Building Systems out of Available Components: Quality and Adaptation Considerations

Tony Gillan (B.InfTech)

6190CIT: Honours Research Project
School of Computing and Information Technology
Griffith University
Nathan, Queensland, 4111, Australia
tony.gillan@student.gu.edu.au
tony@gridloq.com

Submitted in partial fulfillment of the requirements of the degree of:
Bachelor of Information Technology with Honours

November 10, 2002

Abstract

Functional requirements are generally easy to specify, analyse, implement, and verify, compared to quality attributes. There is a well-defined process for taking these requirements through to a finished software product.

Quality attributes, however, have been neglected in many standard software engineering processes. This is primarily because they are hard to quantify, hard to keep track of, and tend to refer to the behaviour of the system as a whole rather than individual functionality.

Component-based engineering requires a system integrator to choose functionality from a number of available components, and meld them together to produce to a complete system. The integrator cannot know what quality attributes are built into each component unless that information is provided by the component developer. Hence the integrator cannot specify in detail and with confidence the quality attributes required in specific components.

By using Behavior Tree$^{\text{TM}}$ architecture descriptions, a direct translation and integration of functional requirements can be made consistently and completely. The architecture can then be analysed for, and adapted to meet, individual quality requirements.

An architecture description, in its XML form, can be provided along with the actual component implementation, allowing a software integrator to determine if a black-box component implements functional and quality requirements.

0.1 Statement of Originality

0.2 Acknowledgements

I would like to thank my supervisors; Senior Lecturer Terry Rout and Professor Geoff Dromey, for their guidance, encouragement, and critical perspectives on this topic. My fellow academics within the Software Quality Institute are also to be thanked for their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This dissertation focuses on aspects of adaptation, and reuse of existing software components. A software component is generally a stand-alone software product with specified interfaces and quality attributes, designed to be integrated into an overall system. At one stage, it has undergone the software development process including requirements, design, implementation, testing, and packaging. The major issue for re-using this existing software is how to integrate it and other components into in a new system. A developer must take into account efficiency, performance, reliability, and safety of the overall system. These are quality attributes.

A detailed, complete set of functional requirements will enable a software development team to create a quality product. However this system can still fail to deliver the user's unstated useability, availability, and performance expectations. The requirements elicitation process generally focuses on deriving the local behaviour of different parts of the system. Quality attributes focus on how well the system performs.

Over the last few decades, a number of researchers and practitioners have examined how systems achieve quality attributes. It seems that no one has systematically and completely documented the relationship between software architecture and quality attributes. Yet there is a widely held premise that architecture determines quality attributes, and very little needs to be known about the functionality of the system in order to analyse the system's quality attributes.

An architectural design is normally derived from a set of requirements as a part of the software development process. It also is the least structured of all processes. Most of this work is left up to the skill, experience, and intuition of the designer, rather than a formalised methodology. This is an obvious process where the quality of the system can suffer.

A part of the problem is that generally a complete set of functional requirements is available for what a component does, but there is only vague information about how the component performs overall, and how it will behave as a part of a new system with different requirements and a different environment.

The software architecture description using Behavior Trees is the first high-level design incorporating the decisions and trade-offs made during the software development process. Decisions made at these early stages have far-reaching consequences, in terms of cost, and quality. Fixing an error found during the requirements phase costs much less than fixing the same error found during testing.

As a result of the identification of the dependent relationship between quality requirements and architecture, and the lack of formalised processes between these domains, these areas will become the focus of this dissertation.

A combination of current requirements, quality standards, and industry best practice will be used to derive a model for the specification of quality attributes. To provide a direct association between functional requirements and an architectural description, the Genetic Software Engineering approach will be used. This approach provides a clear methodology for specifying behaviour for individual requirements, and then composes them into the behaviour for the overall system. This allows for the implementation-independent validation of the requirements. The resulting Behavior Tree descriptions will be used as the basis of comparing similar architectures, identifying quality attributes, encapsulating, and transforming an existing architecture to meet changing quality attribute needs.

Overall the results of this dissertation will show that most quality attributes can be identified and changed within the software architecture. As the software architecture design is independent of the implementation, but still embodies all of the functional and non-functional requirements, it is information that should be included along with the software component. This can allow detailed investigation and validation by the integrator, as a part of the component-based software development process.

Chapter 2

Topic Introduction

This chapter provides a brief overview to the domains of software architecture and requirements. In terms of the software development process, an initial set of customer business needs are translated into detailed software functional and quality requirements. These requirements are then used to create an architectural description for the proposed implementation of the system.

By limiting the scope of this dissertation to these high-level domains, the transformation of these requirements through to an architecture can be traced and analysed, and propriety implementation issues can be avoided.

2.1 Software Requirements

“The path to quality software begins with excellent requirements.” [59, Wiegers(1996)] Not handling the process of requirements development and management is one the most common causes of software project problems.

Requirements have different, valid definitions, depending on your role in the product development process. Customer-provided requirements are really solution ideas, like a high-level product concept or business vision. Developer requirements are often detailed user-interface designs. End-users may provide their requirements, but these may not be detailed or complete enough for the developer to actually implement that functionality. Then there are issues with ambiguous or conflicting requirements, and quality expectations.

Requirements, as specified in a Software Requirements Specification, are generally categorised into two areas; functional and non-functional requirements. Functional requirements describe the basic “concrete” behaviour of individual elements of the system. Non-functional requirements, including quality attributes tend to describe the overall high-level behaviour of the system in terms of safety, security, performance, availability, portability, etc.

For example, in terms of availability, if a client has commissioned an online banking system, they require a certain level of availability; customers will get irate if the system is unavailable from 12pm to 6pm on Friday due to maintenance. Customers want to know that their money, and hence the system, is secure from external and internal violation.

These requirements differ from the specific functionality of the system such as balance checking, money transfer, address updates, and payments that a banking system exhibits. Some of these features can be added on to existing systems, and not effect the existing functionality.

Unfortunately, non-functional requirements, such as “safety” or “security” cannot be added separately to the system once developed. Every code module developed and integrated must take security into account, as it is being written and tested.

The overall security of the system is directly related to the most security-vulnerable part of that system – “the weakest link in a chain”.

Non-functional requirements may not necessarily be localised to one part of the architecture or code, as functional requirements can be. They relate to the overall behaviour of the system.

Overall both functional and non-functional requirements are as important as each other, although handled differently. The system will be developed entirely from them. Hence the success of the software is intrinsically related to how well they are interpreted and implemented.

Sections 4.1 and 4.2 will describe the current practices and standards within the software industry for categorising and using functional and non-functional requirements within the development lifecycle.

2.2 Software Architecture

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them. [7, Bass *et al.*(1998)]

A software architecture is widely recognised as the first key artifact of software engineering. A systems software architecture is the key to communicating and comprehending that system. [4, Bachmann *et al.*(2002)]

From [12, Bosch(2000)]; although software systems have had architectures since the early days of computers, the importance of explicit specification, analysis, and design of architectures has only recently been recognised. Traditional best practice dictated that engineers just add quality to their system as they were implementing the code.

Transforming the original functional and quality requirements into an architecture is the most complex, and least supported activity of the application development process. Detailed design and implementation are better understood, more methodical, and have more technological support available to the software engineer.

An explicit software architecture description allow quality attributes to be designed and assessed, and trade-offs made. This allows an early evaluation of the requirements for the proposed system between the developer and the client.

Extensive work is currently in progress at the Software Engineering Institute developing methods for analysing and designing software architectures, taking quality attributes into consideration. These methods include the Architecture Tradeoff Analysis Method SM (ATAM SM), the Quality Attribute Workshop (QAW), and the Attribute-Driven Design (ADD) method. While these methods efficiently manage the use of quality-attribute knowledge, they don't formalise that knowledge. To properly use those methods, analysts and designers must be armed with quality-attribute knowledge and experience. [4, Bachmann *et al.*(2002)]

Chapter 3

Problem Statement

In continuation from the topic overview presented in Section 2, this section defines and discusses the problem that the dissertation undertakes to address.

3.1 The Problem

Given a statement of quality requirements for a system, how can these be allocated and defined for the identified components?

Assuming that an existing software component meets it's functional requirement specifications, the following breakdown of the problem has been made:

1. How to define, specify, and measure quality attributes within software components?
2. How do quality attributes of individual software components affect the quality of the overall integrated system?
3. How to adapt an existing software component for use within a new system with different quality attributes?

These specific issues will be addressed in Section 5 and results summarised in Section 6.1

3.2 Justification

This section will justify why the problem has not been fully answered as yet.

Using existing software components within new software projects is becoming essential for a developer to meet minimum functionality and time-to-market requirements of the marketplace. The primary issue for the reuse of software is how to evaluate, select, and integrate compatible components. "Although it has been clear for some time that integration is a key to software reuse, little has been done to quantify the integrability of components" [34, Seacord *et al.*(2000)]

The probable reason for this is explained by [12, Bosch(2000), Chapter 1, p6]. A "naive approach to software reuse, takes components, plugs them together, like Lego blocks, and produces a useful system. This may be a simple concept, but does not work in practice for real complex systems."

A planned approach is recommended which requires that the organisation develop reusable assets with the right abstraction and variability levels for the software product devised and evolved. Assets should be developed with predefined interfaces, as parts fitting into a higher-level structure, rather than suitable components being searched for from a collection of developed and mined assets. This seems at odds with current industry practice.

The high-level architectural description of the component, is the best place work with its quality requirements and functionality. From [9, Bergey *et al.*(2002)], the software architecture description is the first high-level design incorporating the decisions and trade-offs made during the software development process. Decisions made at these early stages have far-reaching consequences, in terms of cost, and quality.

To fix an error during requirements or early design phases costs order of magnitudes less than fixing the same error found during testing. [10, Boehm(1981)].

The importance of the architecture within the software development process has been explained, but this doesn't automatically mean clear and consistent methodologies are available now. From [12, Bosch(2000)], although software systems have had architectures since the early days of computers, the importance of explicit specification, analysis, and design of architectures has only recently been recognised.

It is important to any software industry project that not just the functional requirements are defined within the architecture. Traditionally, quality requirements are left up to the programmer to implement as part of their development process. Quality requirements relate more to the overall behaviour of the system, rather than individual parts or functionality. Hence they tend to be hard to identify, and directly implement.

Problems of linking the relationship between quality attributes and architecture descriptions stem from the lack of independent, precise definitions, patterns, and analysis techniques for all attributes. [8, Bass *et al.*(2000)] Availability is an attribute, but it is also a sub-characteristic of security (denial of service attacks can limit availability). Hundreds of patterns at different levels of granularity, do not lend themselves to analysis, categorisation or reuse.

"Achieving an architecture in a reliable, repeatable manner from a precise statement of the quality requirements is a key open research area." [7, Bass *et al.*(1998) Chapter 19.3]

3.3 Discussion

This section undertakes to describe why it is worthwhile for the problem to be answered.

There are number of issues within the software industry today, including;

- the high cost of maintenance,
- the lack of software reliability or high cost to achieve reliability, and
- the failure to deliver products on-time and within-budget.

[12, Bosch(2000), Chapter 1, pp4-6] Any work that addresses these issues will provide direct benefits to the industry as a whole. Reuse of existing software through componentisation [31, McIlroy(1969)] has long been recognised as the most promising approach to deal with these issues. "If we are able to develop systems from existing components, development cost decreases, quality increases because the components have been tested in other contexts, time-to-market is shortened and maintenance cost is decreased because changes to the components benefit multiple systems." [12, Bosch(2000), Chapter 1, p6]

[9, Bergey *et al.*(2002)] states that decisions made at the early stages of software development have far-reaching consequences, in terms of cost, and quality. Time should be spent dealing with functionality and quality requirements issues at the software architecture level, rather than at the costly implementation phase.

There are several benefits to fully defining the relationship between quality attributes and architecture:

1. A standard methodology for implementing functional and quality requirements within the software architecture can be defined and refined. This will allow third-party analysis of the component's behaviour without having to deal with binary code.

2. A standard approach for identifying and specifying quality attributes within software components can be developed. This will allow third-party testing and certification of black-box components to be made.

3. Techniques for reconfiguring or transforming an architecture to provide specified levels of quality attributes, can be developed, allowing reuse of existing components within new application areas.

A component integrator needs to now how a component will affect their system, and whether it will do what it is supposed to. Just supplying the interface specification, with now quality attribute details, is inadequate. This is why most reuse is currently limited to component assets internal to the organisation. Integrators at least have access to source code, and possibly the development team who created it.

Overall, by spending more time dealing directly with the issues of quality attributes and software architecture development, reuse of existing software components by third-party organisations will become more of a reality.

Chapter 4

The State of the Art

In this section a detailed research summary will be given in the seven main areas of:

- Section 4.1 Software Requirements
- Section 4.2 Software Quality Attributes
- Section 4.3 Software Engineering Measurement
- Section 4.4 Process Quality Evaluation
- Section 4.5 Software Component Architecture
- Section 4.6 Architecture Description
- Section 4.7 Component Certification and Selection

This summary is intended to present the main findings, issues, and current best practice for quality attribute specification and component-based software engineering.

4.1 Software Requirements

4.1.1 Functional and Nonfunctional Requirements

The set of all software requirements can be broken down into functional, and its (supposed) antonym; non-functional requirements. [7, Bass *et al.*(1998), ch4, pp76-77] states that functional requirements usually define the behaviour of the system, or the work it is supposed to produce. All other qualities of the system such as performance and maintainability are designated in the non-functional category.

[7, Bass *et al.*(1998)] go further to state that all requirements cannot be cleanly separated into these two categories. All qualities are inescapably bound up in the behaviour of the system. For example, with security, a valid password where expected by the system, allows transition to a state where a set of privileged commands is allowed. Performance dictates that the system will produce a valid response to a command within a set time-frame. Fault tolerance allows system transition to a degraded state for remedial action, when a resource has disappeared.

As such, non-functional requirements cannot be built into the system after the functional requirements have been implemented. They should be an integral part of developing the behaviour of the system.

4.1.2 Software Quality

[56, Voas(1999)] addresses why the research community and software industry has abandoned the search for revolutionary advances in building and assuring software quality. The paper lists the eight major milestones in software quality and the myths associated with each.

1. Process Improvement/Maturity – assumes that if a software development organisation receives a high rating, the software produced by that organisation is as good as that organisation. i. e. organisational process maturity is equivalent to software product quality. *This is the opinion of [56, Voas(1999)]*

only, and is an inaccurate representation of the Process Maturity movement's claims.

2. Formal Methods – formal methods are the "process improvement" answer to any and all security, reliability, and safety problems, by removing ambiguities, inconsistencies, and logically incorrect behaviour. Unfortunately they are hard to implement, expensive, and not foolproof.

3. Languages and OOD – using the current language or design paradigm will solve the problems that older languages could not. This creates complexity with information hiding, threads, inheritance, etc., and makes the software more complex, harder to test, and difficult to produce quality systems.

4. Metrics and Measurement – numerical information about the development process and code will reveal whether the code will do what it was intended to do, not how it does it – semantics. Code metrics are better at assessing the quality of the development process rather than the quality of the code. Metrics are an indirect way of quantifying qualities such as testability and maintainability. i. e. programs with one line of code are more maintainable than million line programs. Metrics are guides only; they are not absolute recipes for quality.

5. Software Standards – following a standard does not give you justification to throw common-sense out the window.

6. Testing – the testing phase of the software development life-cycle is too late to solve functionality and quality issues.

7. Computer Aided Software Engineering (CASE) – assumes that by using pictorial building blocks, more reliable code can be generated. This is still directly reliant on whether the coder is using the right pictures; garbage in – garbage out.

8. Total Quality Management (TQM) – assumes that if you use quality in all aspects of your business, that quality will automatically permeate into your product. This works for manufacturing, but does not apply so well to the inventive process of software development.

4.1.3 A Model for Software Product Quality

[22, Dromey(1995)] identifies the work that has been done in producing a standard model for software quality; from [11, Boehm *et al.*(1978)] and [18, Deutsch and Willis(1988)] to the [35, ISO9126(1991)] standard. Two key points are made about failures in current quality models are:

1. There is inadequate coverage of the product characteristics side of software quality.
2. No direct links are made between quality attributes and the corresponding product characteristics.

The model proposed in [22, Dromey(1995)] is applied specifically at the software product level, but is also able to be applied at the requirements level. The following requirements are given for quality models in general. They must:

- make clear and direct links between high-level quality attributes and explicit product characteristics at all levels.
- provide a systematic guide for building quality into software.
- provide a way to systematically identify and classify software characteristics and quality defects.
- define a structure that is refinable, adaptable, and understandable at all levels.

[22, Dromey(1995)] reviews the current methodology of defining a quality model, such as taken by [35, ISO9126(1991)], to break down each quality attribute into sub-characteristics. They help to further refine the definition of quality, but unfortunately are still vague and have little applicability at the actual product implementation phase. It suggests applying the quality attributes directly to the low-level components of the product, for simplicity and directness, and then building up an overall sense of quality from the qualities of individual components.

The quality model entity sets defined by [22, Dromey(1995)] are:

1. components

2. quality-carrying properties of components

   - correctness properties (minimal generic requirements for correctness)

     C1. Computable – result obeys laws of arithmetic etc.

     C2. Complete – all elements of structural form satisfied

     C3. Assigned – variable given value before use

     C4. Precise – adequate accuracy preserved in computations

     C5. Initialised – assignments to loop variables, establish invariant

     C6. Progressive – each branch/iteration decreases variant function

     C7. Variant – loop guard derivable from variant function

     C8. Consistent – no improper use of side-effects

   - structural properties (low-level, intra-module design issues)

     S1. Structured – single entry and single exit

     S2. Resolved – data and control structures matching

     S3. Homogeneous – only conjunctive invariants for loops

     S4. Effective – no computational redundancy

     S5. Non-redundant – no logical redundancy

     S6. Direct – problem-specific representation

     S7. Adjustable – parameterised

     S8. Range-independent – applies to variables (arrays), types, loops

S9. Utilised – to handle representational redundancy

- modularity properties (high-level, intermodule design issues)

  M1. Parameterised – all inputs accessed via a parameter list

  M2. Loosely coupled – data couples

  M3. Encapsulated – uses no global variables

  M4. Cohesive – the relationships between the elements of an entity are maximised

  M5. Generic – is independent of the type of its inputs and outputs

  M6. Abstract – sufficiently abstract – is no apparent higher level form

- descriptive properties (various forms of specification and documentation)

  D1. Specified – preconditions and post-conditions provided

  D2. Documented – comments associated with all blocks

  D3. Self-descriptive – identifiers have meaningful names

3. high-level quality attributes

[22, Dromey(1995)] describes the application of the constructive theory: If each of the quality-carrying properties associated with a particular structural form is satisfied when that particular structural form is used in a program, then that structural form will contribute no quality defect to the software.

This theory produces two consequences:

1. Building Quality into Software: The task of building quality into software is reduced to systematically ensuring that all the quality-carrying properties associate with each structural form used, and these in turn satisfy all structural applications.

2. Systematic Detection/Classification of Defects: Detecting quality defects in software is reduced to systematically checking whether, for each structural

form, in all of its occurrences, that any of its quality-carrying properties that imply high-level quality attributes are violated.

4.2 Software Quality Attributes

This section reviews some of the key work produced with definition of quality attributes. [26, Gilb(1988), ch19] is one of the first texts to define quality attributes.

4.2.1 Software Engineering Templates

[26, Gilb(1988), ch19] is the primary foundation reference for quality attribute specification referred to most modern texts in this field. The templates give an initial, but not complete, set of frequently used quality attribute definitions and relationships.

An example of an attribute specification format table follows:

Maintainability:

scale = minutes to do simplest repair to software using templates

date (initial delivery to customers) = January next year

test (unit) = at least ten consecutive repair attempts to be done within worst case level for each module

test (system) = at least 50 random, representative system level bugs to be inserted and then repaired within planned level requirement

worst (by initial release date) = 10 minutes

plan (by initial release date) = 5 minutes

plan (by 3 years after initial release date, for online modules only) = 2

record (lab experiment TR23.3 1989) = 10 seconds average

now (old system, last year average) = 30 minutes

see (marketing strategy, Part 2.3.12) = input to this requirement

source (marketing requirement) = MRS Jan 28th 198x, JCP

The following is the breakdown of template attributes proposed by [26, Gilb(1988)].

