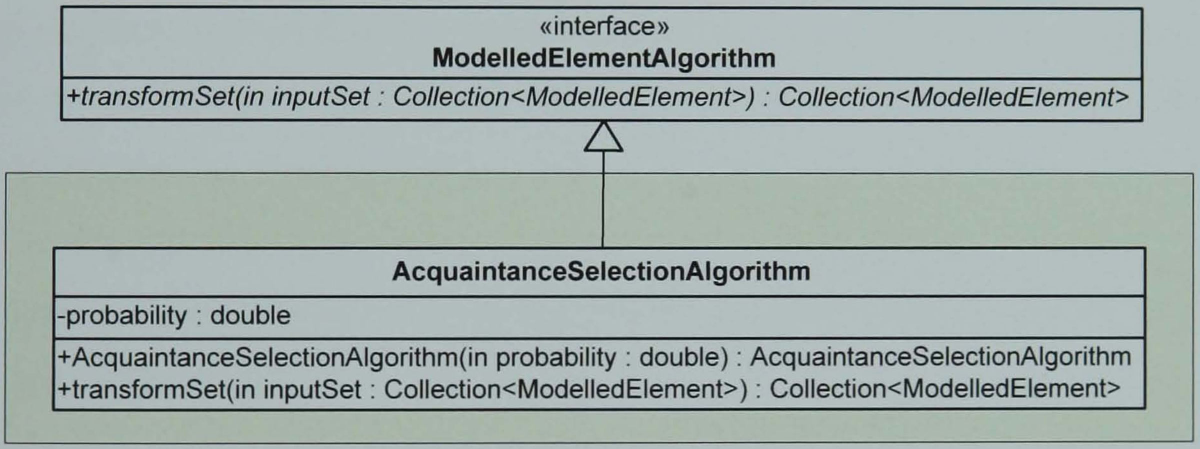

«interface»
ModelledElementAlgorithm
+transformSet(in inputSet : Collection<ModelledElement>) : Collection<ModelledElement>
AcquaintanceSelectionAlgorithm
-probability : double
+AcquaintanceSelectionAlgorithm(in probability : double) : AcquaintanceSelectionAlgorithm
+transformSet(in inputSet : Collection<ModelledElement>) : Collection<ModelledElement>

Figure 20: Generalised Modelled Element Algorithm Interface and Example Implementation

---

The third and final feature under consideration is defining the observation technique itself, making use of the new additions. The Acquaintance Immunisation example defined `AcquaintanceImmunisationTechnique` – encompassing the required logic to generate the required target set. The entire calculation logic is in the algorithm-containing code and the related signature matching set, if available. As such, the technique made simple decisions based on the availability of the signature match data, and returned the appropriate data via its relationships with these objects. Figure 17 also showed this class inheriting from a hypothetical new `ObservationTechnique` abstract class, which acted as a marker in the diagram for the required generalisation. The remainder of this section will complete this generalisation process.

The first stage is to examine the basic responsibilities of the technique. Remembering that an observation technique must produce a suitable target set for deployment of observers and it has the following resources available to it:

- The complete collection of modelled elements that make up the system from which targets should be selected.
- Zero or more Signature Result objects that can be used as a base collection to translate further or to form the result "as-is".
- One or more Modelled Element Algorithms that can be used to translate a collection of modelled elements into only those elements identified as targets.
- Modelled Element Metadata (and its supporting classes) to allow the addition of deployment information to any of the elements in the target set.

From a basic interface perspective, an observation technique needs to define only a single method that returns the selected target element set, taking a specified system-wide element collection as a parameter. The necessary algorithmic logic to perform this translation, as discussed previously, can be specified in a suitable implementation of `ModelledElementAlgorithm`. Therefore, the simplest generalisation that will fulfil the basic requirements must be composed of a suitable algorithm and define a method that returns the target set.

This generalisation is shown in Figure 21 below as a new class `ObservationTechnique`. Note how as with Signature, a Façade method, *`getAssociations()`* is provided to allow other objects to determine which signatures are associated with this technique, according to the specified association object. However, unlike `Signature`, `ObservationTechnique` is shown as a concrete class. Its constructor requires a `ModelledElementAlgorithm`, so instances can perform basic functions without further sub-classing. This default implementation of the *`getTargetSet()`* method will simply pass the specified collection of modelled elements to the algorithm specified in the constructor and return the result.

Therefore, in order to demonstrate how this generalised class can perform more advanced functionality, as required by the Acquaintance Immunisation technique, this relationship is also shown in the class diagram. This Acquaintance Immunisation-specific class subclasses `ObservationTechnique` and provides two constructors; one requiring an instance of `SignatureResults`, and one with no parameters. Both constructors create an instance of `AcquaintanceSelectionAlgorithm` and pass it to their super-constructor. The method that generates the target set – *`getTargetSet`* – is also overridden to check if a `SignatureResults` object has been specified in the constructor. If so, it returns the modelled element collection from the results object. Alternatively, it calls its namesake method in the super-class, which results in the use of the specified algorithm - Acquaintance Selection.

Other domain-specific concerns, such as the addition of metadata to modelled elements, or the amalgamated use of several different algorithms could be specified easily in further-specified subclasses of Observation Technique, whilst retaining a standardised interface for use in a deployment unit; thus utilising the Template Method form of the technique.

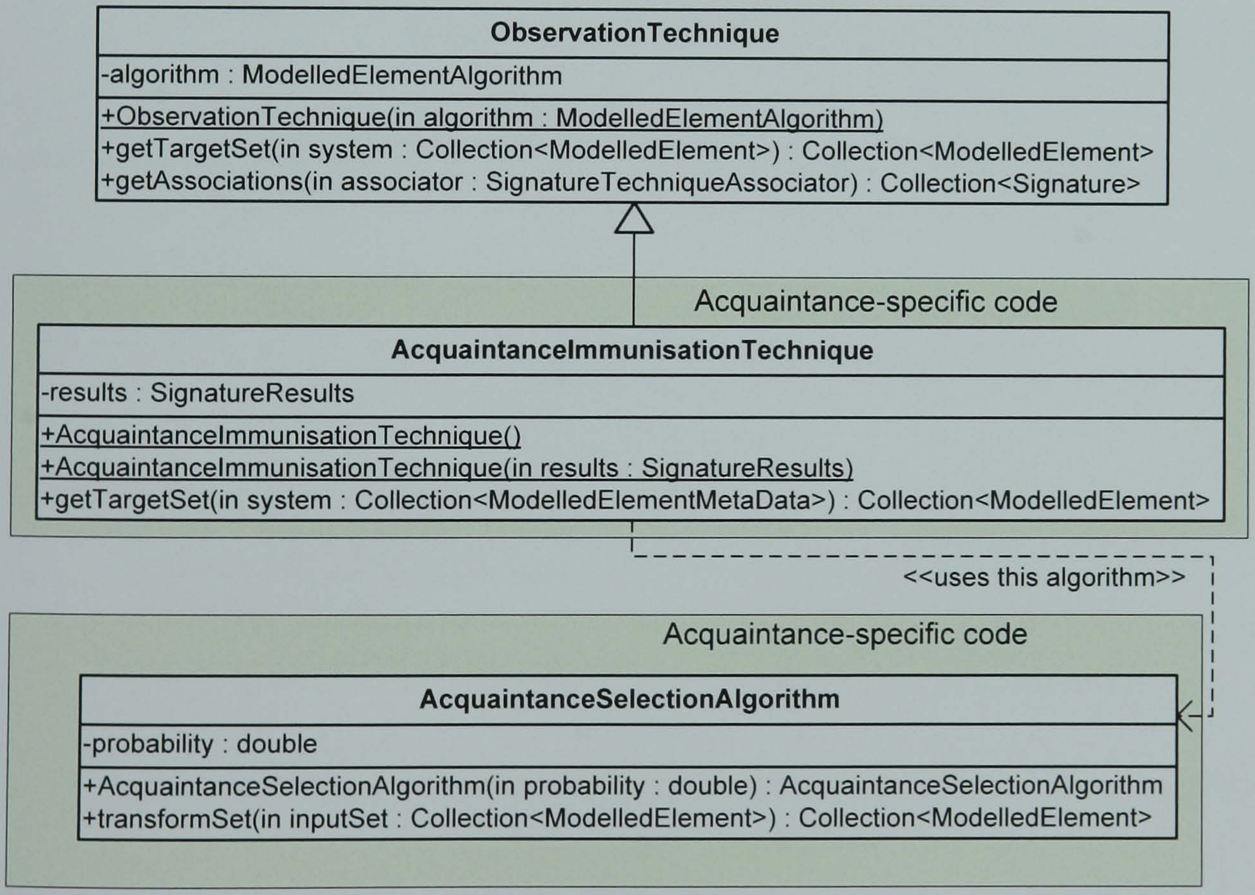

ObservationTechnique
-algorithm : ModelledElementAlgorithm
+ObservationTechnique(in algorithm : ModelledElementAlgorithm)
+getTargetSet(in system : Collection<ModelledElement>) : Collection<ModelledElement>
+getAssociations(in associator : SignatureTechniqueAssociator) : Collection<Signature>
Acquaintance-specific code
AcquaintanceImmunisationTechnique
-results : SignatureResults
+AcquaintanceImmunisationTechnique()
+AcquaintanceImmunisationTechnique(in results : SignatureResults)
+getTargetSet(in system : Collection<ModelledElementMetaData>) : Collection<ModelledElement>
<<uses this algorithm>>
Acquaintance-specific code
AcquaintanceSelectionAlgorithm
-probability : double
+AcquaintanceSelectionAlgorithm(in probability : double) : AcquaintanceSelectionAlgorithm
+transformSet(in inputSet : Collection<ModelledElement>) : Collection<ModelledElement>

Figure 21: Observation Technique class and example concrete implementation

Piecing together the new classes and additions provides a complete overview of the design to support the Observation Technique functionality. The class diagram in Figure 22 below shows all the newly-introduced classes along with their important methods, relationships to and dependencies on other classes. In this diagram, the previously-used Acquaintance Immunisation examples have been removed for clarity.

The faint dependencies on the `ModelledElementMetaData` class indicate that the classes may make use of the metadata additions if required.

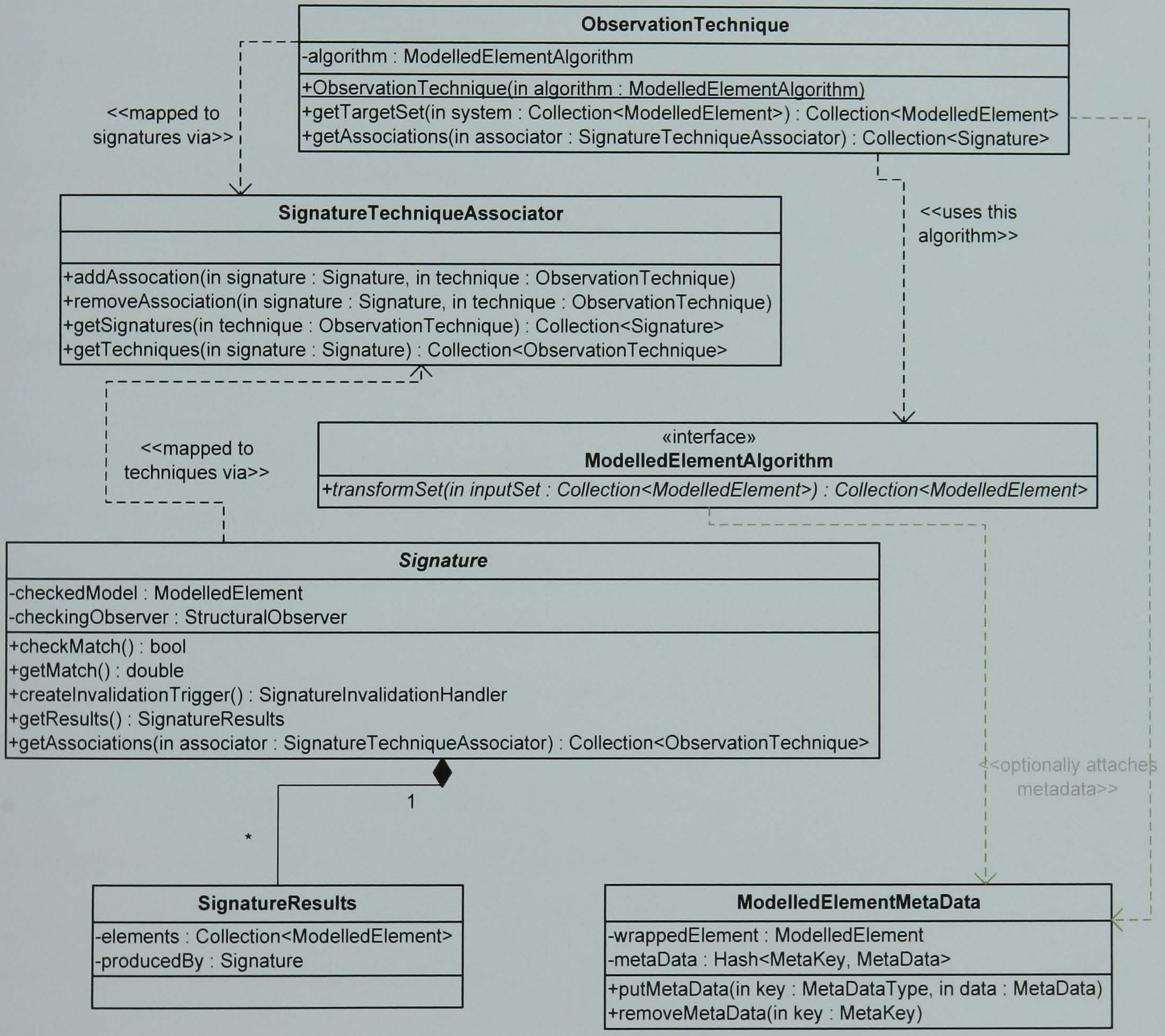

ObservationTechnique
-algorithm : ModelledElementAlgorithm
+ObservationTechnique(in algorithm : ModelledElementAlgorithm)
+getTargetSet(in system : Collection<ModelledElement>) : Collection<ModelledElement>
+getAssociations(in associator : SignatureTechniqueAssociator) : Collection<Signature>
<<mapped to signatures via>>
<<uses this algorithm>>
SignatureTechniqueAssociator
+addAssocation(in signature : Signature, in technique : ObservationTechnique)
+removeAssociation(in signature : Signature, in technique : ObservationTechnique)
+getSignatures(in technique : ObservationTechnique) : Collection<Signature>
+getTechniques(in signature : Signature) : Collection<ObservationTechnique>
<<mapped to techniques via>>
«interface»
ModelledElementAlgorithm
+transformSet(in inputSet : Collection<ModelledElement>) : Collection<ModelledElement>
Signature
-checkedModel : ModelledElement
-checkingObserver : StructuralObserver
+checkMatch() : bool
+getMatch() : double
+createInvalidationTrigger() : SignatureInvalidationHandler
+getResults() : SignatureResults
+getAssociations(in associator : SignatureTechniqueAssociator) : Collection<ObservationTechnique>
1
*
<<optionally attaches metadata>>
SignatureResults
-elements : Collection<ModelledElement>
-producedBy : Signature
ModelledElementMetaData
-wrappedElement : ModelledElement
-metaData : Hash<MetaKey, MetaData>
+putMetaData(in key : MetaDataType, in data : MetaData)
+removeMetaData(in key : MetaKey)

Figure 22: Overview of Observation Technique, support classes and relationships

5.4 Summary

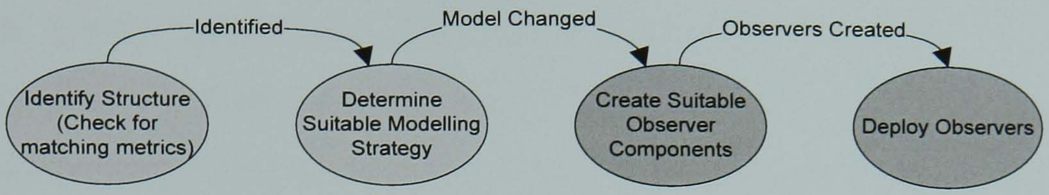
Chapter 5 developed the design of the building blocks of the observation framework, with particular attention to the structural part of the framework. This involved a decomposition of the framework into three main design areas – system modelling, structural characteristic signatures, and observation techniques. The design has been undertaken wherever possible using well-established software engineering techniques and design patterns to solve the problems of how to assign components' responsibilities into classes and interface specifications.

The design has been presented largely as UML class diagrams with surrounding text explaining significant decisions and any implementation issues of note. In order to try and demonstrate the validity of the designs and to augment the basic class diagrams, the designs of the low-level elements have been applied in the associated Appendix I.

It is hoped that this has shown how some of the general designs could be specialised to deal with system requirements. Detailed implementation instructions have been omitted, firstly in the interests of design clarity, and secondly as some of the significant implementation issues relate to the element-specific instrumentation required for the Modelled Element-derivative adapter classes. Outline software designs specifying the structural model's representation are introduced in Section 5.1. They are developed to show how Signatures can specify desired structural characteristics in Section 5.2. The final part of the chapter, Section 5.3, specified the design for techniques that can be used in response to these model and signature matches, in order to guide deployment of observers and provide efficient overlay coverage.

The next chapter will piece together the different units by examining the way in which observation policies are defined.

Chapter 6 – Assembling the Observation Model


Identified
Model Changed
Observers Created
Identify Structure (Check for matching metrics)
Determine Suitable Modelling Strategy
Create Suitable Observer Components
Deploy Observers

As discussed throughout the previous chapter, a significant aim of the observation model is to allow operation on complex systems that exhibit only partial or uncertain system information. This is achieved by the development of an adaptive system that relies only on "typing" characteristics, as discussed in Chapter 5, rather than examining the system model for specific components or a prescribed structure.

Therefore, a necessity in the observer model is a provision for system-specific observation requirements. These requirements may operate in conjunction with the findings of the rest of the observation framework, or may provide a fixed specification that overrides certain areas of the framework's recommendations – such as system elements that must (or cannot) be observed. This chapter will build on the requirements identified in the previous chapter, integrating provision for the system-specific requirements, along with assembling the observation model from the previously defined building blocks.

6.1 Specifying Observation Policies and Process

The scope of an observation policy can range in complexity. It may be as simple as a basic constraint on the amount of resources that are available for observation, or may extend to the specification of external observation concerns. It is impossible to define the specification of such varied policies without resorting to the definition of a full rule/policy-based interpretation system.

It is outside the scope of this thesis to define a policy-based system; there is already active research and practice established (e.g. [23]) in this area. Equally, there are concerns regarding scalability of rule-based systems when applied to complex domains, along with real-time performance issues.

However, the framework must detail how this information can be included in the system's considerations, along with the classes of policy that can be induced upon the rest of the observation framework, and the implications of this additional level of indirection on model creation and observer deployment. The major components needed for policy integration were defined throughout Chapter 5. Significant components such as Signatures and Techniques have been designed in order to promote code reuse and to encourage a separation of concerns and avoid direct one-to-one mapping. This section explains how they join together and provide clear instructions to the deployment process. This must take responsibility for attaching the system-level observers to the real system element, which are abstracted at this level by their Modelled Element representation.

6.1.1 Situating the Policies

Before examining the potential makeup of the policies, it is important to determine their place in the system. Chapter 5 concentrated on the makeup of the significant components that were established in Chapter 4; however, this chapter needs to revisit details of the components' interaction. Further exploring those relationships will help gain a proper understanding of how policy checking controls observation response and technique.

Referring back to Chapter 4 shows that policy control is required within the following elements of the system:

Selection of appropriate observation technique – Section 4.3.3 suggests that system policy, signature results and reasoning are used to determine which Observation Technique is selected as an appropriate response to a signature match.

Deployment of system-level observers – Section 4.3.4 shows that system policy, combined with an appropriate evaluation of current system states (e.g. resource availability) will control the deployment of system-level observers.

Within Chapter 5, basic software designs have been produced for the structural modelling interfaces, along with the signature and technique specifications. While it was possible to design the earlier components in relative isolation within the architecture – only specifying the data they require and that they must produce –

specifying policies requires a more inter-component cohesive approach. Reasoning and decision-making are key aspects of the policy-based areas of the system and as such, it is important to understand the events that lead to policy decisions being made and the outcomes of these decisions. As such, defining the policy and decision making component will necessarily affect the designs of other components; requiring greater specification in terms of how the components will function together as a whole system.

6.1.2 Observation Technique Selection Process

The first policy aspect under consideration is that regarding selection of appropriate Observation Techniques. As discussed in Sections 4.3.3 and 5.3; Signatures can be associated with several Observation Techniques.

The proposed architecture indicated that system-level observation need not be limited to the output from a single Observation Technique, and if applicable, the output from several techniques could be either combined or deployed as several different overlays. The architectural overview does not specify constraints for technique-selection decision logic; however, selected techniques may be chosen according to the quality of the triggering signature match (i.e. the numeric value), design-time guidance (policy), comparisons of different signature result sets, and potentially access to other signatures to determine if there are other signature matches or near matches. Before considering any detail regarding specification of policy itself, it is useful to review the data requirements for the Observation Technique Selection component. It should generate a list of Observation Techniques selected according to criteria within the selection component. The criteria may include system policy, signature match quality, or a comparison of various signatures and their related techniques.

In order to do so, it will require:

- Suitable decision-making information, incorporated entirely within the component, specified in system policy, or a combination thereof.
- The matched `Signature` instance and notification of a signature match
- A suitable instance of `SignatureTechniqueAssociator` providing system-specific associations between signatures and techniques

- Access to other system signatures for comparison of matches and near matches

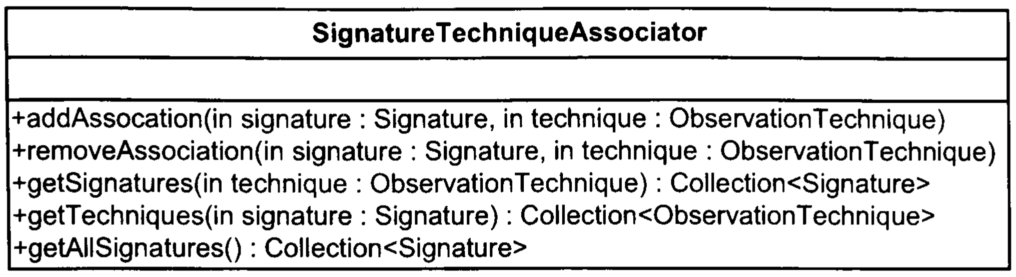
Some of these data requirements are met by existing classes: Instances of `SignatureTechniqueAssociator` provide the Technique selection process with a set of `ObservationTechnique` objects that are associated with any given `Signature`. For clarification, it would be entirely possible to have an observer system operating from a single `SignatureTechniqueAssociator` instance; the intention being that different instances could provide associations for the differing needs of individual overlay types. Equally, instances of the `Signature` class provide sufficient information for the reasoning or decision-making process to function. The interface specifies method access for basic signature requirements such as match checking, and returning the "quality" of a match. Additionally, runtime type checking can provide information about the exact signature subtype that was matched, allowing signature-specific policy (in addition to `SignatureTechniqueAssociator`) to be included in the Technique Selection process.

However, there are new issues requiring further specification. They can be divided into two areas. The first concerns signatures and encompasses issues such as the need for notification upon a signature match, and enumeration of all signatures. The second is that of the decision making process involved in selecting one or more techniques in response to a given signature match.

Support: Signature Enumeration and Match Notification

While the signature interface was defined in Section 5.2.2, the technique selection method requires additional signature-related information. Firstly, signatures within the system must be made available. Secondly, the technique selection process must be notified when a signature is matched, as this is when the selection process must begin.

Given that the `SignatureTechniqueAssociator` class makes provision for associating Signatures with available Techniques and vice versa, it is logical to extend its functionality to make available *all* the system's signatures through this class. The method will return all the signatures available through the association class; its revised class method signature is shown in Figure 23:



```
SignatureTechniqueAssociator

+addAssocation(in signature : Signature, in technique : ObservationTechnique)
+removeAssociation(in signature : Signature, in technique : ObservationTechnique)
+getSignatures(in technique : ObservationTechnique) : Collection<Signature>
+getTechniques(in signature : Signature) : Collection<ObservationTechnique>
+getAllSignatures() : Collection<Signature>
```

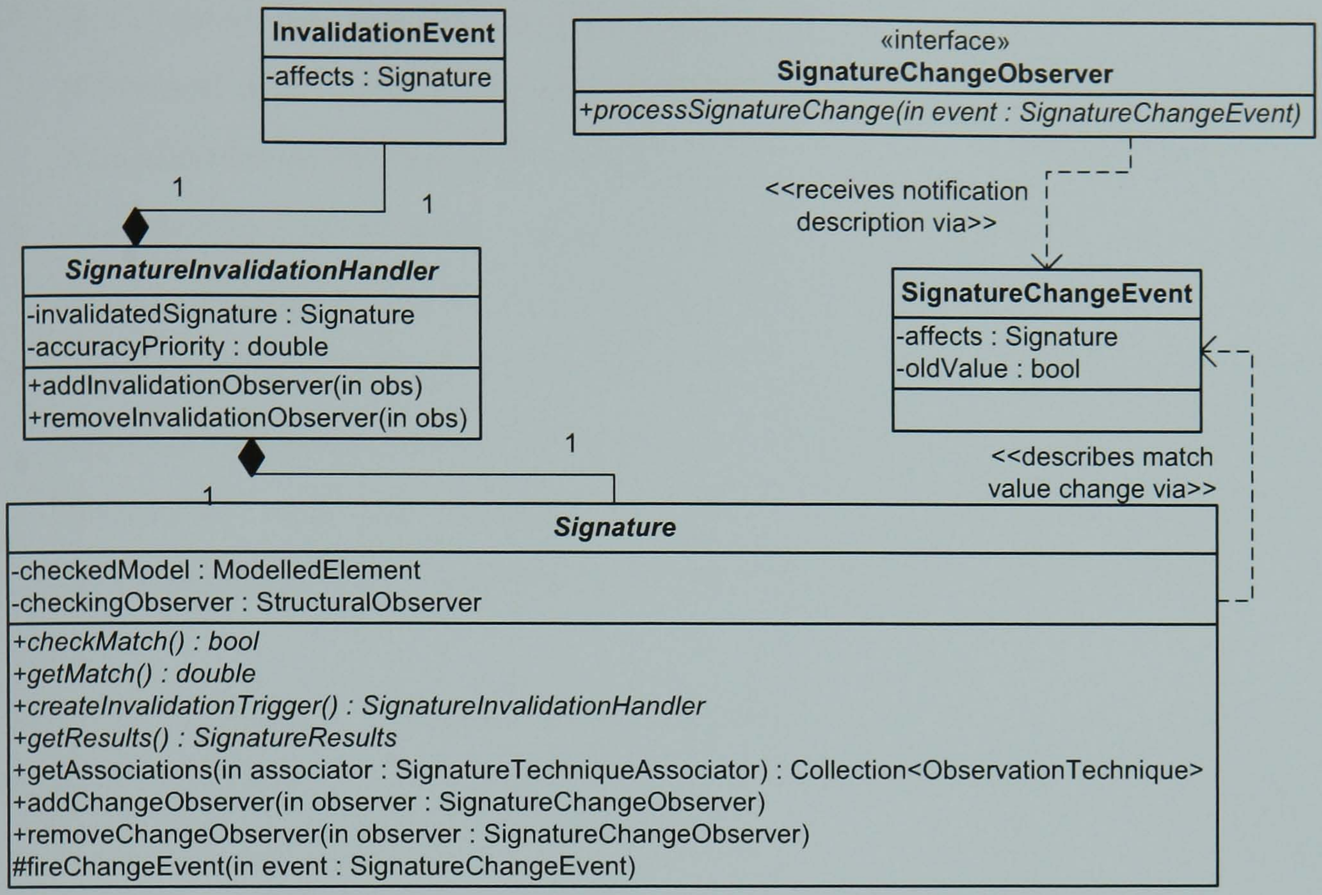
Figure 23: Simply-revised Signature and Technique Association class

Detail surrounding signature match notification has so far been limited, with designs centred on the `SignatureInvalidationHandler` class and its invalidation model. Instances of this class are associated with a given signature, and receive model change events from the set of modelled elements associated with a structural observer. They generate `InvalidationEvents` when the associated signature should be *rechecked.* However, this approach places the onus of signature rechecking with the items that are interested in its status, and does not provide change notification events describing when a signature's match status *has altered.*

To avoid several interested parties having to individually check the status of the signature, support is required to allow a signature to be observed for change. This requires little more than a suitable event-describing class, the observer interface, and suitable methods to add and remove observers in the `Signature` class. The Observer design pattern is a common pattern; a brief class diagram showing the basics of its application is contained in Figure 24, alongside existing definitions for `InvalidationEvent` and related interfaces. The new event description is contained in the `SignatureChangeEvent` class, while `SignatureChangeObserver` defines one method; a receiver for instances of `SignatureChangeEvent`.

The Technique Selection process can now fulfil the requirements as follows:

- Notification of matched `Signature` – by implementing the `SignatureChangeObserver` interface and registering as a listener on all signatures the process is expected to respond to.
- It can now access other available system signatures for comparison purposes by having a reference to the appropriate `SignatureTechniqueAssociator` instance, which can now enumerate all associated system signatures, and provide the Technique associations for a given signature.

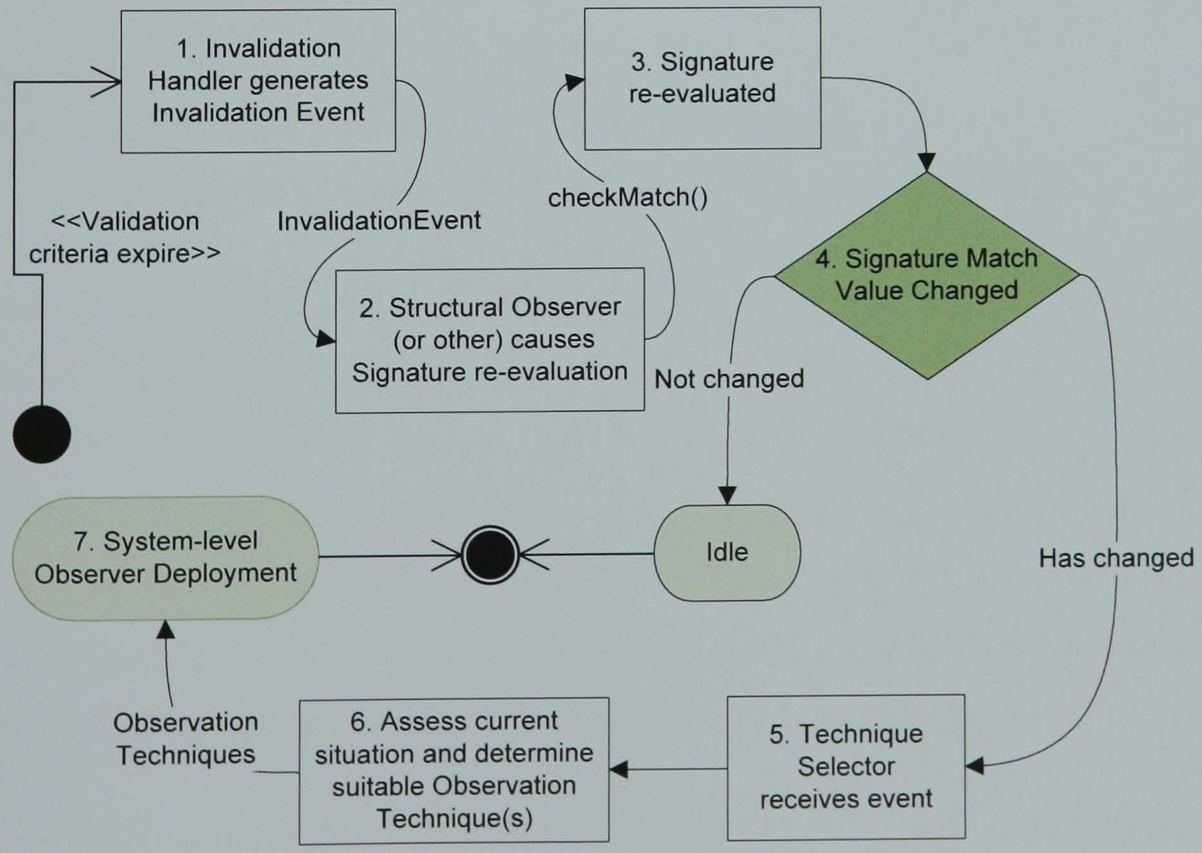


InvalidationEvent
-affects : Signature
«interface»
SignatureChangeObserver
+processSignatureChange(in event : SignatureChangeEvent)
1
1
<<receives notification description via>>
SignatureInvalidationHandler
-invalidatedSignature : Signature
-accuracyPriority : double
+addInvalidationObserver(in obs)
+removeInvalidationObserver(in obs)
SignatureChangeEvent
-affects : Signature
-oldValue : bool
1
1
<<describes match value change via>>
Signature
-checkedModel : ModelledElement
-checkingObserver : StructuralObserver
+checkMatch() : bool
+getMatch() : double
+createInvalidationTrigger() : SignatureInvalidationHandler
+getResults() : SignatureResults
+getAssociations(in associator : SignatureTechniqueAssociator) : Collection<ObservationTechnique>
+addChangeObserver(in observer : SignatureChangeObserver)
+removeChangeObserver(in observer : SignatureChangeObserver)
#fireChangeEvent(in event : SignatureChangeEvent)

Figure 24: Signature design modified for Change Observer

Selecting Techniques

The actual technique-selection policy logic is a system-specific operation and as such, this design will only specify the key process operations. In order to recap and clarify events, the process established thus far is shown in Figure 25:



1. Invalidation Handler generates Invalidation Event
3. Signature re-evaluated
<<Validation criteria expire>>
InvalidationEvent
checkMatch()
4. Signature Match Value Changed
2. Structural Observer (or other) causes Signature re-evaluation
Not changed
7. System-level Observer Deployment
Idle
Has changed
Observation Techniques
6. Assess current situation and determine suitable Observation Technique(s)
5. Technique Selector receives event

Figure 25: Summary of contributory processes to Observation Technique Selection

The key points include:

- A `SignatureInvalidationHandler` triggers an `InvalidationEvent`, processed at an invalidation event observer (e.g. Structural Observer), causing re-evaluation of the invalidated signature.
- If signature re-evaluation causes a change in the match value, it notifies its listeners, including one or more Technique Selectors.
- The Technique Selector queries signature information, current system state or relevant variables, then determines the appropriate technique from those associated with the matched signature. Once applicable techniques are determined, they are passed to System Level Deployment (Section 6.1.3)

Therefore, Technique Selectors must implement the `SignatureChangeObserver` interface. Secondly, they require a `SignatureTechniqueAssociator` object to determine fit techniques. A class diagram is shown in Figure 26, outlining basic requirements a `TechniqueSelector` must fulfil.

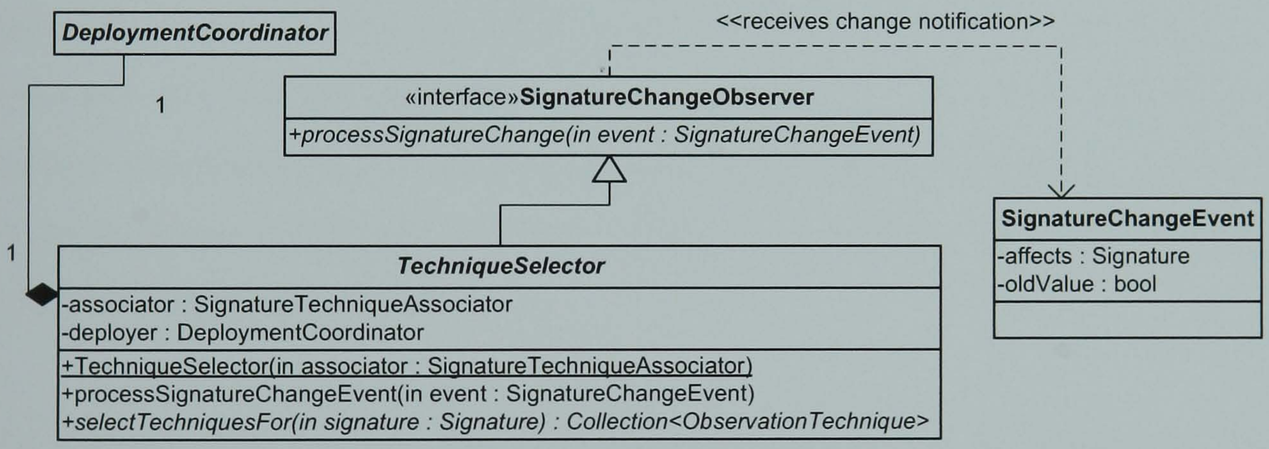

DeploymentCoordinator
<<receives change notification>>
1
«interface»SignatureChangeObserver
+processSignatureChange(in event : SignatureChangeEvent)
1
TechniqueSelector
-associator : SignatureTechniqueAssociator
-deployer : DeploymentCoordinator
+TechniqueSelector(in associator : SignatureTechniqueAssociator)
+processSignatureChangeEvent(in event : SignatureChangeEvent)
+selectTechniquesFor(in signature : Signature) : Collection<ObservationTechnique>
SignatureChangeEvent
-affects : Signature
-oldValue : bool

Figure 26: Basic Technique Selector class definition and relationships

The diagram demonstrates the relationship between classes, utilising the Template Method design pattern to permit concrete subclasses to determine technique selection. This is illustrated via the *`selectTechniquesFor()`* method, which must be implemented according to the subclasses' own policies. The diagram also notes the dependent relationship with the to-be-defined `DeploymentCoordinator`, which is acting as a placeholder for the system-level deployment class(es). The next subsection will discuss the requirements for the system-level deployment module.

6.1.3 System-level Observer Deployment Logic

Finally, the policy logic applicable to system-level observer deployment will be examined and refined. As from the architecture presented in Section 4.3.4, significant decision-making regarding system-level observer deployment is deferred to a separate component – i.e. it is not the responsibility of Observation Techniques to specify exactly how system-level observers should be attached to the underlying components.

System-level deployment is tightly connected to the previous stage along with element-specific technical issues. In order to best separate concerns, the intention is that components in the previous stage deal with *structural* system concerns – signatures, techniques, and the mapping policy. Concerns for observer deployment are centred on the current state of the *observed* system; resource availability, current observer deployments, and management of system observer units. As with Modelled Elements in Section 5.1.1, it is not the concern of these designs to set out the element-specifics; rather specifying a suitable interface for the system-level Deployment Co-ordinator that can be implemented at a component level. This co-ordinator will manage operations such as observer deployment, un-deployment, and provide control to applications of different observer techniques resulting in different sets of system observer operating together.

As with previous sections, the first stage of this component's design is to determine the data it must be provided with, and the output it should produce. The previous section established that Technique Selection processes will provide the system-level Deployment Co-ordinator with one or more techniques that have been identified as appropriate to use in target selection for the system-level observers.

System-specific concerns

Referring back to Section 5.1.1 and the example in Appendix I (Section I.I); the bridge between system-level elements / components and structural-level model elements is represented by instances of the `ModelledElement` class. Therefore, in order to keep access to bridging functionality in a single location, it is sensible to require concrete subclasses of `ModelledElement` to implement the actual system-level component-specific deployment logic.

This permits the Deployment Co-ordinator to rely on simple `deploy()` and `undeploy()` calls to manage creation and destruction of system-level observers. However, there are a number of shortcomings with viewing this as a simple one-to-one deployment relationship:

- Fault reporting mechanism – as the deployment mechanism is the bridge to the system-level components, it is possible that this mechanism could fail for system specific reasons. A simple example of a fault is: the real component represented by the Modelled Element becomes faulty and disconnects.
- Multiple observer deployments for a single Modelled Element – by permitting only deploy/un-deploy commands from the Modelled Element, this prevents a number of System Observers being created to represent the element in different overlays.

Therefore, Modelled Element will control deployment, but the resulting system-level observers will be responsible for handling un-deployment requests. In Figure 27, the `SystemLevelObserver` abstract class defines an *`undeploy()`* method; allowing a one-to-many element→system-level observer relationship. Additionally, the deploy operation facilitates a form of fault report, via the Exception model as shown in Figure 27. The *`deploy()`* method throws a `DeploymentException` to indicate that the deployment did not succeed; allowing the co-ordinator to deal with the fault.

Should additional deployment functionality be required at the Modelled Element level, deployment methods can be overridden and parameterised to support particular types of deployment.