Qualities of Attributes

Workability - raw ability of the system to perform work

- Process Capacity - ability to process units of work in units of time
- Responsiveness - reaction to a single event
- Storage Capacity - capacity of a part of a system to store units of a defined thing
- other work-capacity measures

Availability - how much a system is usefully available to perform the work which it was designed to do

- Reliability - measure of the degree to which the system does what it is intended to do, as opposed to something else
- Maintainability - how quickly an unreliable system can be brought to a reliable state
- Integrity - trustworthiness of the system to be in the right state, with security intact

Adaptability - system's ability to change

- Improvability - efficiency of making minor adaptations, changes and improvements to the system
- Extendability - the ease of adding new factors to an existing system
- Portability - the ease of moving a system from one environment to another

Usability - how well people are going to be able and be motivated to use the system practically

- Entry Requirement - human requirements for success in learning and handling the system
- Learning Requirement - resources, principally time, needed to attain some measurable level of ability in the system

- Handling Ability - net productivity over time, when error time is deducted
- Likability - how well people like to use the system

Resource Attributes - costs of existence (development, use and maintenance) of a system,

Time - calendar time to build a system, and working days needed to accomplish a task

People - all people-related resources such as “work years” to construct a system, and people needed to staff or operate it.

Money - all types of monetary costs of building and maintaining the system

Tools - all physical resources which can somehow be limited, and can be critical to our success.

Other Resources - measurable resources such as marketing, goodwill, reputation, image, etc.

4.2.2 Quality Attributes

[7, Bass *et al.*(1998), ch19.3] attempts to analyse and categorise all the quality attributes, but finds that different qualities manifest themselves differently throughout the software development life-cycle. Overall:

1. Many quality attributes should be analysed and designed into the system at the architectural level.
2. Some quality attributes have no real significance at the architectural level.

Quality attributes should never be developed in isolation. Every attribute will have a negative and/or positive effect on some other attributes. For example, a secure system will tend to be a fault-tolerant system. Almost every attribute implemented will be at the detriment of performance.

The following classes of quality attributes are identified by [7, Bass *et al.*(1998)] with respect to architecture:

1. Discernable while the system executes:

   performance – inter-component communication to exploit parallelism

   security – specialised components such as secure kernels or authentication servers

   availability – fault tolerance with redundant components; controlling component interaction

   functionality – NOT architectural but will interact with other quality attributes

   usability – achieving proper information flow; efficiency related to performance; modifiability helps to achieve

2. Not discernable at runtime:

   modifiability – modularised, encapsulating components

   portability – portability layer

   reusability – loose coupling between components

   integrability – consistent component interfaces; uses relation

   testability – modularised, encapsulating components; uses relation

3. Business qualities affected by the architecture:

   time to market – pressure to use COTS

   cost – ability to use assets from in-house

   projected lifetime of the system – modifiability and portability issues

   targeted market – portability, functionality, as well as performance, reliability, and usability on various platforms

rollout schedule – flexibility and customisability if product to be released only with base functionality

extensive use of legacy systems – careful definition of appropriate integration mechanisms

4. Qualities about the architecture itself:

   conceptual integrity – underlying theme unifying the design of the system

   correctness and completeness – meeting requirements and runtime constraints

   buildability – ease of constructing a desired system on-time within given resources

4.2.3 Software Quality Attributes

[60, Wiegers(1999), Chapter 11] describes the relationship between each of the quality attributes in terms of their trade-offs.

1. Availability

   Reliability – direct

   Robustness – direct

   - important primarily to users
   - critical to mainframe systems
   - percentage of planned uptime during which system is actually available for use and fully operational.
   - can be different for different times of the day/week
   - $Availability = \frac{MeanTimeToFailure}{MeanTimeToFailure+MeanTimeToRepair}$

2. Efficiency

   Flexibility – inverse

Interoperability – inverse

Maintainability – inverse

Portability – inverse

Reliability – inverse

Robustness – inverse

Testability – inverse

Usability – inverse

- important primarily to users
- critical to embedded systems
- how well the system utilises processor capacity, disk space, memory, or communication bandwidth
- important to consider minimum hardware configurations
- allow safety buffer (in memory/cpu usage) for unanticipated conditions, e.g. under peak load

3. Flexibility

Efficiency – inverse

Integrity – inverse

Maintainability – direct

Portability – direct

Reliability – direct

Testability – direct

- important primarily to users
- extensibility, augmentability, extendability, expandability
- how much effort is needed to add new capabilities to the product

- choose design approaches that maximises software flexibility, especially for incremental releases, new drivers, etc.

4. Integrity (security)

   Efficiency – inverse

   Interoperability – inverse

   Reusability – inverse

   Testability – inverse

   Usability – inverse

   - important primarily to users
   - precluding unauthorised access to system functions
   - preventing information loss
   - protection from virus infection
   - privacy protection of data entering/leaving system
   - has no tolerance for error - data and access must be completely protected in specified ways
   - state the user identity verification, user privilege levels, access restrictions, precise data, that must be protected

5. Interoperability

   Efficiency – inverse

   Flexibility – direct

   Integrity – inverse

   Portability – direct

   - important primarily to users
   - critical to desktop systems

- how easily the product can exchange data or services with other systems
- need to specify what other applications/data users will use in conjunction with this system

6. Maintainability

   Availability – direct

   Efficiency – inverse

   Flexibility – direct

   Reliability – direct

   Testability – direct

   - important primarily to developers
   - critical to mainframe systems
   - how easy it is to correct a defect or make a change in software
   - how easily the software can be understood, changed and tested
   - closely related to Flexibility
   - products revised periodically or built quickly require high maintainability
   - average time required to fix a problem, and percentage of fixes made correctly

7. Portability

   Efficiency – inverse

   Flexibility – direct

   Interoperability – direct

   Maintainability – inverse

Reusability – direct

Testability – direct

Usability – inverse

- important primarily to developers
- effort required to migrate a piece of software from one operating environment to another
- design approaches for Portability are similar to approaches for Reusability
- typically immaterial or critical to the projects success
- should state the portions of the product that must be able to migrate to other environments and identify those target environments.

8. Reliability

Availability – direct

Efficiency – inverse

Flexibility – direct

Maintainability – direct

Robustness – direct

Testability – direct

Usability – direct

- important primarily to users
- critical to embedded systems
- probability of software executing without failure for a specified period of time
- Robustness and Availability are often considered aspects of reliability

   - percentage of correctly performed operations
   - the length of time the system runs before revealing a new defect
   - defect density
   - measures based on how severe the impact would be if the failure occurs
   - whether cost of maximising reliability is justifiable
   - software can be still considered to meet its reliability requirements even if it still contains defects.
   - systems designed for high reliability should also be designed for high testability

9. Reusability

   Efficiency – inverse

   Flexibility – direct

   Integrity – inverse

   Interoperability – direct

   Maintainability – direct

   Portability – direct

   Reliability – inverse

   Testability – direct

   - important primarily to developers
   - extent to which a software component can be used in applications other than the one for which it was initially developed.
   - costs considerably more than one-off code
   - code has to be modular, well-documented, application and environment independent, and somewhat generic.

- specify what elements of system are to be reusable, or what libraries are to be generated.

10. Robustness

    Availability – direct

    Efficiency – inverse

    Reliability – direct

    Usability – direct

    - important primarily to users
    - degree to which a system or component continues to function correctly when confronted with invalid input data, defects in connected software or hardware components, or unexpected operating conditions.
    - robust software recovers gracefully from problem situations and is forgiving of user errors.
    - find out what known error conditions the system might encounter and how users would like the system to react.

11. Testability

    Availability – direct

    Efficiency – inverse

    Flexibility – direct

    Maintainability – direct

    Reliability – direct

    Usability – direct

    - important primarily to developers
    - ease with which the software components or integrated product can be tested to find defects.

- critical for product with complex algorithms and logic, or subtle functionality interrelationships.
- important for code that will be modified often, because of the need for frequent regression testing.
- e.g. Cyclomatic complexity measuring number of logic branches in a source code module.

12. Usability

    Efficiency – inverse

    Robustness – direct

    Testability – inverse

    - important primarily to users
    - critical to desktop systems
    - “ease of use” and “human engineering”
    - factors that constitute ”user-friendliness”
    - measures effort required to prepare input for, operate, and interpret the output of the product.
    - need to balance Usability with “easy of learning” how to use the product.
    - encompasses how easy it is for new or infrequent users to learn to use the product.
    - consider how expensive it will be to test the product to determine whether it satisfies the requirements.

4.2.4 Understanding Quality Attributes

[7, Bass *et al.*(1998), ch19.3, pp423-424] discusses the issues with the meaning of quality within software. There are problems with the meaning of “fitness for use" with many of the quality attributes. Developmental qualities have problems because of a lack of suitable mature models for discussing them.

Models such as constructive cost modelling (COCOMO) or function point modelling, only measure development time from an external point of view. They do not have an abstract internal understanding of the development process.

An example of a well-defined model is performance. This is discussed, analysed and measured in terms of resource availability and consumption. Resource consumption is a function of the number of resource requests. The system is modelled as a collection of resources and connections between these resources.

The Software Engineering Institute is currently attempting to define abstract models for all quality attributes. They are defining fulcrum points; "measurable properties of a system that affect two (or more) quality attributes."

For example, the amount of variance of arrival time in a communication pipe will affect system performance, as well as security, and reliability. A lag in arrival leaves the data more open to undetected alteration, as well as the possibility of hardware failures going undetected. This variance, is a fulcrum point.

4.2.5 Software Product Quality Requirements and Evaluation

[36, SQuaRE(2002)] is a working document for the forthcoming ISO 25000 standard for software quality requirements and evaluation. It replaces the previous ISO 9126 - Software Quality Characteristics and ISO 14598 Software Product Evaluation standards. It defines software quality in terms of a changing viewpoint throughout the product development life-cycle. Figures 4.1, 4.2, and 4.3 show that the user's perception of quality is mapped to the external product quality at the requirements stage, which becomes internal product quality at the design and development stage.

Overall, [36, SQuaRE(2002)] attempts to cover all aspects of quality, by taking the view that no one user or developer can explicitly know all the quality attributes required for the system, due to unstated or implied needs, changing needs, and

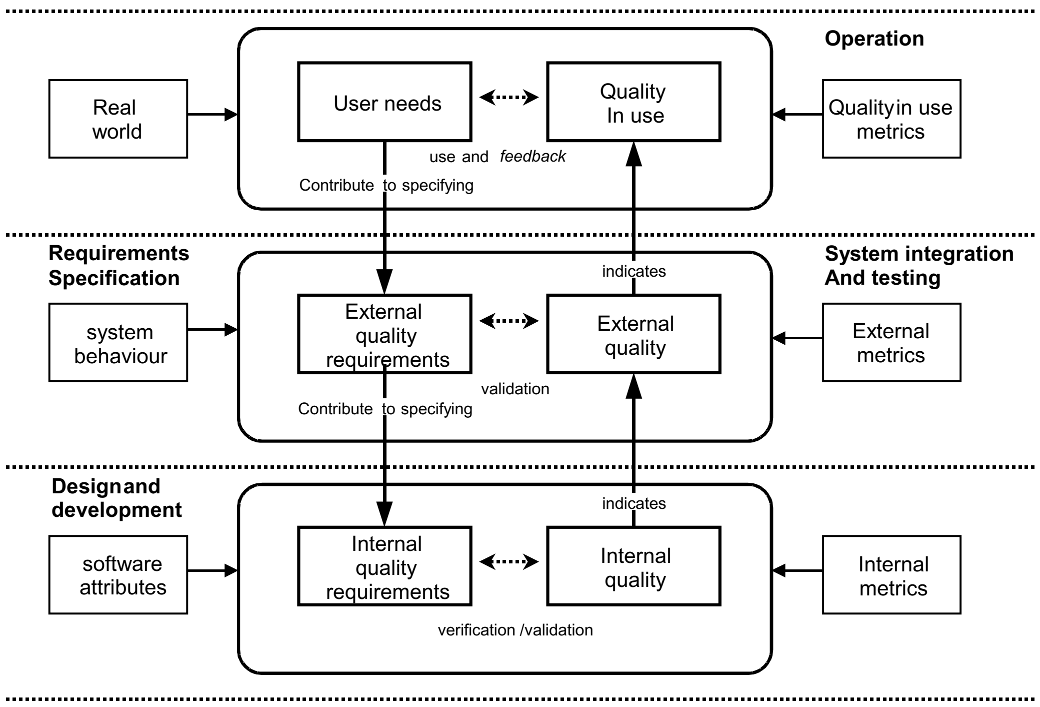


Figure 4.1: Quality in the software life-cycle

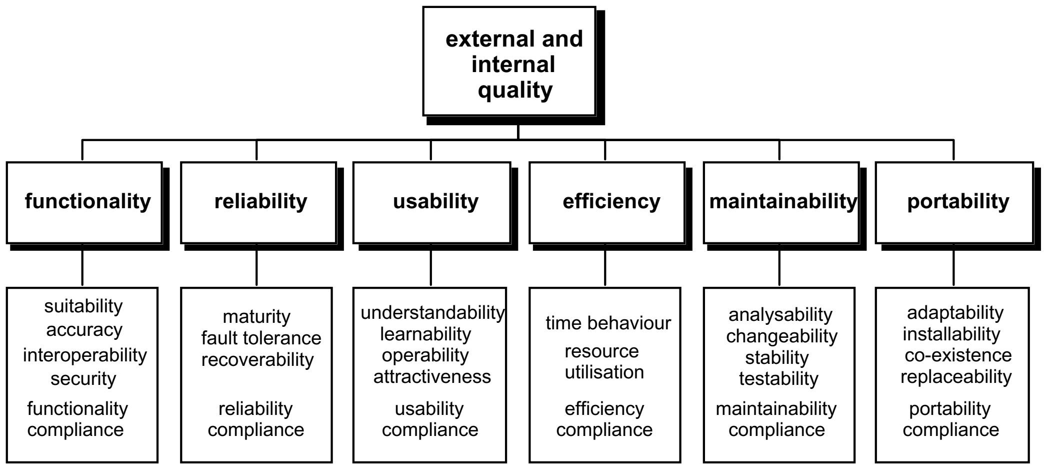

external and internal quality
functionality
reliability
usability
efficiency
maintainability
portability
suitability accuracy interoperability security functionality compliance
maturity fault tolerance recoverability reliability compliance
understandability learnability operability attractiveness usability compliance
time behaviour resource utilisation efficiency compliance
analysability changeability stability testability maintainability compliance
adaptability installability co-existence replaceability portability compliance

Figure 4.2: Model for internal and external quality

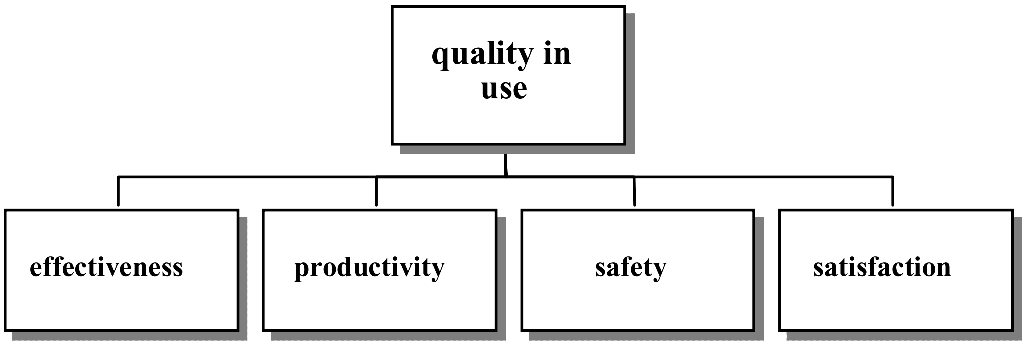


Figure 4.3: Model for quality in use

differing or unknown end-users.

[37, SQuaRE(2002)] details the use of the quality model with regards to metrics collection. Measures are required at all three views; in use, external, and internal, because each individual viewpoint is not sufficient or adequate to encompass the quality measurements required for the other views.

In terms of measures, [37, SQuaRE(2002)] suggests a weighting of high, medium, or low for each quality characteristic, so that evaluators may concentrate on quality aspects which are important to the project in hand. When the product is being assessed, the metrics gathered can be compared against those required, and overall feedback on the product can be provided.

4.2.6 Defining and Specifying Quality Attributes

[1, Abel and Rout(1993)] is primarily concerned with the specification of software product quality attributes. It takes the technique presented by [18, Deutsch and Willis(1988)] and [Mizuno and Yoji] and details a prototype tool that transforms the quality attributes into software engineering specifications.

The original work on quality function deployment was developed in Japan in the late 1960s by Professors Shigeru Mizuno and Yoji Akao. It emphasised the importance of making quality control a part of business management, which eventually became known as TQC and TQM. The purpose of Professors Mizuno and

Table 4.1: Quality Measurement Model

	Activity 1	Activity 2	Activity 3	Activity 4	Activity 5	Activity 6	Activity 7	Activity 8
Phase	requirement analysis	architectural design	software detailed design	software coding and testing	software integration and qualification testing	system integration and qualification testing	software installation	software acceptance support
9126 model reference	required user quality required internal quality required external quality	predicted quality in use predicted external quality measured internal quality	predicted quality in use predicted external quality measured internal quality	predicted quality in use measured external quality predicted external quality measured internal quality	predicted quality in use measured external quality predicted external quality measured internal quality	predicted quality in use measured external quality measured internal quality	predicted quality in use measured external quality measured internal quality	measured quality in use measured external quality measured internal quality
key deliverables of activity	user and external quality requirements specified, internal quality	architecture design of software or system	software detailed design	software code, test results	software product, test results	integrated system, test results	installed system	delivered software product

Akao was to develop a quality assurance method that would design customer satisfaction into a product before it was manufacturered. Prior quality control methods were primarily aimed at fixing a problem during or after manufacturing.

The components of the [18, Deutsch and Willis(1988)] technique are as follows:

1. Step 1: Analyse the user's need for quality:

    - use standard Fitness-For-Use Factors as a checklist

   - collect and organise specific needs for quality
   - determine required levels for each of the fifteen factors
   - resolve conflicts between requirements
   - establish trade-offs with cost and schedule

2. Step 2: Convert quality needs to requirements:
   - use a matrix of Needs vs Requirements to establish a requirements database
   - convert each requirement into a set of relevant, testable, objective statements
   - review each requirement to ensure it meets standards

3. Step 3: Determine the engineering techniques or methods to be applied to achieve each requirement:
   - use a matrix of Requirements vs Techniques

4. Step 4: Document the quality requirement analysis to produce the Software Quality Engineering Specifications.

[1, Abel and Rout(1993)] then presents a mapping between the [35, ISO 9126(1991)] and [18, Deutsch and Willis(1988)] quality characteristics. For example [35, ISO 9126(1991)] defines the sub-characteristics of Reliability as Maturity, Fault tolerance, and Recoverability. Whereas [18, Deutsch and Willis(1988)] uses the overall attributes of Reliability and Survivability to cover these same sub-characteristics. It is found that Safety and Manageability are two characteristics that do not map which are both outside the scope of [35, ISO 9126(1991)].

The trade-off relationships between quality attributes is shown in Figure 4.4. Figure 4.5 is the interpretation of the relationship of attributes defined by [18, Deutsch and Willis(1988)] and reworked to conform to [35, ISO 9126(1991)], as developed by [1, Abel and Rout(1993)].

Results in these characteristics being affected as shown		Functionality					Reliability			Usability			Efficiency		Maintainability				Portability			
Increase in specified quality for these characteristics		Suitability	Accurateness	Interoperability	Compliance	Security	Maturity	Fault Tolerance	Recoverability	Understandability	Learnability	Operability	Time Behaviour	Resource Behaviour	Analysability	Changeability	Stability	Testability	Adaptability	Installability	Conformance	Replaceability
Functionality	Suitability				+					+		+						+				+
	Accurateness	+			+										+	+		+	+			
	Interoperability												-	-								+
	Compliance		+										-		+						+	
	Security			-	+								-							-		-
Reliability	Maturity												-	-			+	+				
	Fault Tolerance												-	-	+	-			-			
	Recoverability			-							-	-		-								
Usability	Understandability	+									+	+	-				+	+				+
	Learnability	+								+		+		-								
	Operability	+			+					+	+						+					+
Efficiency	Time Behaviour														-			-	-			
	Resource Behaviour								-							-		-		-		
Maintainability	Analysability						+									+		+	+			
	Changeability						+							-	+			+	+			
	Stability					+	+	+							+			+				
	Testability				+			+							+							
Portability	Adaptability												-	-								
	Installability								+			+		-								
	Conformance				+								-		+	+			+			
	Replaceability											+	-									

Figure 4.4: Matrix of Interactions - ISO 9126 Quality Characteristics

4.2.7 Quality Attributes and Software Architecture

[8, Bass *et al.*(2000)] focusses on linking quality attributes to software architecture. Even though, over the last 30 years, a number of researchers and practitioners have examined how systems achieve software quality attributes, no one has completely and systematically linked software architecture and quality attributes.

- Lack of precise definitions: Attributes such as reliability, availability, and performance have generally accepted definitions, but modifiability, security,

Quality Characteristics	Functionality					Reliability			Usability			Efficiency		Maintainability				Portability			
Engineering Attributes	Suitability	Accurateness	Interoperability	Compliance	Security	Maturity	Fault Tolerance	Recoverability	Understandability	Learnability	Operability	Time Behaviour	Resource Behaviour	Analysability	Changeability	Stability	Testability	Adaptability	Installability	Conformance	Replaceability
Accuracy						*														*	
Anomaly Mangement						*	*	*													
Augmentability															*	*					
Autonomy							*	*								*					*
Commonality	*		*	*																*	
Completeness	*	*												*							
Consistency		*		*										*	*					*	
Distributivity							*	*													
Quality of Documentation								*	*					*	*						
Efficiency of Communication												*	*								
Efficiency of Processing												*	*								
Efficiency of Storage												*	*								
Functional Scope	*		*													*					
Generality															*	*			*		*
Independence			*											*		*		*			*
Modularity			*											*	*	*	*	*	*		*
Operability									*	*	*										
Safety Management																				*	
Self-Descriptiveness														*	*	*	*	*	*	*	*
Simplicity						*		*						*	*		*		*		*
Support								*						*	*		*	*			
System Accessibility					*																
System Compatibility	*		*	*																	
Traceability	*	*		*				*						*		*	*				*
Training									*	*	*										
Virtuality														*	*	*					
Visibility														*			*				

Figure 4.5: Relationship between ISO 9126 Quality Characteristics and Engineering Attributes of Software

and usability do not.

- Attributes cannot be isolated: Availability is an attribute, but it is also a subset of security (denial of service attacks can limit availability).

- Attribute patterns are not standard: Hundreds of patterns at different levels of granularity, do not lend themselves to analysis, categorisation or reuse.

- Analysis techniques differ between attributes: Analysis results and interactions are difficult to understand.

There is a widely held premise of the software architecture community that architecture determines quality attributes, and very little, must be known about the functionality of the system in order to draw quality attribute conclusions.

Performance predominantly depends of processes, their allocation to processors, communication paths between them. Reliability requires redundancy strategies. In each case, architecture design decisions are necessary to achieve the attribute. Very little knowledge of the functionality is required.

4.3 Software Engineering Measurement

[24, Fenton and Pfleeger(1997)] looks at software quality in terms of measurement of specific software product attributes. This may involve decomposing an attribute into measurable components of quality, guided by published models or standards, or by using defect-based measures.

Defects measures are not an entirely accurate measure of quality as perceived by the user, as some defects discovered during development or testing may not lead to software failures in operation. However, it is a powerful measure for base-lining and monitoring quality changes.

Maintainability involves monitoring of the maintenance process, with measures of time to locate faults and fix.

Usability must involve the people who use the software. As such, there are no internal quality attributes that obviously predict usability.

In terms of setting measurable targets for an quality attributes, [24, Fenton and Pfleeger(1997)] cites [26, Gilb(1988)] in saying that the onus should lie with the user. There are recommended measures such as:

$Portability = 1 - \frac{ET}{ER}$

where ET is a measure of the resources to move the system to the target environment, and ER is a measure of the resources needed to create the system for the resident environment.

Although these measures are dependent on subjective ratings, this is better than no rating at all. Overall, consistency is of high importance for those performing the rating so that there is limited variability.

4.4 Software Process Quality

Software process standards "that detail how to identify, analyse, design, implement, test, deploy, maintain and evolve high-quality components and component-based applications" [30, McClure (2001)] are needed by software developers. The successful reuse of software components cannot occur unless quality processes, using the industry's best practice, are put into place by the providers as well as the integrators of these components.

A quality product must be developed using quality processes. The correlation between the two has been statistically validated by [29, Lawlis *et al.*(1995)]. However process maturity does not automatically imply product quality. If mature technology to predict and control quality attributes is lacking, even a mature organisation will have difficulty producing products with predictable performance and dependability. [6, Barbacci *et al.*(1997)]

The remainder of this section will give an overview of the software industry's process standards.

The ISO 15504 international standard for Software Process Improvement and Capability Determination (SPICE) is primarily concerned with process assessment. It establishes a common framework for software process capability ratings, as well as providing a migration path for existing process models such as CMM.

”The Capability Maturity Model is a widely accepted and used method for evaluating the maturity of an organisation's software development and maintenance processes." [30, McClure (2001) pg.269]. CMM is used as a *de facto* standard for assessment and improving processes internal to the organisation, and evaluating a vendor's capability to produce quality software. Efforts are currently underway to making CMM ISO 15504 compliant.

The defined CMM framework consists of five levels of maturity relating to the software lifecycle:

1. Initial: The software process is mainly *ad hoc*, left to the discretion of the

individuals performing it.

2. Repeatable: The software process includes project management processes to track cost, schedule, and functionality. Disciplines are in place that make the software process repeatable across multiple projects.

3. Defined: A standardised, documented software process is developed for the organisation and is used to collect data that is used in the management and control of both the process and its products.

4. Optimising: Continuous process improvement is achieved by using feedback from process and product measures and by identifying, piloting, and adopting appropriate new technologies and ideas.

Organisations tend to evolve their software processes by using these maturity levels as steps. Typically most organisations are at level 1 and 2, and few have reached level 3 or higher.

CMM and [17, ISO/IEC 12207(1995)] are similar in many regards. They are both requirement level specifications for processes, avoid implementation details, and can be applied to both the software development process as well as the organisational processes. However 12207 describes requirements as only the minimum required to meet the standard, whereas CMM rates the maturity of the requirements currently used. 12207 also describes the processes in far more detail, referencing other standards where required, such as the Joint Review Process, Verification Process, Management Process, etc.

4.5 Software Component Architecture

4.5.1 Models of Evolving Software

[42, Sutcliffe(2002), Chapter 9.3] describes the two reuse paradigms of component engineering and application generation architectures.

Adaption of current systems to a new task appears to be risky strategy because of the potential cost from errors and mal-adaption. Component engineering has lower costs but will be more suitable to technical users and large organisations. Application generators will be suitable more to end-users.

A model is proposed for intelligent reusable components with application generator architectures that can assemble applications as required from semi-autonomous agents. To implement this model, an addition problem arises. What minimum information is required to be distributed between agents, so that they can reliably create harmonious components?

Finally [42, Sutcliffe(2002)] expresses the view that standardisation of the component engineering process must be made across the industry for reuse to become a viable driver in the domain marketplace. It is more likely that will be standardisation only within specific communities of re-users, such as large corporations.

4.5.2 Component-Based Systems

[7, Bass *et al.*(1998), Chapter 15.3] addresses the use of components within a product production line. It is rare that a software system in today's world is coded entirely from scratch, and without the use of third-party functionality. The current trend of "buy, don't build" implies a loss of control and hence quality in the system development process. [7, Bass *et al.*(1998)] asserts that quality primarily is to be found in the architecture, and that a developer must be careful when integrating components into their system, that they do not compromise the quality inherit to this architecture.

[7, Bass *et al.*(1998)] goes further to discuss the case where components are almost compatible – *meaning not compatible*. They may appear to work together; specifications match, code compiles and executes. However subtle errors with timing, operation ordering, and assumptions that no parallel processing is being done, all can combine to produce an unreliable system. [7, Bass *et al.*(1998)] states that "component interfaces are notoriously poor at specifying their quality attributes: How secure is the compiler you're using right now? How reliable is the mail system on your desktop?".

A special case of interface and architectural mismatch when integrating components, is the problem of mismatching assumptions that are embodied in separately developed components. For example, two components may disagree about which one calls the other, causing the system to not compile, link or execute.

There are ways of dealing with interface mismatch:

1. change requirements, and/or make bug a feature
2. avoid it by complete and careful specification of components
3. detect additional cases by careful qualification
4. repair by adaption of components

Techniques for repairing interface mismatch:

1. wrappers - encapsulate component and provide alternative interface. (e.g. good way to introduce privacy)
2. bridges - provides translation between provides assumptions of system and components, determined at build time.
3. mediators - actual translation determined at runtime

Techniques for avoiding interface mismatch:

1. specify as many assumptions about interface as possible (at least a subset with the most important) from earliest phases of design.

2. specify different interfaces for the same component
3. using a parameterised interface where provides and requires assumptions can be changed before invoking the component service by changing value of a variable. This is similar to using environment variables or parameter files.
4. a negotiated interface is a parameterised interface with self-repair logic. Modems and routers routinely use protocols to establish mutually acceptable communication parameters at runtime.

4.5.3 Architecture and Legacy Systems

[7, Bass *et al.*(1998) Chapter 19.2] identifies a research area need for determining or rediscovering the architecture of existing or legacy systems. These systems range from well designed and documented to poorly designed and undocumented. Often intelligent changes need to made to systems that an organisation has a large investment in. The problems encountered are:

1. determining the existing architecture of a legacy system
2. determining the goal state of re-engineered architecture
3. developing a strategy to migrate the system to this new architecture, through re-engineering, wrapping, or complete replacement.

Architecture to system non-conformance problems

1. there is no “documented” architecture
2. relationship between documented architecture and source code is unclear, due to poor representation of architecture.
3. architecture out of phase with actual system, due to maintenance updates performed only on code.

4.5.4 Achieving an Architecture

[7, Bass *et al.*(1998) Chapter 19.3] states that "Achieving an architecture in a reliable, repeatable manner from a precise statement of the quality requirements is a key open research area." Further, the problem is broken down into the following components:

1. gathering meaningful and quantifiable definitions of the various qualities.

2. selecting or generating an architecture based on a complete set of requirements (functional and non-functional).

3. allowing for architecture decision trade-offs.

4.5.5 Architectural Means for Achieving Qualities

[7, Bass *et al.*(1998) Chapter 4.2] introduces the concept of an "architectural style" which are architectural patterns in achieving architectural aspects of software quality.

There are two types of architectural structure design points of view:

1. A system's modular structure is most often used to derive the work-breakdown structure, exploit supplied components, and plan for modifications. This modular structure is only an abstraction because, once the code is compiled and linked, those modules no longer exist.

2. Architectural styles such as pipes, filters, and client-servers do survive at runtime, through the way components interact with each other.

The notions of an architecture as a set of modules, or as a set of processes, are both legitimate, but they are not the same structure. They are optimised to meet completely different criteria.

Allowing for non-execution requirements involves building the system to accommodate the following changes:

1. changes anticipated for this system or experienced by similar systems.
2. changes anticipated by mining the requirements specifications for areas of uncertainty or ambiguity.
3. decreases in functionality due to production of only a system subset.

[7, Bass *et al.*(1998)] concludes that architectures are chosen to "meet behavioral and performance requirements, by exploiting familiarity with architectures of similar systems that produced correct and timely results."

4.5.6 Types of Application Integration

[33, Ruh *et al.*(2000) Chapter 2] investigates the types of application integration at three different levels, by presenting them as models:

1. A presentation integration model, figure 4.6, allows the integration of new software through the existing presentations of the legacy software. This is typically used to create a new user interface but may be used to integrate with other applications. This model uses a white box approach to integration.
2. A data integration model, figure 4.7, allows the integration of software through access to the data that is created, managed, and stored by the software typically for the purposed of reusing or synchronising data across applications. This model uses a white box approach to integration.
3. A functional integration model, figure 4.8, allows the integration of software for the purpose of invoking existing functionality from other new or existing applications. The integration is done through interfaces to the software. This model can use either a white or black box approach to integration. The approaches to application of functional integration are given:
   - Data consistency integration is integration through the code of an application where the purpose is to access or update data. The integration facilitates the communication of data and actions.

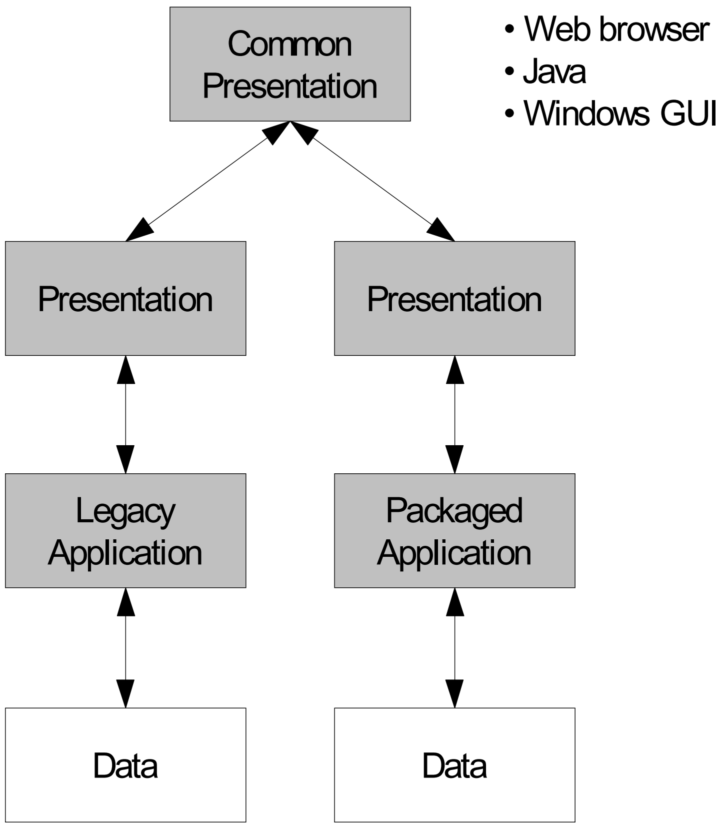


Figure 4.6: The presentation integration model integrates through the user interface of applications.

- Multi-step process integration, also known as straight-through processing, is the integration of applications where there is not only communications of requests but also the coordination and management of these requests across applications. The integration facilitates communication of the request and manages the flow and sequencing.
- Plug-and-play components integration is the integration of applications where a well-defined interface exists that allows a component to be easily connected with other components without modification. The integration facilitates the communication of requests and handles all of the

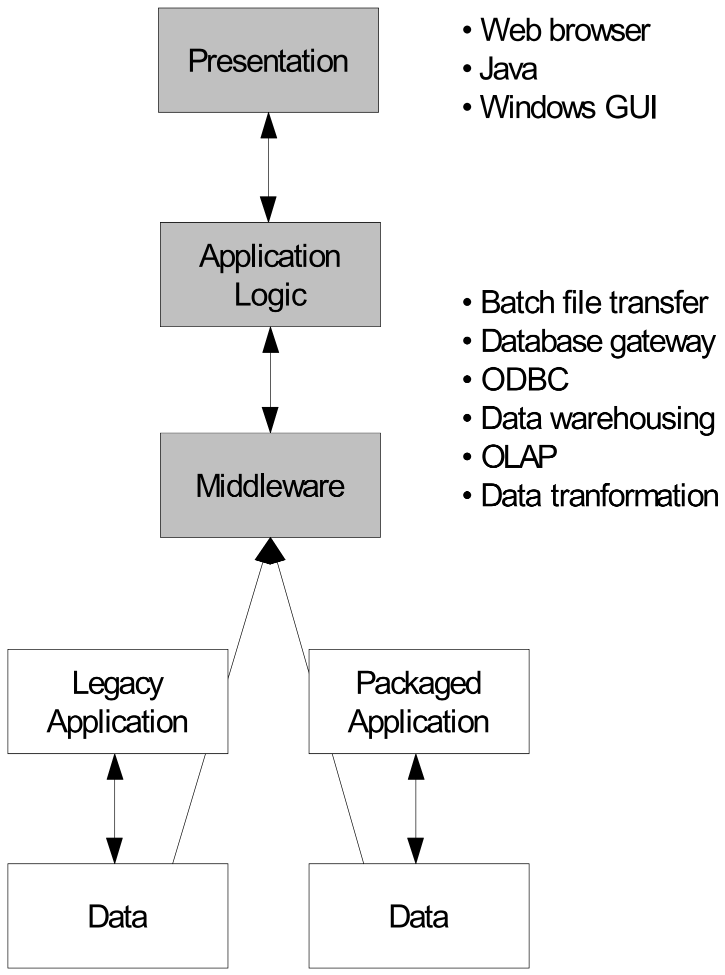


Figure 4.7: The data integration model integrates directly to the data created and managed by the application.

interface definition and management.

Whenever two software components have been integrated there are several factors that heavily influence the quality and utility of that integration.

- The integration model that is applied
- The tools that are used
- The designer's choices

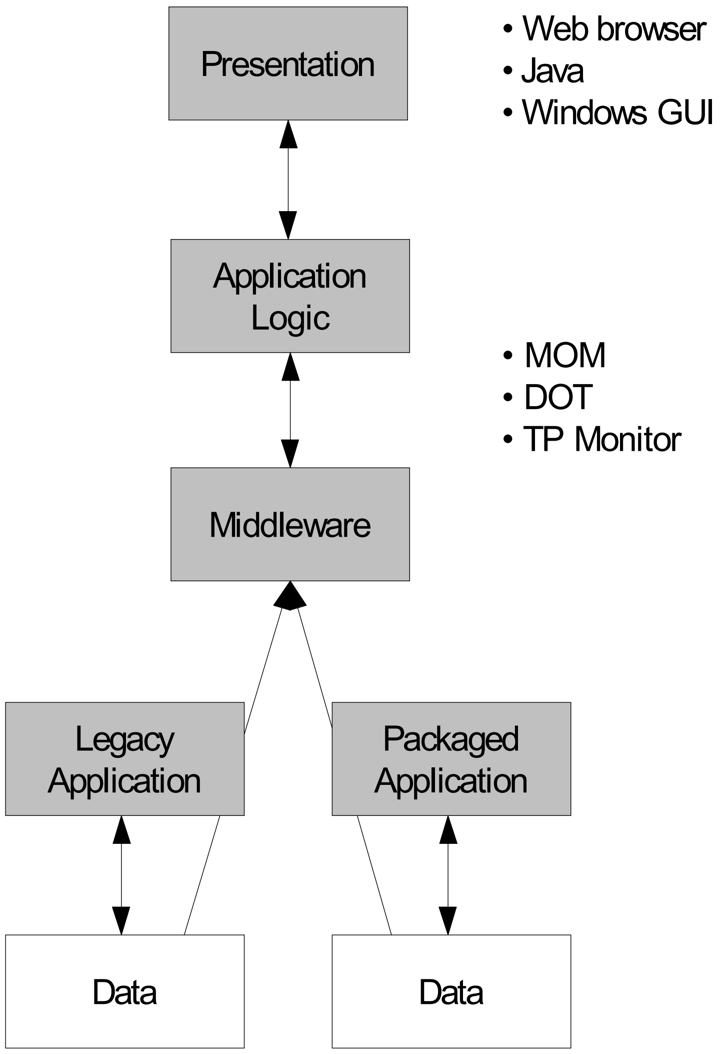


Figure 4.8: The functional integration model integrates directly with the code of the application.

The most important overall goal of integration is to reduce the level of coupling, or degree of interdependency, between the software components. The white box integration approach, as used in the presentation and data integration models, exposes the internals of the application or database to the integrator in order to produce the required integration. This leads to a tight (high) level of coupling.

The black box integration approach usually leads to loose coupling because the details of the component are hidden, and integration is performed through an API, connector, or some form of interface.

4.5.7 Design and Use of Software Architectures

[12, Bosch(2000), Chapter 1, pp4-6] presents a clear review of current software engineering practice: Software development traditionally is a project-based activity, focussed on meeting a delivery deadline, with little or no consideration placed on maintenance or reusability. Typical issues in industry include:

- the cost of maintenance
- the lack of reliability, or high cost to achieve reliability
- and the failure to deliver on-time and in-budget.

Desirable objectives for the health and longevity of the industry include:

- the need to dramatically reduce the cost of development
- increase the reliability, maintainability, and resource efficiency at a reasonable cost
- improve the time-to-market for the product
- considerably reduce maintenance costs

Component Reuse

Reuse of existing software through componentisation [31, McIlroy(1969)] has long been recognised as the most promising approach to achieving the objectives stated previously. "If we are able to develop systems from existing components, development cost decreases, quality increases because the components have been tested in other contexts, time-to-market is shortened and maintenance cost is decreased because changes to the components benefit multiple systems.” [12, Bosch(2000), Chapter 1, p6]

Software reuse seems to have infused into the software industry through:

- operating systems
- database management systems
- compilers
- graphical user interfaces
- component interconnection standard implementations
- web servers
- web browsers

These assets may not be perceived as reuse examples, but each of these, at one time, were developed as part of the application code, rather than developed or sourced as separate components. They have become part of the development infrastructure and are assumed to be present on for the intended platform.