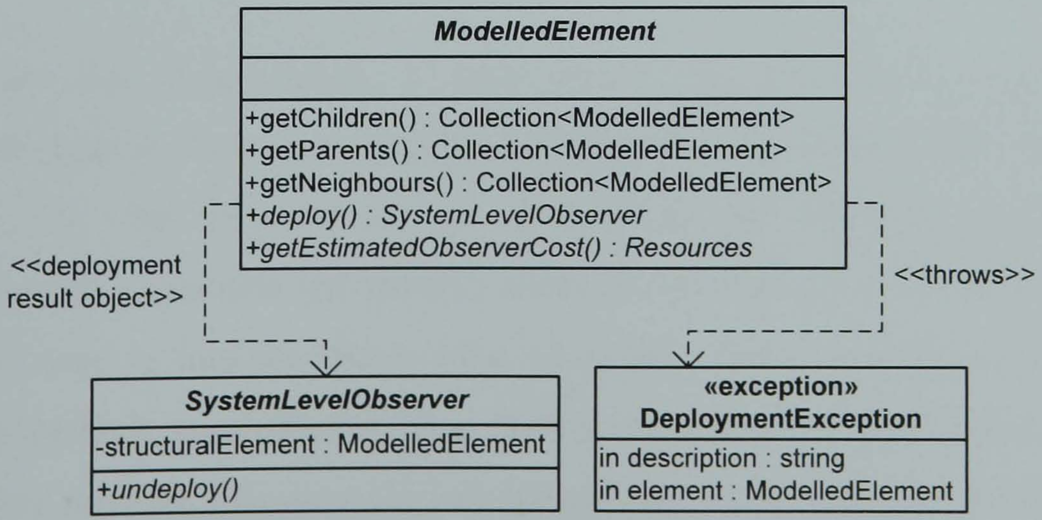

ModelledElement
+getChildren() : Collection<ModelledElement>
+getParents() : Collection<ModelledElement>
+getNeighbours() : Collection<ModelledElement>
+deploy() : SystemLevelObserver
+getEstimatedObserverCost() : Resources
<<deployment result object>>
<<throws>>
SystemLevelObserver
-structuralElement : ModelledElement
+undeploy()
«exception» DeploymentException
in description : string
in element : ModelledElement

Figure 27: System-level Deployment Support: significant methods and classes

Co-ordinating Deployment

As discussed above in accordance with the architectural overview in Section 4.3.4, the Deployment Co-ordinator is responsible for co-ordinating aspects of system-level observer setup and management, significantly:

Requests for changes in observer deployment – the co-ordinator must process requests for observer deployment and translate them into deployment of observers in accordance with the current system state and deployment-related policy.

Currently-deployed observers – the co-ordinator must determine which observers are currently deployed. Even in simple mappings, this affects resource availability, and the makeup of replacement observer deployments. For example, if the new deployment requires many existing observers, it could be wasteful to remove them only to re-deploy moments after.

Observation resource availability – while a technique may recommend a set of elements that require observation, it is the responsibility of the deployment unit to determine which elements *may* have observers deployed. Resource calculation may be a crude observer count, more accurately specified on a per-element cost, or may even be composed of different resource types. Tight resource constraints may mean only a subset of the Technique's recommended elements deploy observers, determined via calculation of (estimated) resource cost and use of Modelled Element metadata.

Other policy-based considerations – in addition to the items outlined above, the deployment co-ordinator may still make its own policy-based decisions on elements requiring observation. As previously, the same caveats apply regarding the use of a rule-based language. The following designs will assume either hard-coded policy-type logic, or sub-classing to provide policy interpretation facilities.

To facilitate this functionality, a new class `DeploymentCoordinator` handles deployment requests from the technique selection process. In the UML class diagram in Figure 28, the co-ordinator is defined as an abstract class, requiring implementations to provide the abstract method, *`requestDeploymentOn()`*, which is the method used to turn an Observation Technique into a real system deployment. Additional methods are defined to deal with manipulation of the deployed set, along with a utility method, *`translate()`*, to return only those Modelled Element objects that are not already observed in a specified collection. The same diagram also shows

`Resources`, which facilitates basic resource functionality as described above. The template interface facilitates basic differentiation of resource types, addition and deduction of resources of a particular type, along with availability checking. Further system-specific requirements could be included via extension.

The diagram also shows resource allocation and costing-related methods in `SystemLevelObserver` and `ModelledElement`. These methods allow access to the planned cost of observing an element against the available resources.

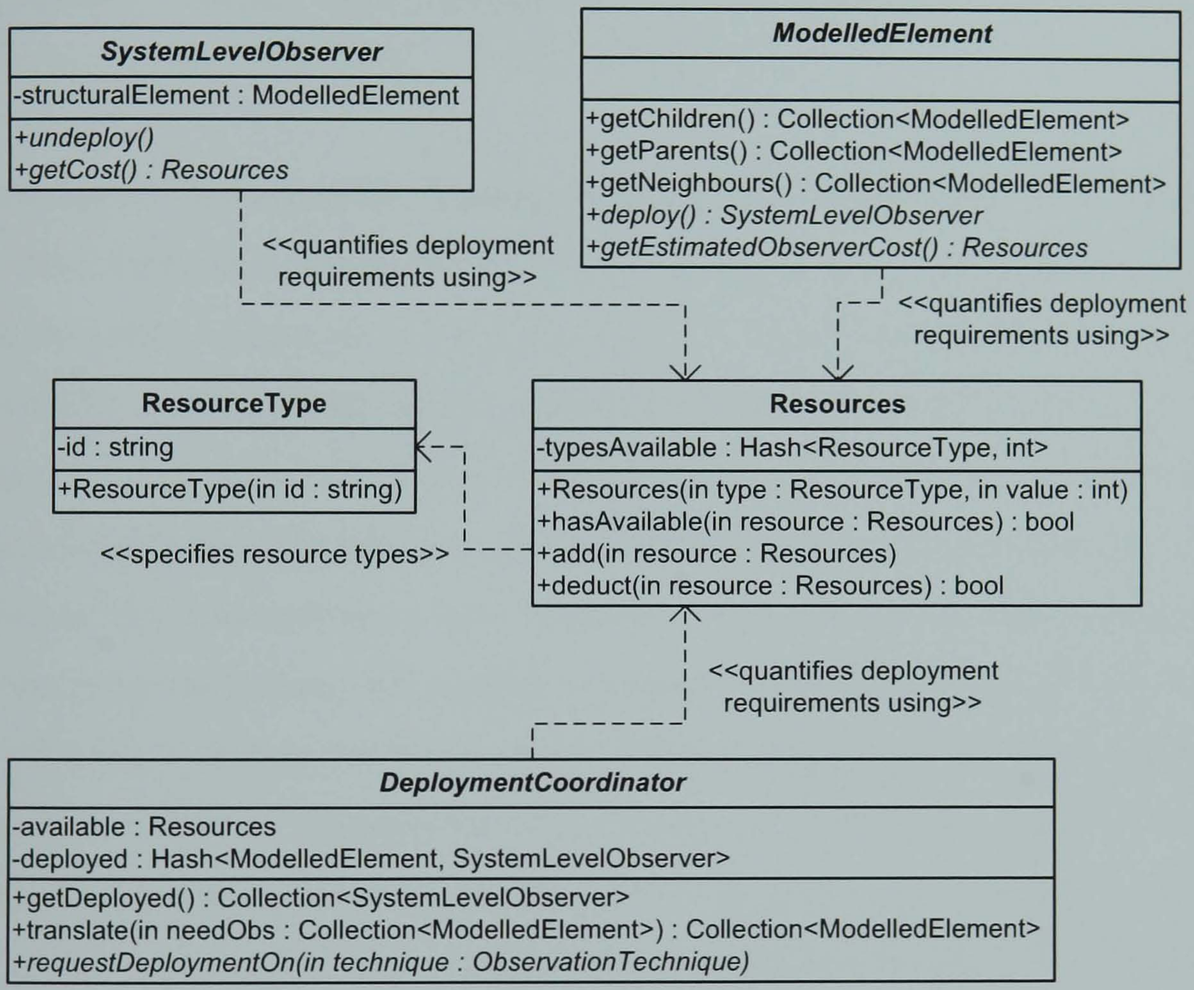

SystemLevelObserver
-structuralElement : ModelledElement
+undeploy()
+getCost() : Resources
ModelledElement
+getChildren() : Collection<ModelledElement>
+getParents() : Collection<ModelledElement>
+getNeighbours() : Collection<ModelledElement>
+deploy() : SystemLevelObserver
+getEstimatedObserverCost() : Resources
<<quantifies deployment requirements using>>
<<quantifies deployment requirements using>>
ResourceType
-id : string
+ResourceType(in id : string)
<<specifies resource types>>
Resources
-typesAvailable : Hash<ResourceType, int>
+Resources(in type : ResourceType, in value : int)
+hasAvailable(in resource : Resources) : bool
+add(in resource : Resources)
+deduct(in resource : Resources) : bool
<<quantifies deployment requirements using>>
DeploymentCoordinator
-available : Resources
-deployed : Hash<ModelledElement, SystemLevelObserver>
+getDeployed() : Collection<SystemLevelObserver>
+translate(in needObs : Collection<ModelledElement>) : Collection<ModelledElement>
+requestDeploymentOn(in technique : ObservationTechnique)

Figure 28: Deployment Coordinator, support classes and revisions to existing classes

The next section assembles individual components for an overview design showing system-wide class interaction and major processes involved in typed observation.

6.2 "Typed" Observers: Applying the Model

In later sections of Chapter 5, new requirements have emerged for existing classes; whereas concise presentation has necessitated inclusion of only the directly-altered classes in new diagrams. This section assembles final designs, explains significant usage processes, and demonstrates how application of this model can create typed-observation for use in monitoring large-scale systems with complex structural characteristics. Firstly, a reminder of the components comprising the system designs:

The Structural Model – provides a low-level overlay on the observed system, effectively the bridge between the structural observation framework and the underlying system. `ModelledElement` objects are responsible for notifying `StructuralObserver` objects of structural change, and on request via the *`deploy()`* method, providing element-specific observation services.
Significant classes and interfaces: `ModelledElement, ModelChangeEvent, ModelledElementFactory, StructuralObserver`

Structural Characteristic Typing (Signatures) – responsible for identifying types by their distinctive characteristics. They interact with a target `ModelledElement`, its neighbourhood and `ModelledElementFactory` to provide information regarding the structural characteristics of the system. Specification is via a `Signature` abstract class, and can make use of `ModelledElementAlgorithm` objects in order to translate sets of elements and produce results. `Signature` objects must provide an indication of whether their characteristics have been met – both as Boolean and continuous values – the latter indicating the quality of match. They must provide two methods of notification – change and invalidation. *Change* notifications are despatched when the signature value is changed from true to false or vice versa. *Signature Invalidation Handlers* monitor change events from the `ModelledElement` objects and generate a suitable `InvalidationEvent` object when the signature should be rechecked.
Significant new classes and interfaces: `Signature, ModelledElementAlgorithm, SignatureResults, InvalidationEvent, SignatureInvalidationHandler, SignatureChangeEvent, SignatureChangeObserver`

Observation Exploit Typing (Techniques) – responsible for specifying different observation exploits; effectively techniques for selecting observation targets. Techniques can also make use of one or more `ModelledElementAlgorithm` objects or can further extend this functionality to provide advanced functionality for target elements selection. Techniques can additionally parameterise their output target elements by associating additional `MetaData` objects.

Significant new classes and interfaces: `ObservationTechnique`, `MetaData`, `MetaKey`, `MetaDataType`, `ModelledElementMetaData`

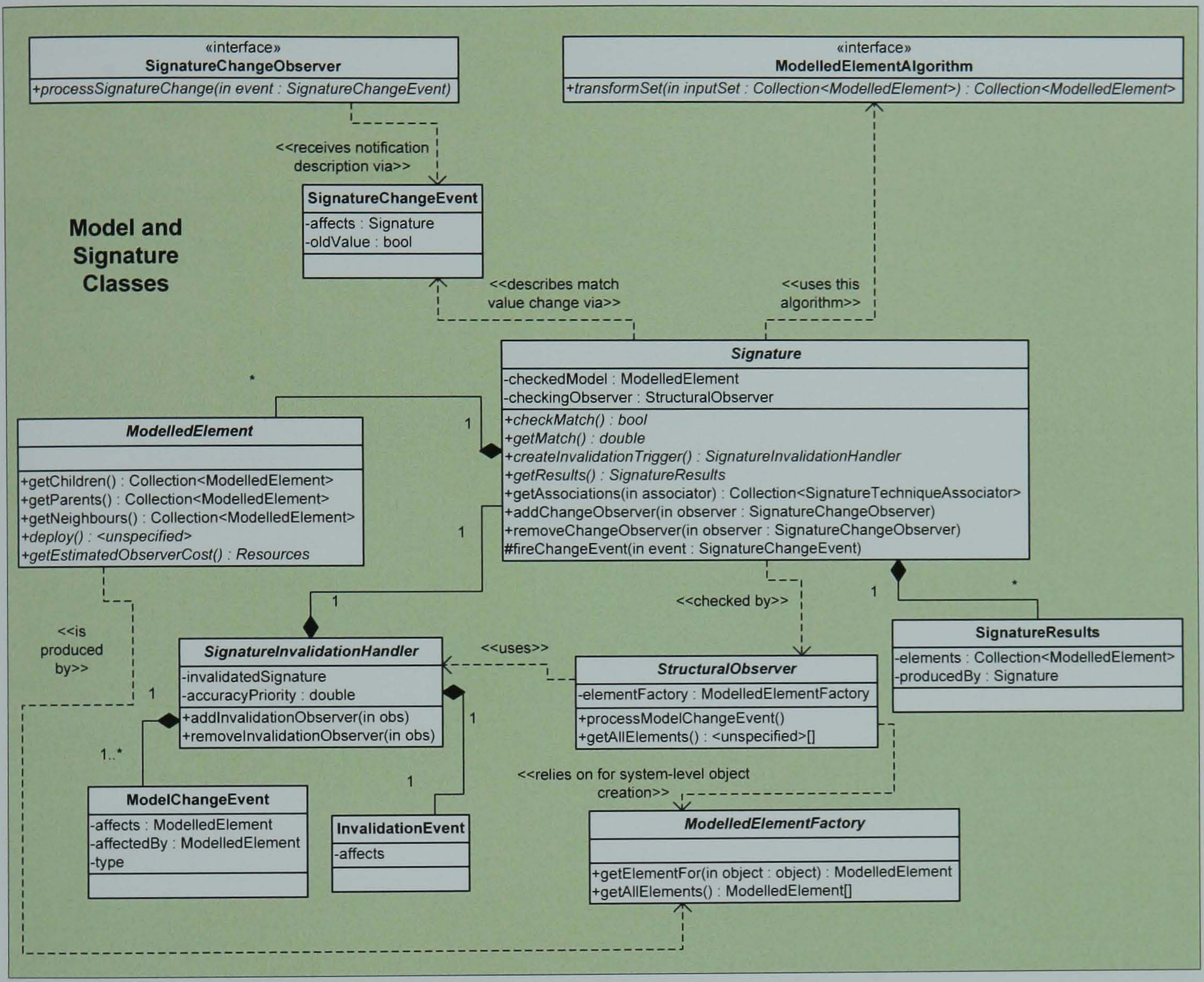
Technique Selection – responsible for *static* association of Signatures with Techniques and for *dynamic* selection of a particular Technique when a signature is matched. The selection process examines structural concerns and policy (such as the quality of signature match and alternative approaches) and selects one or more suitable observation techniques.
Significant new classes and interfaces specifying and supporting this component: `SignatureTechniqueAssociator`, `TechniqueSelector`

System Observer Deployment – responsible for translating a technique's recommendations into observer deployment. It must consider the observed system's runtime state, including resource availability concerns, and the presence of existing observers. Design for this component includes a basic specification for resource management and calculation representation.
Significant new classes defined for this component:
`DeploymentException`, `Resources`, `ResourceType`, `DeploymentCoordinator`, `SystemLevelObserver`

6.2.1 Completing the Model: Overview Class Diagrams

In order to provide a complete overview of the system and describe the relationships between the designed classes and components, a series of complete class diagrams follow. They show the consolidated makeup of the system's classes and are accompanied with only basic explanatory detail; they are collating classes whose functionality has been explained previously. Usual UML notation is used to show abstraction, interfaces and composition, while dependencies and significant interactions are depicted with dotted-line arrows. In the case of dependencies, the arrow is pointing in the direction of the dependency; i.e. away from the dependent.

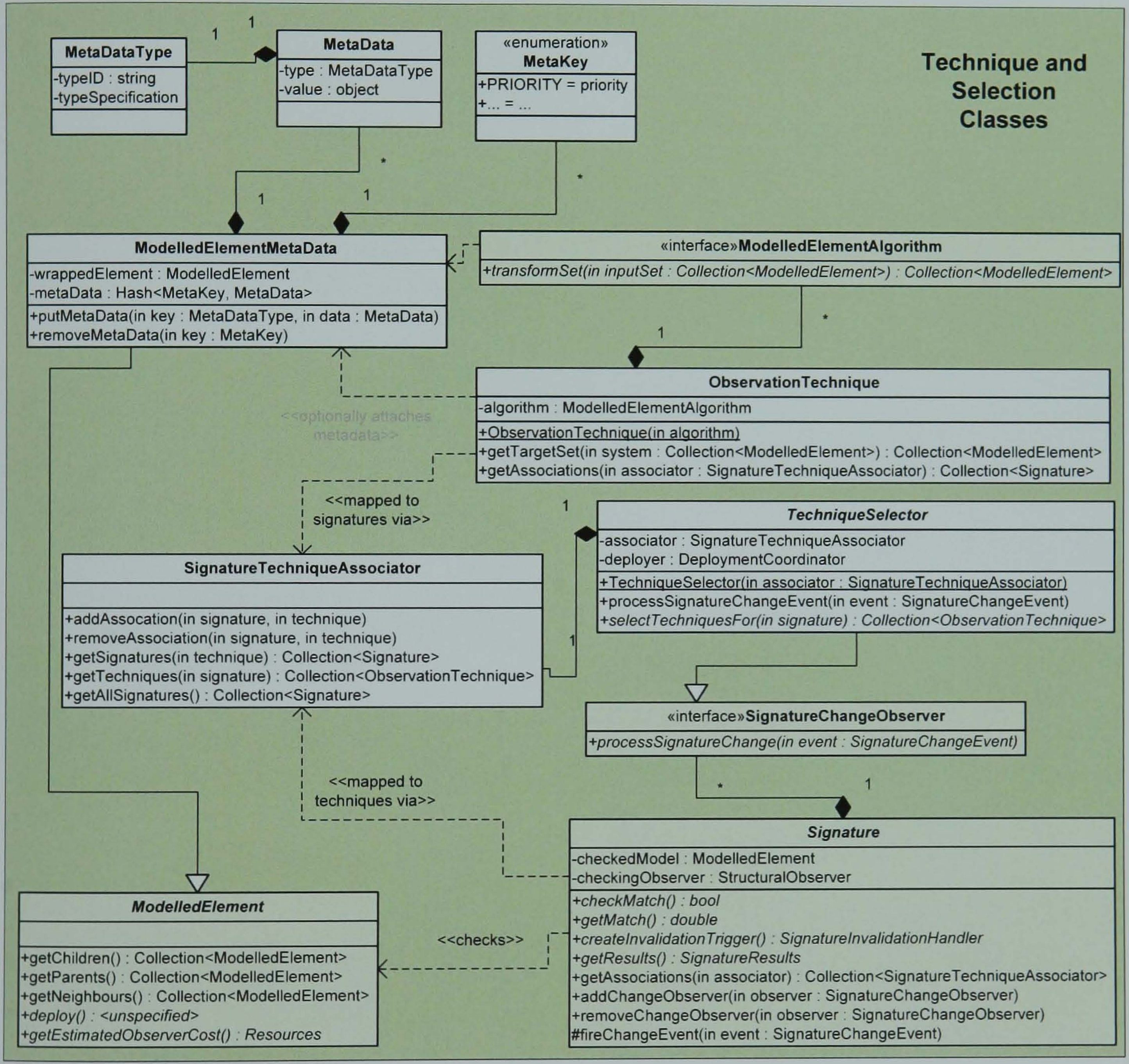
The first class diagram in Figure 29 shows classes required for the **Structural Model** and the associated **Typing (Signature)** functionality outlined in the previous summary. As these classes form the model on which the system is built, a selected few of them will necessarily feature in following diagrams for completeness in providing a reference back to the types defined here. Wherever they appear duplicated, they will be depicted with a shaded background.



«interface»
SignatureChangeObserver
+processSignatureChange(in event : SignatureChangeEvent)
«interface»
ModelledElementAlgorithm
+transformSet(in inputSet : Collection<ModelledElement>) : Collection<ModelledElement>
<<receives notification description via>>
SignatureChangeEvent
-affects : Signature
-oldValue : bool
Model and Signature Classes
<<describes match value change via>>
<<uses this algorithm>>
Signature
-checkedModel : ModelledElement
-checkingObserver : StructuralObserver
+checkMatch() : bool
+getMatch() : double
+createInvalidationTrigger() : SignatureInvalidationHandler
+getResults() : SignatureResults
+getAssociations(in associator) : Collection<SignatureTechniqueAssociator>
+addChangeObserver(in observer : SignatureChangeObserver)
+removeChangeObserver(in observer : SignatureChangeObserver)
#fireChangeEvent(in event : SignatureChangeEvent)
ModelledElement
+getChildren() : Collection<ModelledElement>
+getParents() : Collection<ModelledElement>
+getNeighbours() : Collection<ModelledElement>
+deploy() : <unspecified>
+getEstimatedObserverCost() : Resources
*
1
1
1
<<checked by>>
1
*
SignatureResults
-elements : Collection<ModelledElement>
-producedBy : Signature
<<is produced by>>
SignatureInvalidationHandler
-invalidatedSignature
-accuracyPriority : double
+addInvalidationObserver(in obs)
+removeInvalidationObserver(in obs)
<<uses>>
StructuralObserver
-elementFactory : ModelledElementFactory
+processModelChangeEvent()
+getAllElements() : <unspecified>[]
1
1
1..*
1
ModelChangeEvent
-affects : ModelledElement
-affectedBy : ModelledElement
-type
InvalidationEvent
-affects
<<relies on for system-level object creation>>
ModelledElementFactory
+getElementFor(in object : object) : ModelledElement
+getAllElements() : ModelledElement[]

Figure 29: Significant Model and Signature Classes

The UML class diagram following in Figure 30 shows the **Observation Exploits (Techniques)** and **Technique-selection** policy/logic component classes. Abstract classes such as `ModelledElement` and `Signature` are included from the previous diagram to help show interactions with the underlying structural model and the signature matching that brings about the Technique selection process.



Technique and Selection Classes
MetaDataType
-typeID : string
-typeSpecification
1
1
MetaData
-type : MetaDataType
-value : object
«enumeration» MetaKey
+PRIORITY = priority
+... = ...
*
*
1
1
ModelledElementMetaData
-wrappedElement : ModelledElement
-metaData : Hash<MetaKey, MetaData>
+putMetaData(in key : MetaDataType, in data : MetaData)
+removeMetaData(in key : MetaKey)
«interface»ModelledElementAlgorithm
+transformSet(in inputSet : Collection<ModelledElement>) : Collection<ModelledElement>
*
1
ObservationTechnique
-algorithm : ModelledElementAlgorithm
+ObservationTechnique(in algorithm)
+getTargetSet(in system : Collection<ModelledElement>) : Collection<ModelledElement>
+getAssociations(in associator : SignatureTechniqueAssociator) : Collection<Signature>
<<optionally attaches metadata>>
<<mapped to signatures via>>
1
TechniqueSelector
-associator : SignatureTechniqueAssociator
-deployer : DeploymentCoordinator
+TechniqueSelector(in associator : SignatureTechniqueAssociator)
+processSignatureChangeEvent(in event : SignatureChangeEvent)
+selectTechniquesFor(in signature) : Collection<ObservationTechnique>
SignatureTechniqueAssociator
+addAssocation(in signature, in technique)
+removeAssociation(in signature, in technique)
+getSignatures(in technique) : Collection<Signature>
+getTechniques(in signature) : Collection<ObservationTechnique>
+getAllSignatures() : Collection<Signature>
1
«interface»SignatureChangeObserver
+processSignatureChange(in event : SignatureChangeEvent)
*
1
<<mapped to techniques via>>
Signature
-checkedModel : ModelledElement
-checkingObserver : StructuralObserver
+checkMatch() : bool
+getMatch() : double
+createInvalidationTrigger() : SignatureInvalidationHandler
+getResults() : SignatureResults
+getAssociations(in associator) : Collection<SignatureTechniqueAssociator>
+addChangeObserver(in observer : SignatureChangeObserver)
+removeChangeObserver(in observer : SignatureChangeObserver)
#fireChangeEvent(in event : SignatureChangeEvent)
<<checks>>
ModelledElement
+getChildren() : Collection<ModelledElement>
+getParents() : Collection<ModelledElement>
+getNeighbours() : Collection<ModelledElement>
+deploy() : <unspecified>
+getEstimatedObserverCost() : Resources

Figure 30: Significant Technique, Technique-selecting and support classes

The following class diagram in Figure 31 shows those classes involved in the system-level **Observer Deployment**, and relationships between the significant organisational class – `DeploymentCoordinator` – and its dependencies.

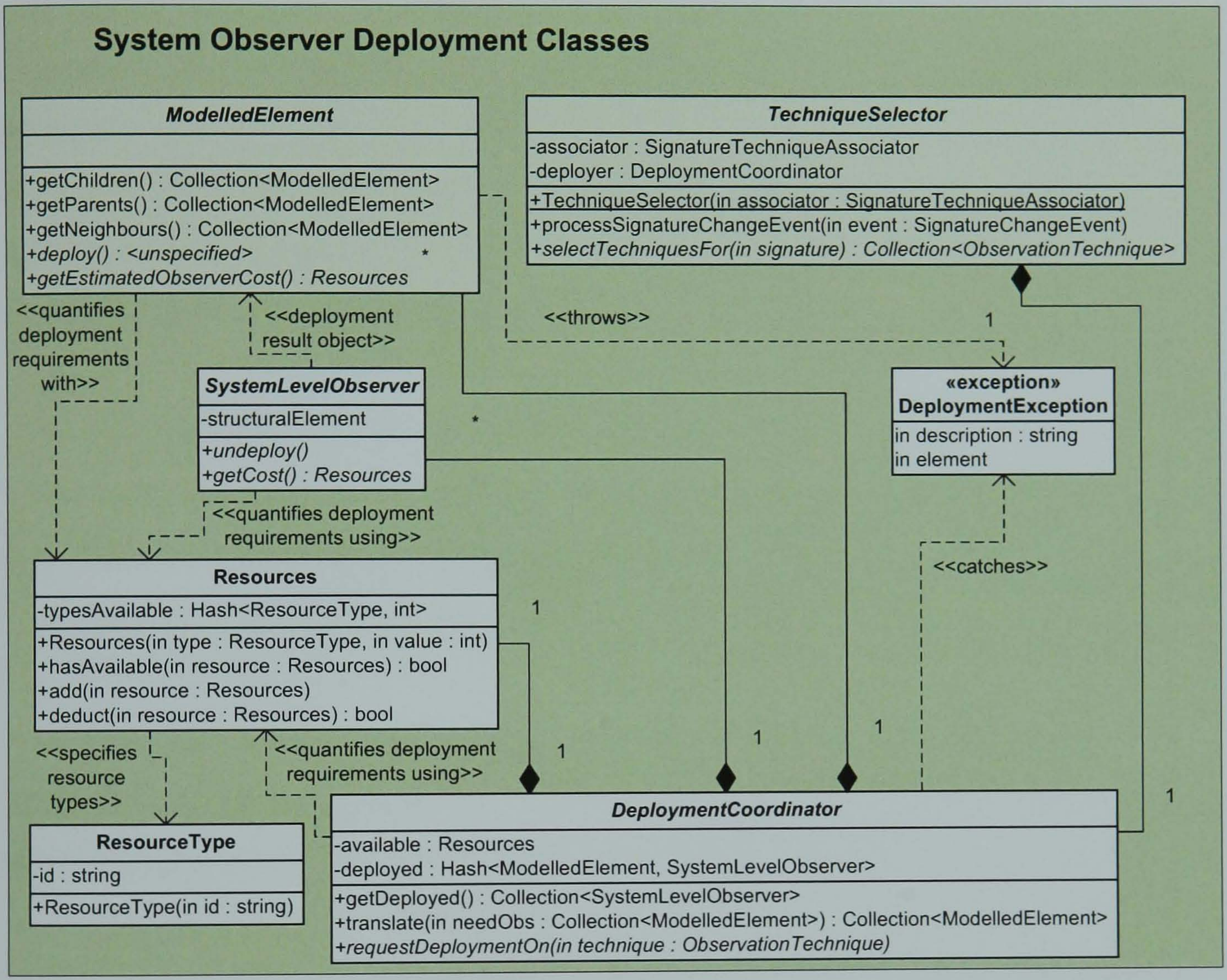

System Observer Deployment Classes
ModelledElement
+getChildren() : Collection<ModelledElement>
+getParents() : Collection<ModelledElement>
+getNeighbours() : Collection<ModelledElement>
+deploy() : <unspecified>
+getEstimatedObserverCost() : Resources
TechniqueSelector
-associator : SignatureTechniqueAssociator
-deployer : DeploymentCoordinator
+TechniqueSelector(in associator : SignatureTechniqueAssociator)
+processSignatureChangeEvent(in event : SignatureChangeEvent)
+selectTechniquesFor(in signature) : Collection<ObservationTechnique>
<<quantifies deployment requirements with>>
<<deployment result object>>
<<throws>>
SystemLevelObserver
-structuralElement
+undeploy()
+getCost() : Resources
«exception» DeploymentException
in description : string
in element
<<quantifies deployment requirements using>>
<<catches>>
Resources
-typesAvailable : Hash<ResourceType, int>
+Resources(in type : ResourceType, in value : int)
+hasAvailable(in resource : Resources) : bool
+add(in resource : Resources)
+deduct(in resource : Resources) : bool
<<specifies resource types>>
<<quantifies deployment requirements using>>
ResourceType
-id : string
+ResourceType(in id : string)
DeploymentCoordinator
-available : Resources
-deployed : Hash<ModelledElement, SystemLevelObserver>
+getDeployed() : Collection<SystemLevelObserver>
+translate(in needObs : Collection<ModelledElement>) : Collection<ModelledElement>
+requestDeploymentOn(in technique : ObservationTechnique)

Figure 31: System-level Observer Deployment and support classes

However, while these diagrams helped to consolidate classes previously shown only iteratively; as a design guide, they suffer from the underlying issue with any UML class diagram – they present only a static overview of the system. Therefore, the following section will help to show how the designs intend the system operates, by summarising component interactions when they are performing system tasks.

6.2.2 Using the Model: Important Runtime Processes

This section examines some key processes that the system is designed to deal with from initial structural addition to a system, through to the changing deployment of system-level observers. As with the previous section, this is largely collating information from Sections 5.1 to 6.1 but with the benefit of presenting the whole picture now previous designs are completed. In order to present a logical overview of the system and to assist developers in implementing the various classes such that they function together correctly, this section will examine how the "reference design" would achieve the following system processes:

- Adding a structural observer to a modelled element
- Initial registration of system signatures, techniques, and deployment processes
- Signature rechecking and Invalidation Handling
- Signature match through to system-level observer deployment

These processes will be broken down into the necessary task descriptions, with clarification of where responsibility lies for each one. Where greater clarity can be gained by using process or flow diagrams, these will be included.

During these functional examinations, reference will be made to super-classes, abstract types and interfaces as if they had been suitably sub-classed or implemented as concrete functional units. It is appreciated that in a real system, programmers would be dealing with the appropriate subtypes that are providing the real functionality.

Adding a Structural Observer to a Modelled Element

This subsection examines how the reference design deals with the process of creating a new structural observer and deploying it on a modelled element. It involves the following sub-processes:

1. Creating the structural observer
2. Creating the modelled element and model exploration*
3. Registering the structural observer as a listener on the modelled element (and accordingly, its neighbourhood)

* = Indicates that this process could be undertaken as a separate operation, allowing many different Structural Observers to then attach to the element. Alternatively, element creation may be encapsulated by the Structural Observer for convenience.

This makes use of functionality from concrete subclasses and implementations of the following types:
`ModelledElement, ModelledElementFactory, StructuralObserver`

Importantly, it also assumes that suitable `ModelledElement` subtypes have been created; capable of adequately describing all system-level components and elements. Additionally, this necessitates a suitable element factory being imbued with correct instantiation logic.

The diagram, Figure 32 below shows the basic process flow of information and execution required in creating the observer and its target `ModelledElement` objects. Attention is paid to the manner in which the factory maintains an object cache and the `ModelledElement` constructor is responsible for providing a factory call-back to get the neighbours. Considering that the process operates recursively, it is clear to see this will result in an exploration (and creation) of the system-level component graph.

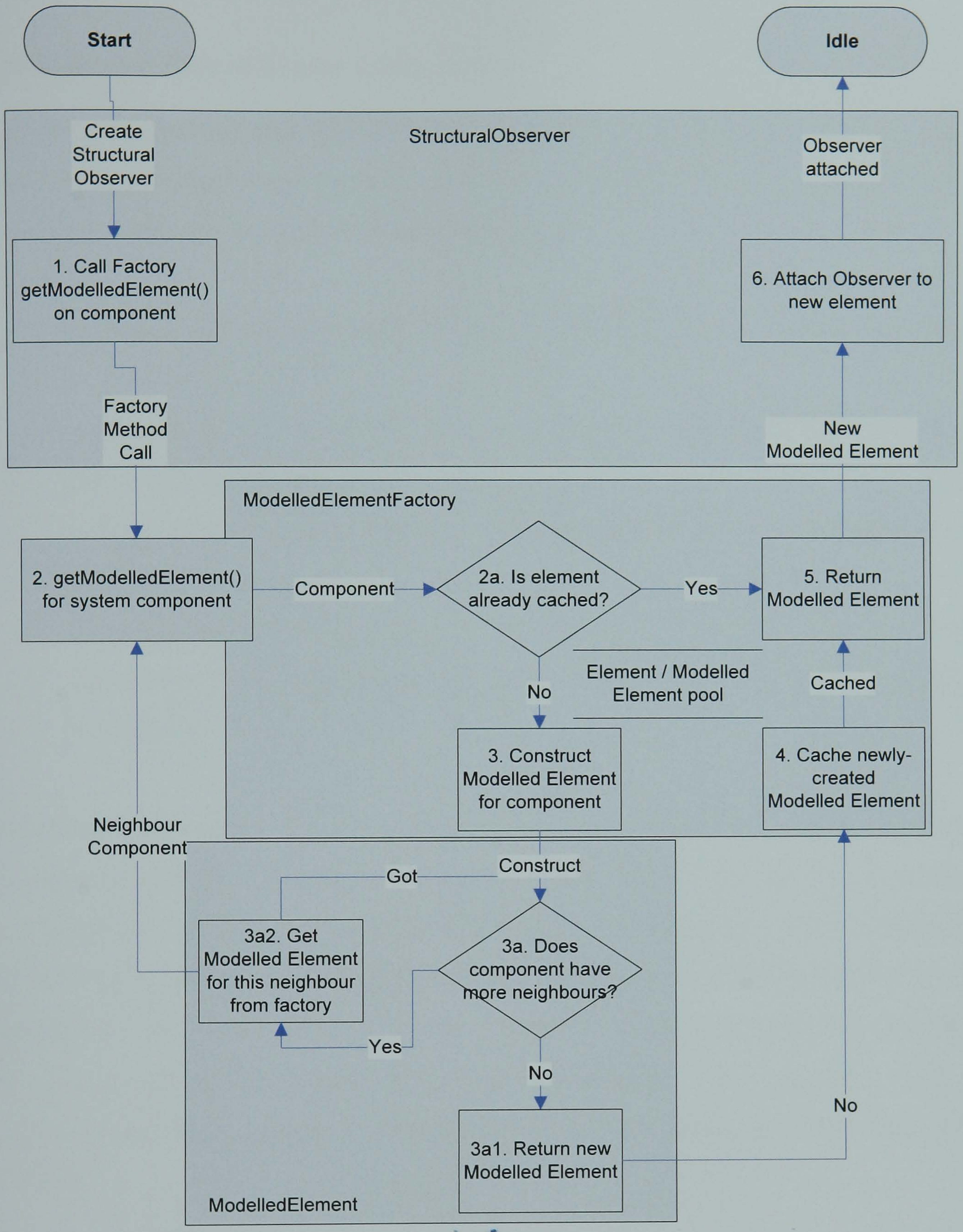

Start
Idle
Create Structural Observer
StructuralObserver
Observer attached
1. Call Factory getModelledElement() on component
6. Attach Observer to new element
Factory Method Call
New Modelled Element
ModelledElementFactory
2. getModelledElement() for system component
Component
2a. Is element already cached?
Yes
5. Return Modelled Element
Element / Modelled Element pool
No
Cached
3. Construct Modelled Element for component
4. Cache newly-created Modelled Element
Neighbour Component
Got
Construct
3a2. Get Modelled Element for this neighbour from factory
3a. Does component have more neighbours?
Yes
No
No
3a1. Return new Modelled Element
ModelledElement

Figure 32: Logical flow for Structural Observer / Modelled Element creation

The limited detail in the flow diagram does not intend to provide line-by-line implementation guidance – there are other issues to be determined; e.g. multithreading and cache handling. However, the flow diagram reiterates the logical delegation of responsibility between the significant objects involved in this process.

Initial Registration of System Components

The next task looks at how various system components are registered with the system once the system model and suitable structural observation units are created. This can broadly be split into the following significant tasks:

1. Instantiation of suitably-written Signatures, Techniques, and shared Algorithms as required
2. Instantiations of suitable signature-technique "associator"(s), Technique Selector(s), and Deployment Co-ordinator(s)
3. Creation of Invalidation Handlers associated with Signatures and attachment to Modelled Elements
4. Attaching Technique Selector(s) to Signatures (as Signature Change Listeners)
5. Association of the appropriate Deployment Coordinator(s) with the Technique Selector(s)

Implementation Note: - Referring back to the end of Section 5.2, the signature invalidation method is discussed along with both an appropriate method to handle it and situations where a low-complexity invalidation mechanism would be preferred (effectively a polling model). Although the concept of an Invalidation Handler is included in the design, it is perfectly feasible to produce a statically-generated `InvalidationHandler` object that simply invalidates the signature every *x* `ModelChangeEvent` events it receives, or even once every predetermined time interval.

This functionality is split across many of the types in the system, specifically: `ModelledElementAlgorithm` which contributes to `Signature` and `ObservationTechnique`. Instances of the signature and technique classes are associated via `SignatureTechniqueAssociator`, which is combined along with `DeploymentCoordinator`, to create `TechniqueSelector` objects.
`TechniqueSelector` implements the `SignatureChangeListener` interface, and is attached to the various signatures in the system, completing the structural "loop" and providing response to signature matching.

Invalidation Handling and Signature (re)checking

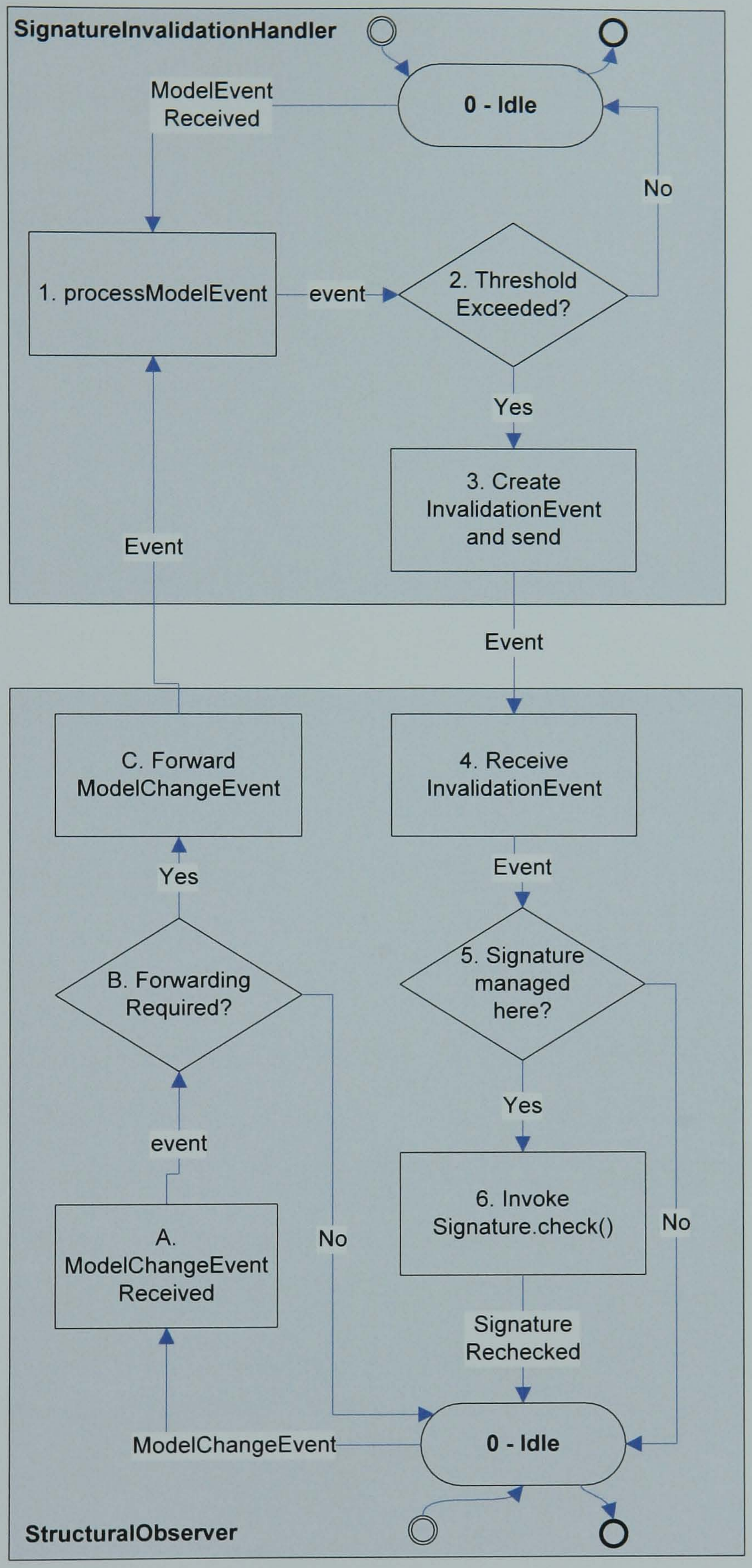
It is appropriate at this point to clarify how the loop between signatures and their invalidation handlers is completed. As discussed in Sections 5.2.1 and 5.2.2, Invalidation Handlers have been selected as the method for Signature objects to notify an interested party that they should be re-evaluated. Implementation details are of no concern at this stage; they could perform complex calculations regarding received connectivity events and the likelihood of signature change, or they could provide what is effectively a polling mechanism.