The changing use of these assets typically follow this life-cycle:

1. functionality is part of application code
2. functionality identified and modelled as a sub-system
3. researchers develop prototype systems generalising the sub-system
4. companies commercialise the system into a product
5. product is accepted market-wide and used by software engineers
6. product incorporated into infrastructure as a library etc.

[12, Bosch(2000), Chapter 1, p6] theorises that the reason software re-use has not been successful within software development organisations is because of organisational structure, engineering culture, and lack of domain understanding. The use of external components is well established, but reuse of software developed within the organisation is often very hard to achieve.

The following paragraphs discusses the approaches to component integration.

The first naive approach to software reuse, takes components, plugs them together, like Lego blocks, and produces a useful system. This may be a simple concept, but does not work in practice for real complex systems.

- The opportunistic approach assumes the software engineer selects and combines pieces of software that fit the current problem and adds them to the software product that they are working on.
- The planned approach requires that the organisation develop reusable assets with the right abstraction and variability levels for the software product devised and evolved.
- The bottom-up approach, allows software engineers to search for a suitable component from a collection of developed and mined assets.
- The top-down approach dictates that assets are developed with predefined interfaces, as parts fitting into a higher-level structure.

[12, Bosch(2000), Chapter 1, p6] states that any successful software reuse programme must be planned, and it must take a top-down approach. Opportunistic and bottom-up reuse does not work in practice.

[12, Bosch(2000), Chapter 1, p6] proposes a method for explicit software architectural design which includes the implementation and assessment of quality attributes. The advantages are:

- The software product built in accordance with the software architecture is much more likely to fulfil both its quality and functional requirements.

- There will be decreased implementation costs and time-to-market because the system has been designed, reviewed, and optimised where necessary, at the earlier architectural stage.
- An allowance is made for future changing requirements, which takes less effort at the architectural level.

There are three levels of component reuse:

1. Subsequent versions of a software product - well established practice in the industry
2. Different products - current challenge of reusing software in different contexts
3. Product versions, various products and different organisations - besides success in domains such as graphical user interfaces, the industry is not yet mature enough to achieve.

The academic and industry viewpoints of reusable components differ:

1. Academia
    - Reusable assets are black-box components.
    - Assets have a narrow interface through a single point of access.
    - Assets have few and explicitly defined variation points which are configured during installation.
    - Assets implement standardised interfaces and can be traded on component markets.
    - Focus is on asset functionality and on the formal verification of functionality.
2. Industry

- Assets are large pieces of software with a complex internal structure and no enforced encapsulation boundary (e.g. OO framework).
- The asset interface is provided through entities (e.g. classes). These interface entities have no explicit difference to non-interface entities.
- Variation is implemented through configuration and specialisation or replacement of entities in the asset. Sometimes multiple implementations (versions) of an asset exist to cover variation requirements.
- Assets are primarily developed internally. Externally developed assets go through considerable (source code) adaptation to match the product-line architectural requirements.
- Functionality and quality attributes have equal importance.

Software Architecture Design Phases

Design of a software architecture is not an independent activity. It is but one step in the development and evolution of the software product. Figure 4.9 represents this graphically.

Phase 1 - Functionality-Based Architectural Design

1. Determine context, external interfaces, and interface behaviour.
2. Abstraction - Archetype identification and relations.
3. Determine architecture structure - System component decomposition, and relationship identification.
4. System instance description using archetypes and system interfaces.

An architectural design created from functional requirements does not preclude the optimisation of quality requirements during the later architectural design phases. An architectural design based on functional requirements only will still have values for its quality requirements; you cannot achieve pure separation of functional and quality requirements. By focussing only on the functional requirements the architecture achieved is more general, and can be iterated over to optimise. On the other hand, it is unlikely that an architecture fulfilling a particular set of quality requirements will be applicable in a domain with different functional requirements.

Phase 2 - Assessing Software Architectures

Traditionally, the software industry will implement a system and then measure the actual values for the quality system properties. This can produce an expensive system that will not meet its quality requirements. Hence there is a need to estimate the quality attributes of a system during the early development stages.

It is not possible to measure system properties at the abstract architectural design phase. What can be measured is the potential of the architecture to reach the required levels of quality. While quantitative statements can be made about the quality attributes of an architecture, it is still very important that these attributes are designed and implemented correctly within the system.

- qualitative assessment - comparison of overall qualities of two systems
- quantitative assessment - compare values of quality attributes between system or versions
- theoretical assessment

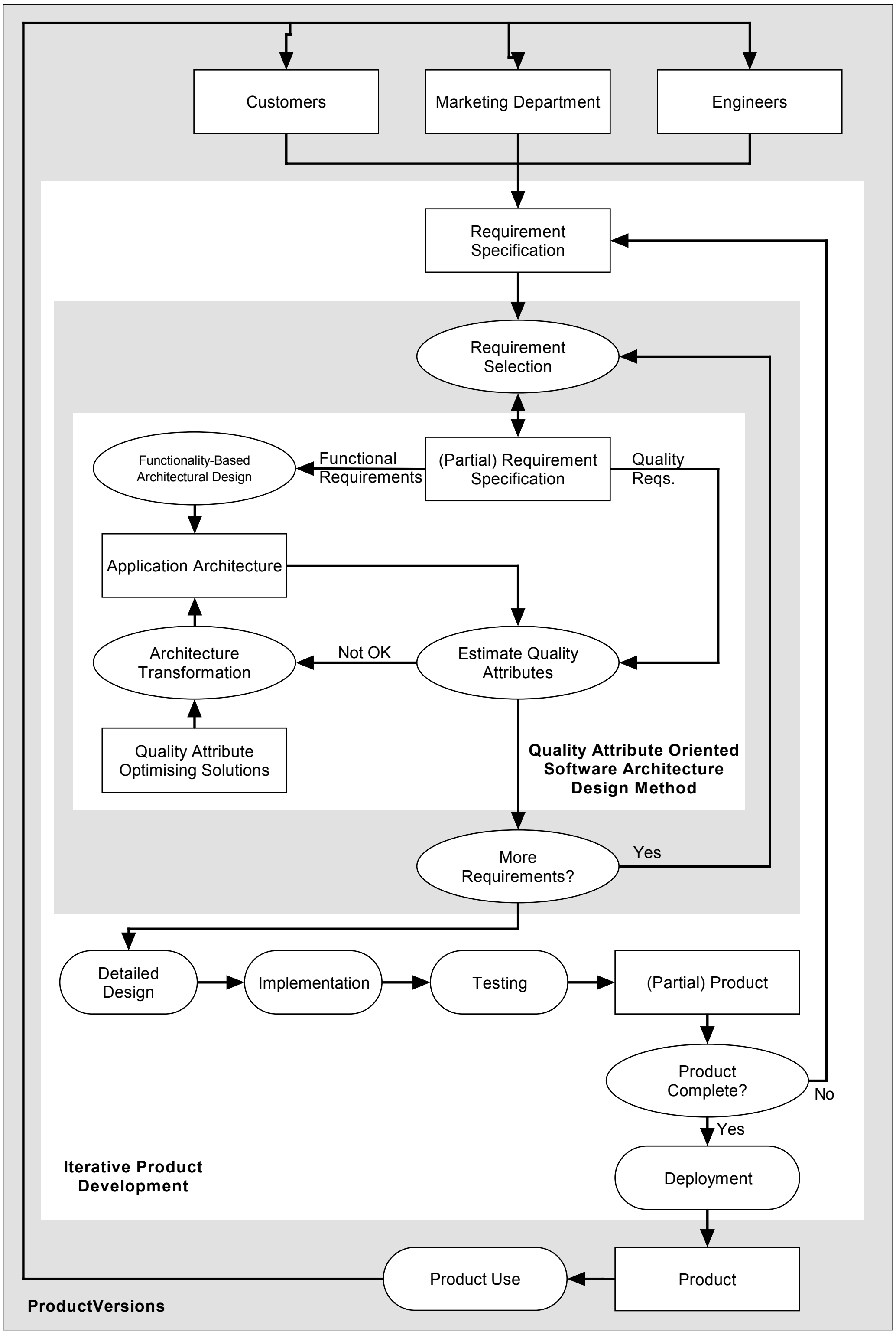

Customers
Marketing Department
Engineers
Requirement Specification
Requirement Selection
Functionality-Based Architectural Design
Functional Requirements
(Partial) Requirement Specification
Quality Reqs.
Application Architecture
Architecture Transformation
Not OK
Estimate Quality Attributes
Quality Attribute Optimising Solutions
Quality Attribute Oriented Software Architecture Design Method
More Requirements?
Yes
Detailed Design
Implementation
Testing
(Partial) Product
Product Complete?
No
Yes
Iterative Product Development
Deployment
Product Use
Product
ProductVersions

Figure 4.9: Context of Architectural Design within the Software Development Process

4.6 Architecture Description

4.6.1 Genetic Software Engineering

[21, Dromey(2002)] and [19, Dromey(2001)] describes the process for creating an architecture design that satisfies a set of functional requirements. The "Behavior Tree"^{TM} model used captures both the static and dynamic interaction between components, normally requiring several separate diagrams in standards such as the Unified Modelling Language (UML).

The primary benefit of using this model, is that it "provides a clear, simple, constructive and systematic path for going from a set of functional requirements to a design that will satisfy those requirements."

In terms of the software development process, it is very good for the clarification and identification of missing, incorrect, or incomplete requirements. The architecture then generated, is clear, concise, and implementation independent. It can be shown that a number of different implementations of the same requirements by different designers, will produce very similar Behavior Trees.

The other advantage for the architectural designer, is that this model can be transformed using techniques of integration, projection, and pruning of individual trees from an overall Behavior Tree.

Please refer to Section A for a detailed specification of this model.

4.6.2 Model Driven Architecture

[40, Siegel and OMG(2001)] and overview by [41, Soley and OMG(2000)] describe the emerging Model Driven Architecture (MDA) technology as a way of integrating different information and systems produced on different operating systems, programming languages, and architectures.

The current main middleware environments today are CORBA, Enterprise JavaBeans, message-oriented middleware, XML/SOAP, COM+ and .NET.

Even though the Object Management Group (OMG) have already produced the CORBA standard that is vendor and system independent, they also realise that middleware standards will continue to change. Companies will also have a need to integrate existing systems, as well as adapt current systems to changing enterprise boundaries, such as from intranet to internet.

MDA is vendor and system independent, and middleware neutral. It uses multiple architecture core models; including the Unified Modeling Language (UML), the Meta-Object Facility (MOF), the Common Warehouse MetaModel (CWM), and XML Metadata Interchange (XMI). Each model will be used to represent different types of structures and requirements, such as enterprise and real-time computing. Refer to Figure 4.10.

[3, ORMSC(2001)] details the benefits of using MDA:

- integration of past, current, and future software
- flexibility despite changing infrastructure
- increase in usable lifetime of software
- interoperability of software between departments, customers and suppliers
- multi-platform industry standards will be more widely used
- each standard can be implemented on the platform that is suits best
- standards will be of higher quality.

An example development process using MDA is described:

1. Create a Platform Independent Model (PIM) using UML.
2. Store the PIM in the MOF and input to mapping step to produce a Platform-Specific Model (PSM) using UML Profiles (e.g. for CORBA).
3. Generate the Application using Java, XML/SOAP etc.

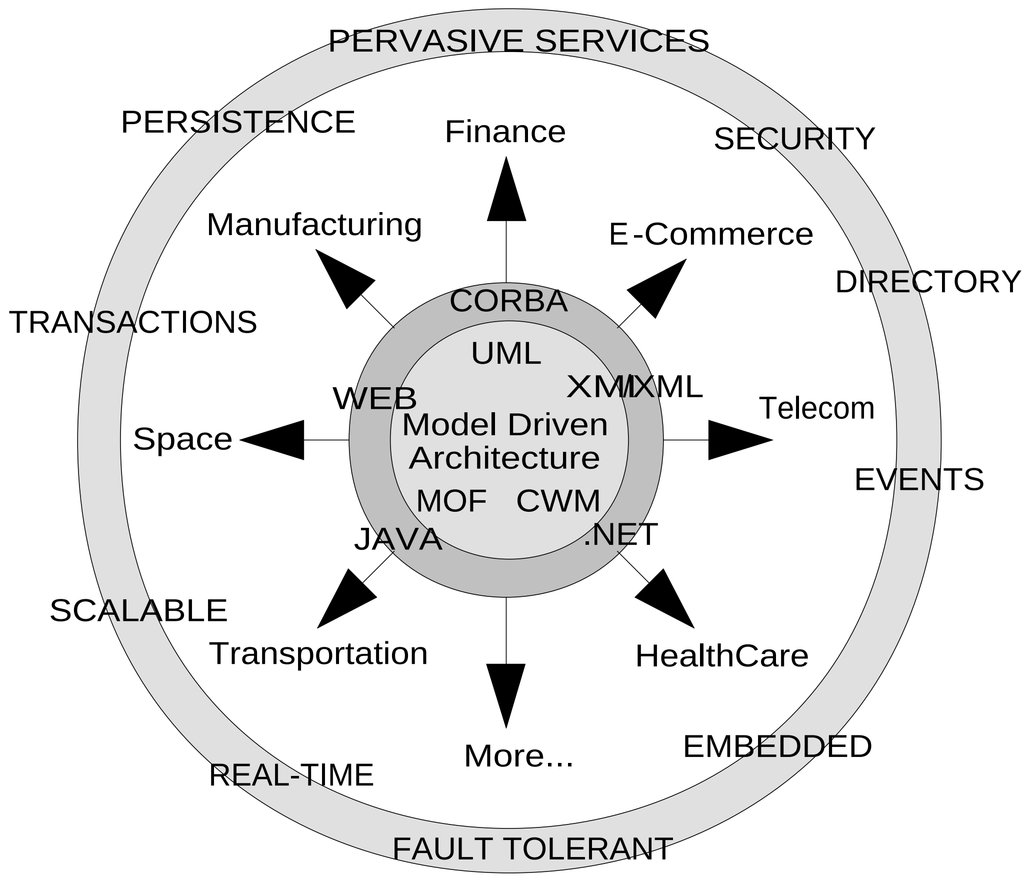


Figure 4.10: MDA showing the pervasive services and specialised computing environments

Model Descriptions

XML Extensible Markup Language, abbreviated XML, describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them. XML is an application profile or restricted form of SGML, the Standard Generalized Markup Language [ISO 8879]. By construction, XML documents are conforming SGML documents.

XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure. [14, Bray *et al.*(2000)]

SOAP is a lightweight protocol for exchange of information in a decentralised, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data-types, and a convention for representing remote procedure calls and responses. SOAP can potentially be used in combination with a variety of other protocols; however, the only bindings defined in this document describe how to use SOAP in combination with HTTP and HTTP Extension Framework. [13, Box *et al.*(2000)]

XMI The main purpose of XMI is to enable easy interchange of meta-data between modelling tools (based on the OMG-UML) and meta-data repositories (OMG-MOF based) in distributed heterogeneous environments. XMI integrates three key industry standards:

1. XML - eXtensible Markup Language, a W3C standard
2. UML - Unified Modelling Language, an OMG modelling standard
3. MOF - Meta Object Facility, an OMG meta-modelling and meta-data repository standard

The integration of these three standards into XMI marries the best of OMG and W3C meta-data and modelling technologies, allowing developers of distributed systems to share object models and other meta-data over the Internet. [27, OMG(2002)]

As derived from [27, OMG(2002) pg.C2] the following Figure 4.11 and XML code listing is an example of using XMI to describe an object hierarchy.

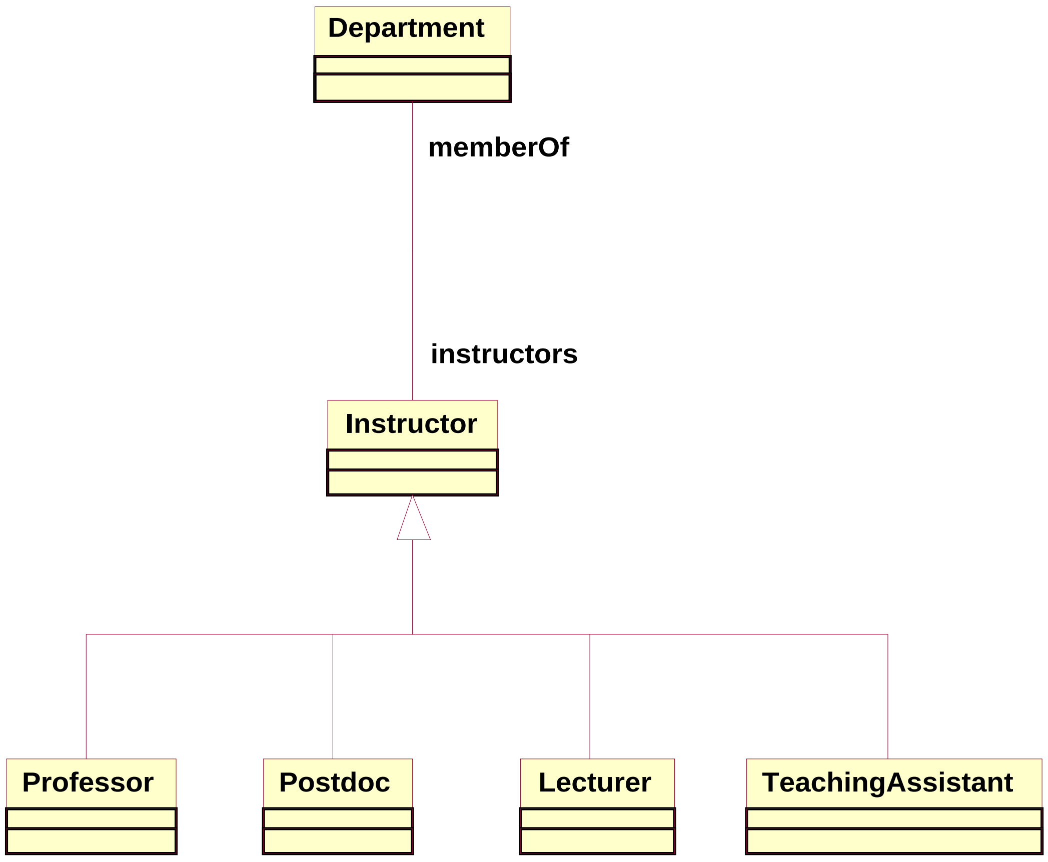


Figure 4.11: Department Relationship Objects using UML

```
File="Department.xml" Namespace="Department": <XMI version="1.1"
         xmlns:UML="org.omg/UML1.3">
   <XMI.header>
      <XMI.model xmi.name="Department" href="Department.xml"/>
      <XMI.metamodel xmi.name="UML" href="UML.xml"/>
   </XMI.header>
   <XMI.content>
      <UML:Class name="Department" xmi.id="Department"/>
      <UML:Class name="Instructor" xmi.id="Instructor"/>
      <UML:Class name="Professor" xmi.id="Professor"
            generalization="Instructor"/>
      <UML:Class name="Postdoc" xmi.id="Postdoc"
            generalization="Instructor"/>
      <UML:Class name="Lecturer" xmi.id="Lecturer"
            generalization="Instructor"/>
      <UML:Class name="TeachingAssistant" xmi.id="TeachingAssistant"
            generalization="Instructor"/>
      <UML:Association>
         <UML:Association.connection>
            <UML:AssociationEnd name="instructors" type="Instructor"/>
            <UML:AssociationEnd name="memberOf" type="Department"/>
         </UML:Association.connection>
       </UML:Association>
   </XMI.content>
</XMI>
```

4.7 Component Certification and Selection

[25, Firesmith and Henderson-Sellers(2001) pg14] states that there has been a goal for many years to build software from components. “Components are replaceable and reusable ‘parts’ from which the application can be directly developed.” Components require standard, clearly specified interfaces to its services offered and resources required. These interfaces are the only way of interacting with the component due to its “black-box” nature. Hence, components must be well tested, reliable, and trusted as they will be integrated with other internal code to produce the system.

Even though industry is focussing on the binary executable side of components, such as with Enterprise JavaBeans and COM+, and how to develop these components from scratch, there is still much work to be done in reusing existing code internal to the organisation, and integrating external third-party code.

This section looks at the process of third-party component certification and selection.

4.7.1 Standards and Product Certification

[2, Apperly *et al.*(2001), Section 42.5] addresses the near-term issues of standardisation of software components. The IEEE have devoted considerable time and resources to software standardisation, but very little to certification that companies adhere to these standards.

Conformity assessments and certification are third-party review activities that evaluate the conformance of development and testing processes, as well as assessment of the products under relevant standards.

[2, Apperly *et al.*(2001)] provides example tables for certification scope (Table 4.2) and concrete documentation requirements for certification activities (Table 4.3).

Certification against a standard assures that specific requirements have been met. Standards also provide the process for constructing software components correctly, and for verification against previous phases. [2, Apperly *et al.*(2001)] also suggests that future component standards, "will specify when and how to record potential risks and their mitigation during the component-based software life-cycle, or in design (ANSI/UL 1998). These standards will establish the means for a component, including an application and OS component, to return to its previous, stable, steady state once it enters a risk-addressed state."

The Unified Modelling Language (UML) is currently being adopted as the standard way to specify software designs, but there is no standard format. Rational Corporation is working on the Reusable Assets Specification to provide a standard format for assets in a software project so that they may be reused for subsequent projects. It uses both UML and textual meta data to define component interfaces, and to define context.

[2, Apperly *et al.*(2001)] states, "the only way to protect investments in costly software components and their elements are to ensure unbiased third-party certification." Standardised component models and associated meta data must be used together, so that consumers can accurately specify and locate components to meet their requirements.

4.7.2 COTS Myths

labelrcsmyth [43, Tracz(2001)] describes a number of misconceptions about creating systems from commercial off-the-shelf (COTS) components, and presents some rules of thumb gained by experience. The following list details the more relevant issues from this dissertation's point of view.

1. It's important to know what COTS components can do to you, not what they can do for you.

2. COTS-based systems are built "bottom-up", not "top-down"

3. There is no standard definition for "open system" for the inter-operability

problem, and “plug-and-play” doesn’t always work.

4. You need to test COTS components more thoroughly because you don’t understand how they were built.

5. You can configure your process to meet the COTS component’s capabilities, not the COTS-based system to meet your requirements. Most integrators never attempt to modify COTS components, and thoroughly understand the requirements.

6. The processes COTS products utilise often only reflect the market schedule and domain experience of the producer, not the current industry best practices.

7. COTS components exacerbate inadequacies in the system development process by compressing the development schedule.

8. The cost of COTS software is 1/100th that of traditional single-use code.

9. You need to evaluate COTS in an environment as close to the operational environment as possible.

10. By using COTS components you decrease development time and increase integration time.

11. The selection of COTS components either creates or mitigates risk.

12. A COTS-based system will never completely satisfy a customer’s needs.

4.7.3 COTS Requirements

[38, SQuaRE(2002)] outlines the requirements for Commercial Off-The-Shelf components with product description, user documentation, and programs and data.

1. Product Description - commercial documentation or packaging

Table 4.2: Certification Scope

1.	Analysis plan
	● Business rules
	● Functional and nonfunctional requirements
	● Use case scenarios and use case diagrams
2.	Various forms of design diagrams
	Development processes and practices, configuration management, and component library management
4.	Evaluation of verification activities
5.	Design plan
6.	Risk analysis and risk mitigation plans
7.	Evaluation of, or participation in, validation activities
	● Assurance that the product under certification actually satisfies the requirements established at the negotiation of the contract or the voluntary business-to-business certification activities for the component or components, the subject of the commerce

Table 4.3: Concrete Documentation According to UL 1998, Sections 3.1-3.4

3.1	A risk analysis shall be conducted to determine:
	a) The set of risk, and
	b) That the software addresses the identified risks
3.2	The risk analysis shall be based on the safety requirement for the programmable component
3.3	An analysis shall be conducted to identify the critical, noncritical, and supervisory sections of the software
3.4	An analysis shall be conducted to identify states for transitions that are capable of resulting in risk

   (a) Availability - product description
   (b) Contents - understandable, complete, ease of overview
   (c) Identification and Indications - unique version and date, supplier, requirements conformance, other COTS interfaces
   (d) Functionality - overview of user-callable functions, options, boundaries, and built-in security
   (e) Reliability - maturity, fault tolerance, recoverability
   (f) Usability - user interface type, knowledge required, tools for adaption, copyright protection
   (g) Efficiency - response time, throughput rate, resource utilisation
   (h) Maintainability - analysability, changeability, stability, testability
   (i) Portability - adaptability, installability, replaceability, and coexistence
   (j) Quality in Use - effectiveness, productivity, safety or satisfaction in specified context of use

2. User Documentation
   (a) Completeness - documentation to use, description of all functions, supplied elements, boundary values, installation
   (b) Correctness - correct and free from ambiguities and errors
   (c) Consistency - free from contradictions
   (d) Understandability - by normal user population
   (e) Ease of Overview - adequate organisation, contents, index

3. Programs and Data - all documented quality attributes in user documentation should be executable
   (a) Functionality - installation, test cases, self-tests, consistency of data and terminology

(b) Reliability - system should not go into uncontrollable state or corrupt data

(c) Usability - understandability, undoes, clarity of execution

(d) Efficiency - only as stated in documentation

(e) Maintainability - only as stated in documentation

(f) Portability - only as stated in documentation

(g) Quality in Use - only as stated in documentation

4.7.4 Quality Attribute Evaluation

[5, Barbacci *et al.*(2001)] states:

- Quality attribute workshops (QAW) provide a method for evaluating the architecture of a software-intensive system during the acquisition phase of major programs.
- architecture is evaluated against a number of critical quality attributes, such as availability, performance, security, inter-operability, and modifiability.
- process of eliciting questions allows stake-holders to communicate directly, thereby exposing assumptions that may not have surfaced during requirements capture.
- report provides a rationale for developing the process and describe it in detail.

[9, Bergey *et al.*(2000)] states:

- software architecture determines the quality attributes of both the software and the entire system.
- It is also one of the earliest artifacts available for evaluation.

- The QAWs provided the acquiring government agency with a means to evaluate each contractors software architectural approach and determine whether it satisfied the systems quality attribute requirements (e.g., performance, inter-operability, security).
- also discusses future opportunities for applying a full-scale architecture evaluation

[6, Barbacci *et al.*(1997)] states:

- software quality is the degree to which software possesses a desired combination of attributes (e.g., reliability, inter-operability).
- describes principles for analysing a software architecture to determine if it exhibits certain quality attributes.
- how analysis techniques indigenous to the various quality attribute communities can provide a foundation for performing software architecture evaluation.
- how the principles provide a context for existing evaluation approaches such as scenarios, questionnaires, checklists, and measurements.
- goal in identifying these principles for attribute-based architecture evaluation is to better integrate existing techniques and metrics into software architecture practice, not necessarily to invent new attribute-specific techniques and metrics.
- A longer-term goal is to codify these principles into systematic procedures or methods for architecture evaluation.

4.7.5 COTS Assessment

[39, Shaw(1996)] states that the conventional doctrine for sufficient, complete, static and homogeneous specifications do not hold for CBSE because developer cannot anticipate all possible future uses of components.

[32, Polze(1999)]

- most of fault-tolerant, real-time systems have been implemented in embedded settings
- Humboldt University has developed the concept of composite objects as a filtering bridge between standard middleware platforms and software frameworks providing services with certain quality-of-service (QoS) guarantees.
- Current research focuses on the CORBA but are also applicable to DCOM and DCE
- concepts in approach are analytic redundancy, non-interference, inter-operability, and adaptive abstraction.

[47, Voas and Payne(1998)] discusses the time-to-market vs. the quality of COTS, and recommends steps for integrators to take before relying on components.

[52, Voas(1998)] states that the arguments for COTS are obvious; required functionality accessed immediately, cheaper, and more expertly implemented. The arguments against include too much functionality, and requirements for longevity; such as Y2K issues. The relationship between the difference in required and actual functionality, with overall quality of system is also discussed.

[48, Voas(1988)] details a methodology for mitigating the potential for dangerous software states to occur.

[44, Voas *et al.*(1996)] presents an assessment technique for studying failure-tolerance of large scale component-based systems, by studying the propagation of information through interfaces in system to determine how much damage will be done by corrupt information.

[57, Voas(2000)] presents a model that generates accurate operational profiles for mass-marketed software with usage-profiling performed by independent organisations, detecting bloat-ware and misused/unused features, and providing information on platforms most executed by users on, user usage-pattern change for new releases. This will provide more accuracy for testing software between releases and improved user manuals and tutorials.

[50, Voas(1996)] provides an assessment of the tolerance of interfaces between components for failures, addressing software maintenance issues.

[55, Voas(1999)] states some of the disadvantages of using COTS components; they are distributed in fixed executable format, their ability to be maintained is reduced, and exposure to damage from malicious components is increased because of hidden code.

[51, Voas(1997)] presents the idea of augmenting software with assertions to improve defect observation qualities.

[56, Voas(1999)] discusses the trust and quality issues of using existing COTS software.

[53, Voas(1998)] presents a consumer-oriented methodology for predicting what impact on system quality a COTS component will have, and outlines ways of safeguarding against COTS software failure.

[46, Voas and Miller(1995)] discusses the difference between traditional verification and testability. Verification tests for likelihood software will fail during use, whereas testability is the new type of verification for testing whether an incorrect program will not fail; i.e testing for hidden faults.

[45, Voas and Miller(1992)] discusses the tendency of code to reveal existing faults during random testing, and quantifies the likelihood that software faults are not hiding after testing.

[54, Voas(1998)] details the maintenance issues of components developed and maintained by different people.

[49, Voas(1996)] introduces the variety of behavioral software quality characteristics, other than “black and white” correctness. They do not replace traditional tests but test additionally for safety/failure-tolerance, and vulnerability/security.

4.7.6 Component Evaluation

[34, Seacord *et al.*(2000)] describes how component reuse suffers from the inability of system integrators to effectively identify ensembles of compatible software components that can be easily integrated into a system. An automated process is developed for identifying component ensembles that satisfy a system requirements specification; and for ranking these ensembles based on a knowledge base of system integration rules.

Chapter 5

Problem Solution

This section will describe how the problem defined in Section 3.1 was addressed.

The primary issue to address is how to apply quality attributes to the software development process. Section 4.1 has demonstrated the need for a complete and consistent set of functional and non-functional requirements to ensure product quality. Section 4.4 identifies that also quality processes and technologies are required within an organisation to ensure product quality.

The need for quality attribute identification, and a trade-off analysis is consistent throughout the literature review. What is not consistent, is how non-functional requirements and quality attributes are categorised. [26, Gilb(1988)] identifies an incomplete list of quality attributes with no categorisation. [7, Bass *et al.*(1998)] identifies which attributes are more important to the user or developer. [36, SQuaRE(2002)] is trying to address this point by working with the concept of a changing viewpoint of quality. The requirements of the user, such as responsiveness are translated into external product requirements such as reliability and then into internal development characteristics such as portability. This seems to be an intelligent approach to how quality actually is used, but whether the emerging standard can be released in a timely and usable manner are two issues.

The initial approach of this dissertation was to define and categorise a set quality attributes applicable to our purposes. This was decided against, as most references are derived from the same primary sources, and [35, ISO 9126(1991)] and [36, SQuaRE(2002)] are well respected sources within the industry.

In terms of this Solution section, the process for using quality attributes will be clarified. Ultimately this work will refer to the work being done in [36, SQuaRE(2002)].

The next objective, will be to define the process for software component development and reuse, derived from Sections 4.5 and 4.7. This will allow the dissertation to identify and address some of the primary issues with component reuse. These include:

- COTS-based systems are black-box, and are not designed to be modified. Evaluation and testing of components must be done in an environment as close to operational as possible. Hence difficulty in true 3rd-party certification.

- COTS-based systems are designed and built “bottom-up”, not “top-down”. Hence they are generally not designed to fit into your processes, or your specific application. You must adapt your processes to allow for the component’s.

- By using COTS components you decrease development time and increase integration time. Effort must be spent creating wrapper and glue-code to interface the component to your system.

- There is a lack of standardisation of the component-based development process, 3rd-party COTS components, and certification.

By defining the process for selecting and adapting components for a system, a technique for embedding quality attribute and architectural information is suggested. This information could be attached as XML data along with the binary executable of the component, assisting 3rd-party certification. Integrity and performance of the code would not be violated. However the means of being able to test and evaluate the component’s behaviour and quality attributes through its architecture are a definite advantage.

Section 4.5 suggest the software architecture is the best place to determine quality attributes. There is an obvious time and cost benefit in identifying issues during the architectural phase over the implementation phase. Additionally, [7, Bass *et al.*(1998)] asserts that quality primarily is to be found in the architecture, and that a developer must be careful when integrating components into their system, that they do not compromise the quality inherit to this architecture. Most quality attributes are visible the architectural level.

This introduces the main scope of work in this Solution Section; the process for creating, adapting, and identifying quality attributes within a software architecture. The Behavior Tree model will be used represent a software architecture description. It provides a clear methodology for representing and validating functional requirements within an architecture. Quality attributes generally relate to the overall behaviour of a system, not just some individual functionality. The primary representation of Behavior Trees is concerned with the dynamic behaviour of the system, so inherently they are related. The model also provides many techniques for transforming the architecture, by pruning, projection, and integration, making this model easier to use and adapt for this dissertation.

5.1 Quality Attributes Model

This section will describe in detail a model for the description and categorisation of quality attributes within a software components. Primarily this will describe how quality attribute specification fits into the software development process. Section 5.2 will describe the component development process.

5.1.1 The Software Development Process

[17, ISO/IEC 12207(1995)], in part, provides an industry standard for the software development process. It provides models for the activities shown in Figure 5.1, as well as component detailed design, external interface detailed design, database detailed design, unit test requirements definition.

This dissertation is concerned with modelling the process from software requirements analysis to software architectural design, as justified in Section 3.2. The remainder of this section is an extract from [17, ISO/IEC 12207(1995)] detailing the activities and tasks associated with software requirements analysis and architectural design.

For Software requirements analysis, the following tasks are described:

- The developer shall establish and document software requirements, including the quality characteristics specifications, described below. Guidance for specifying quality characteristics may be found in ISO/IEC 9126.

  1. Functional and capability specifications, including performance, physical characteristics, and environmental conditions under which the software item is to perform;
  2. Interfaces external to the software item;
  3. Qualification requirements;
  4. Safety specifications, including those related to methods of operation and maintenance, environmental influences, and personnel injury;

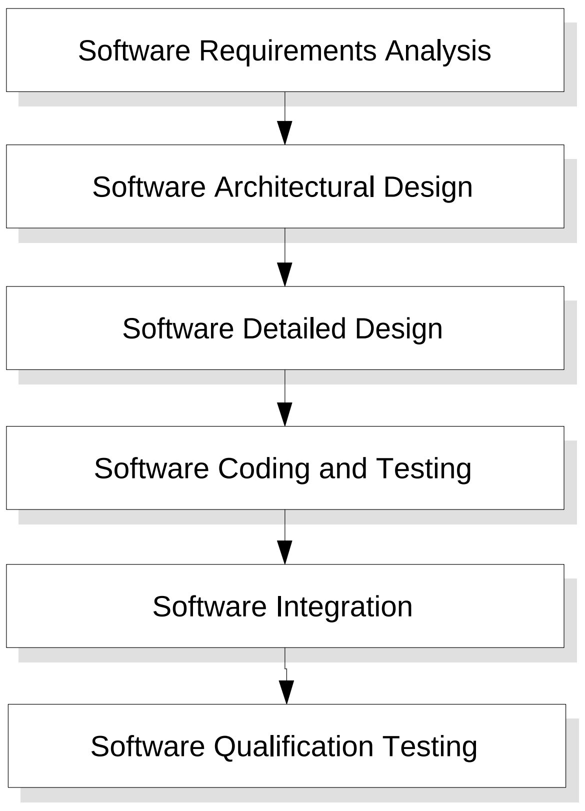

Software Requirements Analysis
Software Architectural Design
Software Detailed Design
Software Coding and Testing
Software Integration
Software Qualification Testing

Figure 5.1: Software Development Activities

5. Security specifications, including those related to compromise of sensitive information;
6. Human-factors engineering (ergonomics), including those related to manual operations, human-equipment interactions, constraints on personnel, and areas needing concentrated human attention, that are sensitive to human errors and training;
7. Data definition and database requirements;
8. Installation and acceptance requirements of the delivered software product at the operation and maintenance site(s);
9. User documentation;
10. User operation and execution requirements;
11. User maintenance requirements.

- The developer shall evaluate the software requirements considering the criteria listed below. The results of the evaluations shall be documented.

    1. Traceability to system requirements and system design;
    2. External consistency with system requirements;
    3. Internal consistency;
    4. Testability;
    5. Feasibility of software design;
    6. Feasibility of operation and maintenance.

- The developer shall conduct joint review(s) in accordance with 6.6. Upon successful completion of the review(s), a baseline for the requirements of the software item shall be established.

For the software architectural design activity, the following tasks are defined:

- The developer shall transform the requirements for the software item into an architecture that describes its top-level structure and identifies the software components. It shall be ensured that all the requirements for the software item are allocated to its software components and further refined to facilitate detailed design. The architecture of the software item shall be documented.
- The developer shall develop and document a top-level design for the interfaces external to the software item and between the software components of the software item.
- The developer shall develop and document a top-level design for the database.
- The developer should develop and document preliminary versions of user documentation.
- The developer shall define and document preliminary test requirements and the schedule for Software Integration.
- The developer shall evaluate the architecture of the software item and the interface and database designs considering the criteria listed below. The results of the evaluations shall be documented.
    1. Traceability to the requirements of the software item;
    2. External consistency with the requirements of the software item;
    3. Internal consistency between the software components;
    4. Appropriateness of design methods and standards used;
    5. Feasibility of detailed design;
    6. Feasibility of operation and maintenance.

5.1.2 Software Requirements Categorisation

Ways of defining and measuring individual attributes are standard now, as shown in the research sections. The long-standing issue of using quality non-functional

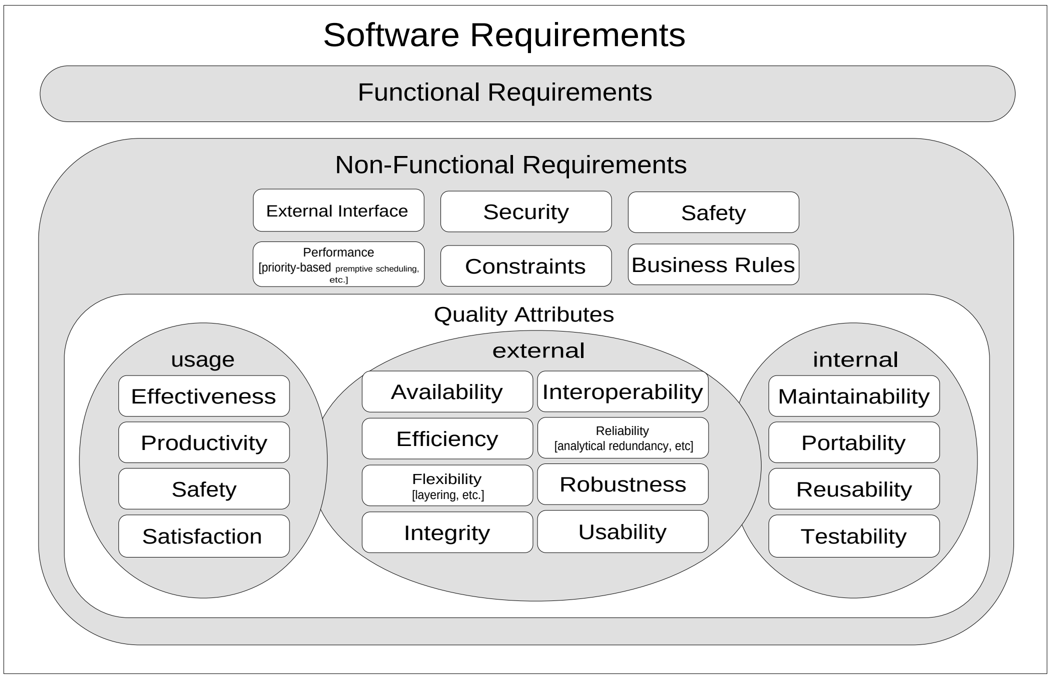


Figure 5.2: Categories of Software Requirements

requirements is how they are categorised and sub-categorised.

As shown in Figure 5.2, software requirements are generally categorised into functional and non-functional requirements as qualified in Section 4.1. Quality attributes are a sub-class of non-functional requirements.