However, in order to understand the way in which invalidation will cause Signature rechecking, it is important to pay particular attention to the definition. It states that concrete implementations of the `SignatureInvalidationHandler` class will act as both event receiver and producer – it receives events of type `ModelChangeEvent` (to allow it to calculate the invalidation "threshold"), and produces events of type `InvalidationEvent` to indicate when the signature should be recalculated. Section 6.1.2 outlined reasoning behind having a single component to process these Invalidation Events (thus specifying that Signatures must also generate change events), and the Structural Observer was proposed as the component that should receive and process these events. When the Structural Observer receives an event, it should locate the appropriate Signature and recheck it.

This functionality will be outlined in a simple flow diagram, Figure 33, highlighting which components can take responsibility for particular tasks. The diagram shows the proposed relationship between `SignatureInvalidationHandler` and `StructuralObserver` when dealing with signature (re)checking. Note how the structural observer should, if required, forward the `ModelChangeEvent` objects on to the invalidation handler. This permits the invalidation handlers to operate without having to maintain their own list of `ModelledElement` objects; exploiting the `StructuralObserver` position as a structural manager.

This process centres on the following classes and interfaces:

`SignatureInvalidationHandler, InvalidationEvent, StructuralObserver, ModelChangeEvent`



SignatureInvalidationHandler
0 - Idle
ModelEvent Received
1. processModelEvent
event
2. Threshold Exceeded?
No
Yes
3. Create InvalidationEvent and send
Event
Event
C. Forward ModelChangeEvent
4. Receive InvalidationEvent
Event
Yes
B. Forwarding Required?
5. Signature managed here?
Yes
event
6. Invoke Signature.check()
A. ModelChangeEvent Received
No
No
Signature Rechecked
ModelChangeEvent
0 - Idle
StructuralObserver

Figure 33: SignatureInvalidationHandler and StructuralObserver relationship

Signature Matching - through to System-level Observer Deployment

This final subsection shows how a signature match event (as detailed above) leads to the system-level observers being (re)deployed. As discussed previously in this chapter, a Signature that has changed its matched value starts the following process:

1. Signature Change Event despatched
2. Technique Selector determines one or more recommended Observation Techniques, considering concerns such as multiple signature matches, the "quality" of the notified signature match and the techniques associated to the matched signature.
3. Deployment Coordinator is invoked with selected technique(s) and deploys the requested technique(s), taking account of concerns including resource availability, currently deployed observers, and dealing appropriately with failures.

Much of what happens within the technique selection and system-level deployment is of necessity domain-specific, and as such, a flow diagram could not provide a generic solution. However, the diagram in Figure 34 shows some potentially common aspects to the whole process, aiming to help clarify each component's responsibility. Of particular domain-specific note is the issue surrounding `ModelledElement.`*`deploy()`*. As discussed in Section 6.1.3, this provides the bridge to `SystemLevelObserver` objects, and effectively to the *real* system observation. Referring back to the architecture in Chapter 4, it is envisaged that there could be several different overlays operating on one system, each monitoring different concerns or states, and each with a different type of system-level observer; each potentially co-ordinated via different deployment coordinators.

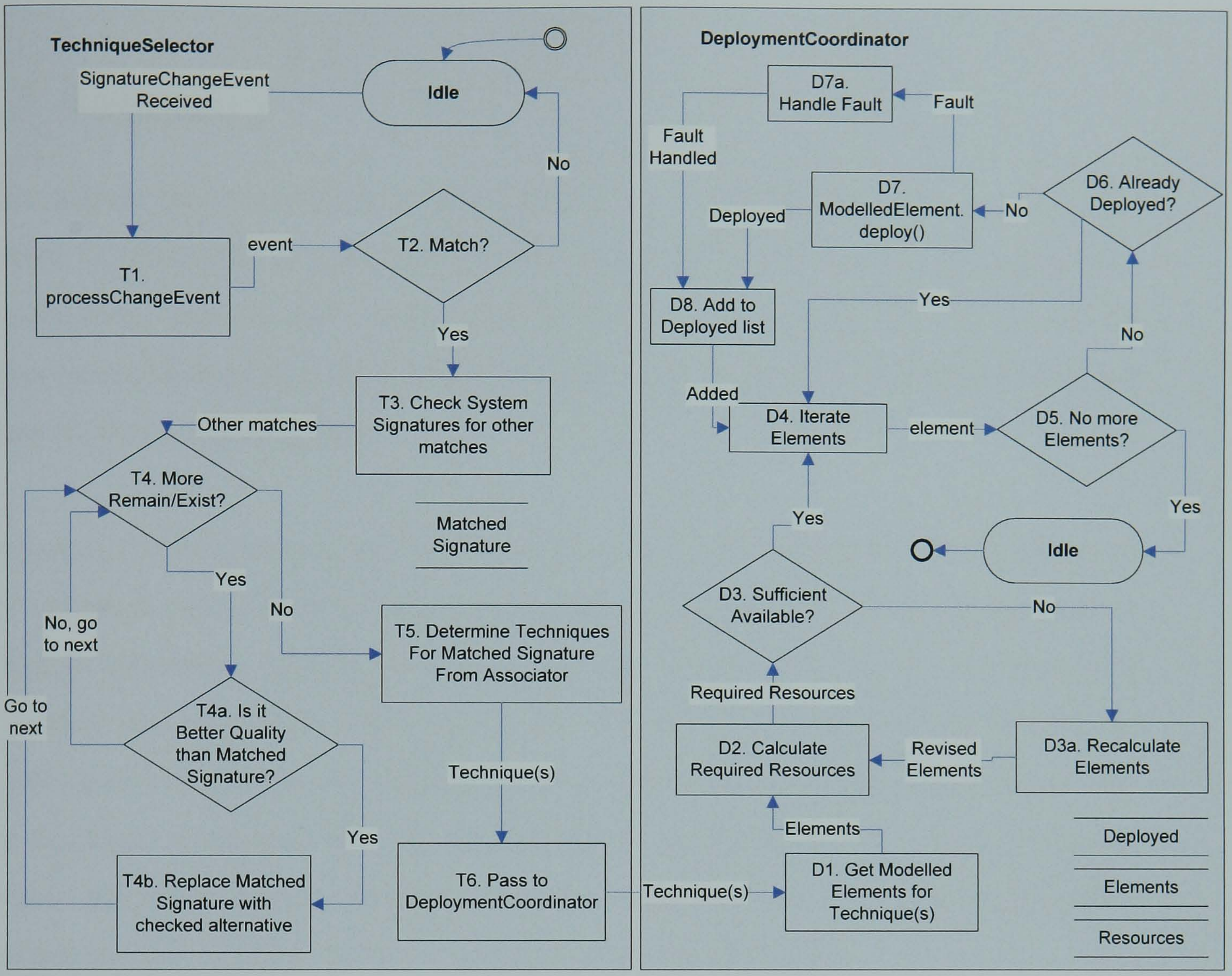

TechniqueSelector
SignatureChangeEvent Received
Idle
No
T1. processChangeEvent
event
T2. Match?
Yes
T3. Check System Signatures for other matches
Other matches
T4. More Remain/Exist?
Matched Signature
Yes
No
No, go to next
T5. Determine Techniques For Matched Signature From Associator
Go to next
T4a. Is it Better Quality than Matched Signature?
Technique(s)
Yes
T4b. Replace Matched Signature with checked alternative
T6. Pass to DeploymentCoordinator
DeploymentCoordinator
D7a. Handle Fault
Fault
Fault Handled
D7. ModelledElement. deploy()
No
D6. Already Deployed?
Deployed
D8. Add to Deployed list
Yes
No
Added
D4. Iterate Elements
element
D5. No more Elements?
Yes
Yes
Idle
D3. Sufficient Available?
No
Required Resources
D2. Calculate Required Resources
Revised Elements
D3a. Recalculate Elements
Elements
Deployed
Technique(s)
D1. Get Modelled Elements for Technique(s)
Elements
Resources

Figure 34: Technique Selection and Observer Deployment process flow

6.3 Summary

This chapter continues the design process begun in Chapter 5, and provides additional design consideration to those summarised in Section 5.4. This new consideration – representing policies and deployment concerns – completes the basics of the observation framework, allowing it to perform controlled observation deployment and observer management duties.

However, the designs here are suited to hardcoded implementations; the flexibility of the system is constrained by options included in the code. The system is designed to adapt to structural changes, but adaptations are catered for by way of hardcoded signatures and techniques bound together by largely hard-coded policy. Changing the system policy would involve editing it at a code level, re-compiling and re-deploying. As discussed in Section 6.1, use of third party rule-based systems could provide a greater degree of runtime flexibility, but this is not without its problems regarding scaling and real-time performance management considerations.

The next chapter will further examine some of the issues briefly tackled in Section 6.1, possible solutions to allow further evolution of the system at runtime, and how the necessary components could be exposed in a suitable externalised format.

Chapter 7 – Evolution and the Observation Model

The previous chapter explored many design concerns involved in the large-scale / complex observer framework. Implementations following this architecture and design guidance should benefit from an observation overlay that will deploy its observation units in accordance with the system structure and at a best-match from the available signatures and techniques provided at design time. However, while the observation overlay will adapt to system modifications, it is still reliant on sufficient information being provided in terms of signatures, observation techniques and observation policy.

The framework should perform efficiently under these conditions as the types of design-time specification are intended to be heuristic rather than specific. However, the previous chapter did not specify how new signatures, techniques or policy may be added, or how existing types may be amended. Additionally, the previous chapter showed designs that specified interfaces and demonstrated how the various proposed classes would interact, but in many cases, did not specify the precise implementation – abstract classes and interfaces gave behaviour definitions but the implementation discussion was deliberately generic. In certain cases, there is value in a hard-coded approach; signatures are defined in code implementing the specified interfaces, then compiled and used in the system. In fact, the entire observation specification could be specified in code, compiled and then deployed on a system, moving between the various combinations of design-time specified characteristics and techniques as required. However, this approach limits the manner in which the observation subsystem can evolve at runtime – it is, at simplest, moving between design-time determined states.

Throughout the architecture and design, reference has been made to the use of policy and associated reasoning to assist the system's deployment decisions. Equally, reasoning has been made for creating loose associations between signatures and techniques, rather than making fixed mappings. Therefore, rather than a simple reflex trigger that determines a particular observed characteristic should lead to a particular observation technique; deferring this decision to a reasoning component allows a more complex and complete definition of structural type and appropriate observation behaviour.

As stated previously, a full discussion of reasoning and policy setup is outside the scope of this work. However, the interfacing between the defined observation sub-system components and any reasoning entity should be well defined. As such, if the behavioural logic behind these signatures and observation techniques is in compiled code, reasoning components cannot be expected to make decisions based on signature criteria, observation technique, and importantly, any mappings between the two. Therefore, this chapter will determine the system components that benefit from openly exposing elements of their data and behaviour, those that would benefit from an externalised specification (be it of behaviour, data or both), and how the necessary exposure and externalisation can be bound to the executed code.

7.1 Considering the Model's Runtime Processes

The bulk of this chapter will look into the individual components that require externalisation and how best to expose them. Firstly, it is worthwhile revisiting the observation model and looking to highlight the process that the observation subsystem uses to deploy an observer, as outlined in Section 6.2.2. This will help to demonstrate how the overall observation behaviour could be specified externally – before going on to consider the required specification points for each of the sub-elements.

As discussed in earlier chapters, the basic premise of this observation system is that when an observer is added to a particular set of elements, it uses signatures to indicate that a particular observation technique should be deployed. The signatures were developed such that they could specify an invalidation event – or trigger – that indicated they should be re-matched. Taking the basic rule-based or event-driven software methodology, **ECA** (Event-Condition-Action) e.g.([120]), this equates to a couple of complimentary ECA definitions. Additionally, it implies a range of definitions that link signatures, observer policy and other conditions together with a particular deployment of an observer system (i.e. the observation techniques):

ECA ID	Causal Event	Required Condition	Resulting Action
1	Observer Added to system elements	Observer not previously observing these elements	Invoke Signature Matching
2	Signature Invalidation	Observer still observing the elements within the signature	Invoke Signature Matching
3…	Signature *x* Matched	Variety of Observer Policy Conditions *y*	Deploy Observers applicable to Signature *x* in consideration with conditional policies *y* using Technique *z*

Table 2: Simplified ECA Breakdown of Observation Subsystem Rules

This may seem a trivial example; ECA rules 1 and 2 make up the straightforward logic forming the basis of the system components outlined in the previous chapter. However, the interest lies in rule 3 and developments thereof. These rules form active system logic determining which observation techniques are applicable in which situations, and how they are deployed.

Signatures and techniques could be hardcoded yet still provide flexibility; their behaviour is governed by the `TechniqueSelector` and `DeploymentCoordinator` classes. Runtime flexibility can be addressed at the link between signatures and the resultant techniques. In exposing the map between signature *x* and observation technique *z* as an ECA-style specification, this allows a simplistic policy specification to be included in this externalised mapping between signatures and techniques. If signatures and techniques were extended with suitable construction parameterisation, limited values could be specified in the externalised form. For example, a basic pseudo specification is shown in Figure 35:

```
ON EVENT Signature Match (SYSTEM, SCALE-FREE)
WITH ( Available Resources (SYSTEM, LOW) )
THEN DEPLOY ( Acquaintance Immunisation (SYSTEM, MINIMUM) )

ON EVENT Signature Match (SYSTEM, SCALE-FREE)
WITH (Available Resources (SYSTEM, HIGH)
   OR Operation Priority (SYSTEM, CRITICAL))
THEN DEPLOY ( Acquaintance Immunisation (SYSTEM, FULL) )
```

Figure 35: Pseudo-Specification Signature-to-Observer Mapping

Assuming that suitable fluents or predicates are defined elsewhere within the system for available resources and the priority of the current system operation, this would allow basic system policy to be encapsulated in the externalised form.

To reiterate, externalisation following this form makes the following assumptions:

- Signatures are defined at design-time, compiled and included in the runtime code-base. Inspection and modification of these signatures is not possible at runtime, other than predefined customisation via suitable parameterisation, specified in both the externalised form and the implemented signature code.
- The same condition applies to Observation Techniques – they must be fully specified in code, and have any configuration variables supplied as parameters.
- The fluent or predicate functions are also defined in the code and are exposed as required to the externalised form. The effect of the predicate on the resulting action (i.e. observer deployment) should be relatively simple: it is intended to allow access to the system's policy or rule base. The externalisation should be, in some ways, a definition language that allows simple Boolean and mathematical comparisons; yet the externalisation is not expected to be a Turing-complete language, nor capable of rule resolution in itself.

The next sections will discuss the required elements as per the example in Figure 35, thus creating a definition schema, before further examining the concerns in exposing a detailed specification of both the Signature and Technique individually.

7.1.1 Basic Observation "Behaviour Definition"

An externalised specification of form shown in Figure 35 will effectively create an Observation Behaviour Definition, allowing the runtime inspection, alteration (addition and removal) of the system's observation strategies (Techniques), in terms of which type indications (Signatures) and system policies bring them about. As such, to achieve this simplest level of *ECA-driven* observer definition, the system would need to provision access for and expose the following items:

Signatures

Firstly, a signature needs to be uniquely identifiable so that it can be used from the externalised definition/mapping. For readability of human editors, the element's identification could be a simple string. In the example pseudo-definition above "SCALE-FREE" was used to indicate a reference to the Scale-Free signature. The externalisation's mapping to the signature's match event was marked in the example definition by the use of the ON EVENT keywords, followed by "Signature Match", marking the appropriate section. Secondly, a signature needs to be suitably parameterised such that necessary information can be "injected" from the externalised observation behaviour definition. In the case of a signature the first (and necessary) parameterisation is that of the element/elements under consideration. Given the hierarchical nature of the structural observer, as described in Section 4.3.2, this parameter would primarily concern the matter of scope – an observer may be interested in structural characteristics of its own system elements (designated by SYSTEM in the example), its parent's scope, or indeed a domain-specific subset of its own scope. Other important parameterisations may be required, which will be examined in Section 7.2, which focuses on behavioural controls.

Observation Techniques

The requirements for exposing control of observation techniques / strategies are broadly similar to those surrounding Signatures. Firstly, they must be positively identifiable, and should at least be parameterised with the scope of observation subjects (as per the SYSTEM example for signatures), along with the priority / strength of observation that should be requested from this technique. Observation techniques are referenced in the example by their unique identifier, and marked by the keywords "THEN DEPLOY". As shown in the example definition, this would permit basic implementation of system policies such as linking resource availability to planned allocation.

System Fluents and Predicate Statements

These permit basic interaction with system control policy, and require the most explanation at this stage. In the example, the marking for a (series of) system fluent(s) is the "WITH" keyword. At an implementation level, results from the fluents or parameterised predicates may come from a suitably-built rule-base query that defines system policy or may simply read and process appropriate data from another settings file. The purpose in using simplistic predicate functions is to remove rule-related complexity (particularly evaluation of rules) from the externalised description and to place it within the observer system. As with the signatures and techniques, it is proposed that predicates will be accessed from the external description via a unique identifier and parameterised suitably. Their purpose is to evaluate its supplied parameters against the current state of the system and to return a result. In the example definition, Figure 35, a fluent "Available Resources" is used to check the currently available observation resources, which then influences the method of observation. The necessary information is provided by way of parameters – in this example, the first to indicate the scope of resource availability checking, and the second to indicate the desired level of availability. Providing the desired state of the predicate or fluent as a parameter reduces the complexity of the description:

1. in-code evaluation is only on an equality basis,
2. guarantees a Boolean return value as the desired value is either a match or it is not
3. constrains parameters to discrete values – thus encouraging fixed categorical definitions, rather than continuous values that are open to differing interpretations

Variables, Constants, Connectors and other Conjunctives

As is apparent from the Figure 35 example, some elements that do not fall into the categories discussed so far. These elements fall into the categories of Variables or Constants along with Conjunctive and other connecting statements. In the example shown, Variables exist in two main roles:

1. In the constant type, used to provide access to system scoping constraints (e.g. use of the SYSTEM constant to scope the applicability of a function check)
2. As a parameter, selected from a suitable (potentially function-specific) enumeration that defines the possible states of the fluent. For example, the Available Resources fluent is used with both the HIGH and LOW fluent-variables.

In these roles, they effectively a constant providing reference to a system entity, or system state, as measured by a function. However, they still must be represented in code and assigned to a particular value for exposure in the OBD. The example also shows two main types of connecting statement. The first defines the structure of the definition and highlights where the various significant component parts are found. The second type of conjunctive is effectively a Boolean join, which helps to define a conditional statement and enables more than one predicate-based function to be used in a single condition.

The structural connectors have been mentioned in each of the appropriate headings, but for reference are listed here:
ON EVENT *Signature Match* – this statement opens a new definition and indicates that the following parameters will specify the type of signature match to be expected
WITH … – this statement follows the signature match specification, and indicates the condition that should be fulfilled on signature match in order to proceed. This is where system policy can be included in the observer definition by way of system fluent checking.
THEN DEPLOY – This statement follows the condition and indicates which observer technique response is appropriate in the specified event and condition.

The Boolean joins will be reasonably self-explanatory to readers – particularly anyone with a familiarity in formal specification or any one of a number of programming languages. However, for completeness, they are listed here:
Statement ordering - The ordering in which statements should be evaluated could be specified by way of brackets surrounding groups of statements that should be evaluated before moving to the next.

- E.g. (X OR Y) AND Z is different to X OR (Y AND Z)

Simple Boolean algebra joins – i.e.

- AND – The new compound statement is true if and only if both halves evaluate to true
- OR – The new compound statement is true if one or both halves evaluate to true.
- A single translation function; negation – NOT – the new statement is the negation of the surrounded statement. True becomes false; false becomes true.

Therefore, the basic Observation Behaviour Definition (OBD) would take the following form, subject to the peculiarities of the exact externalisation format:

```
ON EVENT Signature Match (-signature match information-)
WITH (-fluent defined system policy-
   Joined by AND/OR, translated by NOT as required)
THEN DEPLOY -required observation technique-
```

Figure 36: Generic OBD format

A basic overview of the classes required to support this external format are shown in the following UML class diagram, Figure 37. This class diagram shows the basic division of data and responsibility, in terms of significant public methods. The precise method used for externalisation will be discussed later; however, in this class diagram, note how the requirements for externalisation are defined in an interface `Externalisable`, and the information pertaining to an externalisation's form is contained in the `ExternalisedForm` utility class. `OBDVariable` represents the data types that appear as parameter to fluents and expressions, and can contain any appropriate data type, as constrained on an instance-by-instance basis via the `DataTypeDefinition` mapping. The appropriate externalisation code and mapping is achieved at a class level via its `Externalisable` interface implementation. `BooleanExpression` and `SystemFluent` are considerably more constrained in that they must produce a Boolean value. They are defined as *abstract* classes and are also responsible for handling their own externalisation needs via the `Externalisable` interface. The class that represents the whole definition – `ObserverBehaviourDefinition` – is comprised of all the elements discussed above.

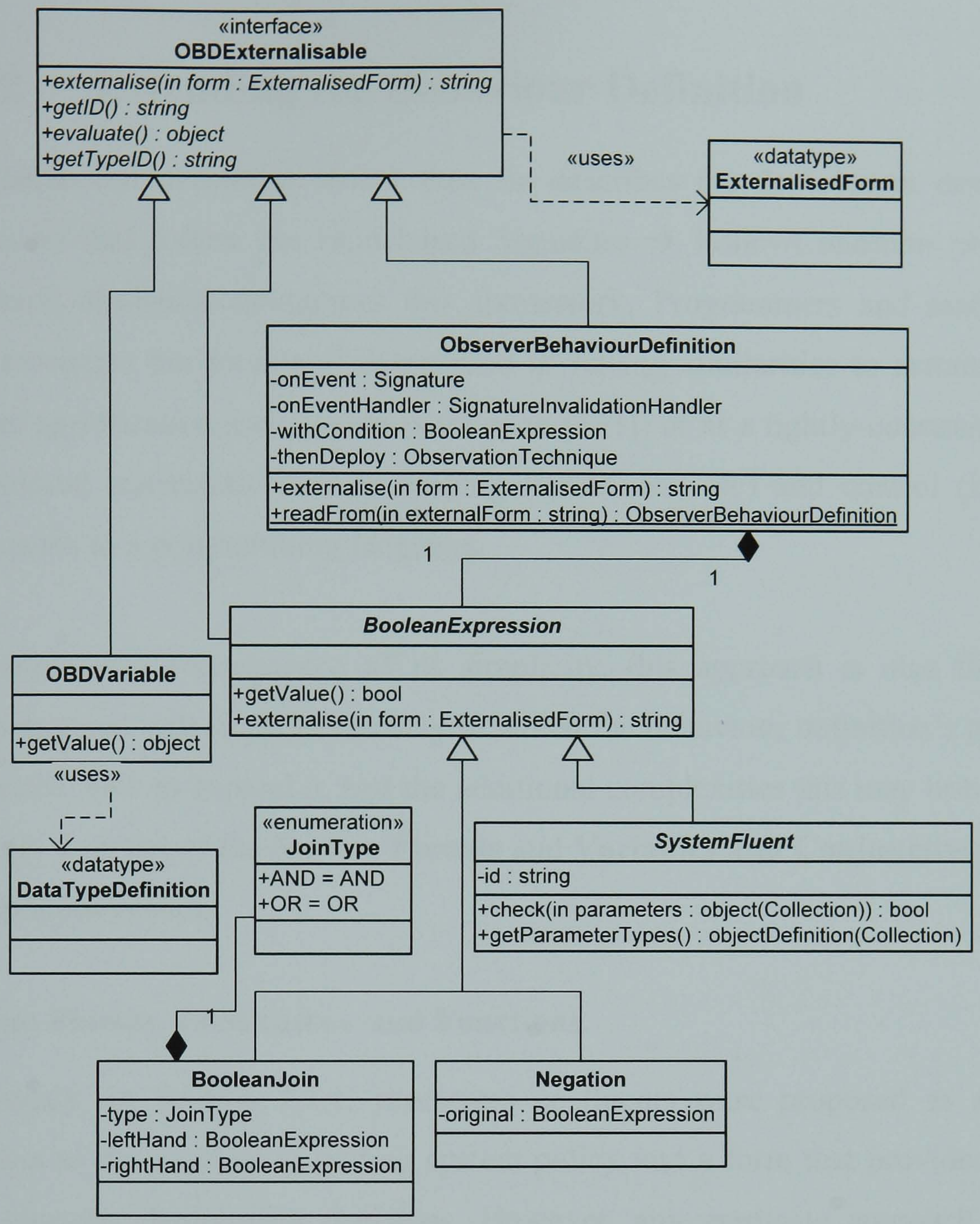

«interface»
OBDExternalisable
+externalise(in form : ExternalisedForm) : string
+getID() : string
+evaluate() : object
+getTypeID() : string
«uses»
«datatype»
ExternalisedForm
ObserverBehaviourDefinition
-onEvent : Signature
-onEventHandler : SignatureInvalidationHandler
-withCondition : BooleanExpression
-thenDeploy : ObservationTechnique
+externalise(in form : ExternalisedForm) : string
+readFrom(in externalForm : string) : ObserverBehaviourDefinition
1
1
BooleanExpression
+getValue() : bool
+externalise(in form : ExternalisedForm) : string
OBDVariable
+getValue() : object
«uses»
«datatype»
DataTypeDefinition
«enumeration»
JoinType
+AND = AND
+OR = OR
SystemFluent
-id : string
+check(in parameters : object(Collection)) : bool
+getParameterTypes() : objectDefinition(Collection)
1
BooleanJoin
-type : JoinType
-leftHand : BooleanExpression
-rightHand : BooleanExpression
Negation
-original : BooleanExpression

Figure 37: UML for Externalised OBD Support Classes

The next section looks at identifying limitations in the behaviour definition proposed thus far and additional propositions aimed at overcoming these limitations.

7.1.2 Extending the Behaviour Definition

The behaviour definition format thus far describes the deployment description for observers that follow the ECA-based Signature → Policy/Condition → Technique approach discussed throughout this framework. Programmers and mathematicians will recognise the format of description as having similarities to certain aspects of formal specification and algebra (such as in [121]) or as a tightly-constrained form of conditional statements such as looping (while, for, etc) and control (if, else, etc) statements in a programming language.

However, as a consequence of its simplicity, this approach is also limiting. The following text will examine the way in which the behaviour definition's present form is limited, how to expand it, and the additional complexities this may bring. Doing so requires a revisit of the **System Fluents** and **Variables and Conjunctives** parts from the previous section.

System Fluents, Predicates – and Functions

Previously, in Section 7.1.1, predicates or fluents were proposed as a simplified solution to the problem of getting system policy into a form that provided easy input into observer deployment decisions. However, any predicate approach (due to its Boolean-only result form) would prevent the description specifying observer behaviour for a range of values in a single statement. The only ways to specify a range of acceptable values with the fluent approach is to:

- Specify a range of values via parameters to a range-checking predicate, or
- Perform a Boolean OR join between a set of statements enumerating all elements within that range.

Therefore, it may prove worthwhile to permit the use of a more general function-based approach that operates in a similar way to the fluent-based system described previously. In addition to returning a Boolean outcome, functions will be able to directly inspect system variables – though again, only those deliberately exposed. The notation proposed for the functions would be as for the previously-discussed system fluent – uniquely identifiable and parameterised as necessary; however, functions would be capable of returning a wider range of data types. Again, this is not without disadvantages. Just as the "fluent" approach was relatively simple, the "fluent + function" approach adds complexity to the whole definition format, namely:

Complications surrounding variety of return types – the fluent approach was constrained to Boolean-only and as such provided the go-ahead indication for observation - or alternatively, prevented a strategy being deployed. If other data types are considered within an observation strategy's conditional functions, then there is the added complexity of validating and evaluating the types used. Opening the function-based approach to allow any data type creates type-related problems that are approached in a variety of different ways by different programming languages.
Type handling can be strongly or semi-strongly-typed as in languages such as C# or Java, where typing errors are detected before and at execution [122], or can operate on a weaker Variant (such as Visual Basic) basis, whereby type conversions are automatic and wherever possible meaningful. However, a significant issue with automatic type-conversion is that in the case of specification mistakes; instead of an explicit error being generated, the automatically-converted data may lead to unexpected behaviour. Conversely, the problem with strong typing is that suitable error reporting must be implemented to report the error to the specification editor.

Conditional statements - The condition specifying whether a given observation strategy should be deployed must still reduce to a Boolean condition – no matter how the condition is made up. Taking the example of an "if-then" statement in a programming language, it is perfectly acceptable for it to be comprised of a variety of data types, providing they combined to make Boolean conditions. For example, assuming x and y are integer variables, "if (x > 3 AND y < 2) ..." is a perfectly valid Boolean statement, whereas "if (x AND y)" is, generally speaking, not. The addition of automatic type conversion (as above) complicates this process further, as the nature of the automated conversion from integer to Boolean is not specified.

Allows free interpretation of functional results – with the fluent approach, the desired state value is provided as a parameter (from an enumeration of possible state values) while the result is provided as a Boolean indication. Allowing functions to return a continuous value means that the result may be open to different (mis) interpretation.

Requirement for (more) conjunctive operators – with Boolean-only type constraints, any connecting statements only needed to fulfil simple Boolean join requirements as discussed in the previous section. Additional data types bring their own requirements for conjunctives, which will be discussed in more detail shortly.

In summary, while the addition of different types can facilitate greater functionality in the behaviour definition, it has a variety of knock-on concerns to the manner in which statements are assembled, and the guaranteed validity of the specification as a whole.

Connecting statements

With Boolean-only values, the required connecting statements are relatively limited. As the "WITH..." condition expects a Boolean value, the only conjunctives needed are those to join a variety of Boolean statements (i.e. the fluents) together, as shown in the previous section. However, when dealing with non-Boolean values methods must be made available to translate (sets of) these values to Boolean.

One of the most widely-applicable methods of translating a non-Boolean value to Boolean is by way of an equality test. Comparing a non-Boolean value to see if it is the same as something else gives a Boolean result. With many data types, a Comparator-type model: Strategy and Interpreter patterns [27], are suitable and provide functionality above and beyond that of a simple equality test. The Comparator model requires that data types each provide functionality to compare a values and return a result to indicate whether one is equal to, greater than, or less than the other. As Boolean return values are required, this necessitates three different operations – an equality test, and greater-than and less-than tests. Mathematical-style notation is used to represent these operations; the operator sits between the two statement values that need testing. For reference, the additional connective operations are:

- EQUALITY (=) – Returns true if and only if the value of the statement on the left hand side equals that on the right hand side.
- GREATER THAN (>) – Returns true if the value of the statement on the left hand side is greater than that on the right hand side
- LESS THAN (<) – Returns true if the value of the statement on the left hand side is less than that on the right hand side.

The precise behaviour of these connective operators depends on the types of data they are comparing. Comparisons of two number data types are quite simple; they should be compared mathematically. Equally, string and character types have well-established patterns for matching and comparisons [123]. The externalisation method should, in common with the Comparator model, place the responsibility for comparisons with the type definition for the data being compared. Specifically, given the relatively weak typing of this externalised definition, the comparison should compare using a meaningful data commonality if possible. This may involve boxing/un-boxing [124] and assessing several data commonalities – akin to the Variant type. A simple example involves a comparison between the "35" (a character string) and 35 (a numeric integer). At a code level, a simple equality test would return false, or even generate an error because of the differing types.

In summary, the extended Observation Behaviour Definition (OBD) will extend the existing format, and featuring the same "ON EVENT, WITH, THEN DEPLOY" structure. However, adding another Boolean evaluation – Comparison – would allow the WITH condition to contain greater complexity, as indicated below in Figure 38:

```
ON EVENT Signature Match (-signature match information-)
WITH (-fluent defined system policy-
 -equality/value comparisons of system functions-
   Joined by AND/OR, translated by NOT as required)
THEN DEPLOY -required observation technique-
```

Figure 38: Extended OBD Format

The following UML class diagram is based on Figure 37, revised to show the additionally-required classes to support the function-based and comparator operations. Note how `Function` and its derivatives are shown inheriting from the `OBDVariable` class; data type permitting, they could be used in place of a straightforward variable – replacing the provided value with a calculation returning an appropriate value.

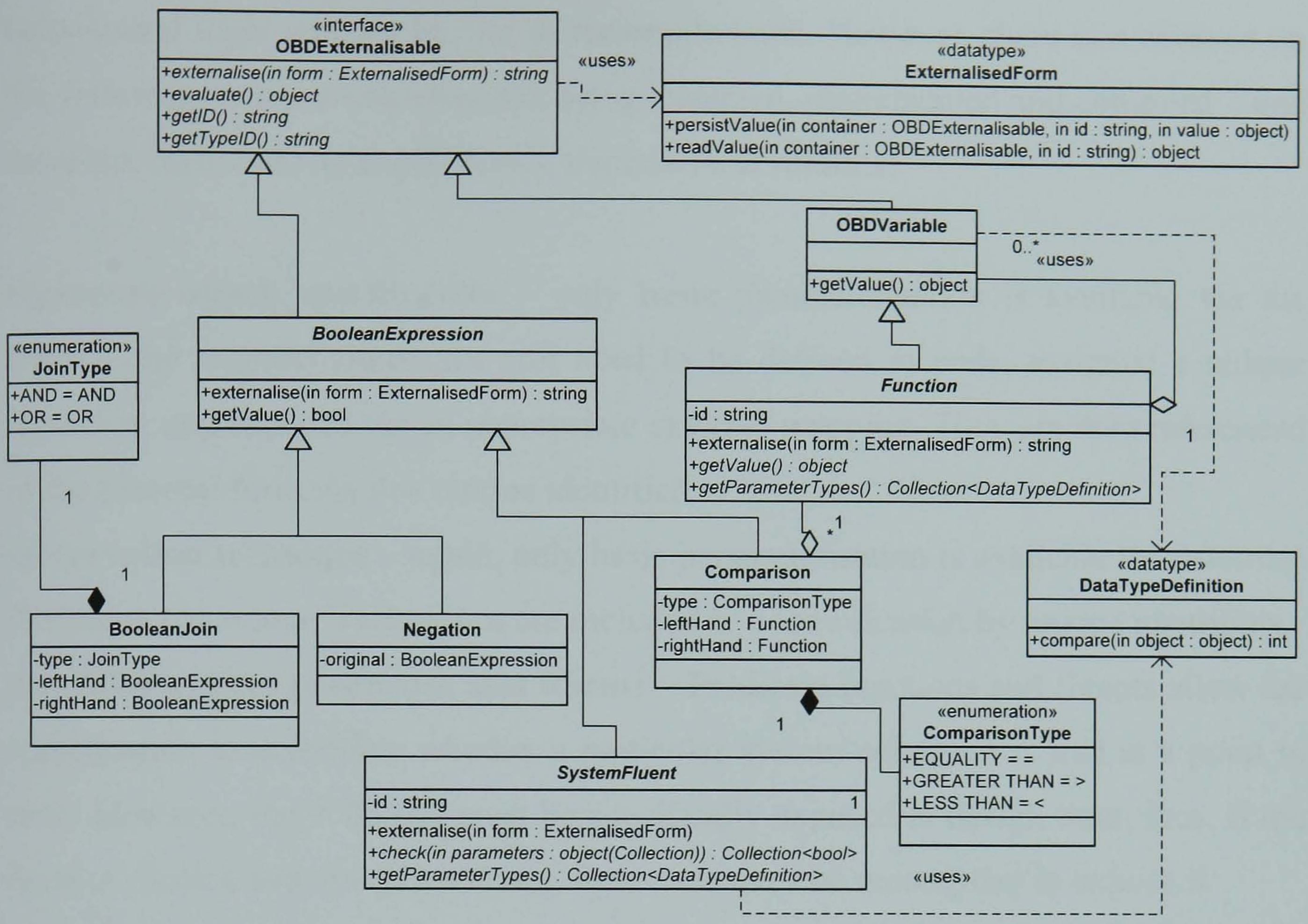

«interface»
OBDExternalisable
+externalise(in form : ExternalisedForm) : string
+evaluate() : object
+getID() : string
+getTypeID() : string
«uses»
«datatype»
ExternalisedForm
+persistValue(in container : OBDExternalisable, in id : string, in value : object)
+readValue(in container : OBDExternalisable, in id : string) : object
OBDVariable
+getValue() : object
0..*
«uses»
«enumeration»
JoinType
+AND = AND
+OR = OR
BooleanExpression
+externalise(in form : ExternalisedForm) : string
+getValue() : bool
Function
-id : string
+externalise(in form : ExternalisedForm) : string
+getValue() : object
+getParameterTypes() : Collection<DataTypeDefinition>
1
1
1
.1
Comparison
-type : ComparisonType
-leftHand : Function
-rightHand : Function
«datatype»
DataTypeDefinition
+compare(in object : object) : int
BooleanJoin
-type : JoinType
-leftHand : BooleanExpression
-rightHand : BooleanExpression
Negation
-original : BooleanExpression
1
«enumeration»
ComparisonType
+EQUALITY = =
+GREATER THAN = >
+LESS THAN = <
SystemFluent
-id : string
1
+externalise(in form : ExternalisedForm)
+check(in parameters : object(Collection)) : Collection<bool>
+getParameterTypes() : Collection<DataTypeDefinition>
«uses»

Figure 39: UML diagram showing additional support classes for extended OBD

Section 7.1 and its subsections have provided an overview of control elements in the observation model that are responsible for specifying the behaviour of the structural framework. The result is specified as a Behaviour Definition; allowing mapping between signatures and techniques, along with basic system policy to be specified by in a set of data objects. These are evaluated to determine how exactly the framework should behave when it encounters a signature match. Specifying system control logic in a separate layer (i.e. the definition's data objects) allows an externalisation to translate between definition objects and a suitable external form (e.g. a flat file, an XML file, or a database of rules); effectively creating an interpreted specification of structural behaviour. Externalising this behaviour specification is discussed in more detail later in the next chapter; the next section examines the need for further preparation of the behaviour definition for externalisation.

7.2 Exposing Components' Behaviour and State

The designs discussed prepared a Interpreter pattern, code-level specification of behaviour for the observation framework's structural operations. Use of a suitable externalisation technique would allow a great degree of control over framework behavioural logic without having to recompile code. However, there is a reliance on the following components/elements being designed, implemented and compiled – and therefore their makeup is effectively immutable at runtime:

Signature match specification – only basic parameterisation is available via the OBD – the mapped signatures still need to be defined in code, assigned a unique identifier, and exposed via an appropriate external mapping. They are then referenced in the external form via this unique identifier.
Observation technique – again, only basic parameterisation is available to customise technique behaviour. Techniques are included in a specification by unique identifiers.
Available system predicates and fluents – Predicate functions and fluents allow the specification to determine whether a particular system condition is true at a point in time. However, these fluents must be specifically exposed at design time; thus, if the need to check a new predicate arises, then code must be recompiled to expose it.
Available system functions – functions extend the above functionality to allow a particular system variable to be examined or the value of a system setting to be used in the OBD. This again relies on a model of explicit exposure, such that any state variables are mapped within code to their unique identifier and external link.