The Software Requirements Specification Document (IEEE 830) provides a standard way of documenting all the requirements for a product. In terms of requirements analysis, defined in 5.1.1, a more detailed way of categorising and analysing software requirements is needed. ISO 9126, is recommended, but there is a new standard, ISO 25000 Software Quality Requirements and Evaluation (SQuaRE) forthcoming.

The emerging SQuaRE standard provides a way of grouping these attributes which differs from earlier references. It uses changing viewpoints (user, external, and internal) and changing attribute relationships between viewpoints. It attempts to capture the dynamic nature of attributes and their use during the development

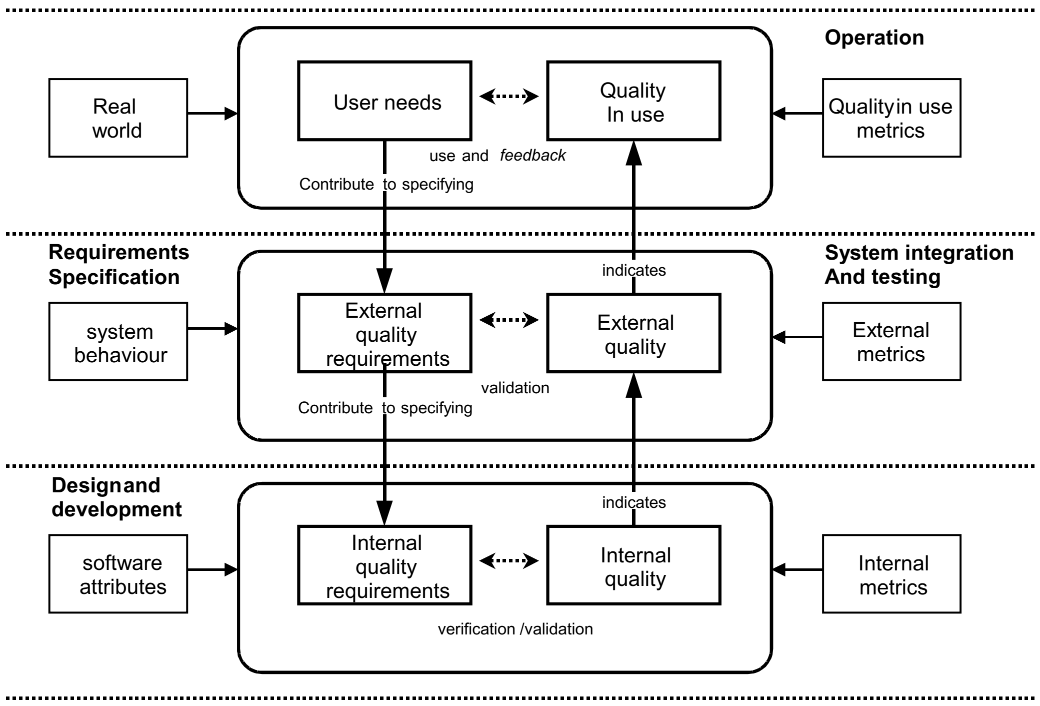


Figure 5.3: Quality in the software life-cycle

lifecycle.

As shown in the Quality in the Software Lifecycle diagram (Figure 5.3), derived from [36, SQuaRE(2002)] in Section 4.2.5; a set of user requirements or needs is translated to external quality requirements at the requirements specification stage, and internal quality requirements at the design and development phase.

The categorisation of individual quality attributes in shown in Figures 5.4 and 5.5.

There are issues with categorisation:

1. incompleteness: all references never profess to include all quality attributes, only the most important ones.

2. complexity: all attributes have complex relationships between themselves, sometime direct or indirect. Hard-line categorisation tends not to reflect these relationships.

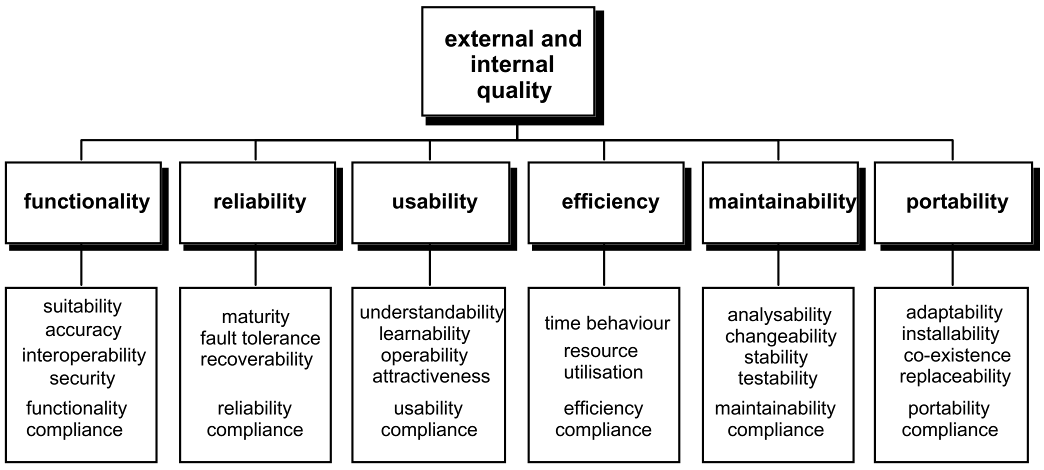


Figure 5.4: Model for internal and external quality

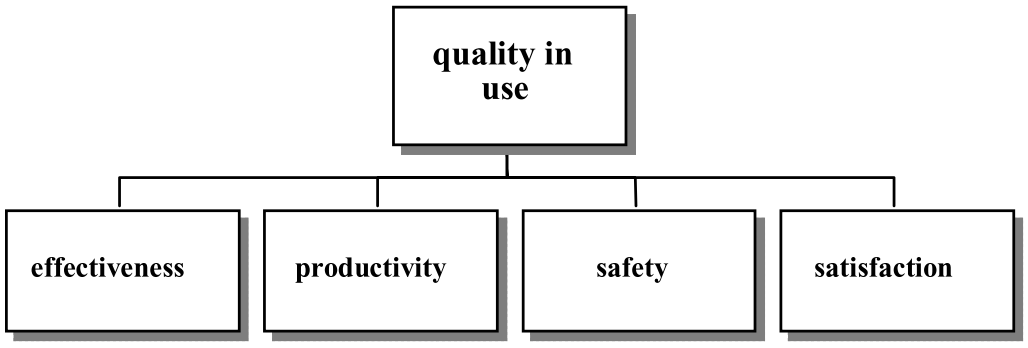


Figure 5.5: Model for quality in use

3. conflicting categories: Safety is often modelled as both a non-functional requirement as well as a quality attribute. Performance is only a non-functional requirement, not a quality attribute.

Ultimately the SQuaRE standard provides a new, detailed look at product quality and metrics, and guides for use. When actually published, it should address most of the issues found to some extent.

5.1.3 Quality Attribute Specification

This section provides an overview of the definitions, and trade-offs for a number of individual quality attributes. Categorisation issues using the SQuaRE standard have been discussed in Section 5.1.2. The purpose of this section was to show a summary of how quality attributes are defined within the literature review. This dissertation will use these definitions as an initial point for identifying quality attributes within a software architecture (Section 5.4). Defining metrics for these attributes is being created in the SQuaRE standard, and is outside of the scope of this dissertation.

- Availability
  - Definition: percentage of planned uptime during which system is actually available for use and fully operational. How much a system is usefully available to perform the work which it was designed to do
  - Context: important primarily to users and critical for mainframe systems
  - Trade-off Relationships:
    - Reliability – direct
    - Robustness – direct
- Efficiency

  - Definition: how well the system utilises processor capacity, disk space, memory, or communication bandwidth
  - Context: important primarily to users and critical for embedded systems
  - Trade-off Relationships:
    * Flexibility – inverse
    * Interoperability – inverse
    * Maintainability – inverse
    * Portability – inverse
    * Reliability – inverse
    * Robustness – inverse
    * Testability – inverse
    * Usability – inverse

- Flexibility
  - Definition: how much effort is needed to add new capabilities to the product
  - Context: important primarily to users
  - Trade-off Relationships:
    * Efficiency – inverse
    * Integrity – inverse
    * Maintainability – direct
    * Portability – direct
    * Reliability – direct
    * Testability – direct

- Integrity (Security)

  - Definition: trustworthiness of the system to be in the right state, with security intact
  - Context: important primarily to users
  - Trade-off Relationships:
    - Efficiency – inverse
    - Interoperability – inverse
    - Reusability – inverse
    - Testability – inverse
    - Usability – inverse

- Interoperability
  - Definition: how easily the product can exchange data or services with other systems. The ability to communicate easily with other assets, on different platforms and machines, and be integrated into an application.
  - Context: important primarily to users
  - Trade-off Relationships:
    - Efficiency – inverse
    - Flexibility – direct
    - Integrity – inverse
    - Portability – direct

- Reliability
  - Definition: measure of the degree to which the system does what it is intended to do, as opposed to something else. The probability of software executing without failure for a specified period of time. Percentage of correctly performed operations. The length of time the system runs before revealing a new defect
  - Context: important primarily to users

  - Trade-off Relationships:
    * Availability – direct
    * Efficiency – inverse
    * Flexibility – direct
    * Maintainability – direct
    * Robustness – direct
    * Testability – direct
    * Usability – direct

- Robustness
  - Definition: degree to which a system or component continues to function correctly when confronted with invalid input data, defects in connected software or hardware components, or unexpected operating conditions.
  - Context: important primarily to users
  - Trade-off Relationships:
    * Availability – direct
    * Efficiency – inverse
    * Reliability – direct
    * Usability – direct

- Usability
  - Definition: how well people are going to be able and be motivated to use the system practically.
  - Context: important primarily to users
  - Trade-off Relationships:
    * Efficiency – inverse
    * Robustness – direct

    * Testability – inverse

- Maintainability
  - Definition: how quickly an unreliable system can be brought to a reliable state
  - Context: important primarily to developers and critical for mainframe systems
  - Trade-off Relationships:
    * Availability – direct
    * Efficiency – inverse
    * Flexibility – direct
    * Reliability – direct
    * Testability – direct
- Portability
  - Definition: the ease of moving a system from one environment to another. The effort required to migrate a piece of software from one operating environment to another. The ability to run on different platforms.
  - Context: important primarily to developers
  - Trade-off Relationships:
    * Efficiency – inverse
    * Flexibility – direct
    * Interoperability – direct
    * Maintainability – inverse
    * Reusability – direct
    * Testability – direct
    * Usability – inverse

- Reusability
  - Definition: extent to which a software component can be used in applications other than the one for which it was initially developed.
  - Context: important primarily to developers
  - Trade-off Relationships:
    - Efficiency – inverse
    - Flexibility – direct
    - Integrity – inverse
    - Interoperability – direct
    - Maintainability – direct
    - Portability – direct
    - Reliability – inverse
    - Testability – direct
- Testability
  - Definition: ease with which the software components or integrated product can be tested to find defects.
  - Context: important primarily to developers
  - Trade-off Relationships:
    - Availability – direct
    - Efficiency – inverse
    - Flexibility – direct
    - Maintainability – direct
    - Reliability – direct
    - Usability – direct

Other Quality Attributes

- Extensive Use of Legacy Systems – careful definition of appropriate integration mechanisms
- Process Capacity - ability to process units of work in units of time
- Responsiveness - reaction to a single event
- Storage Capacity - capacity of a part of a system to store units of a defined thing
- Improvability - efficiency of making minor adaptations, changes and improvements to the system
- Extendability - the ease of adding new factors to an existing system
- Buildability – ease of constructing a desired system on-time within given resources
- Modifiability – modularised, encapsulating components
- Integrability – consistent component interfaces; uses relation

Non-Functional Requirements

- Performance – inter-component communication to exploit parallelism - Discernable while the system executes
- Security – specialised components such as secure kernels or authentication servers - Discernable while the system executes. Since a component contains executable code and is reused as a black box among a community of developers, the reuser must be able to control the origin of the asset, and its access to private resources.
- Resource Attributes - costs of existence (development, use and maintenance) of a system

- Business Attributes
  - Time-to-Market – pressure to use COTS
  - Cost – ability to use assets from in-house
  - Projected Lifetime of the System – modifiability and portability issues - Business qualities affected by the architecture
  - Targeted Market – portability, functionality, as well as performance, reliability, and usability on various platforms
  - Rollout Schedule – flexibility and customisability if product to be released only with base functionality

5.2 Component Development

Extensive use of COTS and product-line components within systems will effect each of the traditional software development phases. Phases such as requirements specification, design, coding, and integration will be changed. Some important component-based activities such as encapsulation, have no equivalent in the traditional lifecycle. Hence [15, Carney(1997)] from SEI outlines a process for software component development and re-use. (Figure 5.6)

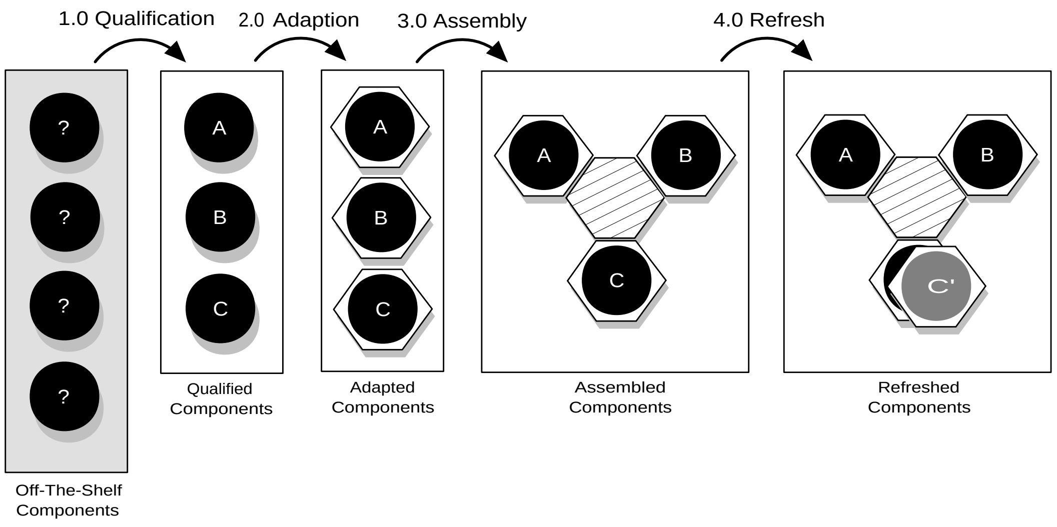


Figure 5.6: COTS System Integration Life-cycle

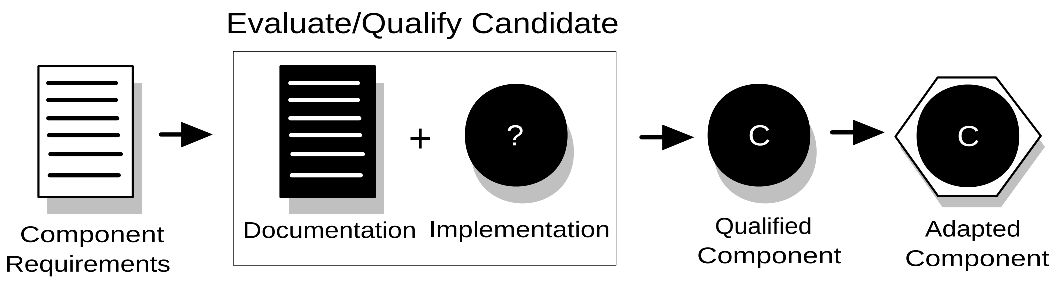


Figure 5.7: COTS Component Qualification Phase

Figure 5.7 illustrates the first activity of the component development lidecycle.

Selecting and qualifying appropriate COTS products means surveying the marketplace in anticipation of the new system development. This in itself is an issue, when there is no one standard for developing and distributing products.

This dissertation presents a variation of this qualification process, (Figure 5.8) which specifically provides more information presented in a standard form. This is an initial attempt to present standardisation, as well as the analytical information required by a 3rd-party system integrator.

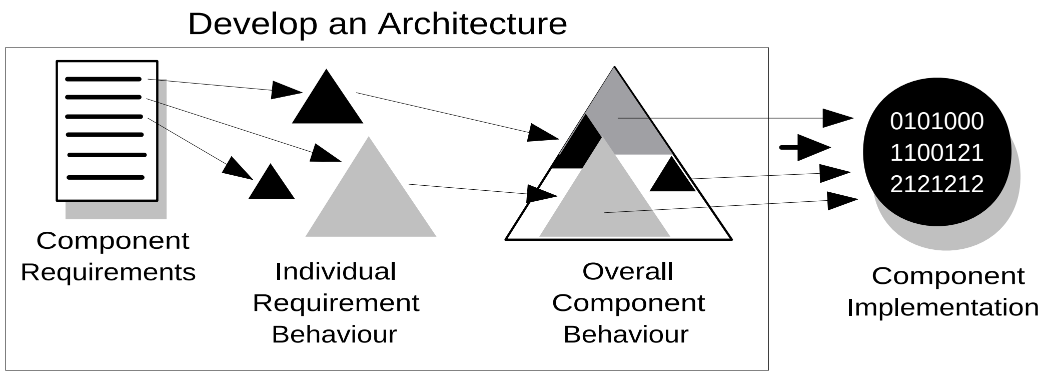


Figure 5.8: COTS Component Architecture Development

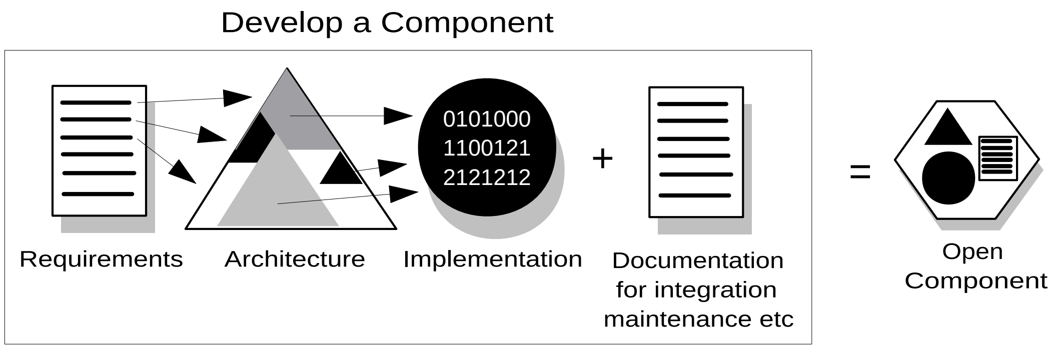


Figure 5.9: COTS Component Augmentation

Figures 5.8 and 5.9 demonstrate a way of including the original requirements specification and the derived architecture description, along with the binary implementation of the component. Current practice is to only include a basic specifica-

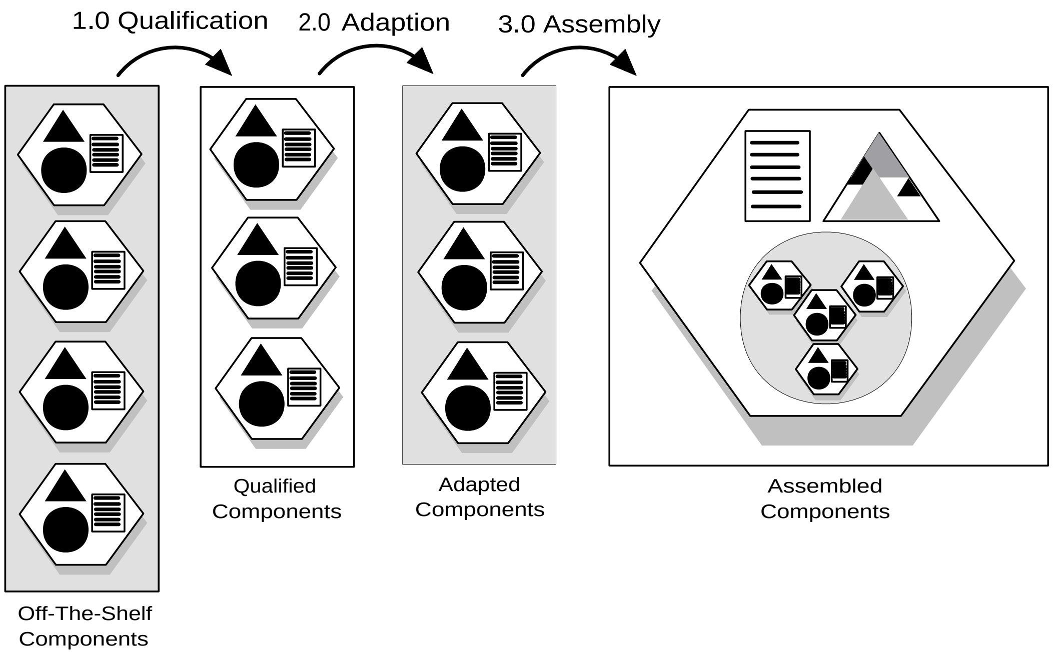


Figure 5.10: Updated COTS Life-cycle

tion and interface guide with the component. Quality attribute specifications and other vital information such as risk mitigation strategies tend not to be included. There are valid reasons why a software supplier does not wish to reveal the contents of their source code, but by at least providing the architecture description, some high level validation of the system can be performed for 3rd-party certification. A system that is required to be safety-critical does not just have to take the supplier's word that it meets its requirements.