Effectively, the OBD designs thus far allow for the observer deployment decision logic to be exposed in whichever external form is most appropriate. Referring back to Chapter 5, the externalised OBD effectively takes much of the responsibility from the **Technique Selection** and to some extent **Deployment Coordinator** components. In order that the observer system could function, basic core-level functionality would be expected of these components similar to that outlined in Section 6.2.2. However, the OBD would specify much of the mapping associated with the **Signature/Technique Associator** and basic policy that would otherwise reside in the selector, such as access to the system's **Resources** object via suitable function exposure.

However, flexibility is limited as event triggers (signatures), responses (observation techniques), and data that makes up the connecting conditions (fluents, functions and variables) must all be predefined in code, and the only configurable aspects must be explicitly exposed via parameterisation.

7.3 Flexibility in Signature and Technique Definitions

In the cases of signature and observation techniques, greater flexibility could be obtained by specifying the makeup of signatures and techniques entirely in an external form. However, simply making the entire signature/technique an interpreted unit of execution, while increasing flexibility, is perhaps a little naïve.

Considering that the externalisation model thus far discusses the *limited* extreme (i.e. signature and observation technique unit code must be prewritten, compiled and explicitly exposed), the *open* extreme would allow a complete code-like definition in external form. The designs presented make use of a basic Interpreter pattern implementation demonstrating basic language constructs expected of the OBD.

However, introducing a fully-featured logical grammar specification to support the OBD brings forward concerns relating to the operation of the system-level observers. Firstly, completely novel (i.e. unforeseen at design time) signatures and their applicable techniques are likely to require additional observation support at the system-level. It is possible that the functionality of the system-level observers may in itself be too complex to represent in a logical runtime adjustable grammar. Additionally performance, scalability, or system-critical concerns of the system-level observation may necessitate a traditional design and implementation approach. In such cases, runtime adaptation, while supported at the structural level, would require more traditional redeployment of the system-level observers. In this case, it is arguable that it would still be beneficial to be able to keep the system running as-is, and then introduce the new observation subsystem as a whole, without recompiling or taking down the observer subsystem at all.

However, the virtual-machine nature of modern OO languages such as Java and C# support approaches such as reflection, dynamic class-loading, and even hot swapping of byte-code (part-compiled classes). Reflection permits the external form to contain only a named reference to the newly revised or added code-unit, rather than a complete definition of the logic contained within. Equally, dynamic class-loading and hot swapping in VM-based languages allow for the part-compiled code to be loaded after system deployment and reloaded by the JVM (Java) or CLR (MS .net) as they are modified. However, this approach raises two major concerns:

Security concerns – Writing code that instructs an executing VM to load new or replacement code exposes a level of security risk. Assuming the developer has control over the entire host environment, the security concerns can be mitigated to some extent by adequate security on the hosts – i.e. the code is at no greater risk than the rest of the hardware system. However, if the observer system requires access to a variety of hosts and security arrangements, the framework requires a suitable method of validating code. Ensuring security concerns on swapped-in code is a research field in its own right (e.g. [125]) and a solution is outside the scope of this research.

Interface concerns – Generally speaking, hot swapping places fairly stringent demands on the interfaces of replaced classes. The strictest requirement encountered is that the replacement class must be absolutely interface-identical to those they replace; they cannot define new methods, remove unused methods or override existing methods [126] – though there has been research to overcome even this limitation [127, 128]. However, given that the system interaction with signatures and techniques are well-specified, this should not pose a particular issue.

However, if entire code-level edits and swaps are envisaged – moving toward dynamic programming – it is perhaps better to look to programming paradigms that better support this notion. Several methods of increasing dynamism in software code are available to programmers and system designers. The most flexible approach centres on the concepts of Meta-programming. While only directly supported in the languages under consideration via reflection, others have explored methods of simulating and closely-reproducing advanced meta-programming [129].

In the context of the OBD, declarative programming would provide useful generative functionality. Just as Signatures and Techniques are examples of Template patterns, a generative approach treats these as class templates. At runtime, the templates would be used in conjunction with observed states to create correctly-specified signatures and techniques in executable code. Equally, more widely-spread meta-programming techniques such as Reflective Programming [130] or any similar approach that relies on a level of descriptive meta-information regarding the code units are suited to this high level of external software influence and runtime dynamism.

Industrial practice and research is active in dealing with some of the initial concerns regarding class loading, class reloading and other *hot*-VM issues (as detailed above) since release 2 of Java - effectively the early 2000s [131-133]. Therefore, remembering that this software framework is intended to extend the traditional Observer pattern and applying existing engineering techniques; some of the levels of flexibility discussed above are an over-complication of the problem. This framework intends that system-level observation will follow more traditional OO patterns, and as such, this degree of flexibility adds a great deal of complexity and most likely would lie redundant without sufficiently dynamic logic in the observer framework – i.e. at the system-level observers. Therefore, the OBD will be based on a simplistic grammar-oriented [134] approach to meta-programming:

As such, extending the OBD to allow a useful degree of inline definition and customisation for signatures and techniques - without creating a fully-featured interpreted language - requires a clear definition of:

- The data available to signatures/techniques (specified in code interfaces)
- The data expected from signatures/techniques (as above)
- The calculations, libraries and other algorithms available to signatures and techniques for their signature matching and target-set producing procedures.

The following subsections will summarise these definitions for the signatures and observer techniques respectively, concluding with an overview of the format and support classes for the extensions to the OBD modelled so far.

7.3.1 Formalising the OBD Signature Definition

In the currently-proposed OBD format, the precise type of signature is indicated by a unique identifier, and the Modelled Element on which it should operate. As mentioned above, in order to try and overcome some of the limitations inherent with this simplistic approach, this section will examine the manner in which the signature could be specified in more detail "in-line" via the externalisation. To introduce how the OBD could give a more detailed definition of a structural signature, let us first recap some of the significant data requirements both in and out for the Signature type from Chapter 5's class designs:

- `checkedModel` - The Modelled Element on which it performs signature checking. This is already considered to some extent by the OBD proposal in Figure 35. This shows a signature being referenced by both its unique identifier and a signature "scope" – in the example case, SYSTEM to indicate it should include the entire Modelled Element neighbourhood.
- `getAssociations()` – The signature class defines a method to allow access to associated techniques via a specified Associator. In the OBD, this mapping/association is already dealt with by virtue of the format of the definition – its purpose is to map signatures to techniques via certain conditions.
- `createInvalidationTrigger()` – hardcoded signatures generate a Invalidation Trigger which specifies how the signature should be rechecked. As discussed previously, this may range from complex statistical evaluation of model change events to a simple polling approach; effectively evaluation of time-based or event magnitude constraints. However implemented; this completes the loop controlling signature checking as discussed in Section 6.2.2. As such, invalidation must feature in the OBD signature definition in some form.
- `getMatch()` – Specification of how to check the metrics to double precision (referred to as *match quality* throughout the designs, as it gives an quantitative indication of signature matching). The designs encourage signature checking to use `ModelledElementAlgorithm` objects to perform necessary manipulation of sets of Modelled Elements. The OBD must specify how this value is calculated.
- `checkMatch()` – Specification of metric check as Boolean. In implementation, this may be a simple check of the value produced by the `getMatch()` method.
- *(dependency on* `ModelledElementAlgorithm`*)* – The Signature may make use of Modelled Element Algorithm(s) to perform set translation for comparison / statistical purposes as part of the Signature checking process.

Therefore to create an inline OBD Signature definition, two significant features must be facilitated, either by flexible definition, or reference to a hardcoded element. Firstly, a flexible and simple match specification must be determined, while secondly, the OBD-based Signature must specify how it is invalidated and therefore rechecked.

Match Checking

The signature definition in OBD defines two match specifications:

- Continuous value match (signature match = 0 to 1) calculation
- Boolean match (signature match = true or false) calculation

In order to retain a simple, yet configurable approach, the OBD continues to extend functional exposure via the abstract `Function` mapping, with a specialisation to allow access to a system-defined `ModelledElementAlgorithm` object via a new class `OBDAlgorithmFunction`. This class exposes `ModelledElementAlgorithm` instances as required; implementations could explicitly map instances or use a reflection approach to find the appropriate Algorithm class at runtime. The class `OBDAlgorithmFunction` must correctly construct `ModelledElementAlgorithm` objects with required parameterisation.

The algorithm results can be examined using special `Function` implementations that allow statistical investigation of the result set, such as sizing functions. Given that signatures discussed thus far are structural, a special implementation of `Function` could implement a variety of graph-manipulating and statistical functions to make them available to the OBD. If the system is extended such that other signatures types are envisaged, as discussed in the architectural chapters, then new `Function` subtypes could be created exposing suitable utility calculations.

Invalidation Trigger

The signature rechecking loop is only closed by the signature itself specifying an invalidation handler that captures `ModelChangeEvents` and determines when a significant magnitude of change has occurred. Earlier designs left implementation details open, allowing as much complexity as required. However, in order to allow specification in OBD, Invalidation Handling must be significantly constrained:

Simple – Signature rechecking is polling-based – The invalidation handler generates a new event every *x* Model Change Events, or alternatively every *t* time ticks.

Complex – Signature rechecking is calculated based on its estimated likelihood of change given the captured model change events. This involves greater considerations than simply the number of events; and may involve a model of what is happening to the previous structure; a mini-signature checking model in itself.

Given the potential for complexity, the reference designs presented here will concentrate on the Simple-type Handler. Externalising a Complex-type Handler is not impossible; rather that it would result in complexity that would be difficult to take advantage of without increasing the complexity of the signature specification – i.e. therefore would be better placed directly in the target language code.

The Invalidation in OBD will expect a `BooleanExpression` statement that has access to two invalidation-specific predicates. They will allow specification of either the *time passed* (in milliseconds) or the *number of model change events* since the last invalidation event. A new concrete type `OBDSignatureInvalidationHandler` subclasses `SignatureInvalidationHandler` providing the extra functionality. This requires construction with a `BooleanExpression` parameter, specifying its invalidation criteria. In order to facilitate rechecking, it records the number of `ModelChangeEvents` it has received and the time since its last `InvalidationEvent`.

Additional mapping is included in order that the two special-use predicates link back to the appropriate `OBDSignatureInvalidationHandler` instance; localising the scope to the Signature and its Invalidation Handler.

Proposed OBD Signature Definition

To summarise, the proposed OBD-based Signature Definition requires the following significant attributes in the external form. Figure 40 shows the definition in the same format as previous OBD-type definitions:

```
SIGNATURE New-Signature-Identifier
MATCHING
-VALUE  (-calls to named Function implementations-
      Joined by calls to Graph Stats functions as required
      Double-precision value expected)
-BOOLEAN (-BooleanExpression- that can self-reference -MATCH
         value   if   required,   joined   by   AND/OR,   NOT,   or
         Comparisons as required)
INVALIDATED (MODELCHANGEEVENTS_PASSED(x) OR TIME_PASSED(t))
```

Figure 40: OBD Signature Format

It is presented in a declarative form; in common with the other externalisable objects introduced during this section, it has a unique identifier. Referring back to Figure 38 showing the extended OBD format; Figure 40 complements this approach, such that the behaviour definition can refer interchangeably to explicitly-mapped hardcoded signatures, or those defined in the format shown above. The following class diagram, Figure 41, shows the OBD Signature classes, and their organisation with previously defined OBD classes. `OBDSignatureDefinition` is the central class, both implementing `Externalisable` while inheriting from `Signature` – linking the OBD and the Signature class. `OBDSignatureInvalidationHandler` and `OBDAlgorithmFunction` also have dual roles, providing OBD externalisation functions for a simplified implementation of the abstract class `SignatureInvalidationHandler` , and a externalised link to suitably mapped (or reflected) implementations of `ModelledElementAlgorithm`, respectively.

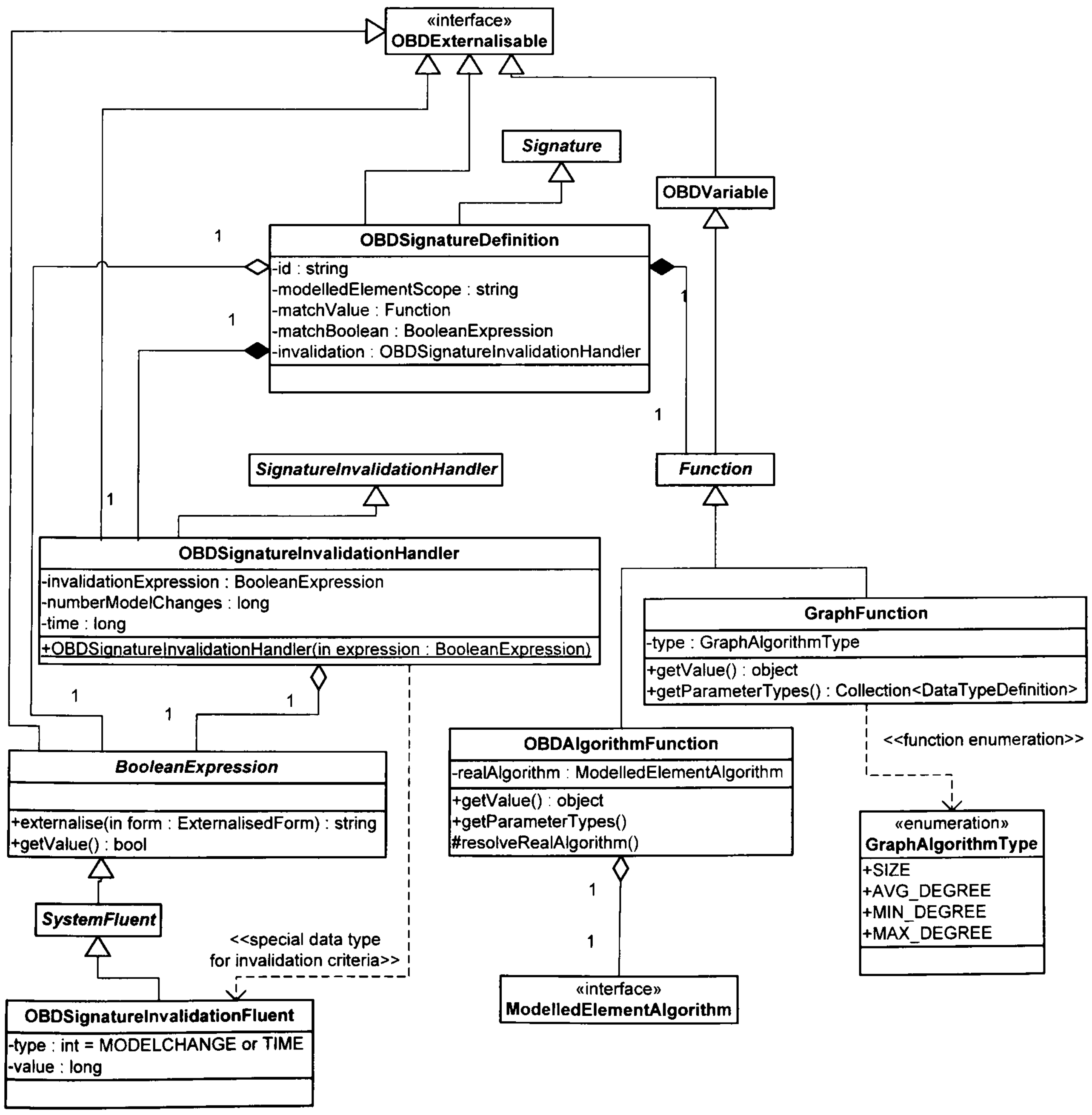

«interface»
OBDExternalisable
Signature
OBDVariable
OBDSignatureDefinition
-id : string
-modelledElementScope : string
-matchValue : Function
-matchBoolean : BooleanExpression
-invalidation : OBDSignatureInvalidationHandler
1
1
1
1
SignatureInvalidationHandler
Function
OBDSignatureInvalidationHandler
-invalidationExpression : BooleanExpression
-numberModelChanges : long
-time : long
+OBDSignatureInvalidationHandler(in expression : BooleanExpression)
GraphFunction
-type : GraphAlgorithmType
+getValue() : object
+getParameterTypes() : Collection<DataTypeDefinition>
<<function enumeration>>
1
1
1
1
BooleanExpression
+externalise(in form : ExternalisedForm) : string
+getValue() : bool
OBDAlgorithmFunction
-realAlgorithm : ModelledElementAlgorithm
+getValue() : object
+getParameterTypes()
#resolveRealAlgorithm()
«enumeration»
GraphAlgorithmType
+SIZE
+AVG_DEGREE
+MIN_DEGREE
+MAX_DEGREE
SystemFluent
<<special data type
for invalidation criteria>>
1
1
«interface»
ModelledElementAlgorithm
OBDSignatureInvalidationFluent
-type : int = MODELCHANGE or TIME
-value : long

Figure 41: Significant OBD Signature support classes

The next section repeats this process for the `ObserverTechnique` class in order to create an OBD-compliant and more flexible technique specification.

7.3.2 Formalising the OBD Technique Definition

`ObservationTechnique`, when compared to `Signature` is a relatively simple class with three main concerns on attributes and method:

- `getTargetSet(Collection<ModelledElements>)` - The Technique must be told from which Modelled Elements it should select targets. As with the signature scope concern regarding Modelled Elements, this is partly covered in the OBD proposal in Figure 35. The technique is shown being applied to a system scope variable alongside an observation priority parameter (HIGH) – in the example case, the scope variable used was SYSTEM to indicate it should include the entire Modelled Element neighbourhood.
- `getAssociations()` – The Observation Technique class defines a convenience method to allow access to associated signatures via the specified *Associator*. In common with discussion on signatures in the previous section, this mapping is already handled by the existing OBD format.
- `algorithm` – The observation technique is composed of one or more algorithms, specified by the `ModelledElementAlgorithm` interface. Just as with the Signature, use of the algorithm was designed to allow for common element-selection tasks to be extracted and reused by other Techniques and Signatures to avoid duplication of code. It is also useful for the OBD externalisation as it allows some control over what the technique does without having to create a complex algorithmic-capable language in the OBD.

Therefore, in order to define the OBD Technique, there is the issue of specifying how the OBD-specification can generate target sets with a flexible use of the algorithm, rather than relying entirely on the hard code. As the OBD algorithm-related design work was undertaken in the previous section, this greatly simplifies the new design requirements for the OBD Technique.

Creating the Target Set

Hardcoded Observation Techniques require a single `ModelledElementAlgorithm` object. Techniques requiring more complex functionality can extend the root class and add features, such as combining multiple algorithms, using additional logic, or manipulating the target set without using an algorithm object.

Allowing OBD-defined techniques to freely manipulate the target set brings the same kinds of complexities rejected previously; therefore, the OBD-techniques will be limited to the use of `OBDAlgorithmFunction` objects as introduced in the previous section. However, as their return and input parameter types are the same, they can be nested, thus allowing combined technique functionality to be introduced at runtime.

As with their use in the OBD Signatures, instances of the `OBDAlgorithmFunction` class's subtypes are responsible for permitting the external OBD-referenced algorithm to map through to the appropriately-identified implementers of `ModelledElementAlgorithm` and to populate them with the correct parameterisation.

Proposed OBD Observer Technique Definition

To summarise, the proposed OBD Observer Technique is simpler than its Signature partner. This is partly because the technique requires less information and interaction with other components, and partly because the significant requirements had already been fulfilled for the OBD Signature definition. The proposed makeup of the OBD Observer Technique is shown below (Figure 42), in the as-previous OBD format.

```
TECHNIQUE New-Technique-Identifier
TARGETS
-FROM  (-desired source set of elements; e.g. SYSTEM-)
-USING (-OBD Algorithm Function identifier; may nest-)
```

Figure 42: OBD Observer Technique Definition

It follows the same form as the Signature definition; it is effectively a declaration. This allows Techniques to be used directly from hardcode if mapped, or to be built up from algorithms in OBD and used elsewhere in the definition.

The new OBD Technique involves a single new class, `OBDTechnique`. It translates the source element specification, along with populating its referenced instance of the `OBDAlgorithmFunction` class. It is shown along with its compositional classes and amongst the significant typing classes within the OBD in Figure 43.

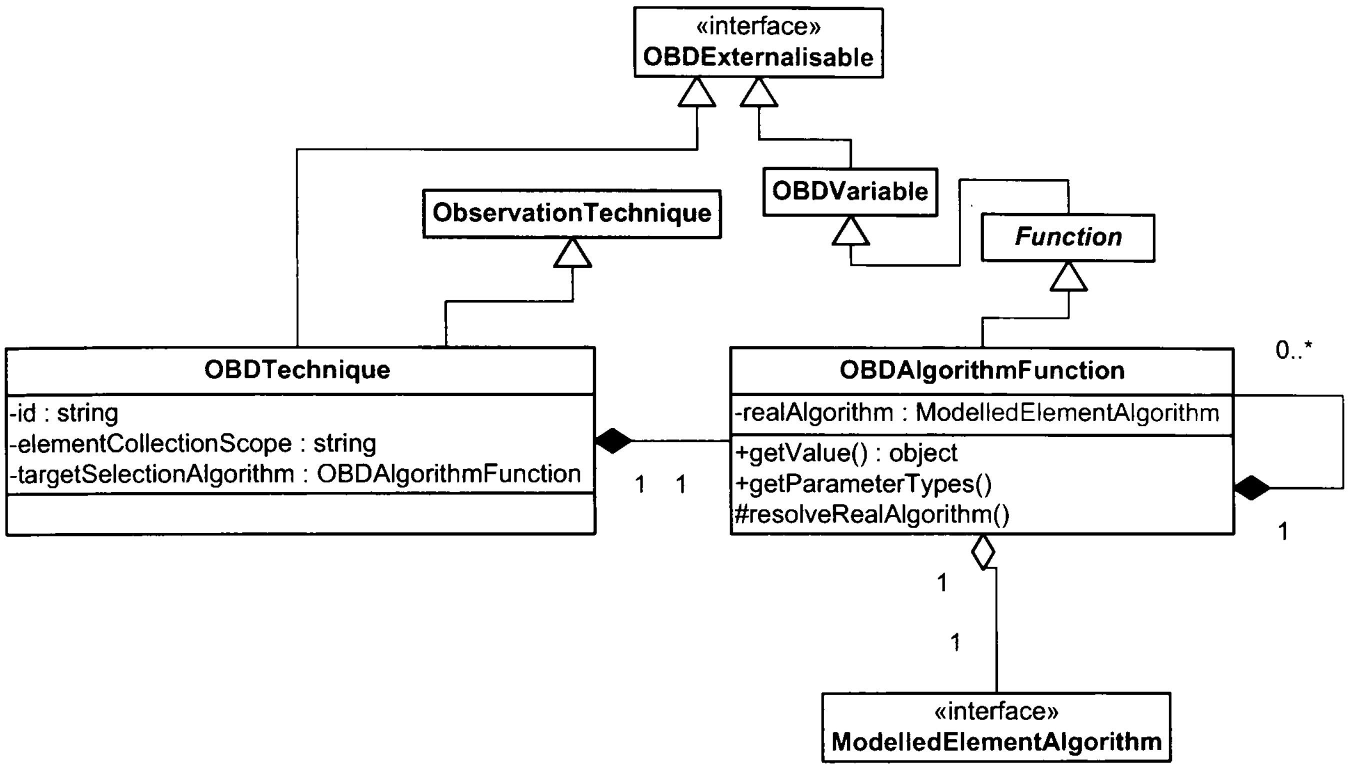


Figure 43: OBD Technique and support classes

The next section draws together the parts of the OBD in summary before moving on to the method of externalisation. It also addresses some potential implementation issues that have not yet been covered in the overview design form.

7.4 The Externalised Specification

The purpose of this section is to piece together previous incarnations of the OBD, along with their OBD-compliant simple definitions of Signature and Techniques. This creates a cohesive overview of the OBD approach to externalised specification, before discussing the externalisation mechanism used to validate this approach.

The externalisation relies on a series of bridging classes which are responsible for translating external elements into classes already presented in the Chapter 5 designs. External elements are marked with a special interface, `OBDExternalisable`, which defines the necessary methods to manage the translation to and from an external form, including a requirement for a unique identifier, and type-identifier strings.

Implementers are responsible for implementing these methods and for ensuring that relevant class-specific data is persisted and read correctly from the external source. For example, the `OBDSignatureDefinition` class both implements the `OBDExternalisable` interface and inherits from `Signature`. As such, it is responsible for ensuring that all the information it requires to construct a `Signature` is suitably packaged in its external form. It must externalise the *unique id* and *type id* (both of which are required for all OBD Externalisable types), and references for the following signature-specific information:

- Modelled element scope (A Variable)
- Match value (A Function)
- Match Boolean value (A Boolean Expression)
- Invalidation handler (An OBD Signature Invalidation Handler)

By referring back to any of the class diagrams in Figure 39, Figure 41, or Figure 43, it is apparent that each of these values are references to other OBD Externalisable types, and therefore the responsibility for persisting each of these values will be referred to the appropriate type. Thus, the designs avoid unnecessary ties to a particular format, with `ExternalisedForm` responsible for the serialisation implementation.

The final matter to cover before presenting an overview of all the externalisation and relevant classes is to explain how "real" code elements should be exposed via the external interface. As mentioned in previous sections, Signatures, Techniques, certain important system variables and states (particularly items related to a structural observer, such as its collection of modelled elements) can be exposed to the external form. Previously, this had been discussed from two points of view:-

- Explicit Mapping – This approach maintains a mapping structure (either per structural observer or per OBD Externalisation type). It requires the definition of the new mapping structure and requires that the framework is modified such that required data and functions are explicitly exposed.
- Open access via reflection – This approach does not use any explicit mapping and as such, eliminates the need for a modification to the framework. This relies on the OBD classes that need access to the framework using fully qualified host language names to access them. While this provides open access

to *code*, it does have some drawbacks. Firstly, it means that the system becomes host-language dependent, and that the externalisation process must either map meaningful names in the external form to fully-qualified classes and method names, or the external form itself must contain direct string references to host-dependent class or method names, e.g. "`structural.techniques.ScaleFreeTechnique`". Equally, the reflection approach is limited in that it is still unable to access existing objects within the framework – so would only be suitable for those instances where creating new objects are required (e.g. creating a new Technique)

Therefore, it is clear that while allowing reflection may in some cases bring greater flexibility, the externalisation must provide a global method of mapping data objects from the structural observer (global meaning *associated with a single structural observer*). Certain OBD classes may need to extend this functionality in order to expose variables with a tighter scope for calculation purposes.

7.4.1 The OBD Format

This section briefly consolidates the various OBD-format definitions that have evolved during this chapter. The schema is shown in a linear format in Figure 44, simply for convenience of presentation. The OBD requires that the externalisation format is capable of delineating each section such that elements can appear in any order, providing that the attribute location is respected (i.e. attributes such as TARGETS must always appear within a TECHNIQUE element). SIGNATURE and TECHNIQUE have been explained fully in Section 7.3 and its subsections, and the behaviour definition form is similar to that initially described by Figure 38; similar to the signatures and techniques - signified by the OBSERVER BEHAVIOUR element. The whole OBD is split into separate sections for Signature, Technique and finally, Behaviour definitions.

The next section presents an overview of the OBD supporting classes combined together from throughout this chapter.

```
SIGNATURE DECLARATIONS
  SIGNATURE New-Signature-Identifier
    MATCHING
    -VALUE  (-calls to named Function implementations-
       Joined by calls to Graph Stats functions as required
       Double-precision value expected)
    -BOOLEAN (-BooleanExpression- that can self-reference
       -MATCH value if required, joined by AND/OR, NOT,
       or Comparisons as required)
    INVALIDATED (MODELCHANGEEVENTS_PASSED(x) OR TIME_PASSED(t))
  [...]
-END SIGNATURES

TECHNIQUE DECLARATIONS
  TECHNIQUE New-Technique-Identifier
      TARGETS
      -FROM  (-desired source set of elements; e.g. SYSTEM-)
      -USING (-OBD Algorithm Function identifier; may nest-)
  [...]
-END TECHNIQUES

BEHAVIOUR DEFINITIONS
  OBSERVER BEHAVIOUR
    ON EVENT
      Signature Match (-signature's unique id-)
    WITH (-fluent defined system policy-
      -equality/value comparisons of system functions-
      Joined by AND/OR, translated by NOT as required)
    THEN DEPLOY -required observation technique-
  [...]
-END BEHAVIOUR
```

Figure 44: Overview of OBD Format

7.4.2 The OBD Classes

This section consolidates classes required for the OBD approach, along with introducing some new features to existing classes and new classes. The new features are introduced where required to support some of the issues identified in Section 7.4, including variable and class mapping and reference management in general.

This explains how elements and `ExternalisedForm` translator implementations can manage the issue of dual-purpose references – that is, a reference to other OBD *or* external elements *or* a reference directly to a Structural Observer type.

In common with the design summary of Chapter 5, the classes will be shown in lightly annotated UML class diagrams, and where classes appear in several diagrams in order to show relationships, they will be shaded to indicate their duplication.

The first class diagram, in Figure 45 shows the classes that make up the backbone of the OBD externalisation. They represent the various operatives, data-types and calculating functions that are available within an OBD definition. It also shows the `OBDExternalisable` interface and `ExternalisedForm` that are jointly responsible for performing the actual serialisation and de-serialisation operations.

Paying particular attention to the entirely new classes; there are two new identifier classes – `OBDIdentifier` and `OBDTypeIdentifier`, which are used to formalise the way in which elements are identified and can refer to one another.

This identification functionality is made use in the second new class – `OBDMapping` – which holds references between one or more Structural Observers and the OBD system, allowing OBD elements to locate explicitly exposed data in code identified by a unique instance of `OBDIdentifier` and to make references to other elements using the same identification and lookup process.

Using `OBDMapping` to create maps between the OBD and Structural Observer variables or constants requires only a reasonably simple and familiar approach – a key/value pair mapping (such as a Hash Table or Set) to a suitable data-carrying mutable object is sufficient. However, in the case of mapping functions, more explanatory detail is required. `OBDMapping` must effectively provide higher-order functions; it must provide a way to associate and then return a method call to a unique ID - not just the method's return value at the time of mapping.

`OBDHigherOrderFunction` is responsible for managing this functionality. Implementing this functionality could be simplified by the target language's runtime dynamism; for example, in Java – reflection may be used to store the method's name and reflected each time it is used (or alternatively a direct reference to the `java.lang.Method` object – eager evaluation of the reflected method). By storing the Method, its relevant object, and any such parameter list, it can be invoked from elsewhere. These classes enable the creation of a useful OBD system; providing the basic elements from which the meaningful elements are made.

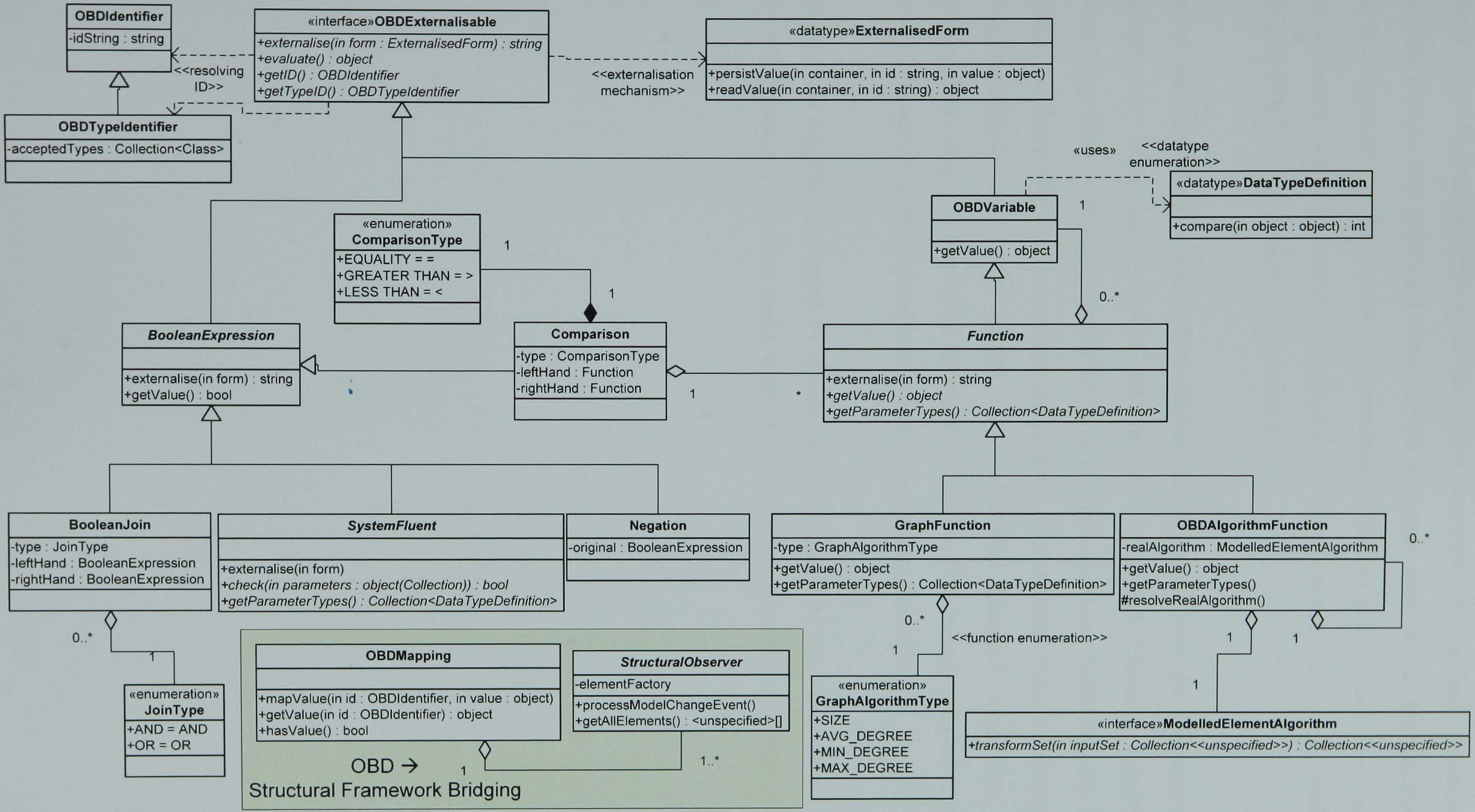
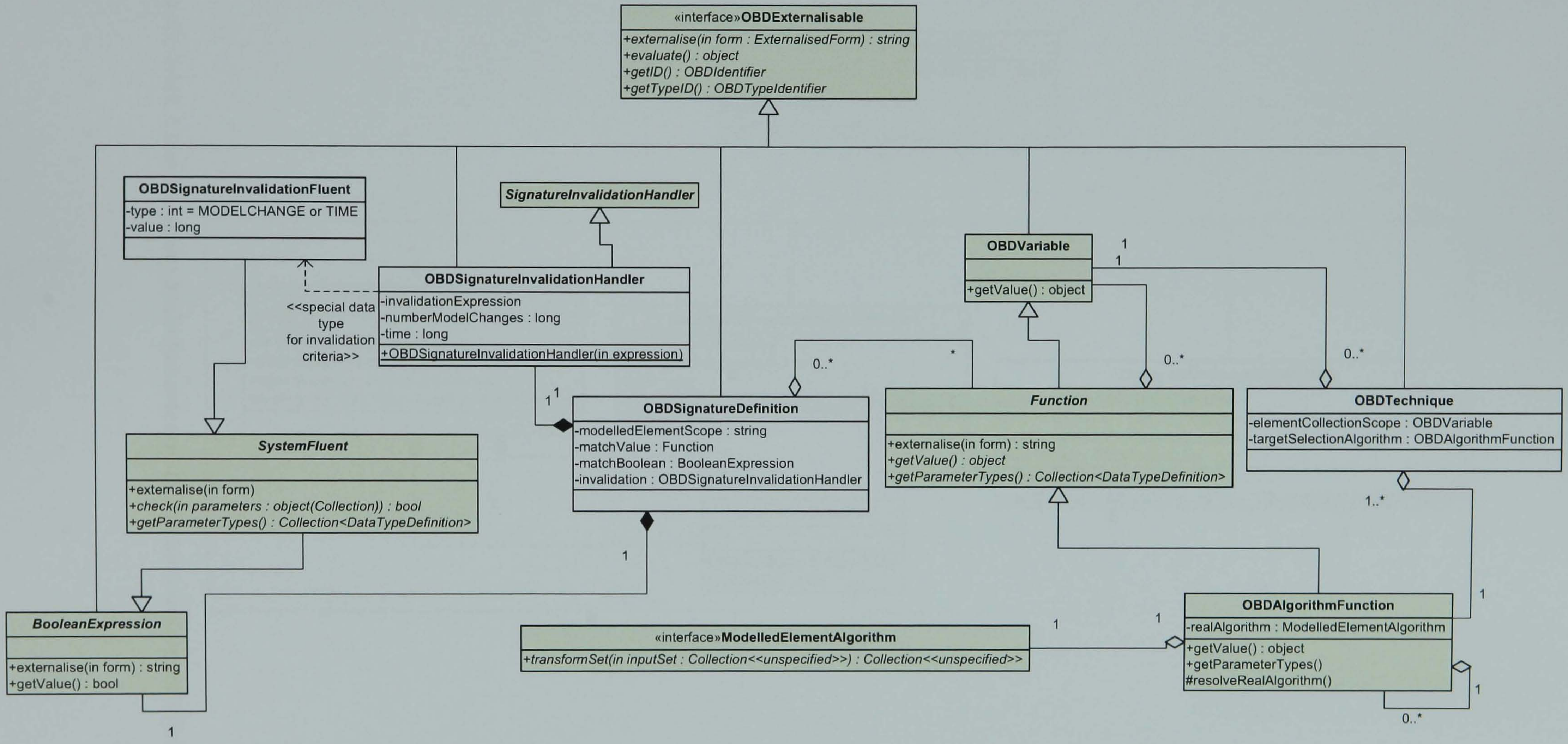

OBDIdentifier
-idString : string
<<resolving ID>>
«interface»OBDExternalisable
+externalise(in form : ExternalisedForm) : string
+evaluate() : object
+getID() : OBDIdentifier
+getTypeID() : OBDTypeIdentifier
<<externalisation mechanism>>
«datatype»ExternalisedForm
+persistValue(in container, in id : string, in value : object)
+readValue(in container, in id : string) : object
OBDTypeIdentifier
-acceptedTypes : Collection<Class>
«uses»
<<datatype enumeration>>
«datatype»DataTypeDefinition
+compare(in object : object) : int
OBDVariable
+getValue() : object
«enumeration» ComparisonType
+EQUALITY = =
+GREATER THAN = >
+LESS THAN = <
BooleanExpression
+externalise(in form) : string
+getValue() : bool
Comparison
-type : ComparisonType
-leftHand : Function
-rightHand : Function
Function
+externalise(in form) : string
+getValue() : object
+getParameterTypes() : Collection<DataTypeDefinition>
BooleanJoin
-type : JoinType
-leftHand : BooleanExpression
-rightHand : BooleanExpression
SystemFluent
+externalise(in form)
+check(in parameters : object(Collection)) : bool
+getParameterTypes() : Collection<DataTypeDefinition>
Negation
-original : BooleanExpression
GraphFunction
-type : GraphAlgorithmType
+getValue() : object
+getParameterTypes() : Collection<DataTypeDefinition>
OBDAlgorithmFunction
-realAlgorithm : ModelledElementAlgorithm
+getValue() : object
+getParameterTypes()
#resolveRealAlgorithm()
«enumeration» JoinType
+AND = AND
+OR = OR
OBDMapping
+mapValue(in id : OBDIdentifier, in value : object)
+getValue(in id : OBDIdentifier) : object
+hasValue() : bool
StructuralObserver
-elementFactory
+processModelChangeEvent()
+getAllElements() : <unspecified>[]
OBD → Structural Framework Bridging
<<function enumeration>>
«enumeration» GraphAlgorithmType
+SIZE
+AVG_DEGREE
+MIN_DEGREE
+MAX_DEGREE
«interface»ModelledElementAlgorithm
+transformSet(in inputSet : Collection<<unspecified>>) : Collection<<unspecified>>
1
1
1
0..*
1
*
0..*
1
0..*
1
1
1
1
1
1..*
0..*

Figure 45: UML Class Diagram: OBD Base and Utility classes

The next class diagram in Figure 46 shows the extensions to the basic functionality and utility classes that comprise some of the elements seen in the specification in Figure 44 – specifically the Signature representation (`OBDSignatureDefinition` plus its invalidation support classes `OBDSignatureInvalidationHandler`), and the Technique OBD representation (`OBDTechnique`).