The updated lifecycle model shown in Figure 5.10 displays one advantage of providing quality information along with the component; the selection, evaluation, and adaptation phases are so much clearer. Figure 5.11 demonstrates a way of including requirements, and Behavior Tree architecture data using the standard XML Schema format. This will allow 3rd-party tools to automatically analysis and evaluate the static and dynamic data within a component prior to implementation.

Lack of standardisation and certification remains the biggest issue within the

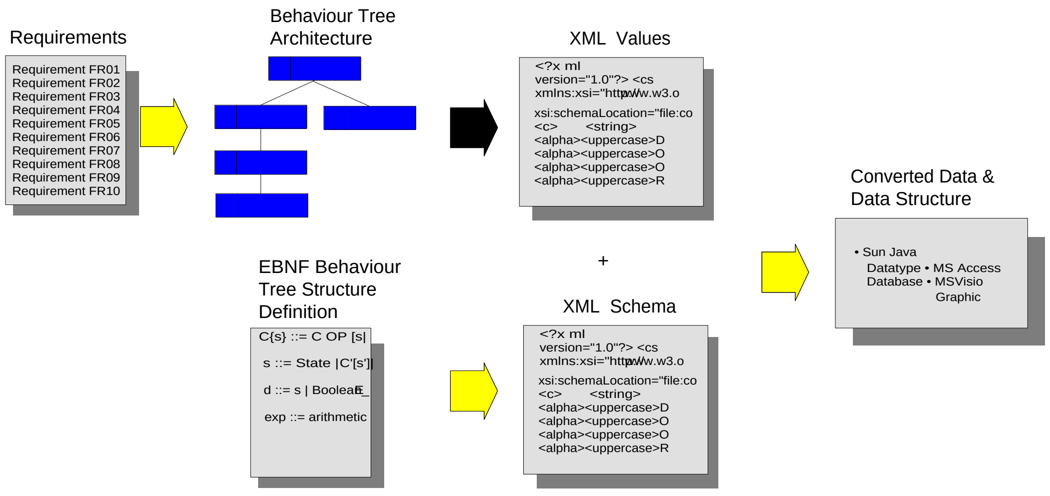


Figure 5.11: XML Specification of Requirements and Architecture

software component industry today. In particular domains, there can be a standard practice and specification for components. This dissertation has merely identified a need for further work. With this in mind, the model proposed is not intended to replace any of the standards in use by the industry today. It only serves to demonstrate that it is possible to provide more useful information which will benefit all people involved in the component development and re-use industry.

The software architecture is the first key artefact of the development process. As this information is required for our proposed model, the next section will examine the process of generating a Behavior Tree architecture description from a set of functional requirements. Later these architectures, and hence the component as a whole, will be analysed and adapted with respect to certain quality attribute requirements.

5.3 Component Development Example

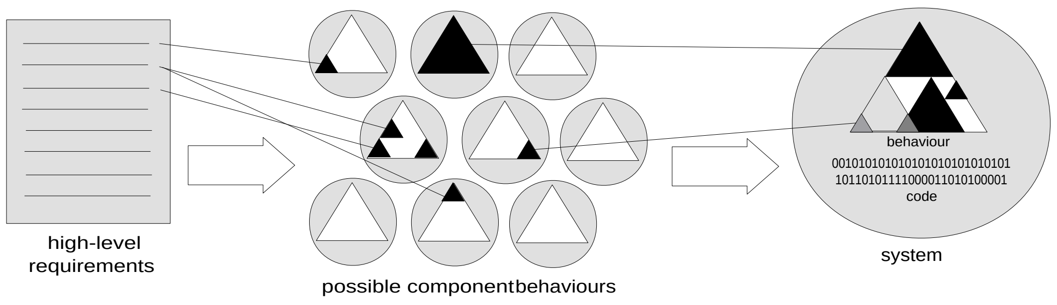


Figure 5.12: Component Specification and Composition

This example will take an initial set of requirements for ways of storing and accessing collections of resources, and produce a number of architectural descriptions. These descriptions will be generalised and combined into software components, that will be reused in a later case-study.

See Appendix C for an example implementation of the generic collection system using Java 1.4.0.

From [28, Lambert and Osborne(2000)] there are four broad categories of collections; linear, hierarchical, graph, and unordered. For this component example a linear queue and stack, and an unordered set will be implemented. In the industry there is no one standard for the naming and implementation of these collections. The software requirements we will follow are derived from the conventions of the Java Software Development Kit.

As an overview:

Stack Each item, except the first and last, has a unique predecessor and successor. All items are ordered, but not sorted. Items can be accessed from end only using operations called *push* and *pop*. e.g. a stack of dinner plates can only be added to and removed from the top plate.

Queue Items are inserted at one end, and removed at the other end using the operations *queue* and *dequeue*. e.g. waiting in line for a bank teller.

Set Items are stored in no particular order, but you can retrieve an item by a unique characteristic such as id or registration number. No matter how many times you add an item to a set, it will never contain more than one reference to that item. e.g you can be only registered once in a hospital at any given time.

5.3.1 Stack

This section lists all the functional requirements for a stack-type linear collection. These requirements will be used as the basis of the architecture descriptions in shown in Figures 5.13, 5.17, and 5.21. A basic stack is augmented with requirements for external operator control of availability, as well as concurrency.

Basic Stack

The following list details the functional requirements for the Basic Stack. Please refer to Figure 5.13 for the Behavior Tree description of this architecture.

FR-01 Resources added to and removed from a stack will follow the process of "first-in, last-out".

FR-02 The stack will be able to keep track of the total number of resources currently in its stack, knowing when it is full or empty. The maximum size of the stack is a pre-defined constant for the system.

FR-03 The stack must be not full, for a resource producer to be able to put (push) a new resource on the stack.

FR-04 The stack must be not empty, for a resource consumer to be able to take (pop) a resource off the stack.

FR-05 The stack is initially has no resources, and is able to accept at least one resource.

Stack with Availability Control

In addition to the requirements specified for the basic stack, these requirements allow specific blocking of the addition and removal processes by an external Operator component. This will allow the stack to be gracefully shutdown when requested, by first blocking all new resource additions, and when the stack is finally empty, the stack will then become unavailable.

The following list details the functional requirements for the Stack with Availability Control. Please refer to Figure 5.17 for the Behavior Tree description of this architecture.

FR-06 An Operator must make the stack available before additions and removals can commence. When the stack is available, the Operator can request that the stack shutdown; initially halting any new resource additions, and when the stack is empty, no further removals will be allowed, as the stack becomes unavailable.

FR-07 The stack must be available for additions, before any new resources can be added to the stack.

FR-08 The stack must be available for removals, before any resources can be removed from the stack.

Stack with Concurrency

In addition to the requirements specified for the basic stack and stack with availability control, these requirements make the stack's addition, removal, and control processes multi-threaded, with mutual exclusion regions. As a demonstration of availability, this will allow simultaneous additions, and removals by multiple producers and consumers.

The following list details the functional requirements for the Concurrent Stack. Please refer to Figure 5.21 for the Behavior Tree description of this architecture.

FR-09 To ensure integrity of data, while a resource is actually being added, or removed, and the stack size is being calculated, no concurrent thread may interrupt (preempt) this operation.

FR-10 The addition, removal, and operator shutdown processes must be implemented as concurrent threads of operations, for high availability.