It shows the relationships with other classes; significantly how the Signature and Technique are both made up of an in-code Modelled Element Algorithm-implementing class – represented by `OBDAlgorithmFunction`. Note that the OBD Signature can have any `Function` type making up its matching value attribute, thus allowing manipulation of the results of a Modelled Element Algorithm. The Technique must supply a Modelled Element collection, thus is only permitted to use Modelled Element Algorithms as these produce the correct data type.

The final class diagram in Figure 47 shows the culmination of these definitions – the Observer Behaviour Definition itself, with its significant relationships to other classes.



«interface»OBDExternalisable
+externalise(in form : ExternalisedForm) : string
+evaluate() : object
+getID() : OBDIdentifier
+getTypeID() : OBDTypeIdentifier
OBDSignatureInvalidationFluent
-type : int = MODELCHANGE or TIME
-value : long
SignatureInvalidationHandler
OBDSignatureInvalidationHandler
-invalidationExpression
-numberModelChanges : long
-time : long
+OBDSignatureInvalidationHandler(in expression)
<<special data type for invalidation criteria>>
OBDVariable
+getValue() : object
1
1
0..*
*
0..*
0..*
1
1
OBDSignatureDefinition
-modelledElementScope : string
-matchValue : Function
-matchBoolean : BooleanExpression
-invalidation : OBDSignatureInvalidationHandler
Function
+externalise(in form) : string
+getValue() : object
+getParameterTypes() : Collection<DataTypeDefinition>
OBDTechnique
-elementCollectionScope : OBDVariable
-targetSelectionAlgorithm : OBDAlgorithmFunction
1..*
SystemFluent
+externalise(in form)
+check(in parameters : object(Collection)) : bool
+getParameterTypes() : Collection<DataTypeDefinition>
1
1
BooleanExpression
+externalise(in form) : string
+getValue() : bool
1
«interface»ModelledElementAlgorithm
+transformSet(in inputSet : Collection<<unspecified>>) : Collection<<unspecified>>
1
1
OBDAlgorithmFunction
-realAlgorithm : ModelledElementAlgorithm
+getValue() : object
+getParameterTypes()
#resolveRealAlgorithm()
1
0..*

Figure 46: UML Class Diagram - OBD Signatures and Techniques and significant relationships

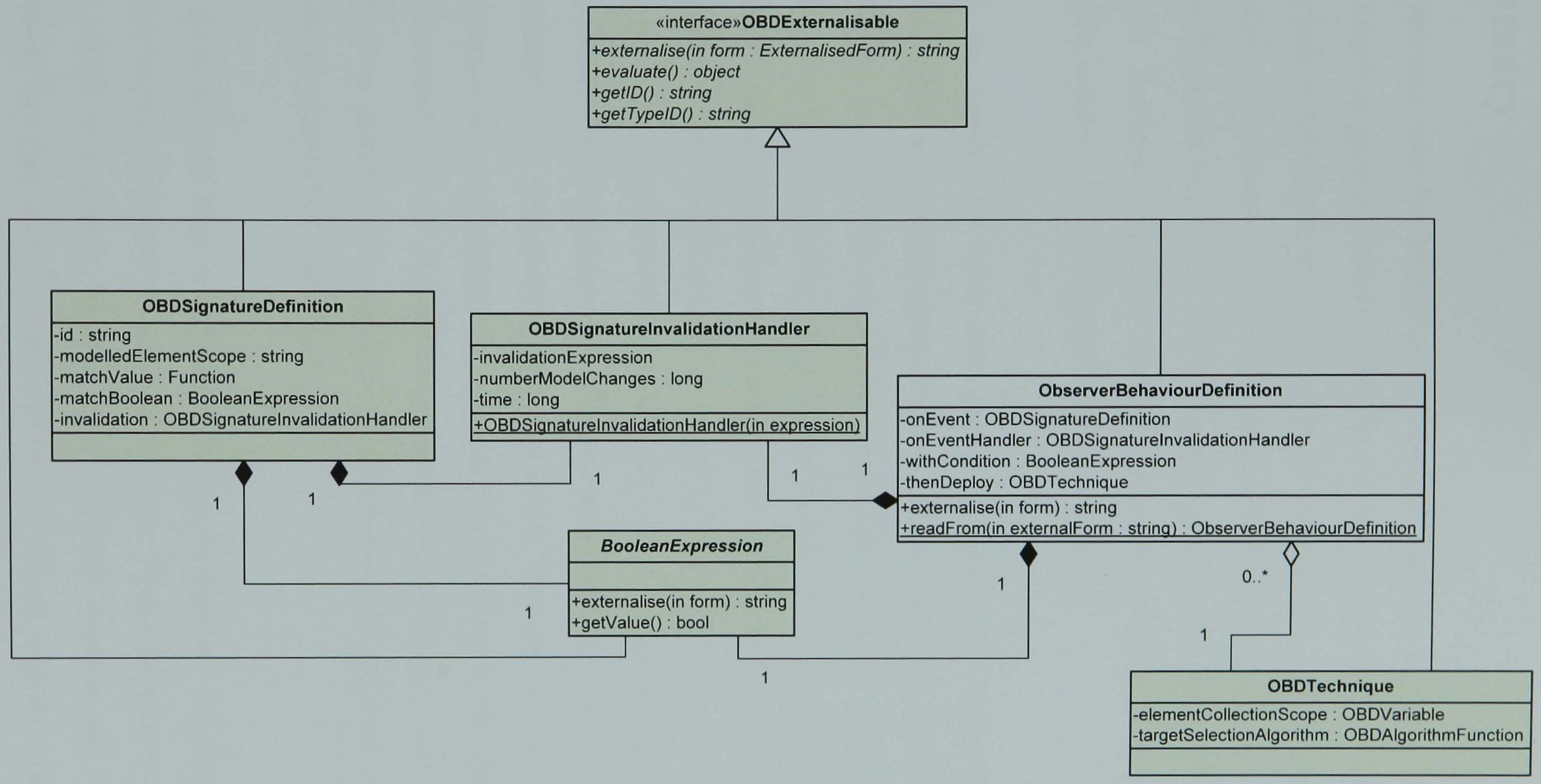

«interface»OBDExternalisable
+externalise(in form : ExternalisedForm) : string
+evaluate() : object
+getID() : string
+getTypeID() : string
OBDSignatureDefinition
-id : string
-modelledElementScope : string
-matchValue : Function
-matchBoolean : BooleanExpression
-invalidation : OBDSignatureInvalidationHandler
OBDSignatureInvalidationHandler
-invalidationExpression
-numberModelChanges : long
-time : long
+OBDSignatureInvalidationHandler(in expression)
ObserverBehaviourDefinition
-onEvent : OBDSignatureDefinition
-onEventHandler : OBDSignatureInvalidationHandler
-withCondition : BooleanExpression
-thenDeploy : OBDTechnique
+externalise(in form) : string
+readFrom(in externalForm : string) : ObserverBehaviourDefinition
BooleanExpression
+externalise(in form) : string
+getValue() : bool
OBDTechnique
-elementCollectionScope : OBDVariable
-targetSelectionAlgorithm : OBDAlgorithmFunction
1
1
1
1
1
1
1
1
0..*
1

Figure 47: UML Class Diagram: Main Behavioural Definition class and relationships

7.5 Summary

Chapter 7 examined how the structural observation framework could be augmented such that it is open to greater runtime flexibility. The system proposed allows runtime specification and modification of key structural observer processes. It gives control over the associations between the Structural Signatures and Techniques, along with a basic level of flexibility in terms of considerations that would previously have been deferred to the Technique Selector and the Associator, along with some of the concerns usually assigned to the Deployment Co-ordinator.

In terms of its interaction with the previously-discussed Structural Observer framework; the OBD externalisation can entirely replace the hardcoded definitions of behaviour (subject to structural algorithms' implementation), or piggyback a basic OBD on core observer functionality implemented in hardcode. For example, structural signatures, techniques and deployment requirements could be specified in hardcode, and the externalised OBD could specify only the associations between signatures and techniques according to an evolving system policy. OBD requires significant control of particular classes; OBD-specific implementations of the Deployment Co-ordinator, Technique Selector and Associator components are required. These stages in the observation process are then subject to evaluation by relevant OBD objects. Finally, suitable implementations of the OBD Mapping specification controller within Structural Observer instances are required, which allows the exposure and control of Observer-determined data and behaviour.

The designs have already been summarised largely in Section 7.4, which aimed to demonstrate the basics of externalisation requirements. This culminated with the OBD specification – both as a generic free-text externalisation schema, and the necessary classes to support that data structure and provide interaction with the classes from the Structural Framework. Next, Chapter 8 will examine the mechanisms by which the OBD can be implemented – both in terms of the externalisation format and mechanism, and how it can be fully integrated into the structural observation framework so that OBD-type specifications could replace, or make up functionality alongside the hard-coded structural observers.

Chapter 8 – OBDXML to Code

This chapter details the method used to externalise the observer definitions and shows how they can be moved out of the code, into an external source of the format shown in the previous chapter. This section also explores how OBD-representative classes can be linked with the external form and Structural Observer code, thus facilitating runtime inspection and modification of the observer system's behaviour.

Firstly, a brief introduction into the actual externalisation format: the required elements of the specification will be stored externally in XML. XML has been chosen for this framework because it:

i. Is defined by its own per-document-type Schema (thus allowing easy lexical and syntax checking), yet is still extensible and flexible,
ii. Allows sufficient parameterisation of required data within a simple type
iii. Is sufficiently "mature" to have been used in many applications and therefore has a wealth of developer support in several languages, and a variety of APIs to allow translation from data objects in code to an XML string/file and back again

8.1 OBD to OBDXML Schema Definition

While it is assumed that the reader will have a basic understanding of the XML format, a brief discussion of XML schemas follows. A variety of XML schema definition formats exist, but OBDXML will be presented in XML Schema (XSD). When trying to create a serialisation of an OO-based object tree, there is the problem of super and sub-classes. They are hierarchical, and if serialising several objects from within the same type hierarchy, the basic DTD format can take two approaches:

i. The first is to represent super-classes by *composition*. Therefore, if type B inherits from type A (and only type A), then the definition of the B element contains an A element. However, while this represents the data efficiently, and conveys an understanding of the represented data, it breaks the OO hierarchy and prevents the correctly-formed use of more specific subtypes in XML.

ii. The second is to duplicate the necessary attributes in each type definition. Following the same example, the B element definition will explicitly present a duplicate declaration of all the attributes that are found in A. While this produces somewhat more intuitive and readable XML, it increases DTD maintenance in the case of changes, and leads to overcomplicated definitions. Additionally, the problem of sub-type substitution still remains.

As sub-typing has been used throughout the OBD specification thus far (e.g. `OBDVariable` implicitly permits Function, which permits the use of `GraphFunction` and `OBDAlgorithmFunction`, and so on), it is appropriate to look towards an XML definition that better facilitates this structural makeup. Therefore, while accepting that it is entirely possible to produce schema designs for OBDXML in DTD, the remainder of this section looks towards XSD as the method of document definition. XSD allows inheritance; restrictive and extensive subtypes of defined element types can be created. For the reader unfamiliar with the XSD format, Figure 48 shows a brief example of a simple inline-XML variant of XSD:

```
<xs:element name="servicecontract">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="serviceID" type="xs:string" use="required"/>
      <xs:element name="providerID" type="xs:string" use="required"/>
      <xs:element name="subsID" type="xs:string" use="required"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 48: XML Schema snippet: Book Example

8.1.1 Re-Examining the OBD Schema

The OBDXML schema follows the form of the summary OBD schema in Figure 44, shown previously in Section 7.4.1. Recapping, there are three main types of element: Signatures, Techniques and Behaviour Definitions.

Multiple Signatures, Techniques, and Behaviour Definitions can exist in a single XML file; thus there is a need to reference a single Signature or Technique from one or more Behaviour definitions. Each element type is made up of a set of attributes, some of which are reasonably complex themselves (i.e. greater than a single attribute). In order to further simplify the schema creation, the following bulleted-lists will hierarchically enumerate the element types, their attributes and their attribute types.

Almost everything that exists in the OBDXML is an implementation of **`OBDExternalisable`**. Its XML representation requires the following information:

- Unique ID (OBDIdentifier)

In the UML class diagram there is also a requirement for a Type ID. This will not appear as an attribute or sub-element in XML. It is the responsibility of the externalisation mechanism to ensure this is populated appropriately based on the XML element type (marshalling) or the runtime object-type (un-marshalling). The following element types are the main OBD elements populating the file:

`OBDSignature` (made up of the following attributes)

- Matching Value – (`Function`)
- Matching Boolean – (`BooleanExpression`)
- Invalidation Handler – (`OBDInvalidationHandler`)

`OBDTechnique` (made up of the following attributes)

- Target Selection Algorithm (`OBDAlgorithmFunction`)

`ObserverBehaviourDefinition`, comprised of the following (note: although the *invalidation handler* appears in the class diagram, this is obtained from the XML-specified *signature*, and not directly from the XML)

- On Event Match (`OBDSignature`)
- With Condition (`BooleanExpression`)
- Then Deploy (`OBDTechnique`)

All types listed in brackets are not easily represented in XML as a simple attribute, with the one exception of the `OBDIdentifier` – which although a class, has just a single attribute and is effectively a string identifier. However, most of the other types are actually just references to either other elements in the external document, or are mapped via the `OBDMapping` object as part of a fluent, function or variable. In the case of constant and variable data, it is sufficient to map (mutable) data-carrying types that can be modified and accessed from both OBD and Structural Observer sides of the OBD Bridge. As mentioned previously, in the case of function and predicate or fluent mappings, it is necessary to simulate higher-order functions by mapping a suitably wrapped `OBDHigherOrderFunction` implementing object containing the desired evaluation method call.

These types are examined hierarchically, following the classes in the UML OBD Utility class diagram (Figure 45), specifying attributes that are required in the XML representation. This is not an exact replica of the information in the UML diagram, as some types are effectively mapped to their code representations; thus from an externalisation point of view, require only an ID and list of parameters.

The utility `OBDExternalisable` types in can be further split into two subtypes:
`OBDVariable` – A variable is any element that has a value usable by a set of other elements. It is comprised of all the information in `OBDExternalisable`, and this subtype supports variable mapping and management. From an externalisation perspective, it is a role marker; i.e. a new subtype with no additional attributes.
`BooleanExpression` – A Boolean Expression is any element having a *Boolean* value usable by other elements. Again, it is comprised of the data from `OBDExternalisable`, and the sub-typing supports Boolean functionality, and provides role indication.

The `OBDVariable` type can then be further split into:
`Function` – A function is a predefined code element that is exposed to the OBD. It is distinct from an `OBDVariable` as it is able to take zero-to-many parameters. At a code level, it returns a value of a type specified by its own Data Type Definition. To reference in XML, it extends `OBDVariable` with:

- Parameter List (`OBDVariable`) (0-* elements)

```
<simpleType name="OBDID">
 <restriction base="string"></restriction>
</simpleType>

<complexType name="OBDExternalisable">
 <attribute name="ID" type="tns:OBDID" use="required"></attribute>
</complexType>

<complexType name="OBDVariable">
 <complexContent>
  <extension base="tns:OBDExternalisable"></extension>
 </complexContent>
</complexType>

<complexType name="Function">
 <complexContent>
  <extension base="tns:OBDVariable">
   <sequence>
    <element name="parameters" type="tns:OBDVariable" minOccurs="0"
    maxOccurs="unbounded"></element>
   </sequence>
  </extension>
 </complexContent>
</complexType>

<complexType name="BooleanExpression">
 <complexContent>
  <extension base="tns:OBDExternalisable"></extension>
 </complexContent>
</complexType>
```

Figure 49: XML Schema snippet: OBDExternalisable, OBDVariable and their extensions

The `Function` type itself can then be further split into the following subtypes:

`GraphFunction` – provides OBD access to a library of algorithms for manipulating graphs. Other functions could be added and whole new libraries, if signatures are extended beyond structural characteristics. Each algorithm requires one parameter; a collection of Modelled Elements. In XML, it extends `Function` with:

- Algorithm Type (SIZE, AVG_DEGREE, MIN_DEGREE, MAX_DEGREE)

`OBDAlgorithmFunction` – provides OBD access to a `ModelledElementAlgorithm`, allowing functional composition of OBD Signatures and Techniques with the same building blocks as hardcoded versions. In addition to the `Function` elements, it needs the following information:

- Modelled Element Algorithm (OBDIdentifier – as algorithms must be mapped from XML to code)

```
<complexType name="GraphFunction">
 <complexContent>
  <extension base="tns:Function">
   <sequence>
    <element name="AlgorithmType">
     <simpleType>
      <restriction base="string">
       <enumeration value="SIZE"></enumeration>
       <enumeration value="AVG_DEGREE"></enumeration>
       <enumeration value="MAX_DEGREE"></enumeration>
       <enumeration value="MIN_DEGREE"></enumeration>
      </restriction>
     </simpleType>
    </element>
   </sequence>
  </extension>
 </complexContent>
</complexType>

<complexType name="OBDAlgorithmFunction">
 <complexContent>
  <extension base="tns:Function">
   <sequence>
    <element name="ModelledElementAlgorithm" type="tns:OBDID"
    minOccurs="1" maxOccurs="1"></element>
   </sequence>
  </extension>
 </complexContent>
</complexType>
```

Figure 50: XSD Snippet: Function derivatives

The `BooleanExpression` type can also be further split into the following subtypes:

`BooleanJoin` – Joins two `BooleanExpressions`, and is comprised of:

- Left Hand Side (`BooleanExpression`)
- Right Hand Side (`BooleanExpression`)
- Join Type (can be "AND" or "OR")

`Negation` – Negates/inverts a given `BooleanExpression`; thus:

- Original (`BooleanExpression`)

`SystemFluent` – Provides access to a system fluent or predicate, and can be thought of as effectively a Boolean-only, potentially situation-bound version of the `Function`. It requires the following, size depending on the implementation:

- Parameter List (`OBDVariable`) (0-* elements)

```
<complexType name="BooleanJoin">
 <complexContent>
  <extension base="tns:BooleanExpression">
   <sequence>
    <element name="leftHandSide"
     type="tns:BooleanExpression" minOccurs="1" maxOccurs="1">
    </element>
    <element name="rightHandSide"
     type="tns:BooleanExpression" minOccurs="1" maxOccurs="1">
    </element>
    <element name="type">
     <simpleType>
      <restriction base="string">
       <enumeration value="AND"></enumeration>
       <enumeration value="OR"></enumeration>
      </restriction>
     </simpleType>
    </element>
   </sequence>
  </extension>
 </complexContent>
</complexType>

<complexType name="Negation">
 <complexContent>
  <extension base="tns:BooleanExpression">
   <sequence>
    <element name="original" type="tns:BooleanExpression" minOccurs="1"
maxOccurs="1"></element>
   </sequence>
  </extension>
 </complexContent>
</complexType>

<complexType name="SystemFluent">
 <complexContent>
  <extension base="tns:BooleanExpression">
   <sequence>
    <element name="parameters" type="tns:OBDVariable" minOccurs="0"
maxOccurs="unbounded"></element>
   </sequence>
  </extension>
 </complexContent>
</complexType>
```

Figure 51: XSD Snippet: BooleanExpression sub-types

Revisiting the original list of significant OBD elements (Signature, Technique and Observer Behaviour Definition), there are just two ancillary elements that need their XML requirements defining, and it is logical to discuss them together. They are both connected to the Signature Invalidation Handling:

`OBDSignatureInvalidationHandler` – This is the signature handler associated with a signature. It is a special subtype of `SignatureInvalidationHandler`, which exists to provide simple invalidation criteria for the associated Signature. It inherits functionality from `OBDExternalisable`, requiring the following extra functionality:

- Invalidation Criteria – The criteria for invalidation is a `BooleanExpression`. It is given special access to invalidation handler specific data via the next fluent (see next item)

`OBDSignatureInvalidationFluent` – This is a special extension of `SystemFluent` that, in order to provide access to important data can be associated with a `OBDSignatureInvalidationHandler`. It allows the invalidation `BooleanExpression` to contain references to the desired level of model change or elapsed time before a signature should be re-evaluated. As such, it requires the following functionality on top of `SystemFluent`:

- Associated Signature Handler (`OBDIdentifier` or XML reference to the `OBDSignatureInvalidationHandler`)
- Type of condition (MODEL_CHANGE_EVENTS or TIME_MS)
- Value for condition (long integer)

```
<complexType name="OBDSignatureInvalidationHandler">
 <complexContent>
  <extension base="tns:OBDExternalisable">
   <sequence>
    <element name="InvalidationCriteria" type="tns:BooleanExpression"
    minOccurs="1" maxOccurs="1"></element>
   </sequence>
  </extension>
 </complexContent>
</complexType>

<complexType name="OBDSignatureInvalidationFluent">
 <complexContent>
  <extension base="tns:SystemFluent">
   <sequence>
    <element name="invalidationHandler"
     type="tns:OBDID">
    </element>
    <element name="conditionType">
     <simpleType>
      <restriction base="string">
       <enumeration value="MODELCHANGEEVENT"></enumeration>
       <enumeration value="TIME_MS"></enumeration>
      </restriction>
     </simpleType>
    </element>
    <element name="conditionValue">
     <simpleType>
      <restriction base="int">
       <minExclusive value="0"></minExclusive>
      </restriction>
     </simpleType>
    </element>
   </sequence>
  </extension>
 </complexContent>
</complexType>
```

Figure 52: XSD snippet for SignatureInvalidationHandler-related elements

8.1.2 The Finalised Compact Schema

Therefore, revisiting the main elements first discussed, it is a matter of formality to produce the XML Schema now the relevant sub-elements have been defined. As a recap, the final schema must contain correctly-specified XML elements for:

```
OBDSignature
OBDTechnique
ObserverBehaviourDefinition
```

Additionally, a structure that permits multiple instances of each element type is required, along with the facility to reference a single Signature or Technique in many Behaviour Definitions. The following snippet, Figure 53, shows the OBD Signature and Technique definitions in isolation:

```
<complexType name="OBDSignature">
 <complexContent>
  <extension base="tns:OBDExternalisable">
   <sequence>
    <element maxOccurs="1" minOccurs="1" name="matchingValue"
     type="tns:Function" />
    <element maxOccurs="1" minOccurs="1" name="matchingBool"
     type="tns:BooleanExpression" />
    <element maxOccurs="1" minOccurs="1"name="invalidationHandler"
     type="tns:OBDSignatureInvalidationHandler" />
   </sequence>
  </extension>
 </complexContent>
</complexType>

<complexType name="OBDTechnique">
 <complexContent>
  <extension base="tns:OBDExternalisable">
   <sequence>
    <element maxOccurs="1" minOccurs="1"
name="targetSelectionAlgorithm"
     type="tns:OBDAlgorithmFunction" />
   </sequence>
  </extension>
 </complexContent>
</complexType>
```

Figure 53: XSD Snippet: Signature and Technique definitions

In order to complete the XML-based OBD, in addition to declaring the whole Behaviour Definition, a suitable element type must be defined to represent Signatures, Techniques and single Definitions, along with permitting the cross-referencing detailed above.

Firstly, this involves the definition of specific signature and technique reference element types – `OBDSignatureRef` and `OBDTechniqueRef` – specialised elements holding only the `OBDIdentifier` and each still a sub-type of `OBDExternalisable`. The new `BehaviourDefinition` element type uses these new types as references to the actual Signatures and Techniques.

The final new element type, `OBDType`, defines a structure that holds a number of `Signature`, `Technique` and `BehaviourDefinition` elements. The XML schema is completed by the definition of an element "instantiation", of type `OBDType`. This has a number of XSD `key/keyref` constraints, indicating that the reference types described earlier actually reference valid OBD Identifiers existing in the document.

The following schema (Figure 54) is presented surrounded in an XSD schema tag, but without all of the previously shown XSD snippets for conciseness only; in the real schema file, all the XSD definitions thus far would reside in the same schema in order to make it complete and correct.

The rest of this chapter, following the referenced Figure, examines how the XML is marshalled and un-marshalled between the OBD-specific code and XML, then discusses the OBD-specific processing that is required to enable the OBD classes to interact with the structural framework.

```
<schema>
 <complexType name="OBDSignatureRef">
  <complexContent>
   <extension base="tns:OBDExternalisable" />
  </complexContent>
 </complexType>
 <complexType name="OBDTechniqueRef">
  <complexContent>
   <extension base="tns:OBDExternalisable" />
  </complexContent>
 </complexType>

 <complexType name="BehaviourDefinition">
  <complexContent>
   <extension base="tns:OBDExternalisable">
    <sequence>
     <element maxOccurs="1" minOccurs="1" name="onEventMatch"
      type="tns:OBDSignatureRef" />
     <element maxOccurs="1" minOccurs="1" name="withCondition"
      type="tns:BooleanExpression" />
     <element maxOccurs="unbounded" minOccurs="1" name="thenDeploy"
      type="tns:OBDTechniqueRef" />
     <element ref="tns:OBD" />
    </sequence>
   </extension>
  </complexContent>
 </complexType>

 <complexType name="OBDType">
  <sequence>
   <element maxOccurs="unbounded" minOccurs="1" name="signature"
    type="tns:OBDSignature" />
   <element maxOccurs="unbounded" minOccurs="1" name="technique"
    type="tns:OBDTechnique" />
   <element maxOccurs="unbounded" minOccurs="1"
    name="behaviourDefinition" type="tns:BehaviourDefinition"/>
  </sequence>
 </complexType>

 <element name="OBD" type="tns:OBDType">
  <key name="sigKey">
   <selector xpath="./signature" />
   <field xpath="@ID" />
  </key>
  <key name="techKey">
   <selector xpath="./technique" />
   <field xpath="@ID" />
  </key>
  <keyref name="sigKeyRef" refer="tns:sigKey">
   <selector xpath="./behaviourDefinition/onEventMatch" />
   <field xpath="@ID" />
  </keyref>
  <keyref name="techKeyRef" refer="tns:techKey">
   <selector xpath="./behaviourDefinition/thenDeploy" />
   <field xpath="@ID" />
  </keyref>
 </element>
</schema>
```

Figure 54: XSD snippet: References, Behaviour and OBD Definition

8.2 The Binding Processes

The purpose of XML binding is to associate one or more XML documents with a series of in-code objects that adequately reflect the data in the XML. Usually this process is referred to as marshalling when translating from in-code objects (the content) to an XML string, and "un-marshalling" when translating an XML string into a representative code object.

This section does not provide a detailed breakdown of XML-code interface techniques, nor is it intended to give a review of the state-of-the-art XML serialisation libraries available at the time. However, it does discuss the basic processes involved in XML binding and gives a brief overview of the chosen XML serialisation library, along with justification of this choice.

There are different models for managing the translation from XML string to code content and vice versa, but the following diagram (Figure 55) aims to show the main stages of several of the common library approaches. Some of the significant stages are described briefly after the diagram. The processes included are split into the pre-deployment actions, and those that happen during un-marshalling (XML→object) iterations. Although the diagram and following text deal primarily with the un-marshalling operation, detail is added to the explanatory notes to provide a basic understanding of some of the considerations involved in the marshalling (i.e. object→XML) operation too.

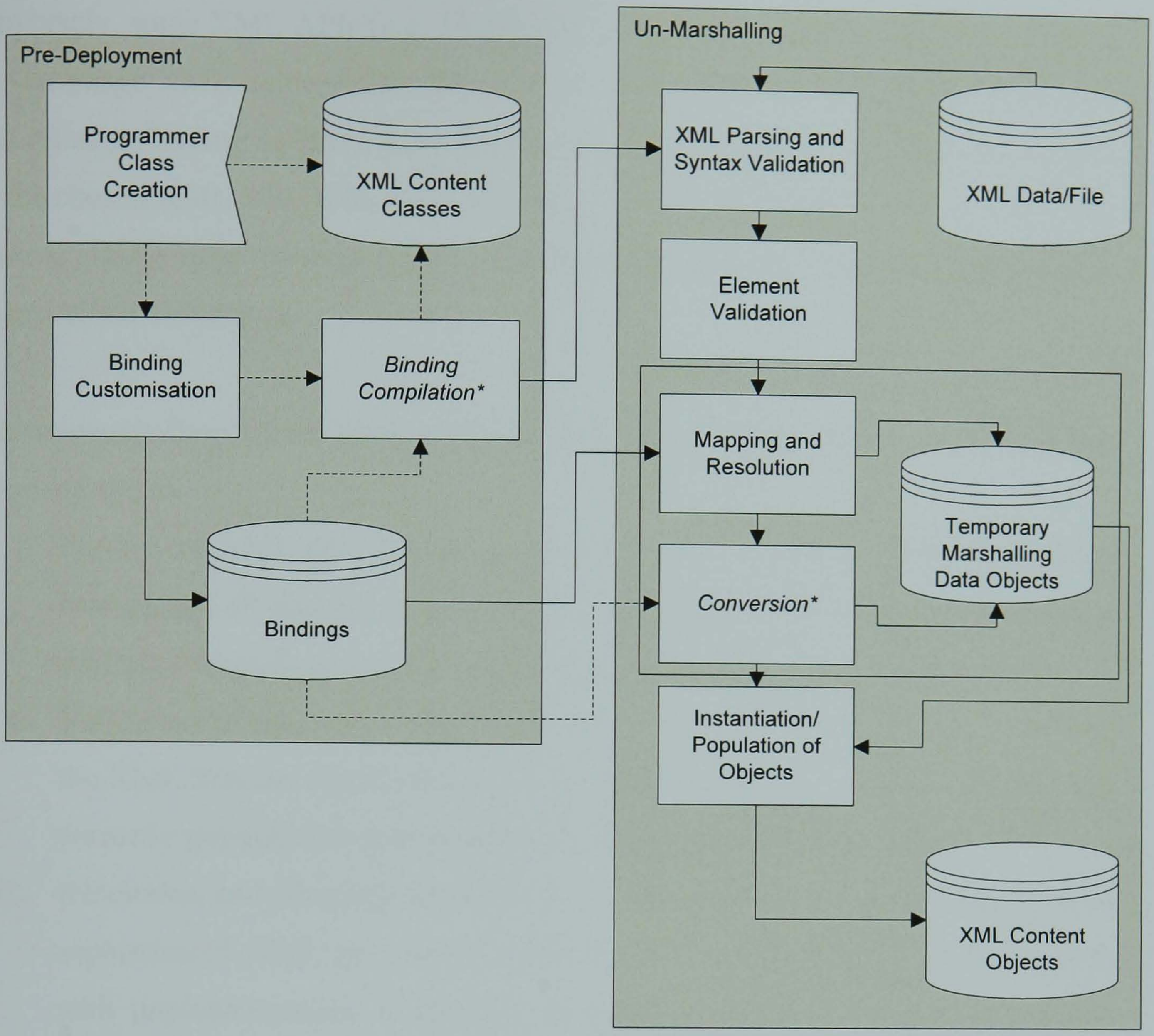


Figure 55: Generic view of Un-Marshalling XML
(*Note:* brief discussion regarding *marshalling* XML and its peculiarities also follow in this text)

Firstly, the pre-deployment points:

i. Binding "Customisation" – The programmer maps XML element definitions to classes in the host language. Typically, most frameworks will do this for language-included types, such as strings and number types, but this stage allows any specific element types to be given a matching allocation in code.

ii. Binding Compilation (if required) – Some XML APIs allow the programmer to map every XML type to a corresponding host language class. This can obviate the need for a complete XML schema in place beforehand and is useful for prototype development and where the makeup is expected to regularly change. However, it does place onus on the programmer to write matching XML content classes, *and* to ensure that the class definitions match the element definitions; a mismatch will have no alternative but to generate a runtime exception.

Conversely, some XML APIs (e.g. JAXB [135]) provide a compiler that will generate host-language XML content class files, based on the definition in the XML Schema. This relies on having an XML Schema correctly written and in an appropriate format for the chosen XML API. Whichever approach is taken, the result is that a set of XML Content classes must be written and compiled that match the XML and Schema that are processed at runtime.

The un-marshalling of an XML file/stream/string typically includes several of the following stages:

i. Parsing the XML file (& Validation of XML) – Checks the XML is correctly-formed and all tags are populated correctly. If an error is encountered here, a runtime exception is usually generated, indicating the XML is malformed.
ii. Validation of elements – Validation that all elements match the types defined the XML Schema (and/or the manually mapped classes), and that elements are correctly populated with the expected data types in the XML string.
iii. Resolution and Mapping of elements [5] – This feature may be omitted in less sophisticated XML persistence methods, and its precise approach can vary with implementations. It involves the process of determining where several elements in XML should represent just a single object in the host-language. It may also extend to allowing particular XML elements access to existing in-code objects, by use of unique identifier as in OBD XML. The same concern applies in the reverse of this process – marshalling. When one object is referenced several times by different objects that will be marshalled, this referencing must be satisfactorily and efficiently replicated in XML.
iv. Conversion (where applicable) – if a particular binding has been set up that has specified a particular conversion routine, this can be set up here. For example, improperly specified, or old (e.g. DTD) style schemas may declare every attribute and element as having string characteristics, and some fields need parsing and converting to numeric data types.

[5] *Note:* Readers requiring further information relating to element resolution may wish to look at an approach which tackles the issue of object references and inter-element references within XML. The W3C's XPath language allows specification of a location within an XML document either absolutely or relative from a particular point, such as that used to locate key/keyref elements in the OBD XML Schema [136]James Clark and Steve DeRose, "XML Path Language (XPath) Version 1.0," in *W3C Recommendations*, vol. 2009: W3C, 1999.

v. Instantiation and Population of code-side objects – the appropriate content objects are instantiated from the content classes and populated with the data in the XML, according to the bindings and conversions that have been set up, and resolutions if found.

The prototyping for this research was implemented in two XML binding mechanisms, following the pattern outlined above. The first mechanism is XStream [137], used throughout the project. XStream is an open-source, Java-based XML serialisation library available under a BSD licence. It has the following significant advantages:

- Simplicity of binding – XStream doesn't describe itself as a binding library; rather a serialisation tool. As such, the level of specification is fairly minimal; it need only map the elements' type names to appropriate classes. Given that the design thus far has already produced suitable classes, this saved effort in terms of Schema definition (and redefinition) during prototyping. Equally, it avoids any in-schema duplication of inherited attributes as discussed in the previous section's comments on DTD; the attributes only appear if they require serialisation. The downside is that as the Schema is an optional part of the whole process, the resulting XML is not always a valid, well described and correctly-formed XML Document; although it is easy to read.
- Tidy handling of inter-element referencing/self-referencing – XStream is configured by default to use the W3C-XPath Relative Location Path handling, which made the XML relatively easy to understand for debugging purposes and dealt with the issue of multiple references neatly and simply.

In comparison, built-in Java support for XML at the project start was inflexible and clumsy to use. Java included support for several XML-access methodologies; however the discussion concentrates on the binding methodology – Java Architecture for XML Binding, otherwise known as JAXB. The first version of JAXB requires a schema definition, compiling a set of special XML content classes that represent XML document data. However, these classes contained calls to special reflective constructors, and forced inheritance with proprietary JAXB classes.

However, JAXB version 2, released with JDK 6 [138] was a significant improvement. While it still requires greater specification than XStream for binding, it made use of other new-to-JDK6 features; supporting inline XML-serialisation annotations of code, which greatly simplifies the binding operation, allows already-written data classes to be made XML-compatible easily, and even generates suitable XML Schema for a particular set of XML-annotated classes; an approach more suited to this design scenario than previous versions of JAXB.

The next section examines what happens once the XML has been un-marshalled, and discuss behavioural aspects of the classes outlined in the previous sections – specifically in Figure 37 and Figure 39.

8.3 Interpreting and Processing

This section discusses the outstanding concerns regarding the OBD subsystem's method of acquiring, interpreting and processing its specification in conjunction with the Structural Observer framework.

8.3.1 Acquiring the OBDXML String

The designs thus far have assumed that an XML string is available for processing; overlooking the method by which new or updated XML would be introduced into the system. There are several potential options available to manage the specification XML string. Firstly, there is the prospect of re-reading the XML string from its source whenever OBD objects are evaluated. However, this has the potential to become computationally inefficient, particularly as the XML string could contain the entire system specification; re-reading, re-parsing and re-processing it each time a set of elements need re-evaluating is wasteful.

Therefore, once created, the OBD objects should be created from the XML, kept in cache, and evaluated when they are required. The OBD object pool should only revalidate itself as and when the source XML string has changed. XML change notification could be managed by a variety of methods dependent on how the string is stored or provided. Some examples follow:

- If the XML string is contained in a locally-sourced file, then the externalisation mechanism has to simulate its own change notification. A simple implementation involves a threaded file listener, sleeping for a regular period x, then polling the file; if the last-modified time/date has changed (e.g. in Java, `java.io.File#lastModified()` provides this data), it must reload and process it.
- Equally, if the XML string is located at a remote-source location, the externalisation must again simulate change notification. However, in this instance, last-modified file information may not be available. Therefore, the change component must devise another method, such as string comparison, to determine when to generate change events.
- Alternatively, the XML string may be provided by users/administrators or other services, by way of uploading an XML-containing file to a specially-written server. In this case, once the stream has been read; the server listening thread could notify the externalisation of an XML change event.

8.3.2 Un-marshalling XML strings to OBD objects

In addition to the serialisation and de-serialisation processes outlined in the previous section, the un-marshalling of XML to OBD type objects must consider the following additional OBD-specific requirements:

- Allocation of appropriate `TypeID` to each OBD object. In implementation terms, the XML-based element type is used to select the destination code class, and the constructor for each class must ensure it is populated with the correct `TypeID`, in order to support externalisation forms.
- Resolution of mapped `OBDVariable` derivative types; `OBDVariable`, `SystemFluent`, `OBDSignatureInvalidationFluent`, and `OBDAlgorithm` types. Each association must describe which object or collection it should acquire the data from, and how it should acquire it (e.g. method name).

8.3.3 Piggybacking OBD on the Structural Framework

As discussed earlier in this chapter, the OBD subsystem provides an externalisation approach to some of the Structural Framework's observation processes. However, while units and classes have suitable functionality to provide for these processes; the intended method of integration into the structural framework has not been specified.

Therefore, this subsection will clarify the interfaces between the Structural Framework and the OBD subsystem's classes, and specify how they should interact in order to facilitate the operation of a Structural Framework with the appropriately determined degree of externalisation.

Firstly, **Signatures, Techniques and the required OBD bridging**:-
`OBDSignature` and `OBDTechnique` are OBD-specialisations of the `Signature` and `ObservationTechnique` classes defined by the Structural Framework. Therefore, dealing with these classes in isolation, the modifications required to the framework are such that the Structural Observer is registered as a Signature Change Listener on the OBD Signatures. Additionally, the Observer's `SignatureTechniqueAssociator` must be programmed to obtain its list of Techniques and Signatures from the OBD (in addition to its hardcoded ones). As such, the OBD external definition would be contained to new Signatures and Techniques.

Several OBD support classes provide functionality for the OBD Signature and Technique instances – this includes OBD's Signature Invalidation Handler, which is again an OBD specialisation of `SignatureInvalidationHandler`, specifying its associated Signature's invalidation criteria in terms of special OBD data. All of the tests in `Signature`, `SignatureInvalidationHandler`, and `Technique` make use of the `OBDVariable` (and its `Function` derivatives), and `BooleanExpression`. `BooleanExpression` objects evaluate their contained statements appropriately to produce a Boolean value from the joins, negations and contained comparisons of `OBDVariable` and derivative objects, mapped as described previously.

Completing the OBD to Framework Bridge

However, even with this functionality, there are still missing links; the Technique Associator does not receive instruction on how to associate the OBD techniques to any particular Signature(s). As such, the new OBD Signatures and Techniques, while included in the system's available items, will not actually do anything. In the

Structural Framework, the decisions leading from a Signature match to a Technique's deployment are made by firstly the `TechniqueSelector`, then the `DeploymentCoordinator`. *Technique Selector* selects a suitable Technique from those that are associated, then *Deployment Co-ordinator* tunes the output from the Technique to make it suitable for the current system state and current system policy (e.g. resource constraints).

With OBD-associated techniques, much, though not all of this process is managed by the appropriate `ObserverBehaviourDefinition`. With OBD, the selected Technique is chosen based on whether the `BooleanExpression`-typed `withCondition` attribute evaluates to true, replacing `TechniqueSelector`'s functionality. However, `DeploymentCoordinator` must manage the deployment of the actual observers, as it is ultimately responsible for co-ordinating which units need deploying and which units should be un-deployed. Therefore, in order to support OBD, the `TechniqueSelector` must be able to check with the OBD Coordinator to determine if any `BehaviourDefinition` objects exist that specify the matching signature. If so, they must be evaluated, and if their `withCondition` evaluates to true, then the associated Technique should be added to the `TechniqueSelector`'s recommended Technique set.

Therefore, a revised OBD-compatible `TechniqueSelector` is a requirement, along with consideration to an OBD-compatible `DeploymentCoordinator`. The `DeploymentCoordinator` may prioritise information contained in an OBD, or even override certain definitions with hardcoded policy if necessary.

8.3.4 Runtime (Typing) Errors

As the OBD system introduces both an external input format and a form of dynamic typing, the potential for runtime errors is greater than a standard POJO-based piece of software. Constraints on the externalisation format (e.g. XML key/keyrefs, requirements on the types of data) goes some way to mitigating these problems, but they are to some extent unavoidable – for example, in XML, the Function Element allows any number of parameters and the type checking necessarily occurs when the appropriate OBD Function is called. Admittedly, an externalisation format with greater complexity and customisability (such as a bespoke definition language) could determine the expected types and encode this in the definition. Equally, the use of variant types while allowing for flexible definitions; increases the possibility that a type will not convert at runtime, leading to another *deferred error* situation.

Therefore, the OBD and its externalisation mechanism must be capable of dealing with two main types of errors and determining the appropriate response:

Externalisation validation error – when the provided specification is determined invalid before attempting execution. With XML, this may be an error that arises when the document is parsed and validated against its schema. In this case, it is possible that the external source (a user or other component) can be informed immediately with a detailed error. The malformed part of the document must be discarded, but the system must have an error-handling policy that determines whether to:

- Discard the whole document (i.e. ignore it, do not change current OBD)
- Discard the malformed part of the document, and load the rest, overwriting the current OBD
- Discard the malformed part of the document, and merge it with the current OBD

OBD execution error – this type of error can occur even when the specification has loaded, parsed and validated according to its schema correctly. It may include an incorrect number of parameters for a function, a parameter type that cannot be converted into the appropriate type, or a referenced element that does not exist. In this case, the domain-specific error handling policy has different decisions to make. It must firstly determine how and where to log the error with sufficient detail, and then has the following options:

- Ignore the failed element execution and try to continue if possible, stopping otherwise.
- Ignore the failed element execution, and try to find a near alternative.
- Remove the failed element from wherever it occurs in the OBD, and continue without its presence in the specification.
- Produce a default value for this type of functional error and continue.

System error handling policy can be produced to deal with these errors in a satisfactory way, and in critical systems or systems where it is not envisaged that the OBD will alter regularly, it would be possible to make the OBD perform a *complete validation* on its new specification when one is provided. While this would not eliminate the errors, it would at least report them in a timely fashion, increasing the possibility of suitable alternative action.

8.4 Summary

This chapter provided an overview of the OBD externalisation mechanism, along with a discussion of some externalisation-related concerns. The design can be split into the OBD data with custom implementations, and the Externalisation Mechanism itself. Although there is a small degree of overlap, they have separate responsibilities.

The OBD provides a method of external control to the Structural Observer framework; the functional and Boolean OBD classes effectively form a basic *OBD language specification* which allows predicate-based and functional comparisons to create a basic Boolean-driven policy. The Externalisation part of the OBD takes control of the mechanism by which OBD data is serialised and de-serialised, along with controlling the method of updating, resolution and error handling.

It is worth mentioning in the summary that while the externalisation has been designed with the aim of allowing flexibility and policy change during execution, it is not a substitution for dynamic programming. While the OBD may exhibit certain dynamic traits, such as flexible specification of policy, and aspects of dynamic data typing; the OBD still relies on the implementation of underlying algorithms, fluents and functional tasks and exposure via mapping. This feature simplifies OBD policy development, and reduces the amount of runtime type errors that can be raised, whilst still allowing reasonable flexibility.

The next chapter evaluates various aspects of both the structural observation framework along with the OBD externalisation.

Chapter 9 – Evaluation & Case Study

The thesis considered an architectural overview of the framework in Chapter 4, and specified the software along with detailed component design in Chapter 5 and Chapter 6. Finally, the framework OBD and its externalisation bridge are detailed in Chapter 7 and Chapter 8. This chapter provides evaluation details and results of key features of the framework using both qualitative and quantitative measures.

A case study and experiment are used to assess both the generality and applicability of the framework from an SE perspective, and the potential performance of components under conditions that simulate their likely application.

9.1 Testing & Evaluation Methods

The evaluation method is used to assess the validity of the basic approach; that is, the *type signature* based observation mechanism and associated framework. In addition, the evaluation demonstrates the applicability of the method to systems that do not wholly conform to the expected problem domain. For instance, how to provide developer support for bespoke observation overlays creation for systems that do not necessarily require all the functional components specified by the framework designs.

Therefore, the chosen experiment is used to assess how well the approach is suited to the observation of large-scale dynamic systems by means of monitoring for specified topological events; in addition, how well the software engineering design guide can be adopted as needed to provide a typed observation overlay. The following subsection will describe the conditions under which experiments were conducted.

9.1.1 Evaluation Conditions and Specifications

The framework developed through this thesis is not tied to a particular programming language or methodology. However, it is intended for use as part of an OO design model and implementation platform, and makes use of design concepts, particularly sub-typing and polymorphism.

The evaluation used for the experimental simulated component-systems environment was conducted using an agent-based network simulation testbed. This was developed for this research by extending the open source Repast agent-based discrete event simulation framework [139]. Repast was selected primarily due to its open-source availability and inclusion of basic visualisation tools.

Repast experiments and software evaluations were conducted on a small sample of JREs, with the following specification being a fair representation:

- Intel P4 3.0Ghz or higher
- 1Gb+ RAM
- Windows Platform (XP SP3 or Vista)
- Sun's JDK 1.6.0 for Windows
- Repast v J 3.0 [6],

(Experimental code developed and debugged in the Eclipse IDE, v 3.3 and greater)

The case study following the simulation is based on a piece of software developed by the author known as the Email Exploration Tool (EET), full details of which will be included alongside the evaluation in Section 9.3. EET is also written in Java, yet differs from the experimental work as it operates on real data, rather than a simulated component set.

9.2 Quantitative Evaluation: Identification and Deployment

As discussed throughout this thesis, the framework is built on several architectural components. These components allow a traditional type observer pattern to observe a large-scale, dynamic computer system. The complexity and scale of the system should be hidden from the business-logic observers (the system-level) observers, by way of a structural overlay that reacts to system change and deploys the system-level observers in an efficient placement, and in accordance with system-wide goals.

[6] This project deadlines and requirements did not allow or necessitate for the upgrade of the simulation to use the new version of Repast ("Sim-phony") as it was only released towards the end of 2008.

Concentrating particularly on the structural overlay and efficiency of placement, this is highly reliant on good performance of both the Structural Signatures (Section 5.2), and the Observation Techniques (Section 5.3), along with their interaction with the system model (Section 5.1). This section will examine the effectiveness of the structural signatures particularly, along with the effectiveness of selected observation techniques when used to protect a simply-constructed (yet large-scale) system.

9.2.1 The Infection Experiment

This evaluation example assesses the system's ability to keep an observed system protected against an introduced infection. The protection mechanism, which will be explained in more detail below, relies on the use of a set of pre-defined signatures that specify structural characteristics associated with several types of topology. The deployed observation strategy is entirely reliant on the indicated type characteristics as provided by the signature. The infection is introduced on demand to a randomly selected set of nodes, of size ${}_{i}n$ (a proportion of the total nodes, n). The infection then propagates from one node to another, with each transmission along the propagation occurring according to a certain probability, p. A screenshot of the simulation's network is shown in Figure 56. The significant visualisation features will be explained in the following paragraphs.

The system is protected by an arrangement of observers across a certain set of nodes, which are then immune to the infection. Immunity is used here to mean that these nodes can neither have the infection nor spread it. Immune nodes are shown green in the screenshot. Infected nodes are shown as red, while "normal" nodes are blue. The size of the node indicates the number of connections to other nodes (the degree). Note how even from the screenshot; it is clear that in this simulation the majority of immunised are well-connected (i.e. high degree) nodes.

The propagation of infection is used solely as a measure of success of the observation's deployment, as is described below. Infection does not alter an infected node's connectivity, or its ability to spread infections that are propagated through it. Network connectivity changes are made by user adjustments to the simulation's controls. The implementation of the structural observer model was assessed on two main criteria in a system with a changing topology:

i. The cost of the selected observation strategy (i.e. how many nodes needed observation within a particular strategy on a particular topology)
ii. The effectiveness of protection (i.e. how many nodes were infected/clean after the system was infected according to runtime parameters)

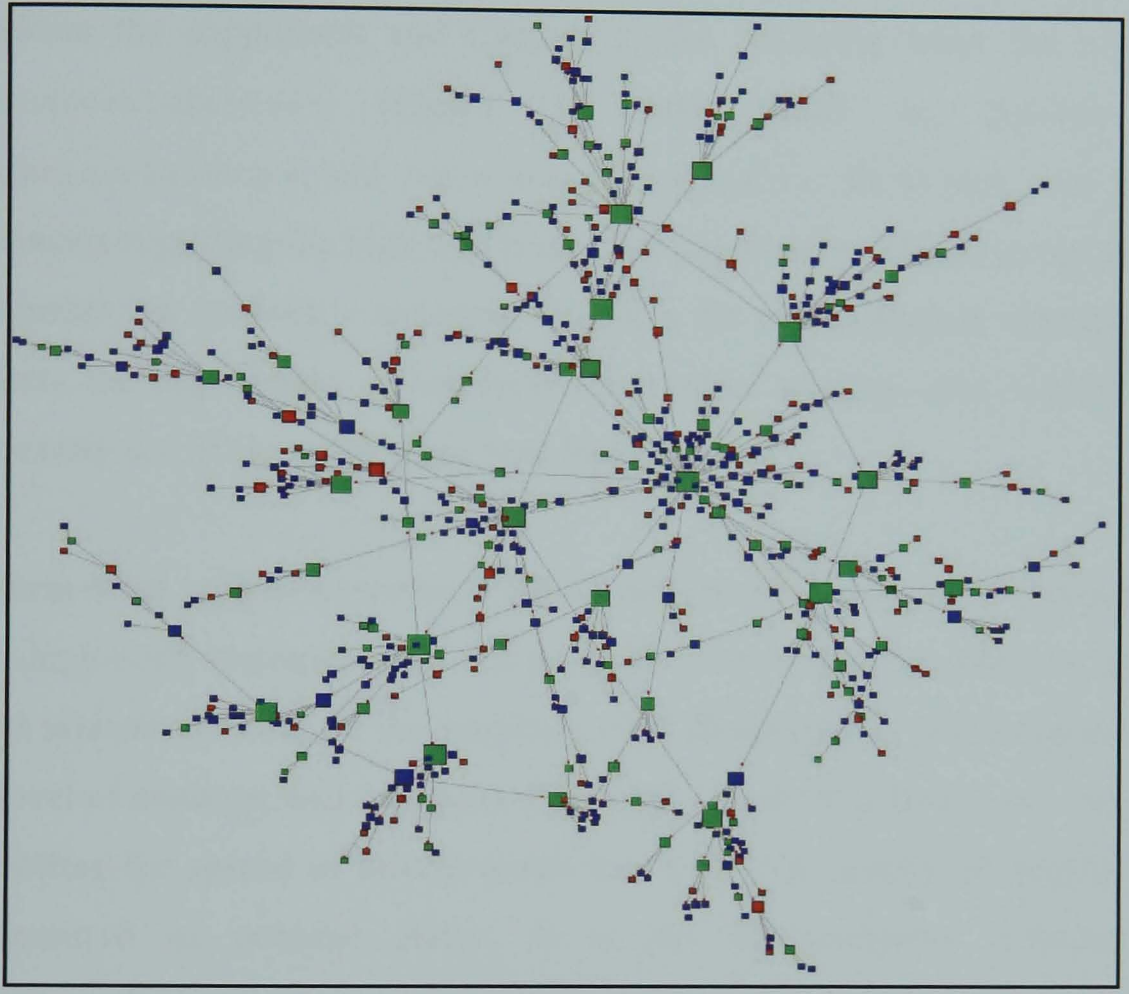


Figure 56: Infection Simulation Network Screenshot

In accordance with the proposed framework, this experiment's observation control was organised as an overlay. Specifically, the system operated this overlay level using two levels of listening & management components:

i. Firstly, an exhaustive set (i.e. one at each system node) of simple low-level listeners monitored the network for base-level topological events such as nodes being connected or disconnected from other nodes. These low level listeners form the basics of the **Structural Observer** part of the architecture, as discussed at the start of Chapter 5. These listeners effectively provide the bridge between the real system components and the structural model, represented as

`ModelledElement` and `ModelChangeEvent` classes identified in the system design, (Section 5.1.1 and the associated Section I.I of Appendix I).

ii. Secondly, an observation overlay controller [7] node monitors each of these low-level nodes. Each time a network structure change occurred, the controller node was notified. This caused the *identification* phase to begin operation. This co-ordinator is responsible for the *Identification* and *Deployment of* **System Observers** (Chapter 5). Again, to provide a precise code-to-design mapping between the experiment and Chapter 5, the controller takes the role of the `StructuralObserver` (Figure 15 and detail in Section 5.1.1), `TechniqueSelector` and `DeploymentCoordinator` all in one – as it makes decisions regarding the high cost protective observation. The observer system re-evaluates the network's structure against a set of topological signatures, then selects the appropriate *system-level* observation overlay, and re-deploys it if necessary (i.e. if the topological type had changed).

This **system-level** type of observer, from an architectural viewpoint, represented the real (i.e. high-cost) resources required to protect, duplicate, or otherwise guarantee important system elements. In the simplistic case demonstrated in the experiment, the system-level of observer was represented by a set of infection observers – responsible for controlling the spread of infection and their cost (i.e. number of deployed units) were measured as detailed above. It is the characteristics surrounding the identification of observed structure and placement of these system-level observers that make up the assessment criteria for this experiment. The next section discusses the results of the experiment on a variety of different system topologies, and presents results according to the criteria defined above.

9.2.2 Results: Random, Regular and Scale-Free topologies

As described earlier, the main assessment criteria for this experiment were the cost and effectiveness of the selected strategy. In order to provide a comparison of this framework's effectiveness, it was decided to introduce two other monitoring

[7] While there is overlap of responsibility into the roles of Observation Techniques (Section 5.3) and Policies (Section 6.1), this experiment intends primarily to give an indication of the success of the identification phase, and as such, detail related to the techniques and associated deliberation is omitted.

strategies that each acted in a predictable and simple manner. Therefore, this resulted in the following framework implementations:

- *Intelligent* – This implemented the key parts of the framework – signatures and observation techniques, and appropriate associations. It used the structural-level observers to assess signatures, which in turn controlled the observation technique used to deploy the system-level (i.e. infection-protection) observers.
- *Random* – This did not implement any signatures and only a single technique. Upon a system model change, the technique would simply deploy system-level observers at a variety of randomly-selected component locations, totalling ¼ of the system's components. As such, the cost of this strategy was always known to be at most ¼ of system resources, yet the effectiveness was variable.
- *None* – This did not implement any signatures or techniques, and as such, did not have a cost overhead on the system. It predictably did not inhibit the progress of the introduced infection, and is included as a control measure.

To recap, an experiment's iteration would consist of:
1. Constructing "system-maps" (effectively large graphs representing several thousand interconnected nodes, representing system components),
2. Using the selected framework to control the observation
3. Introducing "infection" to the modelled system
4. Recording the results

A variety of iterations on increasing system sizes and differing system topologies was conducted. This was followed by the current framework implementation being unplugged and swapped for the next one, and the process would start again. The following graphs show some results from the most striking set of iterations, obtained when the experiment was running through its Scale-Free topology for the different framework implementations.

This first graph (Figure 57) shows the cost of observation – in terms of the number of deployed units at the system-level. Predictably, the *None* Strategy is the lowest cost, while *Random* increases steadily with the size of the system.

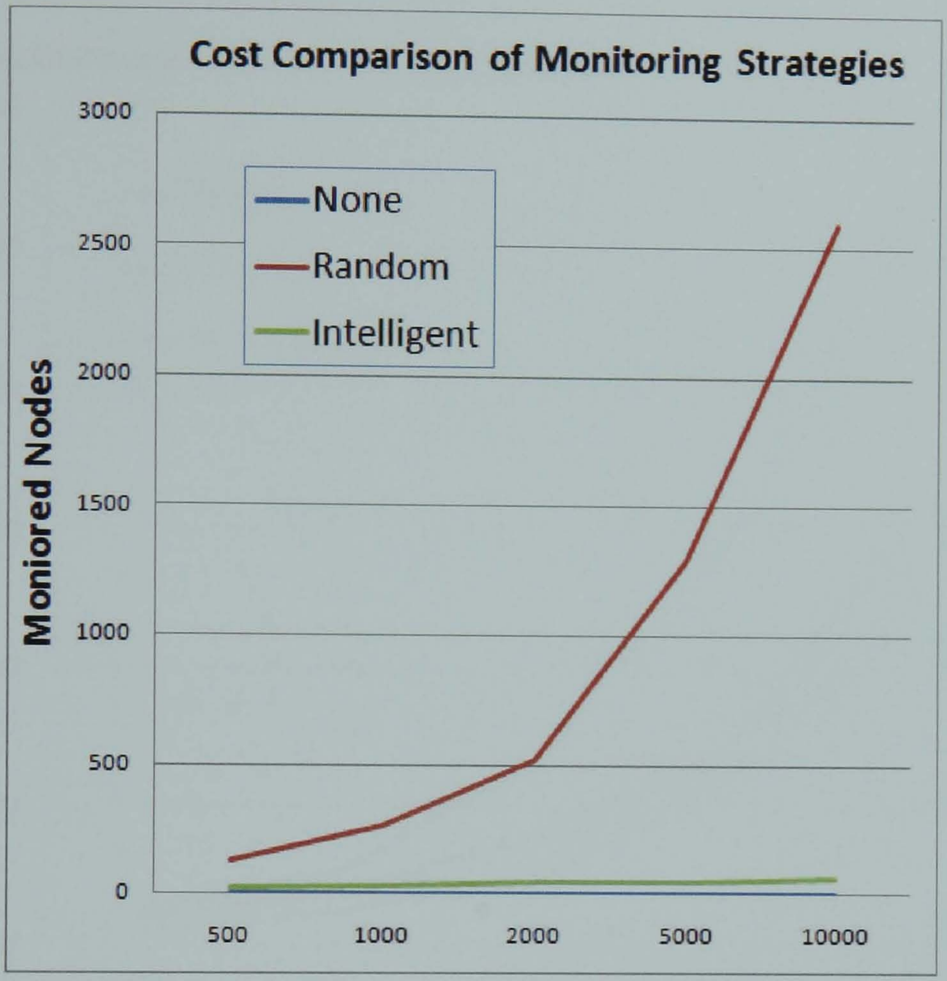


Figure 57: Cost of Monitoring Strategies (x: Network Size vs. y: Nodes Observed)

Interestingly, the Intelligent Strategy, while increasing with the overall size of the system, is a very low-cost option. This suggests that the intelligent strategy is potentially selecting observation targets efficiently. However, in order to make any meaningful conclusions, the effectiveness of the observation strategy must be assessed. The effectiveness measure calculates the effectiveness of the deployed observers in terms of inhibiting progress (Section 9.2.1) of the hypothetical infection. The measures shown on the graph in Figure 58 indicate the number of nodes infected after the infection attempt; the plot for *None* indicates an open control – the effective spread of the infection algorithm without any observer protection. Plotting the results of the *Random* approach indicates the spread of the infection with observers placed at a quarter of the system's components, demonstrating the effectiveness of constrained but considerable resource-usage, yet insensitive placement of observers. The *Intelligent* plot, showing the effectiveness of the structural overlay and its identification, indicates that it stops approximately half the infection paths; when compared with no strategy at all (*None*). The results also show that it prevents 40% - decreasing and stabilising towards 20% (as system size increases) - of infection paths that exist with the *Random* strategy.

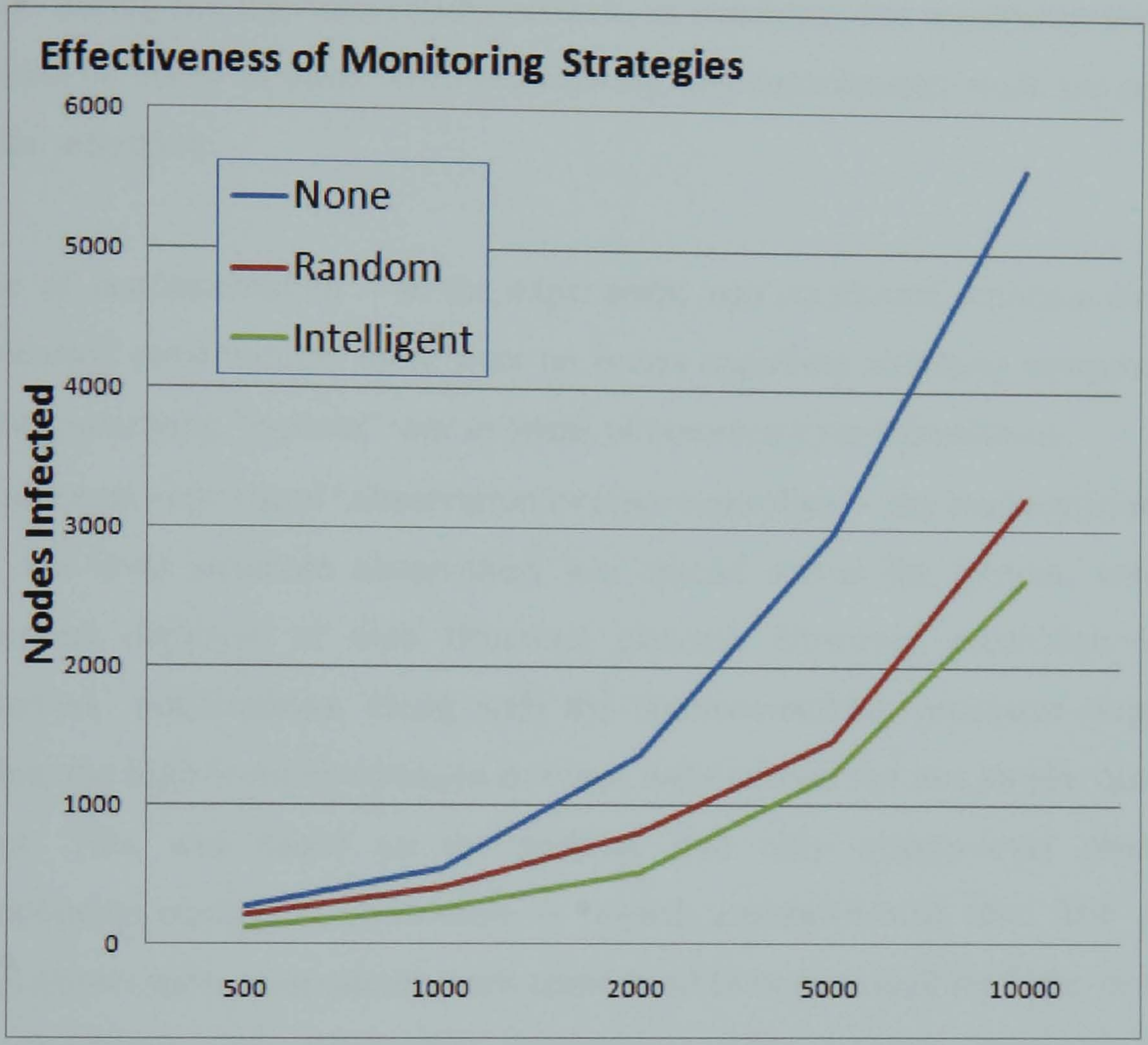


Figure 58: Effectiveness of Monitoring Strategies (x: Network Size vs. y: Nodes Infected)

While the *Intelligent* results do not appear as significant an improvement when compared with the *Random* pattern, they still outperform it. This demonstrated improvement is evident while using a small fraction of the resources of the *Random* approach.

However, while this experimental approach effectively demonstrated the validity of an automatic topology-dependent observation framework, it did not address all of the requirements that had been outlined in the suggested component architecture (Section 4.3). The next section will discuss some of the important limitations that directed the rest of the research work.

9.2.3 Limitations of the experimental work

The limited-framework implementation in this experiment had limitations both in scope, and as a basis for a scalable engineering approach; some of which have been discussed during the previous results section. In summary, the significant limitations that should be borne in mind when considering this experimental work are presented below for reference:

- Ease of instrumentation – as the experiment was conducted within a controlled simulation environment, there were no issues regarding attaching instrumentation to the underlying "system", nor in terms of common event translation.
- Reliance on centralised "observation overlay controller" – the load associated with the low-level structure observation was spread across the system, with basic observers deployed at each structural element. However, resolution of these observers' notifications, along with the decision-making processes required to update the high-level observation overlay, were carried out at a single, centralised point. This was based on the premise that only system-level observation components needed consideration as regards computational cost. The overlay-level observation components were considered to be provided for little, or at most, no significant cost. While this assumption sufficed while demonstrating the practicality of the basic approach, the centralised architecture would present reliability and robustness concerns, along with scalability concerns as the observed structure became ever more complex.
- With reference to the previous point, the small pool of potential signature matches and deployment strategies ignores problems associated with scaling to huge signature databases. Equally, the experimentation does not consider the use and/or impact of multiple positive signature matches. In the case of a system matching several topological characteristics, the resulting observation would only be deployed on a first-matched, first-deployed basis.
- Predetermined (i.e. hard-coded) response lacks flexibility; the potential effectiveness that could result in combining responses for several topological characteristics.
- The "special case" of Scale-Free topology and its predetermined Acquaintance observation & protection response was shown to be extremely efficient and

effective. While research indicates [37, 57] that many complex structures are observed to demonstrate this type of topology, not all topological characteristics can be expected to have such a clear and favourable response.

Equally, it demonstrated the potential effectiveness and basic principles involved in using a signature trigger for a given pattern of observation. In summary, the most severe limitation to this early simulation approach was the lack of flexibility. This manifested itself as an inability to customise the system to make the most appropriate use of observation patterns not explicitly identified at design time (i.e. pre-programmed).

9.2.4 Implications on "Observability"

Section 2.3 introduced characteristics of large-scale complex software systems. Of particular interest were the concepts within Systems-of-Systems and ULS Systems, as discussed in Section 2.3.1. To recap, these systems may – owing to the emergence of new behaviour and structures – exhibit dynamic properties in the elements requiring observation. This characteristic is replicated in this simulation environment by runtime topology change in response to user input, while the observers must handle this change via events noted in their structural probes – Modelled Elements.

At the end of the discussion on ULS Systems in Section 2.3.1, a note is made that when dealing with dynamic system configurations, system monitoring is expected to be tasked with *assurance* rather than *assertions* of behaviour and state. Therefore, it is appreciated that a reduced-complexity or reduced-size observation set is not necessarily going to have the capabilities to make absolute guarantees of *full system observation* – herein termed "*observability*". However, in certain system domains, such as safety-critical systems, the quality of provided assurance and level of *observability* will need to be quantified such that the resulting system can operate within the constraints applied to the safety-critical elements [140]. In these cases, the deployment of observers will need assessing in order to determine the resulting *quality* or *coverage* of observation. This may take the form of a system-wide bounded uncertainty measure, or policy that specifies certain system areas that *require* observation. As this experiment demonstrates, the observation deployment will automatically adjust as the simulation alters the topology type, thus providing an efficient coverage for the detected topology. However, outside of a simulated dataset, can the observers' placement be reliably assessed?

To answer this question, the discussed framework should be considered a foundation in the case of *observability*. The author believes there is no single solution that will fit all requirements; however, the use of simulation provides some useful techniques supported by the structural model to help bound *uncertainty-in-observation* within the observed system:

1. Visualisation provides a useful tool by which the system model can be examined by human operators for placement of particular observers, just as in this simulation.
2. Simulation of deployment and model analysis according to domain-specific *observability* requirements. The structural observer components are responsible for maintaining an exhaustive model of the system's structural elements via the Modelled Element objects, while a Deployment Coordinator is responsible for managing the model of each layer of system-level observation.

9.2.5 Further development of the experiment

The experiment, also reported in [114] and the associated technical report [118] can be extended to adopt more advanced features of the framework and OBD specification language (see Chapter 7 and Chapter 8). This allows for runtime support for adjustable systems' observation and actuation. In particular, provisioning for:

1. Runtime-specifiable operation of the observation subsystem; in its simplest form, if the triggers and/or behaviour were altered by another editor (even a human editor), the observation subsystem would alter its behaviour to match the externalised specification.
2. Alteration of the observation subsystem behaviour by one or more other subsystems; such that the observation subsystem itself or other concerned parties could (request to) alter the behaviour of the observation system

Thus the proposed extension to this experiment should test the effectiveness of the method and the associated performance overhead and other issues incurred to support this feature of the framework.

9.2.6 Summary

The experiments in the paper showed that the envisaged subsystem structure could provide the basis of an efficient observation management subsystem for large scale systems. When equipped with suitable structural characterising metrics and appropriate guidance for co-ordinating deployment, it provided a workable framework which responded accordingly to topological changes, identifying known topological characteristics and deploying a cost-effective observation strategy accordingly.

Experiment results indicated that exploitable structural characteristics would benefit greatly from tailored observation responses, and that required resource allocation could be greatly reduced with no perceivable reduction in observation effect, and in many cases, an improvement. Importantly, it demonstrated that the information required to exploit system structure could be gained without necessitating a full, predefined "designer's" viewpoint of the system; the notion of Typed Observation. Importantly, it showed that a sufficiently-detailed model could be built at runtime (albeit through exploration) and then kept updated, rather than continuously rebuilt. The experimental work did admittedly utilise a degree of centralised control, but identified areas in which distribution of computation could be introduced (such as exploration) while retaining central co-ordination of model events and management of distributed processing (e.g. factory object creation control and co-ordination).

As such, the experimental work was able to simulate some of the aspects of a large-scale topological arrangement by treating a distributed component-based system as, conceptually, a very large data structure which undergoes change – an abstraction that guided the idea of Modelled Element-based structural bridge with an Observer-pattern overlay. Additionally, the implementation of this simulation provided the basis for future research work, some of which helped to guide the direction of this project and was addressed during this research. In terms of experimental gains, it provided an opportunity to implement classes to support the system and helped to complete the architectural design of the system and to introduce suitable levels of system indirection, thus allowing concerns to be suitably delegated among separate processing units.

9.3 Qualitative Evaluation: Applying the Model to "EET"

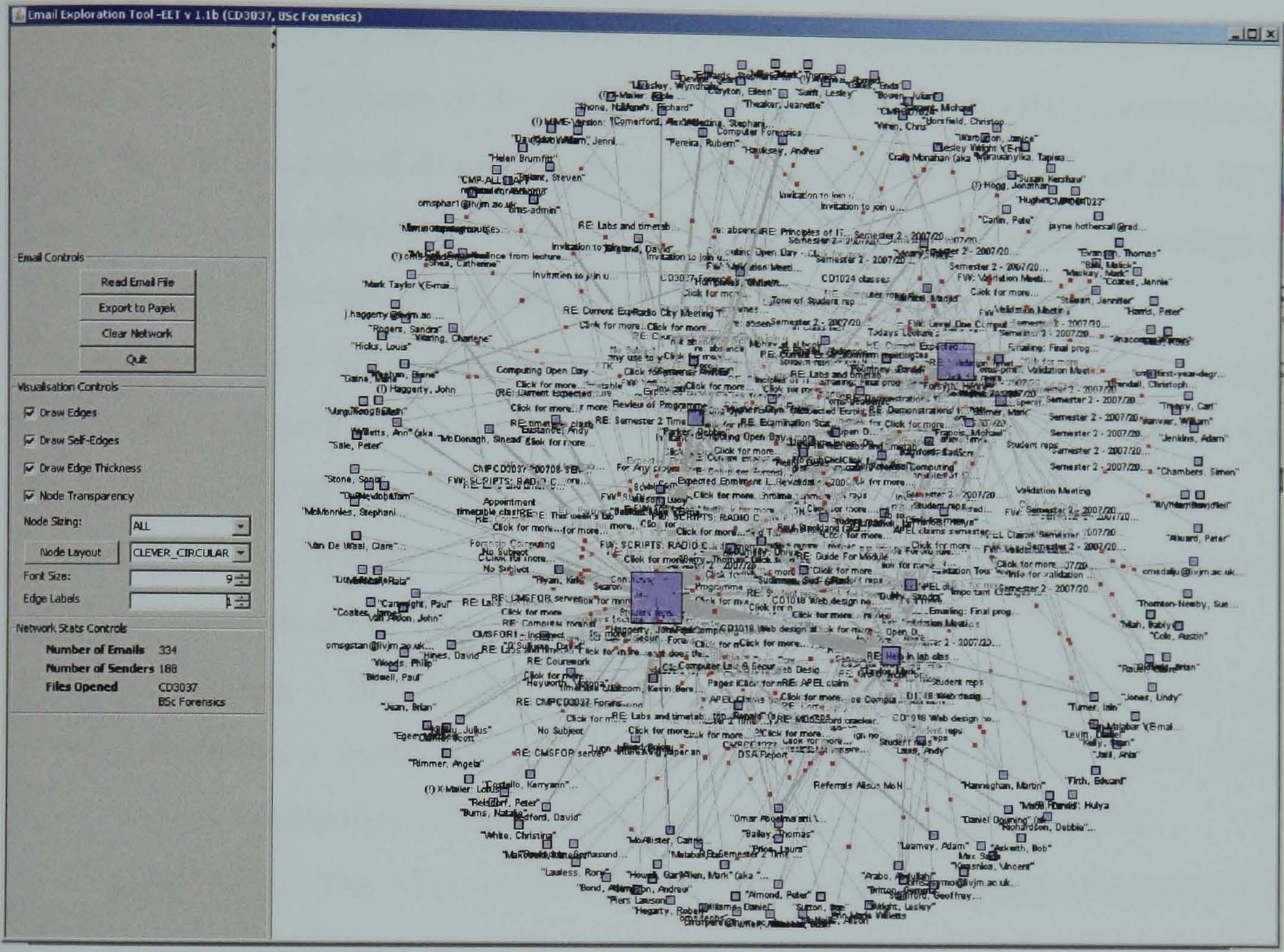
The Email Exploration Tool (EET) is an application to consolidate and visualise any number of email repository files in a given social network, developed for computer forensics analysis of social networks found in electronic communication [141]. Unlike the simulated synthetic autonomic network type of experiment presented in Section 9.2 above, EET operates on very large-scale dataset of legacy data, loaded from distributed collections of email files.

The observer model discussed throughout the thesis is intended for application to systems of systems, which are known to exhibit scale-free connectivity patterns. Equally, social networks often exhibit a scale-free connectivity structure (see Section 3.3). As EET operates on large-scale email repository data, the underlying structure of the dataset being examined and visualised represents a social network; therefore providing a suitable problem domain for the applicability of this work.

This evaluation will provide a qualitative assessment of the results of this research work by applying the findings to the development of a new software utility for EET. This will concentrate on the application of the scaling-management, complexity management and runtime-alteration, their specific requirements along with how they were implemented.

9.3.1 The Email Exploration Tool: Overview

EET is a Java-based Computer Forensics software utility, which allows users to import a set of emails into the software and then visualise the email traffic according to a variety of forensic investigation criteria. EET can import emails on a one-by-one basis, or in "repository" form; whereby files are parsed directly from the user's email client format. Partial support is included for all known email repository formats, but full support in the referenced prototype is limited to Thunderbird email data files. During email loading, basic efforts are made to resolve duplicated email content and senders from duplicated or similar email aliases. The net result of the import process is a graph-type model whereby senders and recipients are connected by their joint email traffic. Further detail regarding the EET software can be found in Appendix II. A screenshot is shown in Figure 59; the software is demonstrated having loaded its dataset, presenting the contents according to the selected settings.



[✓] Draw Edges
[✓] Draw Self-Edges
[✓] Draw Edge Thickness
[✓] Node Transparency
Node Sizing: ALL
Node Layout: CLEVER_CIRCULAR
Font Size: 9
Edge Labels: 1

Number of Emails 334
Number of Senders 188
Files Opened CD3037 BSc Forensics

Figure 59: Email Exploration Tool Screenshot

9.3.2 Model Application: Overview

Despite this software operating only on a single machine and dealing with functionally simple components, it was considered a relevant application for this framework as it *is* managing observation functionality – albeit with a human end-user as the ultimate Observer. Additionally, this evaluation will demonstrate the manner in which the programming model is capable of managing and resolving:

- Scale – the examined datasets can potentially consist of many different emails (forming links / edges) between senders and many different senders/recipients (nodes / vertices) in an investigation's email network.
- Complexity – complex social structures that undergo change – firstly as new datasets are loaded, and secondly, as the "perspective" of the system undergoes change – i.e. the settings that the user inputs as regards importance in the "observed" emails.

- Runtime alteration – the user selection preference was demonstrated via the alteration of OBD-XML which was re-interpreted by the system and reflected in the visualisation.

As such, this evaluation will concentrate on a brief evaluation of the design involved in the following components of the observation framework and its OBD counterpart:

- Simple *ModelledElement* wrapping functionality – the design of the EET-specific Modelled Element class will be explained through a basic coverage of its responsibilities to and interaction with the EET data.
- Algorithms – basic algorithms required to support the social email networks will be described, with their implementation in code and exposure via the observer's OBD mechanism.
- Signatures – design for the signatures will be specified, along with some detail regarding their implementation.
- Techniques – the limited application of observation techniques – in this software as a method for visualisation selection only – will be noted, along with an explanation of their necessary association with the OBDXML.
- OBDXML – The XML-based OBD will be used to demonstrate the manner in which the various observation rules are linked to the user interface, in order that the underlying logic can be altered to represent the current visualisation selections.

9.3.3 Modelled Element: Core EET Support

As described in Section 5.1, the programming model defines Modelled Elements as providing the adapter between the real system under observation and the observer framework itself. Within the EET scenario, the Modelled Elements provide the interaction between the EET data and the observer framework – which itself provides visualisation feedback.

Therefore, in EET, Modelled Elements are synonymous with senders and recipients – effectively email addresses. Therefore, the class used to represent email address must either inherit from Modelled Element, or if this is not possible, must be wrapped by a specialist subtype of the Modelled Element class that can perform the necessary event functionality. Modelled Elements must generate events when connections are added and removed.

As this system is being designed around the observation framework, it is possible to have the main data class (i.e. email senders/recipients) simply extending Modelled Element and firing events when necessary. In terms of required events, this translates in EET to a sender/recipient having a new email (i.e. either sent or received) and therefore another sender/recipient email address associated with it. The alternative approach (i.e. a prewritten class that needs a `ModelledElement` adapter) would require a wrapping `ModelledElement` class capable of capturing the prewritten class's events, then translating and forwarding them in a `ModelledElement`-agreeable fashion. In the prototype version of EET, new data sets can be loaded, but they cannot be *individually* unloaded, so the core support required is only to generate events for new links (as an existing email will not be removed). As such, support for the basics of the add event is shown in Figure 60 alongside the visualisation-specific graph manipulation.

```
public class EmailAddressNode extends ModelledElement
{
 ...
   public void connectTo(EmailAddressNode node, Email email)
   {
     // if (node != this)
     if ((getOutNodesCount() < getMaxConnections()) ||
         (getMaxConnections() == -1))
     {
       EmailEdge edge = new EmailEdge(this, node, email);
       edge.setColor(Color.LIGHT_GRAY);
       this.addOutEdge(edge);
       node.addInEdge(edge);
       ModelChangeEvent mce = new ModelChangeEvent(this, ModelChangeType.ADD);
       fireEvent(mce)
     }
   }
}
```

Figure 60: EET Code Snippet: "Add" Model Change Event generation

As the system is providing a visualisation service, the notion of an observer deployment is subtly different to that in the discussed self-managed systems. EET's high-level Observer – so to speak – is the human operator, and as such, EET Observer Deployment is concerned with making prominent the *selected-for-observation* Email Address Nodes. As discussed in Section 6.1.3, deployment and un-deployment is managed at the Modelled Element via a pair of methods, suitably parameterised if necessary. As such, the Email Address Nodes' implementation of the `deploy()` method causes the node to highlight itself within the visualisation and the `undeploy()` method causes the node to return to its normal state. Additionally, the `deploy()` method is parameterised such that the Observer Deployment Co-ordinator can specify the extent of node highlighting; allowing the visualisation to reflect the number of emails sent, or any other measure of importance of a given Email Address Node, based on the Co-ordinator's knowledge of others.

9.3.4 Developing Algorithms, Signatures and Techniques

Signatures and Techniques were described in Sections 5.2 and 5.3 as the components that provide the system characterisation and observational response, fitting broadly as the specification for the Event and method for the Action in the ECA model, respectively. Implementations of Algorithm classes provide results, in terms of sets of Modelled Elements (in this case, Email Senders/Recipients) translated based on algorithmic and parameter-specific criteria. They can therefore be used by both signatures and techniques to reduce: the complexity or amount of XML-based OBD specification that is required, the amount of duplicated algorithmic logic that must be written, or both.

It is beyond the scope of a short evaluation to go into a full breakdown of all of EET's features, however, this section will examine one of the visualisation's features, based on a recurring theme throughout this thesis – Scale-Free Connectivity - and how it is supported by the observation framework.

- Acquaintance Selection – makes use of the Acquaintance Immunisation and related techniques, as discussed in Section 3.3.4. This identifies senders and recipients that form the "hubs" of the email communication network in the loaded email repositories. This is implemented as an algorithm in order to simplify the externalised specification and to allow its use by both signatures and techniques. A simplified version of `AcquaintanceSelectionAlgorithm` is shown below in Figure 61:

```
public class AcquaintanceSelectionAlgorithm
      implements ModelledElementAlgorithm
{
  private double prob;

  public AcquaintanceSelectionAlgorithm(double probability)
   ...

  public Collection<ModelledElement>
      transformSet(Collection<ModelledElement> in)
  {
    Collection<ModelledElement> interrogated =
      selectRandom(in, in.size() * prob);
    Collection<ModelledElement> neighbours =
      selectRandomNeighbours(interrogated);
    return neighbours;
  }
}
```

Figure 61: EET Code Snippet: Simplified Acquaintance Selection Algorithm

Using any algorithm from a signature or technique is fairly self explanatory; the algorithm object is instantiated with the correct parameters (in the referenced figure, the parameter is the appropriate probability), and then its method `transformSet` is invoked on the collection of `ModelledElements` that requires translation. As the evaluation deals with a relatively simple visualisation tool, the use of signatures and techniques is also simply described. Continuing the case of the Acquaintance Selection algorithm, the associated Signature and Technique provides the following support to the application:

- An appropriate Signature is specified that uses the Acquaintance Selection Algorithm in conjunction with the Acquaintance Nomination algorithm (Section 3.3.4) to detect the presence of scale-free connectivity in the email network.
- An appropriate Technique is also specified using the same algorithm which produces a limited-size model, based directly on the results of the algorithm.

The Signature and Technique are then associated, as described in the next section. A simplified Deployment Co-ordinator is then responsible for creating and managing the observers at each Email Address Node via the `deploy()` and `undeploy()` methods as discussed in the previous section on Modelled Elements.

9.3.5 OBDXML Governance in EET

The XML form of Observer Behaviour Definition (OBD) is used to specify the behaviour in a runtime-examinable and alterable manner. OBD is described throughout Chapter 7 as a mechanism by which the behaviour of the observer can be specified through the instantiation of various OBD objects, rather than embedded entirely in hard code. OBDXML utilises these objects but specifies their makeup in XML, allowing runtime alteration and a reasonable degree of human-readability of the observer system's operation.

The EET software can be operated in either user-configurable or automatic mode. In user-configurable mode, the operator selects the desired visualisation method by setting the appropriate controls, shown on the left-hand side of the screen in Figure 59. In automatic mode, the software evaluates the OBD and uses this to determine Observer (and therefore visualisation) behaviour. In both cases, the resulting visualisation is produced by the application of Techniques to select a set of Email Address Nodes, and the use of the Deployment Co-ordinator to enact the required per-node adjustments.

The behaviour of EET's OBD was initially specified by hardcoded Technique Selector and Deployment Co-ordinator, then extended to use the proposed OBDXML format. Figure 62 shows a simple example, specifying a single OBD and the Technique it uses referring to a common (hardcoded) algorithm as in the previous section. Associating several techniques and signatures in this way provides at a minimum a flexible mapping between them, whilst allowing for additional governing logic to be specified in a form that is easily inspected and altered at runtime. Thus, visualisation behaviour under OBD control could still be customised just as in the user-configurable mode though with the added benefit of automated visualisation changes in accordance with the currently specified OBD.

However, the OBDXML Signature has been omitted from this snippet due to its verbosity, though is included in Appendix II (Section II.I). Comparison of the XML snippet in the Appendix with that in the Figure below demonstrates how OBDXML is compact and understandable when used simply to map between hardcoded elements, though becomes more verbose and deeply-nested, reflecting the binary-tree structure of the operators when the specification involves further computational direction.

```
<tns:techniques ID="AcquaintanceImmunise">
  <tns:targetSelectionAlgorithm ID="AcquaintanceSelectRef">
    <tns:ModelledElementAlgorithm ID="AcquaintanceSelection">
    <tns:parameter>SYSTEM</tns:parameter>
    <tns:parameter>0.3</tns:parameter>
   </tns:ModelledElementAlgorithm>
  </tns:targetSelectionAlgorithm>
</tns:techniques>

<tns:behaviourDefinitions ID="ScaleFree">
  <tns:onEventMatch ID="AcquaintanceSignature"/>
  <tns:withCondition ID="simpleTrue">
   <tns:parameter>true</tns:parameter>
  </tns:withCondition>
  <tns:thenDeploy ID="AcquaintanceImmunise"/>
</tns:behaviourDefinitions>
```

Figure 62: EET Compact OBDXML Snippet

9.3.6 Summary

Section 9.3 provides a brief evaluation of the application of a selection of components from the observation framework; applying the model to a problem that may initially have been considered outside the proposed domain of the framework. In particular, this evaluation demonstrated the use of:

i. Modelled Element as a method of wrapping existing system elements (including data) or providing a base on which to model.
ii. Modelled Element Algorithms as a method of describing collection-translating algorithms and their reuse within Signatures and Techniques
iii. The externalisation support for OBD (OBDXML) and how it provides easy mapping between signatures and techniques, thus effectively externalising the technique selector. Additionally, the relatively high size of the OBDXML string required to support even a simple set of comparisons or mathematical operations was noted.

Managing and visualising large data sets that consist of complex arrangements of connectivity is an example of how the framework can be applied in a traditional/small-scale software engineering and design environment. Application of the framework's design guidance to EET enables it to successfully produce a visualisation overlay; this displays a complex social network that is contained within one or more email repositories.

Additionally, OBDXML was used in a real specification situation: simple visualisation behaviour for an email-based social network according to the results of EET algorithms. This behaved as expected, resulting in a flexible runtime environment, whereby the behaviour of the overlay can be adjusted at runtime – according to social/email characteristics that are deemed important.

9.4 Evaluation Summary

This chapter has described the methods of evaluation used in this research, starting with a brief explanation of the environments and methods used. The evaluations presented in this chapter took a two part approach to evaluating the success of the observer programming model and the framework's approach.

The first part of the evaluative approach assessed via controlled experimentation: the potential success of the signature and technique approach as an underlying control and adaptation model for observer deployment and co-ordination. This used hypothetical, generated "system models" to quantify the level of observer deployment under different strategies and to quantify the effectiveness of limited, targeted observer deployment. This provided positive results within the simulated environment along with highlighted areas for improvement in the model. This evaluation made some simplifications in its approach, which were detailed in Section 9.2.3. While some of these provided useful feedback governing the further development and generalisation of the model, some – such as the component-specific instrumentation – were considered outside the scope of this research and have remained abstractions in the proposed model; via the Modelled Element adapter.

The second adopted a qualitative approach whereby the applicability of the programming model was evaluated. The model was applied to the forensics software developed by the author, EET; demonstrating the generalised nature of the model. In automatic OBD mode, the underlying observer control model was adapted to control elements within the user's visualisation; thus highlighting certain Email Address Nodes according to their importance as defined by:

i. The topological characteristics outlined in a set of signatures
ii. The visualisation response, as defined by a combination of the matching technique and the control (in this case, visual sizing on degree) embedded in the system's deployment co-ordinator.

Chapter 10 – Conclusions and Further Works

The research described in this thesis is centred on the development of a global observer software design pattern, and associated programming model and framework. The framework aims to present a method to manage some of the problems associated with large and complex software systems, through mathematical abstraction, layering, and importantly, modelling in such a way that the system-level observers can attach and operate in a standardised manner, absent from the concerns of scaling.

The next section discusses the motivating factors for this research, and re-examines the approach used. This chapter and the thesis concludes by outlining some specifics in which the author considers further research may deliver worthwhile results and developments in the area.

10.1 Motivation and Research Approach

Software systems continue to evolve in complexity; in addition to increasing product complexity, evolving software development approaches have contributed by increasing system organisational complexity yet vastly decreasing development time. Increased component reuse, composition and service-based systems, via levels of indirection, create process workflow paths necessarily abstracted from those using the system.

This creates a great number of issues for software engineers providing management, debugging and configuration tools for these systems. These can range from technical concerns surrounding interoperability and interfacing through to acquisition and the algorithmic complexity in interpretation of policy governance and control; the problem area encompasses many different fields of research. However, the underlying objective of this research has been to approach the problems of large-scale, complexity of structure and resource constraints upon these large-scale systems.

In order to attempt to tackle this problem, this necessitated an investigation of the following research areas:

- **Identifying the nature of the problem**: In order to start to develop this research, in common with any research, it was important to identify some of the key aspects of the problem domain. With complex and large scale systems, it was clear that there were a variety of aspects to consider, such as component interoperation, service quality guarantees, problem resolution and AI type approaches. For this research, the author chose to concentrate on the aspects of reliably modelling and monitoring large scale systems, along with extracting an engineering-style approach to this problem; ultimately aiming for the specification of a programming and software development model. It was also important to be able to assess other research approaches to complex system management in order to avoid duplication of research effort.
- **Examining commercial and research approaches to software system complexity**: investigating some of the existing mainstream approaches to this problem helped to guide the research inasmuch as areas of weakness or omission could be identified. What became apparent was that autonomic computer system design centred on policy-based control, propagating downward to system levels that could effectively function autonomously with only basic rule guidance. However, methods to achieve that translation between system goal and component behaviour have not been specified, nor the mechanisms by which the underlying control loops would operate. Therefore, the research investigation took the direction of examining modelling and observation techniques in order to provide one of the required underlying support elements.
- **Characterising complex and large-scale systems**: understanding the challenges that are present in complex and large scale systems was a key component in beginning the architecture of a framework to help overcome these challenges. Complex behavioural characteristics such as evolution and emergence were identified, along with examining frequently-occurring structural characteristics such as scale free connectivity. In particular, understanding the characteristics that present obstacles to traditional modelling approaches helps to develop modelling methods that can better deal with, and if possible exploit these aspects.

- **Abstract modelling of systems**: based on the notion of viewpoint-dependent complexity, an examination of a variety of approaches to reduction and abstraction-based modelling provided an overview of several different approaches to complexity reduction. This included traditional top-down functional abstraction, along with introducing methods of mathematical modelling of systems:
    - **Mathematical modelling of large connectivity structures**: While the functional abstraction approaches provide logical, hierarchical overviews, they generally rely on decomposition requiring a viewpoint with full understanding of the system and its behaviour – creating more granular, specified sublevels from abstract part-specifications. Therefore, investigation of mathematical modelling allows reduced-size modelling of systems by statistical methods, or graph-theory-centred approaches. Examinations of graph theoretical approaches in particular allowed further explorations of some of the complex connectivity descriptions, such as Scale Free connectivity, along with their characterisations, and exploits, such as Acquaintance Immunity. This provided scope for the proposed Acquaintance Nomination metric as a basic trigger for a particular type of mathematical *exploit* of connectivity structure.

Determination of sufficient basic information in these fields led to a period of simultaneous design and experimentation, which prompted the following lines of enquiry in terms of architectural and further design considerations:

- **Experimentation Environments and Requirements**: Experimentation was proposed as a method to determine the validity of the whole observation-triggering signature approach. Given the requirement for control over experiments, and the need for repeatable structural characteristics, experimentation in closed simulation environments was identified as the preferred way to proceed. The ideal simulation environment would provide support for large-scale data structures that can be changed during execution, along with sufficient visualisation support to provide user overviews of the *systems* undergoing trial, along with results of the trials.

- **Relevant Software Engineering Design Techniques**: A limited-scope application of structural exploitation presented in simulation form is of research value. However, in order to development the usefulness of the underlying techniques, it was necessary to a) generalise this approach, and b) present it in such a way that it would be understandable and usable by software engineers, from analysis stages through to implementation. Therefore, this work used the Observer pattern as the design mechanism providing the base for adaptation, and included other relevant good-practice approaches; all of which exposed the need to adequately model the system structure needing observation:
    - **Incorporating and Modelling large mathematical structures in code**: While the purpose of the framework is to effectively reduce complexity of the observed system, that does not relieve the obligation for the observer subsystem to access and manage information - across the entire system. Therefore, a suitable method was required in order to co-ordinate this information and put it to use in checking signatures for observation triggering. This *model-based* observation consolidated the concept of *typed observer* recognition, and eventually, deployment.

As the designs were refined and finalised, the issue of runtime dynamism, which eventually led to specification of the OBD, became more important. This required research investigation into the following areas:

- **Approaches to runtime-dynamism**: The approach to runtime adaptation of observation was to consider systems in terms of their identifiable characteristics, rather than via full specifications of expected systems and the desired observation response. However, even by specifying selection and deployment components; the author accepted that given the wide variety of complex arrangements, this may not provide sufficient support for runtime adaptation. Therefore, research was required to briefly investigate mechanisms that would support runtime dynamism of code. The significant areas investigated included:
    - **Domain-specific meta-programming and code (re)generation** – describes, in this context, a variety of approaches to meta-programming as an applicable paradigm for the OBD specification; including the definition

of class specifications or even templates that are then used at runtime to generate the executed code. Several variants of the meta-programming approach were considered, including the following points.

- **Recompiling and "Hot-swapping"** is the process of code generation either in the manner described above, or via standard development channels, recompilation and exchanging it for executing/executed code on the host machine. While this approach has the potential to improve performance in the longer term, and allows greater magnitude of change to a running system, a detailed evaluation was considered outside the scope of the thesis, in order to concentrate on full specification of OBD attributes.
- **Open access via reflection** – allows software to inspect and modify its own program code. In OO this approach typically makes use of object representations of code units such as classes and methods. This work dismissed reflection as a primary access method to dynamism due to the risks to security and excessive complexity; however, it is used to facilitate some of the chosen methods of exposure.
- **Higher-order functions** – describes primarily, in this context, the issue of functions that return other functions in their return set, rather than just a value. While this has greater implications in the context of dynamic software, it was discussed with relevance to the explicit exposure of easily-customisable system predicates and functions via OBD, hence the focus on *functions returning functions.*
- **Interpretation** – allowing the software, at runtime, to interpret and execute externally-specified code – be it a near full-set language, a reduced-set language, or even a precise specification. It is a useful method of dynamism as it can be extremely flexible, but brings disadvantages in the form of complexity, security risks, and ease-of-maintenance plus performance concerns. OBD opted for a reduced-set language to try and mitigate some of these issues whilst providing a degree of runtime dynamism to the observers.

- **Externalisation methods and mechanisms**: were considered a necessary topic in order to describe, understand and appreciate the processes involved in an externalisation. The issues to tackle include taking a piece of executing code, exposing elements of its state to an external (i.e. non-host language) source, potentially allowing change, and then interpreting the changes and reflecting them in execution.
- **XML as a serialisation format:** was discussed during the work in Chapter 7 as a simple demonstration, with widespread usage and thorough language support, in order to use as a *soft* code external specification. While XML has both proponents and opponents in the software development world, it provides a convenient and human-readable form in which external specifications can be demonstrated.

10.2 Summary of Thesis

This research work within this thesis is centred on the concept of resilient self-managing software, with particular focus towards addressing this problem in large-scale, complex system-of-systems architectures. Observation is crucial in the management of any software system, and as such, this forms the motivation for the work.

Trends, particularly as discussed in Chapter 2, indicate that software will continue to increase in complexity, and these software systems will continue to need management and configuration. Commercial research proposals include ideals of self-managing software; entirely abstracting the underlying system complexity such that the administrator's management tasks are greatly simplified, and wherever possible, managed by the system itself. However, many of these proposals take for granted the concept that the balance between autonomy and control has been struck perfectly; the system configuration is sufficiently detailed yet not overwhelmed with confusing constructs.

However, providing a simplistic abstraction of a complex model does not usually translate to a genuine removal of complexity; it is hidden, apparently reduced, and organised in such a way as to make it appear simple. Details are omitted, and levels of external control are reduced. It is the *autonomous* nature of the proposed next-generation software that is supposed to manage this balancing act. Therefore, at the system-level, methods must be in place to reduce the *apparent* complexity manifesting at the next higher level, while still recognising and interacting appropriately with the local complexity at the lower level. Importantly, these methods must be able to operate in a variety of conditions, on a changing structure, whilst remaining efficient.

The potential of graph theoretical techniques as a method to manage the scale and structural complexity of a system is clear; both in terms of the efficient identification of characteristics, and of special features and areas of the structure that are considered vital, abnormal or otherwise significant. The discussion of Scale-Free connectivity led to research examples studying interesting perspectives on the emergence of the connectivity, its unique features and the criticality of structures with such connectivity. Additionally, it led to the development of a novel metric that utilises existing work in Acquaintance Immunisation.

Therefore, addressing the stated problems, the thesis has presented a programming model and framework, taking the form of overview architecture, subsystem level designs, and implementation guidelines. This framework aims to help software engineers develop software components that can deal with these system-level issues of scale and complexity. The approach results in an adaptive overlay of observation components that can adequately characterise or *type* the configurations they are dealing with, identify key areas and suggest suitable observation exploits or techniques, and select them according to defined policy. This overlay is intended to support the implementation of system management, monitoring and visualisation components.

Additionally, given the evolving and emergent nature of the software systems under investigation and those likely to develop, the need for a degree of runtime adaptability has been identified. The framework facilitates this through the Observer Behaviour Definition (OBD) design concept, specifying the behaviour of the observer (Signature, Observation Technique and mapping policy) in an ECA-like manner. The OBD forms an externalisation-friendly description, which has been externalised into

XML during this work; though the designs encourage a loose coupling thus permitting any suitable externalisation mechanism's use.

The validity of the proposed *typed observation* has been evaluated by both experimentation and case study. A simulated Repast environment was used to test the likely performance of reduced-cost observation and to assess the improvements that could be gained by selecting targets based on the observed topology. The programming model was then incrementally applied to the development of an email-collating forensics visualisation tool; thus demonstrating both the method of utilisation for these design methods on a software system along with the framework's flexibility in terms of applying the model to systems that exhibit only a few of the characteristics discussed throughout this work.

10.3 Significant Contributions

The research work contained in this thesis has led to the specification of a programming model, aimed to assist engineers in moving toward autonomic management of large-scale and complex systems. Early on in the research, it became clear that Autonomic Computing is a large research field in its own right, attracting significant research input from IBM, and bringing together several research communities outside of computer science. As such, given the author's position as a software engineer, the research focused towards the specification of technical and programming aspects concerning the management aspects required to support autonomic control in software.

One significant concern identified surrounding these management aspects was the matter of observation and monitoring. As such, the programming model looked to address the issues associated with observation of a large-scale complex system; particularly issues associated with management of resources (and resource costs) – typically associated with the traditional exhaustive approach, and those associated with approaches that relying on unrealistic quantities of design knowledge. Equally, the model specification was approached with a conscious awareness that research into complexity within self-managing systems very often led towards low-level, biologically-inspired, emergent, self-regulating systems that could maintain a state with true autonomy. At the other end of the microscope, overall system control is often discussed in terms of well-specified policy and goals; the two levels often seem disconnected; the engineering problem of governance and control vs. the art of self-regulation and autonomy.

This project has approached the problem retaining the model-based design of the observer system with an understanding that an exhaustive model of systems undergoing observation is not a feasible approach for observing large-scale or complex system. As such, the approach taken has concentrated on specifying the relationships between a logical "observer" and its actual system-level monitoring points within the system. The programming specification has done so in a way that is intended to support efficient reduced-set observation, along with evolution and any form of system change. This was enacted through both characteristic-based *typed* observation and the facility to adjust the measures and associations the observer subsystem uses at runtime, without recompilation. It necessitated the following significant contributions, as outlined originally in the research aims:

Identification of significant issues that complicate the management of software (and software engineering as a whole) of a large scale and with complex system structures: Issues connected with structural complexity, including evolving structures and a lack of design-time specification present challenges that complicate traditional approaches to management; specifically monitoring and observation. Additionally, challenges presented purely by the size of the system being monitored create issues regarding co-ordination of the system's units, resource availability vs. requirements and the difficulty in creating an exhaustive observation model.

Investigation into methods that can integrate complexity management and software engineering: Methods discussed include ideas mooted under the Autonomic Computing umbrella, including the self-regulative and managing approach; specifying normative overview control of otherwise-autonomic system-level units [9, 142]. Additionally, mathematical modelling of structures and relevant software engineering techniques were collated in the research stages:

- **Collection and assessment of complexity management techniques:** Autonomy at the lower levels is often compared to biologically-inspired computing paradigms [85, 143-145]. Equally, observed characteristics of complex topologies can be exploited to reduce requirements for exhaustive monitoring, such as the Scale-Free and Acquaintance-based approaches [37, 43, 57, 109]. However, joining this autonomy with the software engineering concepts of control, parameters and regulation presents the issue of exercising control over this autonomy and self-regulating behaviour. As such, modelling approaches based on mathematical exploitation [39, 101] allow for the adoption of model-based approaches that can overcome some of the identified scaling issues.
- **Collection of relevant software engineering practice:** The issue of relevant software engineering design practice was effectively split into two separate categories; those investigated in preliminary research, and those during system design. The first was a consideration of the architectural approaches and design patterns that support the notion of large-scale monitoring and complexity management. This included those that have featured strongly in the research such as the Observer pattern [27], modelling techniques rooted in graph theory, and other relevant overlay design techniques to support modelling abstractions and overlays, such as Adapters, Factories and Façades . The second considerations centred on runtime dynamism, and included collecting and considering approaches towards *Meta-programming*, including considerations towards Reflection, and a variety of potentially-applicable programming approaches [134].

Specification of software engineering model for designing within large scale and complex systems: Based on the preliminarily research, an observation approach was proposed that retained a model-based overview of the system. However, the model-based overview was implemented via a series of unit-hosted modelled elements, adapting the real system model to the *low-level* observation model. The *low-level* model undergoes translation to the reduced-size *high-level* observation model utilising the Signatures, Techniques, and connecting Association and Deployment components that make up the bulk of the framework. Change is propagated from the low-level modelled elements via events. This was accompanied by an optional runtime specification and alteration plug-in sub framework that allowed further configuration of certain components and specification of the main conditions of the observer system's behaviour – the OBDXML.

Evaluation of methods; case studies and real-world applications: The programming model, its useful application, and the validity of important components were assessed in a series of experiments and applications to real world case studies. The case studies were aimed to assess the quality of generalisation in the design and the ease of application throughout different problem domains. The case studies and simulations used as evaluations were:

- **Signature and Deployment *Infection* Simulation** – The simulation, described in Section 9.2, created a mock system structure representative of service-sharing components, deploys an observation overlay, and then attempts *infection* of the simulated system. The infection is intended to represent a viral-type denial of service, whereby components infect neighbours and reduce or remove their functionality. The observation overlay represents protection from this infection; both directly for the observed component and in terms of prevention of propagation to others. Signatures and Deployment techniques were used in the observation deployment mechanism to provide appropriate observation coverage and were found to provide useful improvements over uninformed approaches. Additionally, the experimental work led to additional functionality being proposed for addition to the framework; further details of which can be found by referring back to Section 9.2.5.
- **Email Exploration Tool (EET)** – The development of EET, while outside the scope of *this* research specifically, has been undertaken during the PhD research programme in conjunction with another researcher. While this was primarily a Computer Forensics software prototype, the author (as its developer) saw this as an opportunity to use the framework as a design tool in order to facilitate the visualisation of a social network of email contacts based on imported raw email data. In this evaluation, the overlay took the form of a visualisation tool, rather than an observer in the usual sense. The usefulness of the framework was demonstrated along with the ease with which it could be partially adopted to provide functionality outside its initial scope.

10.4 Critical Review

This research has set out to produce a software engineering approach – specifically a programming model – for the observation of large scale and/or complex systems; in order to facilitate the monitoring, and ultimately, management of such systems. Monitoring and Observation forms a well-established software design principle; required as in important part of the control mechanism/loop in even the smallest of scale systems. It is widely used and formalised from an Object Oriented perspective by way of the Observer design pattern; defining a subject/target and observer/monitor relationship where the observer(s) takes the responsibility for registering on subjects in which they are interested, who in turn take responsibility for notifying interested observers via events. However, as evidenced by this relationship, observers must explicitly elect to observe those elements that they wish to observe, and of course, take responsibility for receiving (and processing) any events that are sent.

This basic observer model is incompatible with the systems under consideration for two main reasons. Firstly, scale; as system size increases, the observer's set of target elements (and therefore, potential for received event size) increases. As such, a point will be reached where the event load – and/or the associated deliberation for incoming events – reaches too high a computational load. Secondly, complexity; complex systems tend to exhibit properties that make them difficult to model consistently. One such property is that of an evolving or otherwise unstable structure, with components being introduced and removed; thus, in addition to scaling concerns, the observer must constantly re-determine its targets and manage its own redeployment.

This research is therefore primarily situated with others in the fields of self-managing and adaptive software; largely under the umbrella of Autonomic Computing. Current research in this field is active – from software engineering, mathematics, and "biologically-inspired" or Organic research schools. Existing research work ranges from the discussion of engineering architectures through to the specification of domain-specific techniques that can help to manage certain aspects of complexity and/or scale in systems that follow a prescribed makeup or certain characteristics.

However, current research is lacking in terms of a software engineering design approach – more specifically, a model for the implementation of software management systems providing autonomic-type functionality on complex systems. As described throughout Chapter 2, existing research has as yet either avoided the software design considerations or has produced a specific design targeted at a specific application or at its most generic, a problem domain.

As such, this thesis has aimed to address this problem and to formalise the solution; taking the form of a two-part specification for a programming model to manage the observation concerns, facilitating the monitoring of large and complex systems as a step towards control in autonomic software:

- Part one of the programming model – the Structural Observer Framework – addresses the concerns of scalability using best-match exploitation of structural types, facilitated with signatures, associated techniques and deployment models, allowing domain specific and system-wide concerns to influence observation tactics. System-level instrumentation and evolution are abstracted by the overlay adapter model – Modelled Elements – thus allowing the system to report changes to the framework, which arranges alterations to deployment as required.
- The second part of the programming model – the **O**bserver **B**ehaviour **D**efinition – looks to provide greater flexibility in the adaptive part of the Structural Observer Framework; acknowledging that as the system evolution progresses the initial design-time policy for deployment and management may alter and removing and re-implementing the overlay may not be a viable solution.

10.5 Suggested Further Works

The work contained in this thesis looked to develop an efficient, manageable observation framework that would be easily understandable by developers and engineers familiar with common and domain-specific approaches toward instrumentation, observation and simple OO design patterns.

However, it is appreciated that this problem is situated in a field with much greater scope than can be covered sufficiently in a single research project; matters such as alternatives to design issues, new issues identified along the way, along with greater specification of the rest of the autonomic control problem. This section looks to highlight some areas in and around the framework in its current form that may benefit from further research and problem-solving. The suggested future work is therefore split into two categories; Project and Framework-Specific, and General and Wider Framework related.

10.5.1 Project and Framework Specific

The following items of future work are tightly constrained to this current project and as such, are extensions of the design, implementation details or minor architectural additions.

- **Increased Developer support for this programming model** – The thesis has provided discussion and documentation, including overview software architectures and designs, and detailed designs for the individual components. However, in order to increase the ease with which the framework can be adopted and model implemented, the creation of developer tools to support common design and implementation tasks would be beneficial. Popular IDEs such as Eclipse include a plug-in development kit; thus allowing development of such features and easy distribution and integration into the developer's workstation.
- **OBD Extensions:** Further investigation into the integration of XPath, a standard XML-based language; thus allowing greater extensibility of system policy specification in XML, while helping to reduce the string size for simple comparisons and operations. Additionally, alternative forms of specification languages could be utilised to help facilitate greater externalised OBD functionality; thus allowing a more system-appropriate balance between, for example, portability of format and computational features.
- **Beyond Scale-Free Connectivity:** Much of the work in this thesis regarding structural identification and exploitation has centred on well-researched elements of scale-free connectivity. In order to provide better-tailored support to a greater number of structural features and sub-features, further research is required into complex system structure in terms of features, exploits, system partitioning.

Additionally, further research into, and general collation of complexity management techniques in order to increase the applicability and base "library"; thus facilitating greater sophistication in terms of the structural's observation deliberation.

- **Integration of runtime-adaptability technologies and patterns** – Given the potential computational load involved in a many-signature Structural Observation model, additional overhead associated with an interpreted specification may be problematic. While the OBD model separates the externalisation mechanism – thus only reparsing strings where necessary, the OBD specification objects must be re-evaluated at each call. As such, there is further useful research involved in a greater investigation of meta-class, part-compiled, Just-In-Time or similar approaches to runtime adaptability. Some potential options for future research work were identified in Section 7.3 alongside existing established technologies.

10.5.2 General and Wider Framework Related

The following items discuss some potential alternative or additional framework approaches and an extension of the framework beyond observation; moving closer to a full specification for the behaviour and co-ordination of self-managing systems. These points are therefore considered important research questions for any given approach to equipping large-scale software with global self-management capabilities:

- **Structural and System-level observation systems** – The research in this thesis adopted a level of separation between the structural or deployment control and the underlying system-level, business observation. Specification of the system-level observation level is deliberately abstracted due to the likely domain-specificity of these components. Some design methodologies, such as multi-agent systems do away with this separation of concerns, placing responsibility for attachment and adaptation management with system designers. However, for observation system designs that retain this separation; it is conceivable that in certain applications, a greater level of cross-observation co-ordination will be required. As such, further investigation into management at the Observer Deployment level - between the structural and system observers – and the concerns and new interfaces required would no doubt bring greater flexibility to the system-level adaptation.

- **Feedback to Observed System** – This work focussed on the attachment of and observation on the software systems. As discussed in Section 4.2, the issue of feedback was identified as a complex subject in its own right. Assuming a high-level deliberative system is capable of providing such feedback; further investigation and design work is required in terms of specification for the observational response. Concerns such as how to specify the feedback, how to resolve or evaluate it against undesirable behaviour, translate it between levels of hierarchy and finally, how to ensure that the system-level component(s) responsible for enacting the feedback are capable of doing so - are all important issues.
- **New or Novel Event, Situations and Feedback Types** – Observation and management systems operating on complex and particularly *evolving* systems must have the capability to described newly-observed events, event *types* along with new feedback types. In the case of events, this may involve development of existing research such as IBM's Common Base Event [146] to ensure that all system events can be described in a commonly-understood format. Equally, the author suggests research into associated methods for correlating lower level events into composite events to increase the ease with which higher level governance policy and components could be written.

Glossary

AC:	Autonomic Computing (see OC)
ACL:	Agent Communication Language
ADL:	ADL – Architectural Description Language
API:	Application Programming Interface – also typically used to refer to a library whose instructions and specifications are provided by way of a detailed method-by-method interface instruction.
BSD:	Berkeley Software Distribution – now used to describe a type of free software licence that usually allows unlimited redistribution providing certain conditions are met.
CBE:	Common Base Event (IBM implementation of WSDM WEF)
CI	Cognitive Immunity; see Section 1.1
CIL/MSIL:	Common Interpreted Language (MicroSoft Interpreted Language) – the core language of the .NET framework. See also JVM.
CLR:	Common Language Runtime – Microsoft's VM created for executing CIL (and its bytecode) within the .NET framework. See also JVM.
DARPA:	Defence Advanced Research Projects Agency; USA Military and Technology research agency
DTD:	Document Type Definition – see XSD, XML
ECA:	Event-Condition-Action – a description for a broad range of simple event-driven rule-based architectures; where the event triggers the evaluation of a set of rules and then performs an action
EET:	Email Exploration Tool (by the author – see Evaluation)
FIPA:	Foundation for Intelligent Physical Agents; trade and standards association for software agents; IEEE-accepted.
JDK:	Java Development Kit – broadly considered to be the bare minimum software utilities required to create and (part) compile a piece of Java software.
JVM:	Java Virtual Machine – a VM specifically for executing Java byte code. See also CLR
JRE:	Java Runtime Environment – the minimum software (and hardware) required to execute a piece of Java software created with a JDK.
OASIS:	Organization for the Advancement of Structured Information Standards; trade-based data standards organisation, whose standards are usually XML-based.
OBD:	Observer Behaviour Definition / Description *(originated in this research)*
OC:	Organic Computing (see AC)

OO:	Object Oriented / Object Orientation (as context); software design and programming methodology based on Objects – i.e. structures that combine data attributes and related data-manipulation methods.
POJO	Plain Old Java Object – Refers to standard Java objects; i.e. those that do not have to implement interfaces or extend specific classes.
SRS:	Self-Regenerative Systems (see DARPA)
UML:	Unified Modelling Language
VM:	Virtual Machine
W3C:	World Wide Web Consortium; independent standards organisation
WSDM:	Web Services Distributed Management – Collection of Web standards defined by OASIS. See WEF and OASIS
WEF:	Web Event Format; definition within WSDM. See WSDM and CBE
WXS:	W3C XML Schema Document – (used interchangeably with XSD)
XSD:	XML Schema Document – see WXS, XML
XML:	eXtensible Mark-up Language

Appendices

Appendix I – The Programming Model: Additional Discussion

This appendix provides additional information in support and discussion of the observer's programming model, as discussed in Chapter 5. The following text details some relevant features and a detailed example that was omitted from the main text in order to improve readability.

I.I An Example Structural Model Implementation

Section 5.1.1 gave a software design overview for the structural modelling module of the proposed framework. In order that the designs remained suitably generic it was necessary to omit detail in some areas. Of particular note was the manner in which Structural Observers and Modelled Elements would manage the instrumentation of the real underlying system component. This section considers the design and partial implementation of a Structural Observation subsystem on a simple "service-oriented" software system.

This example system is made up of a number of components and a number of different service types. Each component is capable of providing a number of different service types, and may require one or more service types. Each component's goal is to ensure that it has all its required service dependencies fulfilled. A snapshot of the hypothetical system's configuration is shown in Figure 63.

Each component is represented within the observation system as an instance of `ModelledElement`, which demonstrates the use of an implementation of Modelled Element Factory, facilitating necessary separation provided by each element between component-level instrumentation and structure-level observation.

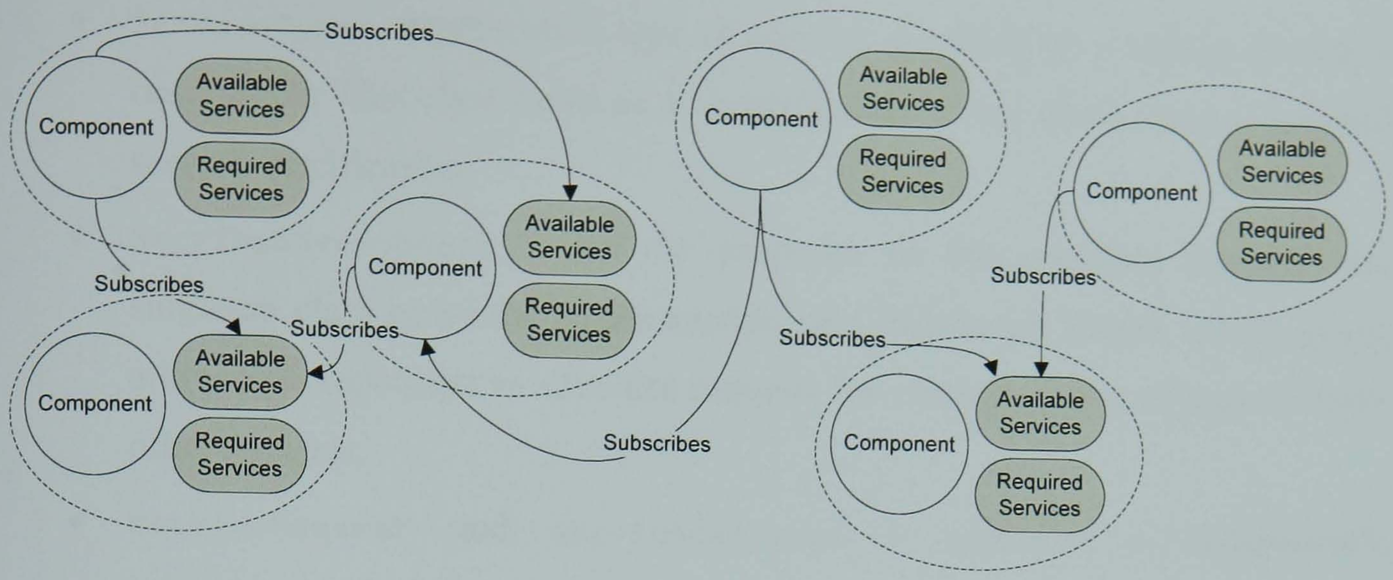


Figure 63: Example System Setup Showing Component / Service dependencies

During execution, existing components are removed and new components added, crudely simulating the evolution associated with a complex system. The service provisioning and requirements of new components will be generated randomly. This is to demonstrate how the Structural Observers will be expected to keep the system model updated based on the `ModelChangeEvents` being fired by the remaining components' representations.

At this level, brief consideration will be given to the creation of and communication between individual observers – via a measure of how successful the system is connected (in terms of dependency fulfilment). However, full details on the observer network's maintenance and control will be deferred to the later sections – describing the application of the observation techniques.

The example system is built around the following types of object, shown in context in the UML class diagram found at Figure 64:

- `Component` – represents the working elements of the system that are responsible for providing and subscribing to services from other components. When a component has acquired contracts for all its required services, it is said to be in a "fulfilled" state. At this point that it can offer its own available services to other components. It is each component's fulfilment state that will be used to measure the success of the system. This type of architecture requires that there are some standalone components – those that do not have any dependencies.

- `ServiceType` – represents a type of service; specified by a unique ID and a description. This class exists as a simplification of any given service-oriented service specification.
- `ServiceAdvertBoard` – for the purposes of this example system, this singleton class provides the abstraction of a distributed shared advert space, allowing components to advertise requests for services and to arrange to fulfil other requests.
- `ServiceRequest` and `ServiceContract` – represent a component's interaction with the service advert board, to either request a service or offer to fulfil one respectively.

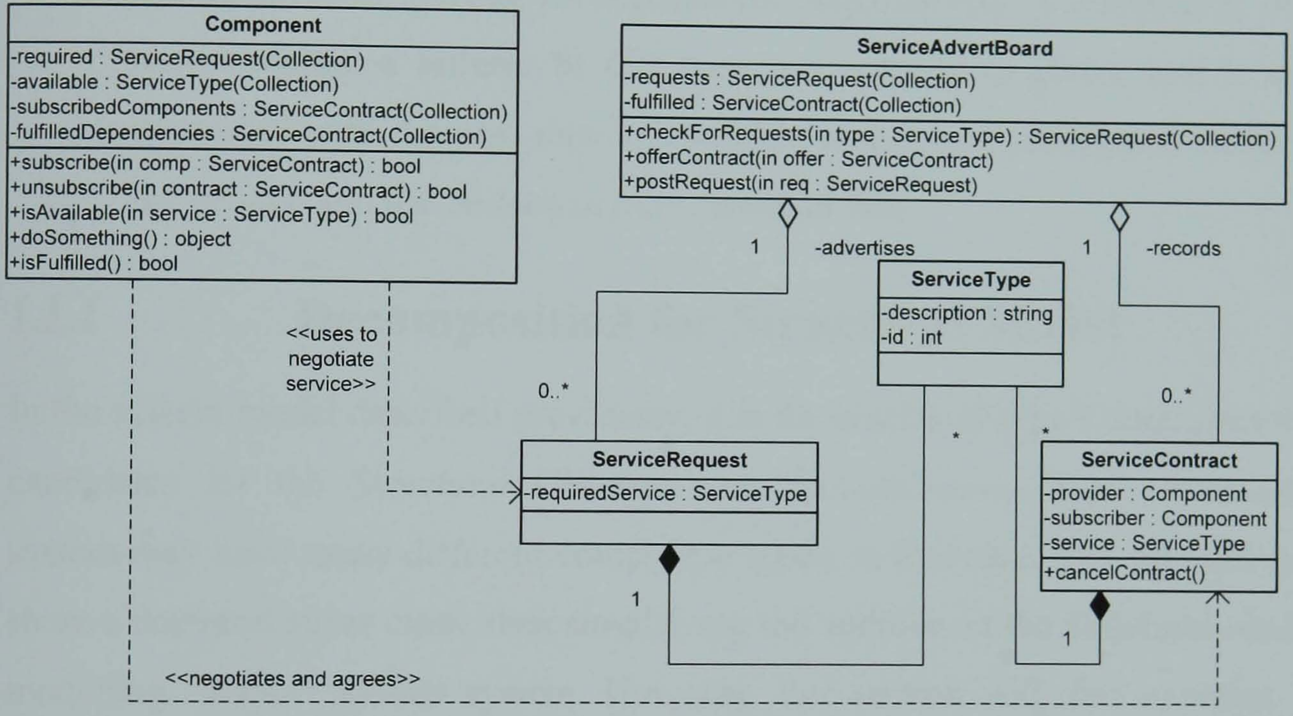

Component
-required : ServiceRequest(Collection)
-available : ServiceType(Collection)
-subscribedComponents : ServiceContract(Collection)
-fulfilledDependencies : ServiceContract(Collection)
+subscribe(in comp : ServiceContract) : bool
+unsubscribe(in contract : ServiceContract) : bool
+isAvailable(in service : ServiceType) : bool
+doSomething() : object
+isFulfilled() : bool
ServiceAdvertBoard
-requests : ServiceRequest(Collection)
-fulfilled : ServiceContract(Collection)
+checkForRequests(in type : ServiceType) : ServiceRequest(Collection)
+offerContract(in offer : ServiceContract)
+postRequest(in req : ServiceRequest)
1
-advertises
1
-records
ServiceType
-description : string
-id : int
<<uses to negotiate service>>
0..*
0..*
*
*
ServiceRequest
-requiredService : ServiceType
ServiceContract
-provider : Component
-subscriber : Component
-service : ServiceType
+cancelContract()
1
1
<<negotiates and agrees>>

Figure 64: Example Service-based System: Simplified Class Diagram

The system operates within the following logical constraints:

- When a Component is created, it is randomly assigned a number of **required** and **available** Service Types. Service types may be duplicated, indicating that the Component has a number of a given service types that it requires (or can offer).
- If the component requires any services, it then creates a number of Service Requests to describe those it needs, and then posts them on the system's Service Advert Board.
- When a Component has all of its service dependencies fulfilled, it then regularly checks with the advert board to determine if it can fulfil any outstanding requests.

- If there are any outstanding requests, the component can offer a new Service Contract to the requesting component. If the requesting component agrees, it accepts the contract, and at this point, a dependency is set up.
- Components can then make use of their contractually-subscribed services. This is simulated by regularly invoking the *doSomething()* method on the component providing the service, parameterised with the appropriate service type.
- At random intervals, a component may be removed or added from the system. Removal can take one of two forms: a component may voluntary opt out of the system; in which case, it can choose to cancel any contracts it may have agreed to, thus alerting its dependents. Alternatively, a component removal may simulate a failure. In this case, no warning is given, and it may go unnoticed until the next time a dependent component attempts to make use of the service and the `doSomething()` method fails.

I.I.I Decomposition for Structural Model

In the system model described previously, it is the objects of type Component that are candidates for the Structural Observer's `ModelledElement` type. Although this system may have many different component types, in this case, they all conveniently share a common super class, thus simplifying the addition of the structural observer's modelling "hooks" to this system. However, this section will also examine which design methods are best suited for integrating the required functionality in a variety of code locations.

Given that each instance of Component contributes to the overall success of the system and has the potential to fulfil a dependency for another component, it is also clear that there is a direct relationship between a modelled element and a component – each component will be represented by a single modelled element.

The decision whether to model a system element should be based on the following consideration – does the addition or removal of this component directly affect the system's ability (now or in the future) to achieve its goals. Generally speaking, the decision on an element's "importance" will be effected at a level above the structural observation; therefore, a compelling argument would have to be made to omit a system process or data store. One such example would be software architecture exhibiting a number of clearly separated concerns - that together affect system operation - but can be modelled entirely independently.

As outlined in the previous section, the modelled element's primary responsibility is to provide an interface between the real system-level component, and the observation system's structural level. Modelled element creation is deferred to a factory which would be implemented such that it creates a suitable Modelled Element subclass supporting the required functionality.

The design requires modelled elements to generate suitably descriptive model change events that will be sent to the structural observers – which in turn will ensure the correct state of the observation model. The earlier suggested minimum set of event types are:

-an event to represent **adding (or connecting)** a new model element,

-and another to represent a **removal (or disconnection)**.

These event types allow an observer to create a "bare minimum" graph of modelled elements – encompassing connection information – and if the events are implemented rigorously across elements, the resulting graph can therefore contain directional information. However, even the creation of events to support this simplified graph requires some design thought, considering the business logic of the system. Therefore, to support this model, responsibility for firing the appropriate events must be determined. To do so, it is imperative to find out at exactly which point a service contract dependency is set up, and when it is cancelled, or otherwise rendered void. Examining the specification above; it is the responsibility of components to manage their relationships with other components. However, they do so utilising a set of supporting classes, thus negating the assumption that a "connect" or "disconnect" (i.e. subscribe to / cancel contract) operation is a synchronous, atomic operation. The creation and validation of a contract requires the agreement of two components, while the termination of a contract effectively only requires action from a single component. Equally, while the classes provide methods to formalise cancellation operation, it is not always possible that they are adhered to - as dictated by the system's "business" constraints (particularly that regarding component failure). Therefore, the notification of a component disconnection may also need to be sent when it is realised a subscribed component has failed – effectively when a `doSomething()` method call fails.

As such, the events should all be generated within the Component class at the following points:

- Exception handling within `Component.doSomething()` – marks a Remove/Disconnect Event
- At the success marker in `Component.subscribe()` – marks an Add/Connect Event
- When `Component.unsubscribe()` is called – marks a Remove/Disconnect Event. A new type of remove event may be required to indicate that this was an "unauthorised" disconnection.

Therefore, if the Component class has not been designed to generate events that describe its connectivity alterations, then this is the first element that must be amended. This logic must be included in the Component class, as per the example code snippet in Figure 65, (assuming a method `fireEvent` is defined that will send the specified event to all the component's listeners).

```
public void doSomething(){

  try {
      // do normal service comms
  } catch (Exception e) {
      ServiceEvent se =
        new ServiceEvent(this, ServiceEvent.DISCONNECT);
      fireEvent(se);
  }
}
```

Figure 65: Example "hook" into Component for failure disconnect

Then, a new class, `ModelledComponentElement`, must be created, inheriting from `ModelledElement`. Instances of this class are examples of the Adapter design pattern [147] and are responsible for translating the events from the Component class into a form suitable for the structural modelling elements. In this case, this will involve `ModelledComponentElement` receiving events from `Component`, determining what they represent and then generating new `ModelChangeEvents` and sending them to its list of observers. An example of the code related to translating the component-specific event to a Model Change Event is shown in Figure 66:

```
public class class ModelledComponentElement
      extends ModelledComponent
      implements ServiceListener {

    public void processServiceEvent(ServiceEvent e)  {
      ModelChangeEvent mce = null;

      if (e.getType() == ServiceEvent.CONNECT) {
        mce = new ModelChangeEvent(this, ModelChangeType.ADD);
      } else if (e.getType() == ServiceEvent.DISCONNECT ||
          e.getType() == ServiceEvent.FAILED) {
        mce = new ModelChangeEvent(this, ModelChangeType.REMOVE);
      }

      if (mce != null)
        fireEvent(mce);
  }
}
```

Figure 66: Snippet showing example Component to ModelledElement event translation

At this point, the existing classes have been edited (if required), and a new Modelled Element subclass has been created to wrap the existing classes to provide a consistent structural model for the observation framework. The final stage is to create a suitable Modelled Element Factory (if one does not exist), and then to add suitable instantiation logic for this new type to the factory's creation method. Typically, this logic will consist of determining the runtime type of the requested target object, and if it is a Component, then creating a Modelled Component Element. An example of the relevant logic is shown below in Figure 67.

```
public class ServiceSystemModelledElementFactory
      extends ModelledElementFactory {

    public ModelledElement createElementFor(Object object)  {
      ModelElement me = null;

      if (object is Component) {
        me = new ModelledComponentElement(object);
      } else {
        // do other type determinations /creations
      }
      return me;
  }
}
```

Figure 67: Example Modelled Element Factory snippet

In summary, the main steps involved in the modification of elements to make them compatible with the Structural Model:

- Add or identify suitable "hooks" into system-specific components or elements that require observation.
- Create wrapper class implementing Modelled Element, which acts as a dual-purpose Observer and Subject, translating component-specific events into those suitable for Modelled Element.
- Add suitable runtime-type determination and instantiation logic into appropriate Modelled Element Factory

This section has, using the requirements identified in Section 5.1.1, given a simple example of a service-based system modelled from a variety of components, and then identified the necessary decomposition of that system model in order to implement the interfaces required by the structural model.
The code snippets within this section are reasonably simple, demonstrating the single case identified within this design. It should be noted that outside of this simplified example the steps identified above will be required for each unique element type that should be modelled by the system, leading to a variety of type-specific "translator" Modelled Elements and a factory capable of creating each of them.

Appendix II Additional EET Data

This appendix contains important additional data relating to the Email Exploration Tool (EET) evaluation undertaken in Chapter 9, when it has been omitted from the evaluation for brevity.

EET v1 is a prototype software utility developed by the author as part of a separate Computer Forensics software research project with a colleague, Dr John Haggerty; detailed information regarding its Computer Forensics features and significant results of which can be found in [141]. While its primary purpose was to fulfil a set of Computer Forensic requirements; the implementation and as such, the exact features, design and implementation methods were left to the author. Therefore, this allowed the software to be developed with some of the functionality designed in accordance with the proposed framework. Unlike the Repast-based simulations, EET operates on real data, includes its own visualisation interface and runs as a standalone Java application. While it is not a large-scale distributed software system, it deals with large-scale dataset, managing emails and contacts, which may be made up of a variety of individually distributed repository files.

The required software functionality allows the user to import a set of emails into the software and then visualise the email traffic according to a variety of forensic investigation criteria. EET can import emails on a one-by-one basis, or in "repository" form; whereby files are parsed directly from the user's email client format. Partial support is included for all known email repository formats, but fully tested support in the referenced prototype is limited to Thunderbird email data files. During email loading, basic efforts are made to resolve duplicated email content and senders from duplicated or similar email aliases. The net result of the import process is a graph-type model whereby senders and recipients are connected by their joint email traffic.

The following section provides extra detail pertaining to the OBDXML support contained within EET.

II.I Additional EET Data

As discussed in Section 9.3.5, the OBDXML snippet defining the Signature was omitted. For completeness, it is shown in Figure 68, referencing the shared hardcoded Acquaintance Selection algorithm, nested within several Math and GraphFunction tags used to make the Acquaintance Nomination Metric as discussed in Section 3.3.4.

```
<tns:signatures ID="AcquaintanceSignature">
  <tns:matchingValue ID="acqVal">
   <tns:parameters ID="acqMatchParams">
    <tns:Math type="DIVIDE">
     <tns:leftHandSide>
      <tns:GraphFunction AlgorithmType="SIZE">
        <tns:ModelledElementAlgorithm ID="AcquaintanceSelection">
         <tns:parameter>SYSTEM</tns:parameter>
         <tns:parameter>0.3</tns:parameter>
        </tns:ModelledElementAlgorithm>
      </tns:GraphFunction>
     </tns:leftHandSide>
     <tns:rightHandSide>
      <tns:Math type="MULTIPLY">
       <tns:leftHandSide>
        <tns:GraphFunction AlgorithmType="SIZE">
          <tns:parameter>SYSTEM</tns:parameter>
        </tns:GraphFunction>
       </tns:leftHandSide>
       <tns:rightHandSide>0.3</tns:rightHandSide>
      </tns:Math>
     </tns:rightHandSide>
    </tns:Math>
   </tns:parameters>
  </tns:matchingValue>

  <tns:matchingBool ID="acqBool">
   <tns:Compare type="GREATER_THAN">
    <tns:leftHandSide>
     <tns:Function>acqVal</tns:Function>
    </tns:leftHandSide>
    <tns:rightHandSide>0.1</tns:rightHandSide>
   </tns:Compare>
  </tns:matchingBool>
  <tns:invalidationHandler ID="simpleHandler">
    <tns:InvalidationCriteria ID="simpleCriteria">
 <tns:invalidHandler>simpleInvalidationHandler</tns:invalidHandler>
    <tns:conditionType>MODELCHANGEEVENT</tns:conditionType>
    <tns:conditionValue>1</tns:conditionValue>
   </tns:InvalidationCriteria>
  </tns:invalidationHandler>
</tns:signatures>
```

Figure 68: Additional EET OBDXML Signature Snippet

Appendix III – Publications by the same author

1. Social Network Visualization for Forensic Investigation of E-mail
   Haggerty, J., Lamb, D., Taylor, M.
   Proceedings of the 4th IEEE Annual Workshop on Digital Forensics and Incident Analysis (WDFIA 09), pp TBC, Athens, Greece, 25 - 26 June, 2009.

2. Monitoring Autonomic Networks through Signatures of Emergence
   Lamb D., Randles, M., Taleb-Bendiab, A.
   Proceedings of the 6th IEEE Conference and Workshop on Engineering of Autonomic and Autonomous Systems pp56-65, (EASe 2009, San Francisco), 14-16 April 2009.

3. Autonomic Monitoring in Large-Scale Digital Ecosystems via Self-Organisation
   Randles, M., Lamb D., Taleb-Bendiab, A.
   Proceedings of the 2nd IEEE International Conference on Digital Eco-Systems and Technologies (DEST '08, Phitsanulok, Thailand)

4. Engineering Autonomic Systems Self-Organisation
   Randles, M., Lamb D., Taleb-Bendiab, A.
   Proceedings of the 5th IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe 2008, Belfast)

5. Cross Layer Dynamics in Self-Organising Service Oriented Architectures.
   Randles, M., Taleb-Bendiab, A., Lamb D.,
   In Self-Organising Systems (Editors: K.A. Hummel and J.P.G. Sterbenz) Springer LNCS 5343, 2008.

6. Software Engineering Concerns in Observing Networks of Autonomic Systems
   Lamb D., Randles, M., Taleb-Bendiab, A.
   Presented at SOAS 2007, (3rd International Conference on Self-Organization and Autonomous Systems in Computing and Communications), 24-27 September 2007, Leipzig, Germany. Published in System and Information Sciences Notes Journal, ISSN 1753-2310, V.2, N.1, Sept. 2007, pp101-104, (cosiwn.2007.09.149)

7. An Adaptive Observation Framework for Self-Organising Networks of Autonomic Systems
   Lamb D., Randles, M., Taleb-Bendiab, A.
   Proceedings of the 8th Annual Conference on the Convergence of Telecommunications, Networking & Broadcasting (PGNET 2007), pp296-302, Liverpool, 27-28th June 2007. ISBN: 1-9025-6016-7

8. Using Signatures of Self-Organisation for Monitoring and Influencing Large Scale Autonomic Systems
   Randles, M., Taleb-Bendiab, A., Miseldine, P., Lamb D.
   Proceedings of the 4th IEEE Workshop on Engineering of Autonomic and Autonomous Systems (EASe 2007), pp24-26, Tucson (Arizona), 26-29th March 2007.

9. Towards the Automated Engineering of Autonomic Systems
   Miseldine, P., Randles, M., Lamb D., Taleb-Bendiab, A.
   Proceedings of the 7th Annual Conference on The Convergence of Telecommunications, Networking & Broadcasting (PGNET 2006), pp393-398, Liverpool, 26-27th June 2006.

References

[1] Jeffrey C. Mogul, "Emergent (mis)behavior vs. complex software systems," *ACM SIGOPS Operating Systems Review*, vol. 40, pp. 293 - 304, 2006.

[2] Andreea Vescan and Horia F. Pop, "Automatic configuration for the component selection problem," Proceedings of 5th international conference on Soft computing as transdisciplinary science and technology, pp. 479-483, Cergy-Pontoise, France, 2008.

[3] Brandon Morel and IEEE Perry Alexander, "SPARTACAS Automating Component Reuse and Adaptation," *IEEE Transactions on Software Engineering*, vol. 30, pp. 587-600, 2004.

[4] Kent Beck and Cynthia Andres, *Extreme Programming Explained: Embrace Change*: Addison Wesley, 2004.

[5] Lee Badger, "Self-Regenerative Systems (SRS) Programme Abstract," Agency, Ed.: DARPA, 2004.

[6] Howard Shrobe, Robert Laddaga, Bob Balzer, Neil Goldman, *et al.*, "AWDRAT: A Cognitive Middleware System for Information Survivability," Proceedings of National Conference on Artificial Intelligence (AAAI-06), Boston, MA, 2006.

[7] Howard Shrobe, Robert Laddaga, Bob Balzer, and Neil Goldman, "Self-Adaptive Systems for Information Survivability: PMOP and AWDRAT," Proceedings of Self-Adaptive and Self-Organising Systems (SASO 2007), pp. 332-335, 2007.

[8] D. J. T. Sumpter, G. B. Blanchard, and D. S. Broomhead, "Ants and agents: A process algebra approach to modelling ant colony behaviour," *Bulletin of Mathematical Biology*, vol. 63, pp. 951-980, 2001.

[9] Paul Horn, "Autonomic Computing: IBM's Perspective on the State of Information Technology," IBM, 2001.

[10] Jeffrey O. Kephart and David M. Chess, "The Vision of Autonomic Computing," 2003.

[11] Gregory R. Ganger, John D. Strunk, and Andrew J. Klosterman, "Self-*Storage: Brick-based storage with automated administration," Carnegie-Mellon University, 2003.

[12] David Lorge Parnas, "Software engineering programmes are not computer science programmes," *Annals of Software Engineering*, vol. 6, pp. 19-37, 1998.

[13] J.R Speed, "What do you mean I can't call myself a Software Engineer?," *Software*, vol. 16, pp. 45-50, 1999.

[14] Ian Sommerville, *Software Engineering*, vol. 7: Addison Wesley, 2004.

[15] C. Jones, "Strategies for managing requirements creep," *Computer*, vol. 29, pp. 92-94, 1996.

[16] D Sotirovski, "Heuristics for iterative software development," *Software*, vol. 18, pp. 66-73, 2001.

[17] Kent Beck, Martin Fowler, and Jennifer Kohnke, *Planning Extreme Programming*: Addison-Wesley, 2000.

[18] Fred Brooks, *The Mythical Man-Month*: Addison-Wesley, 1995.

[19] T Mens, M Wermelinger, S Ducasse, S Demeyer, *et al.*, "Challenges in Software Evolution," Proceedings of Principles of Software Evolution, pp. 13-22, 2005.

[20] R.J.K. Jacob and J.N. Froscher, "A software engineering methodology for rule-based systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, pp. 173-189, 1990.

[21] P.K. McKinley, S.M. Sadjadi, E.P.Kasten, and B.H.C.Cheng, "Composing Adaptive Software," in *IEEE Computer*, vol. 37, 2004, pp. 56-64.

[22] Michael Wooldridge, *An Introduction to MultiAgent Systems*. Chichester: John Wiley & Sons, 2002.

[23] M. Sloman, "Policy Driven Management for Distributed Systems," *Network and Systems Management*, vol. 2, pp. 333-360, 1994.

[24] Nicholas R Jennings, "An agent-based approach for building complex software systems," *Communications of the ACM*, vol. 44, pp. 35-41, 2001.

[25] Franco Zambonelli and Andrea Omicini, "Challenges and Research Directions in Agent-Oriented Software Engineering," *Autonomous Agents and Multi-Agent Systems*, vol. 9, pp. 253-283, 2004.

[26] Melanie Mitchell, "Complex systems: Network thinking," *Artificial Intelligence*, vol. 170, pp. 1194-1212, 2006.

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable OO Software*: Addison-Wesley, 1995.

[28] IBM, "IBM Research - Autonomic Computing," IBM, 2001-2006.

[29] Julie A. McCann and Markus C. Huebscher, "Evaluation Issues in Autonomic Computing," in *Grid and Cooperative Computing – GCC 2004*, vol. 3252, *Lecture Notes in Computer Science*: Springer Berlin / Heidelberg, 2004, pp. 597-608.

[30] M. E. J. Newman, S. H. Strogatz, and D. J. Watts, "Random graphs with arbitrary degree distributions and their applications," *Physical Review E*, vol. 64, pp. 026118, 2001.

[31] Jon M Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, *et al.*, "The Web as a Graph: Measurements, Models, and Methods," Proceedings of International Conference on Combinatorics and Computing, pp. 1-17, 1999.

[32] P. Erdos and A. Renyi, "On the evolution of random graphs," *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, vol. 5, pp. 17-61, 1960.

[33] Murray Gell-Mann, *The Quark and the Jaguar: Adventures in the Simple and the Complex*: Abacus, 1994.

[34] M. Mitchell Waldrop, *Complexity: The Emerging Science at the Edge of Order and Chaos*: Pocket Books, 1993.

[35] J.A. Zachman, "A Framework for Information Systems Architecture," *IBM Systems Journal*, vol. 26, pp. 276-292, 1987.

[36] Stafford Beer, "The Viable System Model: Its Provenance, Development, Methodology and Pathology," *The Journal of the Operational Research Society*, vol. 35, pp. 7-25, 1984.

[37] Bela Bollobas and Oliver Riordan, "Robustness and Vulnerability of Scale-Free Random Graphs," *Internet Mathematics*, vol. 1, pp. 1-35, 2003.

[38] Duncan J. Watts, "Networks, Dynamics and the Small-World Phenomenon," *American Journal of Sociology*, vol. 105, pp. 493-527, 1999.

[39] Reinhard Diestel, *Graph Theory*, vol. 173, 3rd ed. Heidelberg: Springer-Verlag, 2005.

[40] Steven Berlin Johnson, *Emergence: Connected Lives of Ants, Brains, Cities and Software*. New York: Scribner, 2001.

[41] Kurt Bittner and Ian Spence, *Managing Iterative Software Development Projects*: Addison Wesley, 2006.

[42] Craig Larman, *Agile and Iterative Development*: Addison Wesley, 2003.

[43] Reuven Cohen, Shlomo Havlin, and Daniel ben-Avraham, "Efficient Immunization Strategies for Computer Networks and Populations," *Physical Review Letters*, vol. 91, 2003.

[44] HA Simon, *The Sciences of the Artificial*: MIT Press, 1996.

[45] David S Rosenblum and Alexander L Wolf, "A Design Framework for Internet-scale Event Observation and Notification," *SIGSOFT Software Engineering Notes*, vol. 22, 1997.

[46] Nigel Goldenfeld and Leo P Kadanoff, "Simple Lessons from Complexity," in *Science*, vol. 284, 1999, pp. 87-89.

[47] Martin Randles, Hong Zhu, and A. Taleb-Bendiab, "A Formal Approach to the Engineering of Emergence and its Recurrence," Proceedings of Engineering Emergence in Decentralised Autonomic Systems, 2007.

[48] C. Mohan, "A Survey of DBMS Research Issues in Supporting Very Large Tables," *Lecture Notes In Computer Science*, vol. 730, pp. 279-300, 1993.

[49] Andrew Johnson and Farshad Fotouhi, "Adaptive Indexing in Very Large Databases," *Journal of Database Management*, vol. 1995, 1996.

[50] V Ganti, J Gehrke, and R Ramakrishnan, "Mining very large databases," in *IEEE Computer*, vol. 32, 1999, pp. 38-45.

[51] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel, "The X-tree: An Index Structure for High-Dimensional Data," Proceedings of 22th International Conference on Very Large Databases pp. 28 - 39, 1996.

[52] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, *et al.*, "SETI@home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, pp. 56-61, 2002.

[53] Celso L. Mendes and Daniel A. Reed, "Monitoring Large Systems Via Statistical Sampling," *International Journal of High Performance Computing Applications*, vol. 18, pp. 267-277, 2004.

[54] Mark W. Maier, "Architecting Principles for Systems-of-Systems," *Systems Engineering*, vol. 1, pp. 267-284, 1999.

[55] M. Turner, D. Budgen, and P. Brereton, "Turning software into a service," in *Computer*, vol. 36, 2003, pp. 38-44.

[56] F. Heylighen, "Self-organization, emergence and the architecture of complexity," Proceedings of 1st European Conference on System Science, pp. 23–32, Paris, 1989.

[57] Albert-Laszlo Barabasi, Reka Albert, and Hawoong Jeong, "Scale-free characteristics of random networks: the topology of the world-wide web," *Physica A: Statistical Mechanics and its Applications*, vol. 281, pp. 69-77, 2000.

[58] John Toner and Yuhai Tu, "Flocks, herds, and schools: A quantitative theory of flocking," *Physical Review E*, vol. 58, pp. 4828-4858, 1998.

[59] Guy Theraulaz, Jacques Gautrais, Scott Camazine, and Jean-Lous Deneubourg, "The formation of spatial patterns in social insects: from simple behaviours to complex structures," *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, vol. 361, pp. 1263-1282, 2003.

[60] Julia K. Parrish, Steven V. Viscido, and Daniel Grünbaum, "Self-Organized Fish Schools: An Examination of Emergent Properties," *Biological Bulletin*, vol. 202, pp. 296-305, 2002.

[61] Frederick R. Adler and Deborah M. Gordon, "Information Collection and Spread by Networks of Patrolling Ants," *The American Naturalist*, vol. 140, pp. 373-400, 1992.

[62] Hartmut Schmeck, "Organic Computing – A New Vision for Distributed Embedded Systems," Proceedings of Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005), pp. 201-203, Seattle, Washington, 2005.

[63] Urban Richter, Moez Mnif, J¨urgen Branke, Christian M¨uller-Schloer, *et al.*, "Towards a generic observer/controller architecture for Organic Computing," Proceedings of INFORMATIK 2006 – Informatik f¨ur Menschen!, pp. 112-119, 2006.

[64] Huaglory Tianfield, "Some Reflections on Intelligent Control," *Artificial Intelligence Review*, vol. 23, pp. 57-91, 2005.

[65] David Lamb, Martin Randles, and A. Taleb-Bendiab, "Monitoring Autonomic Networks through Signatures of Emergence," Proceedings of Engineering of Autonomic and Autonomous Systems, EASe 2009, pp. 56-65, San Francisco, CA, 2009.

[66] S. Milgram, "The small world problem," *Psychology Today*, vol. 2, pp. 60-67, 1967.

[67] Reynold Cheng, Yuni Xia, Sunil Prabhakar, and Rahul Shah, "Change Tolerant Indexing for Constantly Evolving Data," Proceedings of International Conference on Data Engineering (ICDE) 2005, pp. 391-402, 2005.

[68] Jeff Magee and Jeff Kramer, "Dynamic Structure in Software Architectures," *ACM SIGSOFT Software Engineering Notes*, vol. 21, pp. 3-14, 1996.

[69] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor, "Architecture-based runtime software evolution," Proceedings of International Conference on Software Engineering, pp. 177-186, Kyoto, Japan, 1998.

[70] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, James Whitehead Jr, *et al.*, "A Component- and Message-Based Architectural Style for GUI Software," *IEEE Transactions on Software Engineering*, vol. 22, pp. 390-406, 1996.

[71] John C Georgas, Andre van der Hoek, and Richard N. Taylor, "Architectural runtime configuration management in support of depedendable self-adaptive software," Proceedings of International Conference on Software Engineering (ICSE), Workshop on Architecting Dependable Systems (WADS), pp. 1-6, 2005.

[72] Linda Northrop, P.H. Feiler, B. Pollak, and D. Pipitone, *Ultra-large-scale systems: The software challenge of the future*: Software Engineering Institute, Carnegie Mellon University, 2006.

[73] M.N. Huhns and M.P. Singh, "Service-oriented computing: key concepts and principles," in *IEEE Internet Computing*, vol. 9, 2005, pp. 75-81.

[74] J. Branke, M. Mnif, C. Muller-Schloer, and H. Prothmann, "Organic Computing – Addressing Complexity by Controlled Self-Organization," Proceedings of 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, pp. 185-191, 2006.

[75] Emre Cakar, Jorg Hahner, and Christian Muller-Schloer, "Creating Collaboration Patterns in Multi-Agent Systems with Generic

Observer/Controller Architectures," Proceedings of Autonomics, Turin, Italy, 2008.

[76] T. DeMarco and T. Lister, "Software Development: State of the Art vs. State of the the Practice," Proceedings of 11th International Conference on Software Engineering, pp. 271-275, 1989.

[77] Ivica Crnkovic, "Component-based Software Engineering - new challenges in software development," Proceedings of 25th Itnl Conference on Information Technology Interfaces (ITI2003), pp. 9-18, 2003.

[78] S.C. Chu, "From Component-based to Service-Oriented Software Architecture for Healthcare," Proceedings of 7th International Workshop on Enterprise Networking and Computing in Healthcare (HEALTHCOM 2005), pp. 96-100, 2005.

[79] J. Kral, M Zemlicka, and M Kopecky, "Software Confederations - An Architecture for Agile Development in the Large," Proceedings of International Conference on Software Engineering Advances (ICSEA 06), pp. 39-44, 2006.

[80] Pin Chen and Jennie Clotheir, "Advancing Systems Engineering for Systems-of-Systems Challenges," *Systems Engineering*, vol. 6, pp. 170-183, 2003.

[81] L Duboc, E Letier, D. S. Rosenblum, and T. Wicks, "A Case Study in Eliciting Scalability Requirements," Proceedings of 16th IEEE International Conference on Requirements Engineering (RE08), pp. 247-252, Catalunya, Spain, 2008.

[82] Richard P. Gabriel, Linda Northrop, Douglas C. Schmidt, and Kevin Sullivan, "Ultra-large-scale systems," Proceedings of Dynamic Languages Symposium; 21st ACM SIGPLAN: Object Oriented Programming Systems, Languages and Applications, pp. 632-634, Portland, Oregon, 2006.

[83] Christian Müller-Schloer and Bernhard Sick, "Emergence in Organic Computing Systems: Discussion of a Controversial Concept," in *Lecture Notes In Computer Science*, vol. 4158/2006, *Autonomic and Trusted Computing*, 2006, pp. 1-16.

[84] Urban Richter, Moez Mnif, J¨urgen Branke, Christian M¨uller-Schloer, *et al.*, "Towards a generic observer/controller architecture for Organic Computing," Proceedings of INFORMATIK 2006 – Informatik f¨ur Menschen!, Lecture Notes in Informatics (LNI), pp. 112-119, 2006.

[85] Uwe Aicklen and Steve Cayzer, "The Danger Theory and its Application to Artificial Immune Systems," HP Labs, 2002.

[86] Jie Wang, Jian Cao, J.O. Leckie, and ShenSheng Zhang, "Managing e-government IT infrastructure: an approach combining autonomic computing and awareness based collaboration," Proceedings of Computer and Information Technology - CIT'04, pp. 998-1003, 2004.

[87] Jeffrey O. Kephart, "Research Challenges of Autonomic Computing," Proceedings of International Conference on Software Engineering (ICSE), pp. 15-22, 2005.

[88] IBM, "Autonomic Computing Toolkit Overview," in *IBM DeveloperWorks - Autonomic Computing*, Accessed May 2009.

[89] David Bridgewater, "Standardize messages with the Common Base Event model," in *IBM DeveloperWorks*, 2006.

[90] Bart Jacob, Richard Lanyon-Hogg, Devaprasad K Nadgir, and Amr F Yassin, "A Practical Guide to the IBM Autonomic Computing Toolkit," vol. 2009: IBM Redbooks, 2004.

[91] R Desmarais and H Muller, "A Proposal for an Autonomic Grid Management System," Proceedings of Software Engineering for Adaptive and Self-Managing Systems - SEAMS '07, pp. 1-11, 2007.

[92] Foundation for Intelligent Physical Agents - FIPA, "FIPA ACL Message Structure Specification," 2002.

[93] Michael E. Bratman, *Intention, Plans, and Practical Reason.* Stanford, California: Stanford University, 1987.

[94] Martin Randles, A Taleb-Bendiab, Philip Miseldine, and Andy Laws, "Adjustable Deliberation of Self-Managing Systems," in *EASe 2005.* Greenbelt, Maryland, USA: Liverpool JMU, 2005.

[95] Hong Jiang, Jose M. Vidal, and Michael N. Huhns, "EBDI: an architecture for emotional agents," Proceedings of International Joint Conference on Autonomous Agents and Multiagent Systems, Honolulu, Hawaii, 2007.

[96] Norbert Glaser, *Conceptual Modelling of Multi-agent Systems: The CoMoMAS Engineering Environment (Multiagent Systems, Artificial Societies, and Simulated Organizations)* Springer, 2002.

[97] Dylan Dawson, Ron Desmarais, Holger M. Kienle, and Hausi A. Muller, "Monitoring in Adaptive Systems using Reflection," Proceedings of International Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 81-88, Leipzig, Germany, 2008.

[98] Patrick Eugster, "Uniform proxies for Java," Proceedings of 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, pp. 139-152, Portland, Oregon, 2006.

[99] Matthew L. Massie, Brent N Chun, and David E Culler, "The Ganglia Distributed Monitoring System: Design, implementation and experience," *Parallel Computing*, vol. 30, pp. 817-840, 2004.

[100] Andres J. Ramirez and Betty H.C. Cheng, "Design patterns for monitoring adaptive ULS systems," Proceedings of 2nd International Workshop on Ultra-large-scale software-intensive systems in International Conference on Software Engineering (ICSE '08), pp. 69-72, Leipzig, Germany, 2008.

[101] Brian Hayes, "Graph Theory in Practice: Part 2," *American Scientist*, vol. 88, pp. 104-109, 2000.

[102] Bela Bollobas, *Random Graphs*, vol. 73. Cambridge: Cambridge University Press, 2001.

[103] Nisha Mathias and Venkatesh Gopal, "Small worlds: How and why," *Physical Review E*, vol. 63, pp. 12, 2001.

[104] Lun Li, David Alderson, John C. Doyle, and Walter Willinger, "Towards a Theory of Scale-Free Graphs: Definition, Properties, and Implications," *Internet Mathematics*, vol. 2, pp. 431–523, 2005.

[105] Daniel Le Metayer, "Describing Software Architecture Styles Using Graph Grammars," *IEEE Transactions on Software Engineering*, vol. 24, pp. 521-533, 1998.

[106] Dan Hirsch, Paola Inverardi, and Ugo Montanari, "Graph Grammars and Constraint Solving for Software Architectural Styles," Proceedings of 3rd ACM International Software Architecture Workshop - ISAW'98, Orlando, FL, 1998.

[107] Robert Bruni, Antonio Bucchiarone, Stefania Gnesi, and Hernan Melgratti, "Modelling Dynamic Software Architectures using Typed Graph Grammars," *Electronic Notes in Theoretical Computer Science*, vol. 213, pp. 39-53, 2008.

[108] Tom Mens, Gabriele Taentzer, and Olga Runge, "Analysing Refactoring Dependencies using Graph Transformation," *Software and Systems Modelling*, vol. 6, pp. 269-285, 2007.

[109] Albert-Laszlo Barabasi and Reka Albert, "Emergence of Scaling in Random Networks," *Science*, vol. 286, pp. 509-512, 1999.

[110] Lian Wen, Diana Kirk, and R.G.Dromey, "Software systems as complex networks," Proceedings of IEEE International Conference in Cognitive Informatics, pp. 106-115, 2007.

[111] Lian Wen, R.G.Dromey, and Diana Kirk, "Software Engineering and Scale-Free Networks," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 39, pp. 648-657, 2009.

[112] Yuumi Kawachi, Kenta Murata, Shinichiro Yoshii, and Yukinori Kakazu, "The structural phase transition among fixed cardinal networks," Proceedings of Complex2004, pp. 247-255, Cairns, Australia, 2004.

[113] Duncan J. Watts and S. H. Strogatz, "Collective dynamics of small-world networks," *Nature*, vol. 393, pp. 440-442, 1998.

[114] David Lamb, Martin Randles, and A. Taleb-Bendiab, "Software Engineering Concerns in Observing Networks of Autonomic Systems," *System and Information Sciences Notes Journal*, vol. 2, pp. 101-104, 2007.

[115] David Lamb, Martin Randles, and A. Taleb-Bendiab, "An Adaptive Observation Framework for Self-Organising Networks of Autonomic Systems," Proceedings of PGNET 2007, pp. 296-302, Liverpool, 2007.

[116] Ernest Friedman, *Jess in action: rule-based systems in java*: Manning Publications Co, 2003.

[117] Tian He, John A. Stankovic, Michael Marley, Chenyang Lu, *et al.*, "Feedback control-based dynamic resource management in distributed real-time systems," *Journal of Systems and Software*, vol. 80, pp. 997-1004, 2006.

[118] David Lamb, Martin Randles, and A. Taleb-Bendiab, "Software Engineering for Large-Scale Observation Systems," in *DASEL Technical Reports/07/07/DL01*, 2007.

[119] B Eckel, *Thinking in Java*. Englewood Cliffs, New Jersey: Prentice-Hall, 1998.

[120] W. Mepham and S. Gardner, "A software framework for translating ECA sequences from OWL-DL into Java," Proceedings of Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT '08, pp. 540-543, 2008.

[121] Adrian Paschke, "ECA-RuleML/ECA-LP: A Homogeneous Event-Condition-Action Logic Programming Language," Proceedings of International Conference of Rule Markup Languages (RuleML '06), Athens, GA, 2006.

[122] J. Gosling, "The feel of Java," *Computer*, vol. 30, pp. 53-57, 1997.

[123] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg, "A Comparison of String Metrics for Matching Names and Records," Proceedings of KDD Workshop on Data Cleaning and Object Consolidation, 2003.

[124] W. Clay Richardson, Donald Avondolio, Scot Schrager, Mark W. Mitchell, *et al.*, "Boxing and Unboxing Conversions," in *Professional Java JDK 6 Edition*: Wiley Publishing, 2007, pp. 19-21.

[125] Angela Nicoara, Gustavo Alonso, and Timothy Roscoe, "Controlled, systematic, and efficient code replacement for running java programs," Proceedings of 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, pp. 233-246, 2008.

[126] Sun, "Java Hotswap API."

[127] Dong Kwan Kim and Eli Tilevich, "Overcoming JVM HotSwap constraints via binary rewriting," Proceedings of International Workshop On Hot Topics In Software Upgrades, pp. 1-5, Nashville, Tennessee, 2008.

[128] Algis Rudys and Dan S. Wallach, "Enforcing Java Run-Time Properties Using Bytecode Rewriting," in *Lecture Notes in Computer Science*: Springer Berlin / Heidelberg, 2003, pp. 271-276.

[129] Eelco Visser, "Meta-programming with Concrete Object Syntax," in *Generative Programming and Component Engineering, Lecture Notes in Computer Science*: Springer Berlin, 2002, pp. 299-315.

[130] Jonathan M. Sobel and Daniel P. Friedman, "An Introduction to Reflection-Oriented Programming " Proceedings of Reflection 96, 1996.

[131] N Feng, G Ao, T White, and B Pagurek, "Dynamic evolution of network management software by softwarehot-swapping," Proceedings of 2001 IEEE/IFIP International Symposium on Integrated Network Management, 2001.

[132] Shigeru Chiba, Yoshiki Sato, and Michiaki Tatsubori, "Using HotSwap for Implementing Dynamic AOP Systems," Proceedings of 1st Workshop on Advancing the State-of-the-Art in Run-time Inspection, 2003.

[133] Jonathan Appavoo, Kevin Hui, Craig Soules, and Robert Wisniewski, "Enabling autonomic behavior in systems software with hot swapping," *IBM Systems Journal*, vol. 42, 2003.

[134] RG Taylor, *Models of computation and formal languages*: Oxford University Press, 1998.

[135] Ed Ort and Bhakti Mehta, "Java Architecture for XML Binding (JAXB)," in *Sun Java Technical Articles*, 2003.

[136] James Clark and Steve DeRose, "XML Path Language (XPath) Version 1.0," in *W3C Recommendations*, vol. 2009: W3C, 1999.

[137] Joerg Schaible, "XStream Tutorial," 2008.

[138] Wolfgang Laun, "A JAXB Tutorial," in *Glassfish Tutorials*, 2009.

[139] Open Source / SourceForge, "Repast API Documentation," 2005.

[140] Nancy G. Leveson, "A Systems-Theoretic Approachto Safetyin Software-Intensive Systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 66-86, 2004.

[141] John Haggerty, David Lamb, and Mark Taylor, "Social Network Visualisation for Forensic Investigation of E-mail," Proceedings of 4th IEEE Conference on Digital Forensics and Incident Analysis (WDFIA 09), Athens, Greece, 2009.

[142] Martin Randles, A Taleb-Bendiab, and Philip Miseldine, "Using Stochastic Situation Calculus to Formalise Danger Signals for Autonomic Computing," in *PGNET2005*. Liverpool John Moores University: LJMU, 2005.

[143] Alfred Tauber, "The Biological Notion of Self and Non-self," in *The Stanford Encyclopedia of Philosophy*, Zalta, Ed., 2002.

[144] Markos Markou and Sameer Singh, "Novelty Detection: a review - part 1: statistical approaches," *Signal Processing*, vol. 83, pp. 2481-2497, 2003.

[145] P Matzinger, "Tolerance, Danger, and the Extended Family," *Annual Review of Immunology*, vol. 12, 1994.

[146] IBM, "Autonomic Computing Toolkit: A Developer's Guide," 2006.

[147] J Bosch, "Design Patterns as Language Constructs," *JOURNAL OF OBJECT ORIENTED PROGRAMMING (JOOP)*, vol. 11, pp. 18-32, 1998.