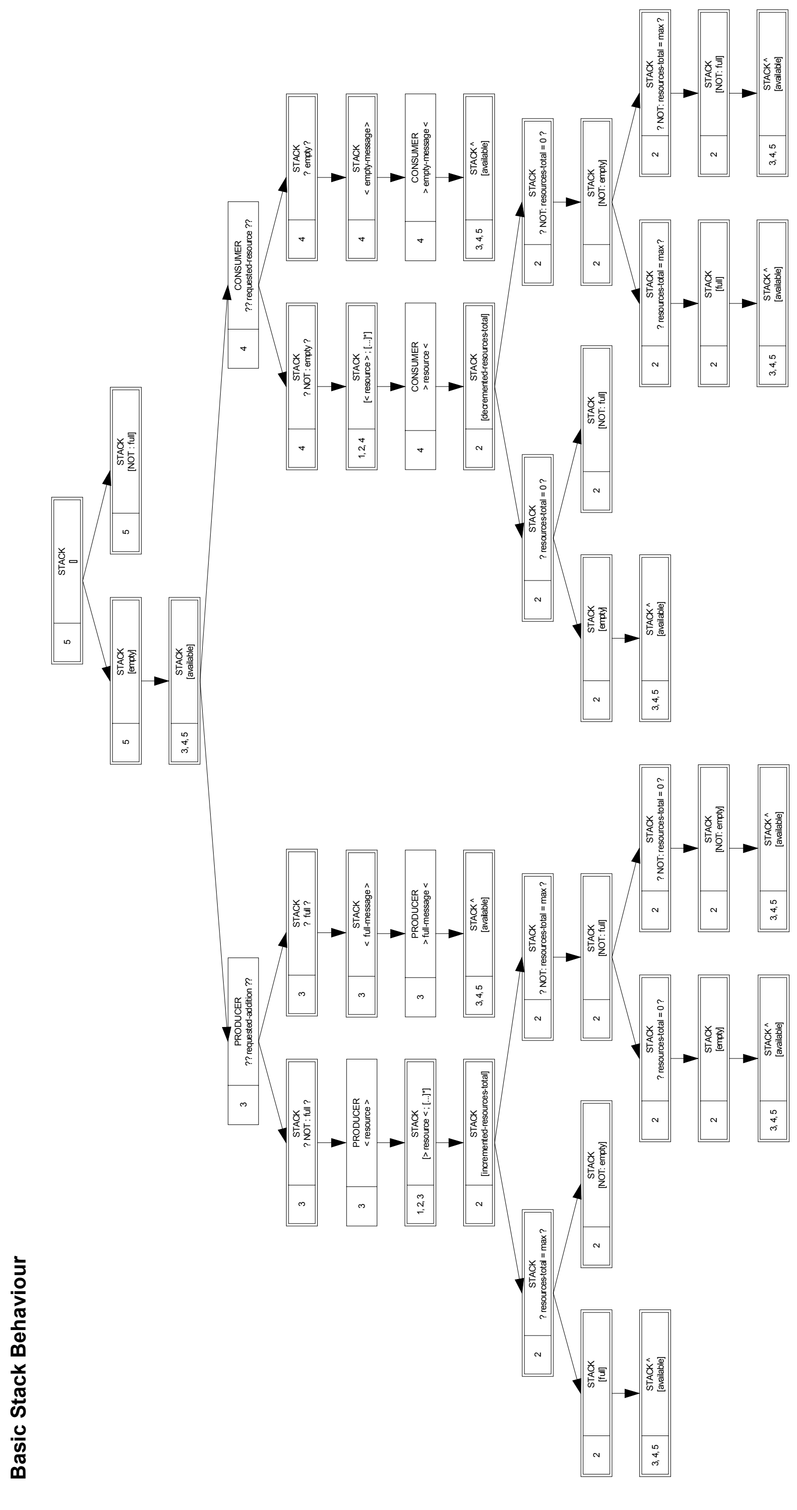


Figure 5.13: Basic Stack Behaviour

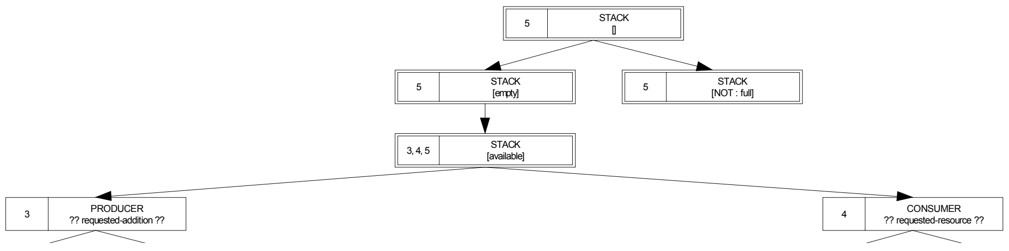


Figure 5.14: Initialisation Part of Stack

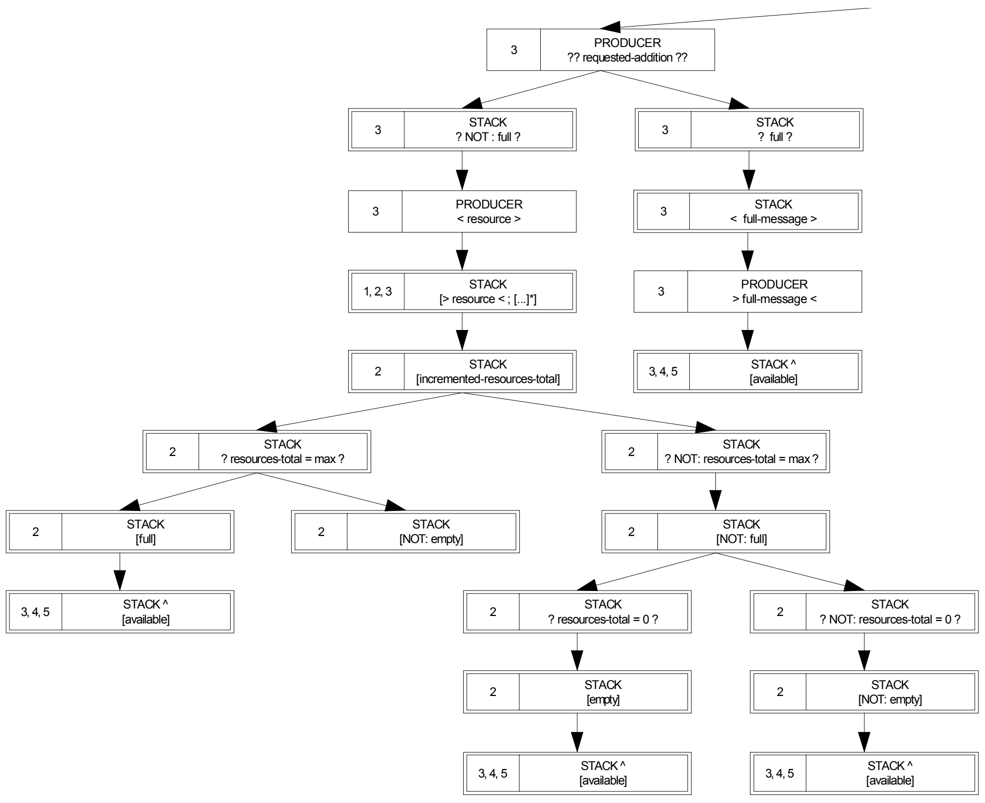


Figure 5.15: Producer Process for Stack

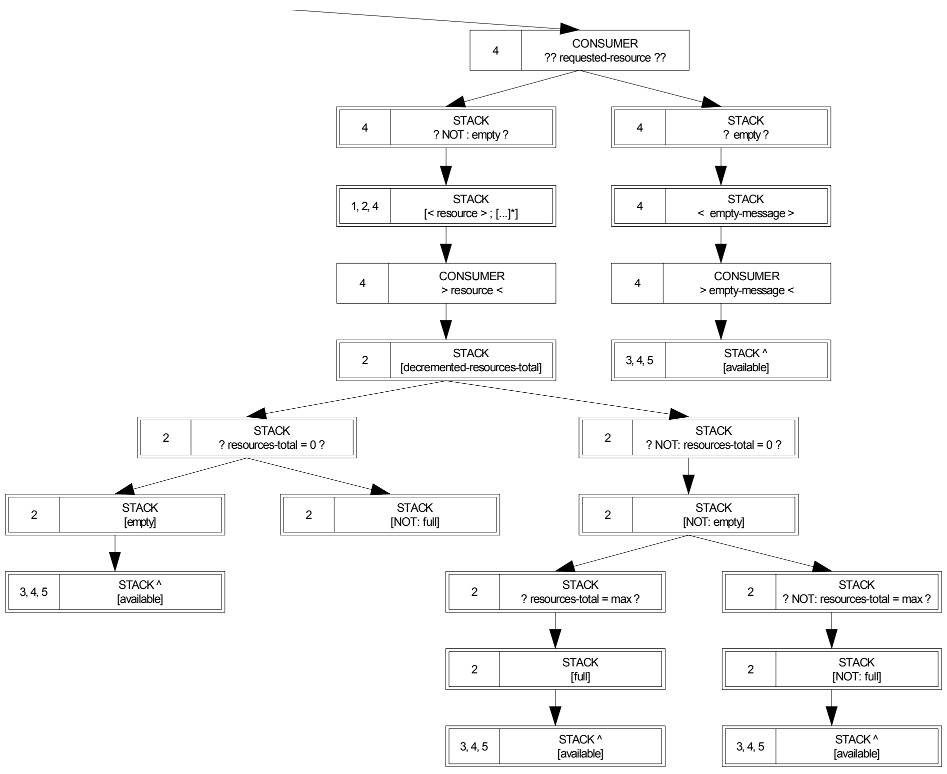


Figure 5.16: Consumer Process for Stack

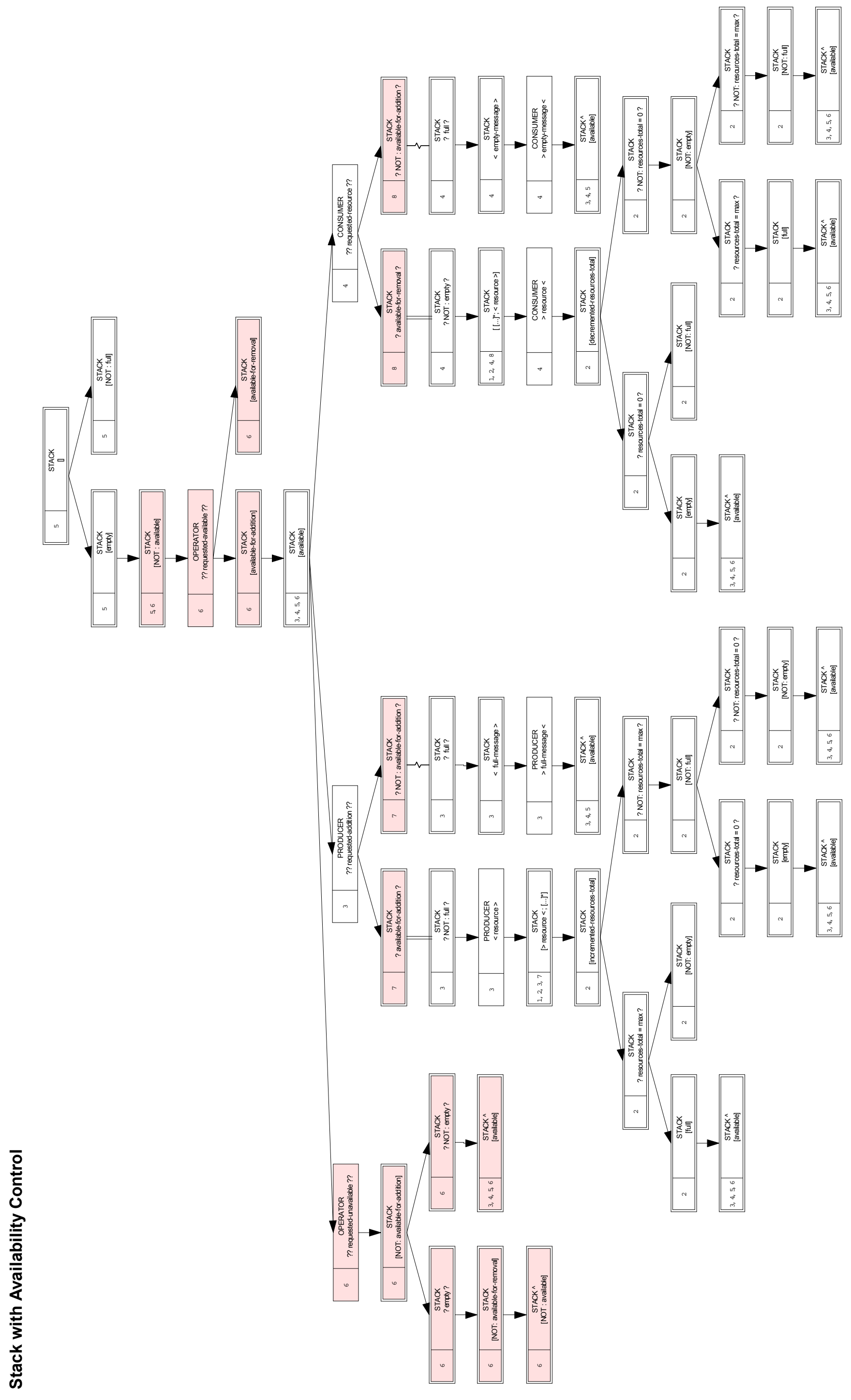


Figure 5.17: Stack Behaviour with External Operator Control Augmentation

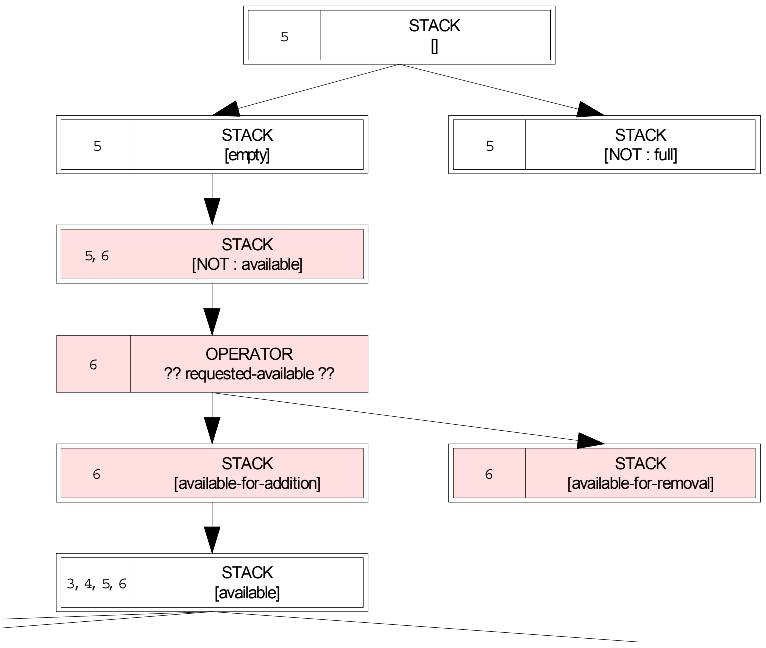


Figure 5.18: Updated Initialisation Part of Stack

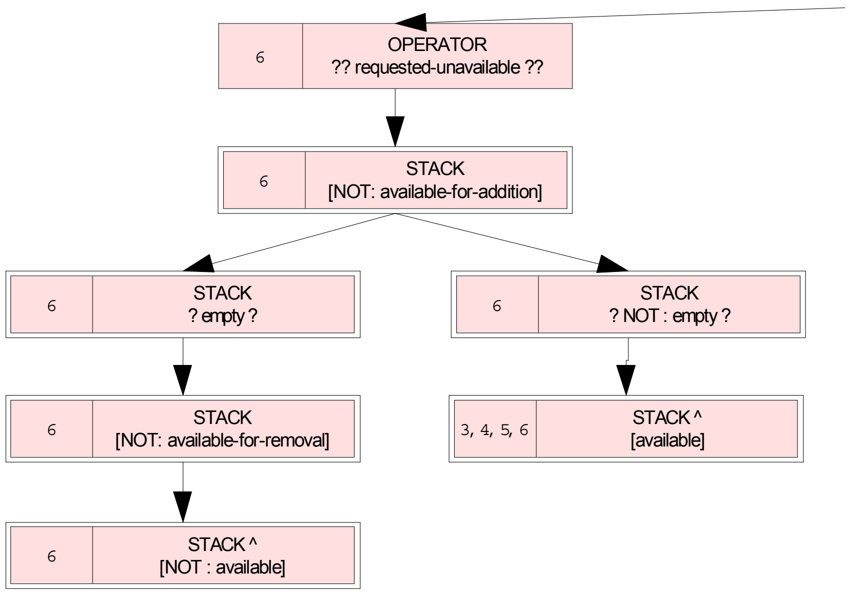


Figure 5.19: Additional Operator Control Process for Stack

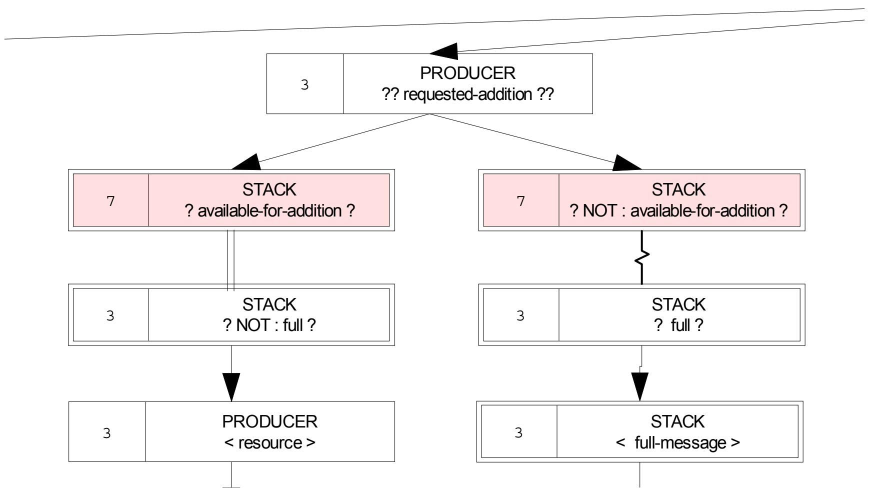


Figure 5.20: Alterations to Producer Process

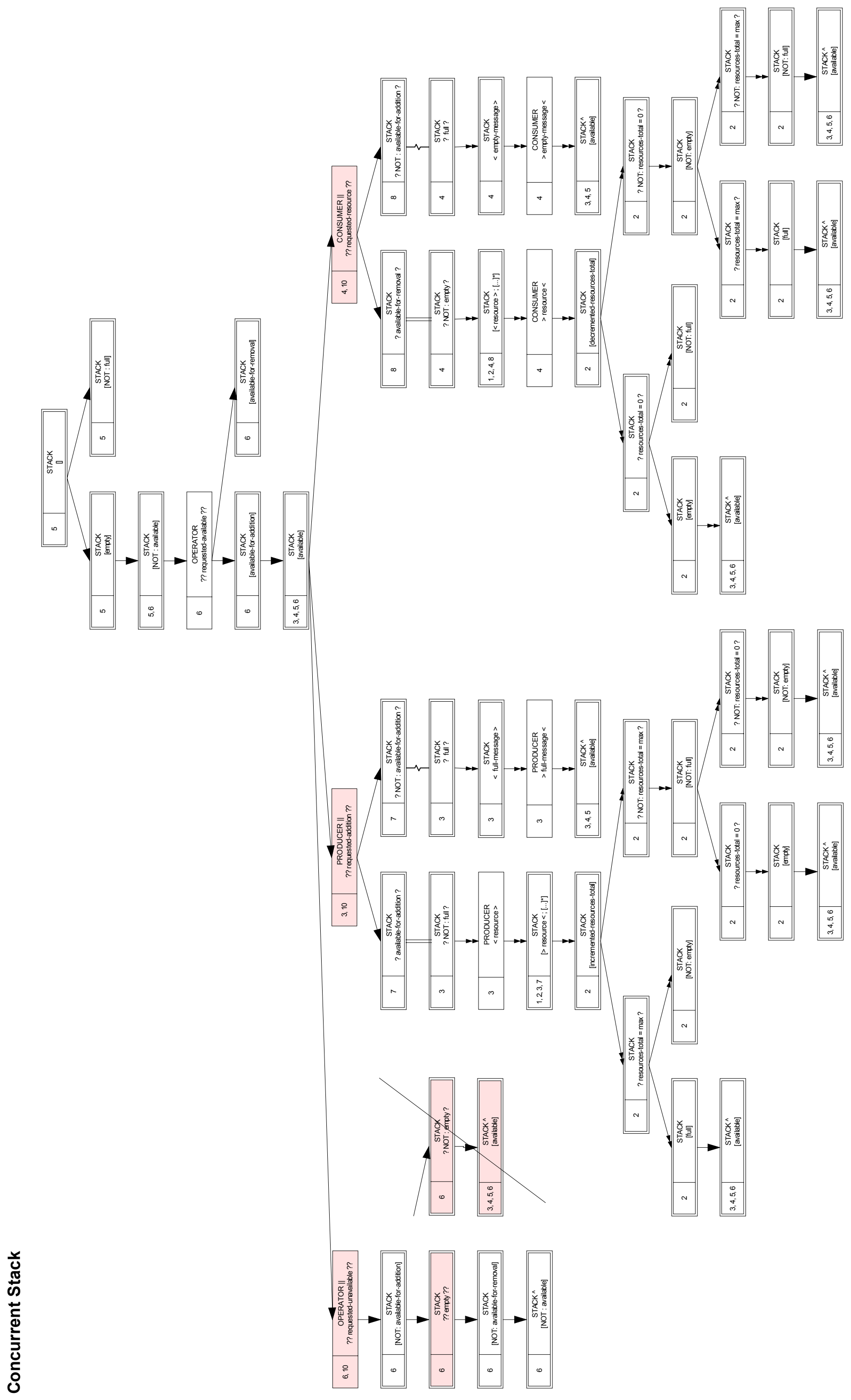


Figure 5.21: Stack Behaviour with Concurrency Augmentation

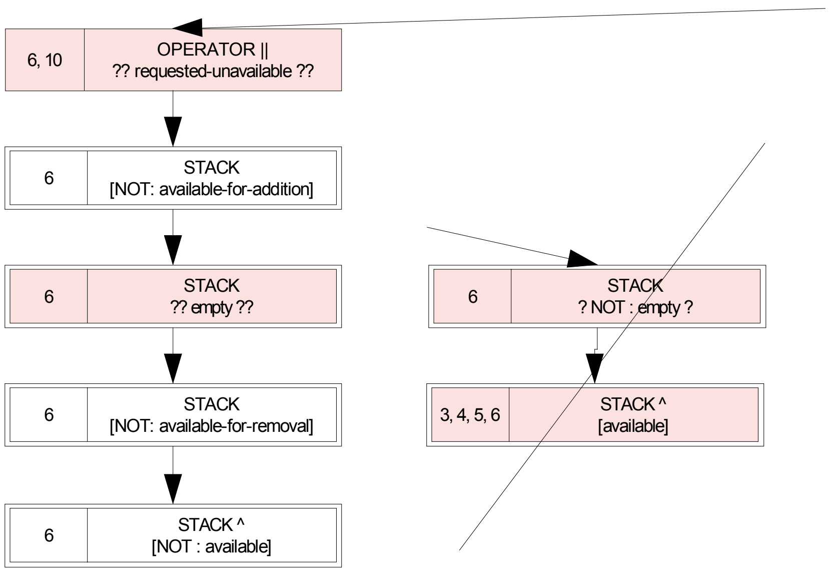


Figure 5.22: Updated Operator Control Process for Concurrency

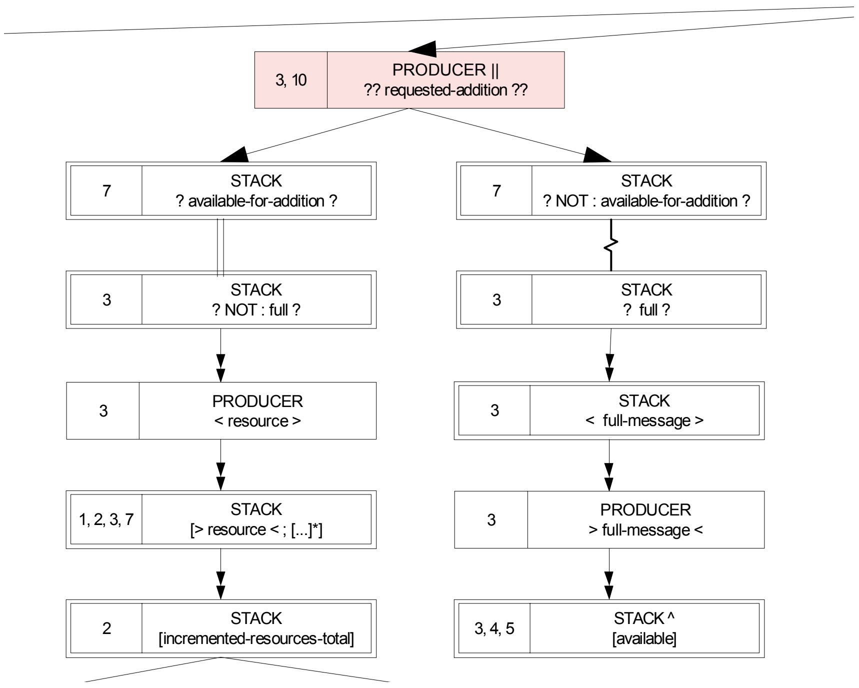


Figure 5.23: Updated Producer Process with Threads and Critical Regions

5.3.2 Queue

This section lists all the functional requirements for a queue-type linear collection. These requirements will be used as the basis of the architecture descriptions in shown in Figures 5.24, 5.25, and 5.26. A basic queue is augmented with requirements for external operator control of availability, as well as concurrency.

As can be seen the requirements between the Stack from Section 5.3.1 are identical except for FR-01, with how resources are removed from the collection.

Basic Queue

The following list details the functional requirements for the Basic Queue. Please refer to Figure 5.24 for the Behavior Tree description of this architecture.

FR-01 Resources added to and removed from a queue will follow the process of “first-in, first-out”.

FR-02 The queue will be able to keep track of the total number of resources currently in its queue, knowing when it is full or empty. The maximum size of the queue is a pre-defined constant for the system.

FR-03 The queue must be not full, for a resource producer to be able to put (enqueue) a new resource to the queue.

FR-04 The queue must be not empty, for a resource consumer to be able to take (dequeue) a resource from the queue.

FR-05 The queue is initially has no resources, and is able to accept at least one resource.

Queue with Availability Control

In addition to the requirements specified for the basic queue, these requirements allow specific blocking of the addition and removal processes by an external Operator component. This will allow the queue to be gracefully shutdown when requested, by first blocking all new resource additions, and when the queue is finally empty, the queue will then become unavailable.

The following list details the functional requirements for the Queue with Availability Control. Please refer to Figure 5.25 for the Behavior Tree description of this architecture.

FR-06 An Operator must make the queue available before additions and removals can commence. When the queue is available, the Operator can request that the queue shutdown; initially halting any new resource additions, and when the queue is empty, no further removals will be allowed, as the queue becomes unavailable.

FR-07 The queue must be available for additions, before any new resources can be added to the queue.

FR-08 The queue must be available for removals, before any resources can be removed from the queue.

Queue with Concurrency

In addition to the requirements specified for the basic queue and queue with availability control, these requirements make the queue's addition, removal, and control processes multi-threaded, with mutual exclusion regions. As a demonstration of availability, this will allow simultaneous additions, and removals by multiple producers and consumers.

The following list details the functional requirements for the Concurrent Queue. Please refer to Figure 5.26 for the Behavior Tree description of this architecture.

FR-09 To ensure integrity of data, while a resource is actually being added, or removed, and the queue size is being calculated, no concurrent thread may interrupt (preempt) this operation.

FR-10 The addition, removal, and operator shutdown processes must be implemented as concurrent threads of operations, for high availability.

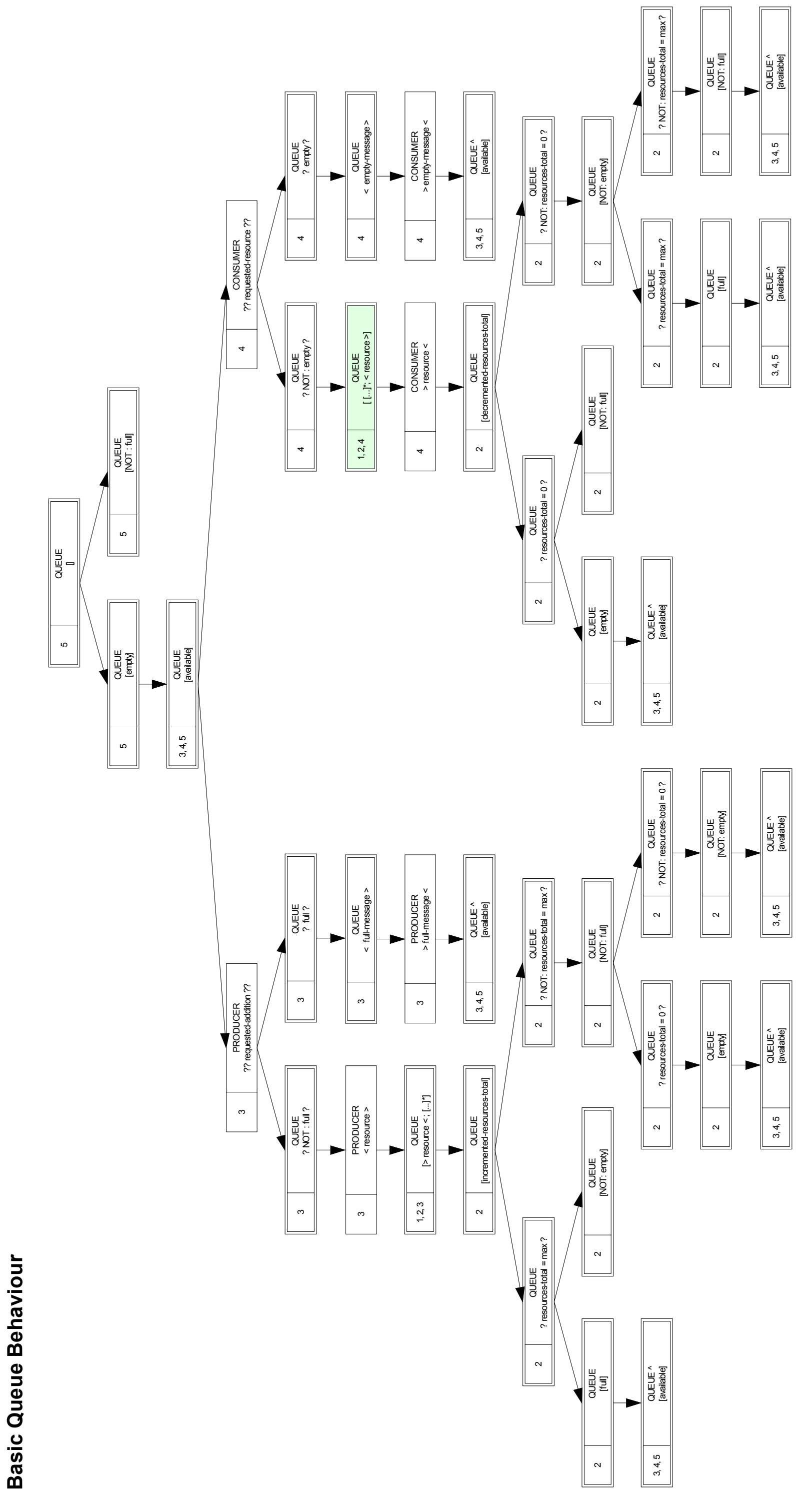

Basic Queue Behaviour
QUEUE []
QUEUE [empty]
QUEUE [NOT : full]
QUEUE [available]
CONSUMER ?? requested-resource ??
PRODUCER ?? requested-addition ??
QUEUE ? NOT : empty ?
QUEUE ? empty ?
QUEUE [[...]*; < resource >]
QUEUE < empty-message >
CONSUMER > resource <
CONSUMER > empty-message <
QUEUE [decremented-resources-total]
QUEUE ^ [available]
QUEUE ? NOT : full ?
QUEUE ? full ?
PRODUCER < resource >
QUEUE < full-message >
QUEUE [> resource <; [...]*]
PRODUCER > full-message <
QUEUE [incremented-resources-total]
QUEUE ? resources-total = 0 ?
QUEUE ? NOT: resources-total = 0 ?
QUEUE ? resources-total = max ?
QUEUE ? NOT: resources-total = max ?
QUEUE [NOT: full]
QUEUE [NOT: empty]
QUEUE [full]
QUEUE [empty]
5
4
3
2
1, 2, 4
1, 2, 3
3, 4, 5

Figure 5.24: Basic Queue Behaviour

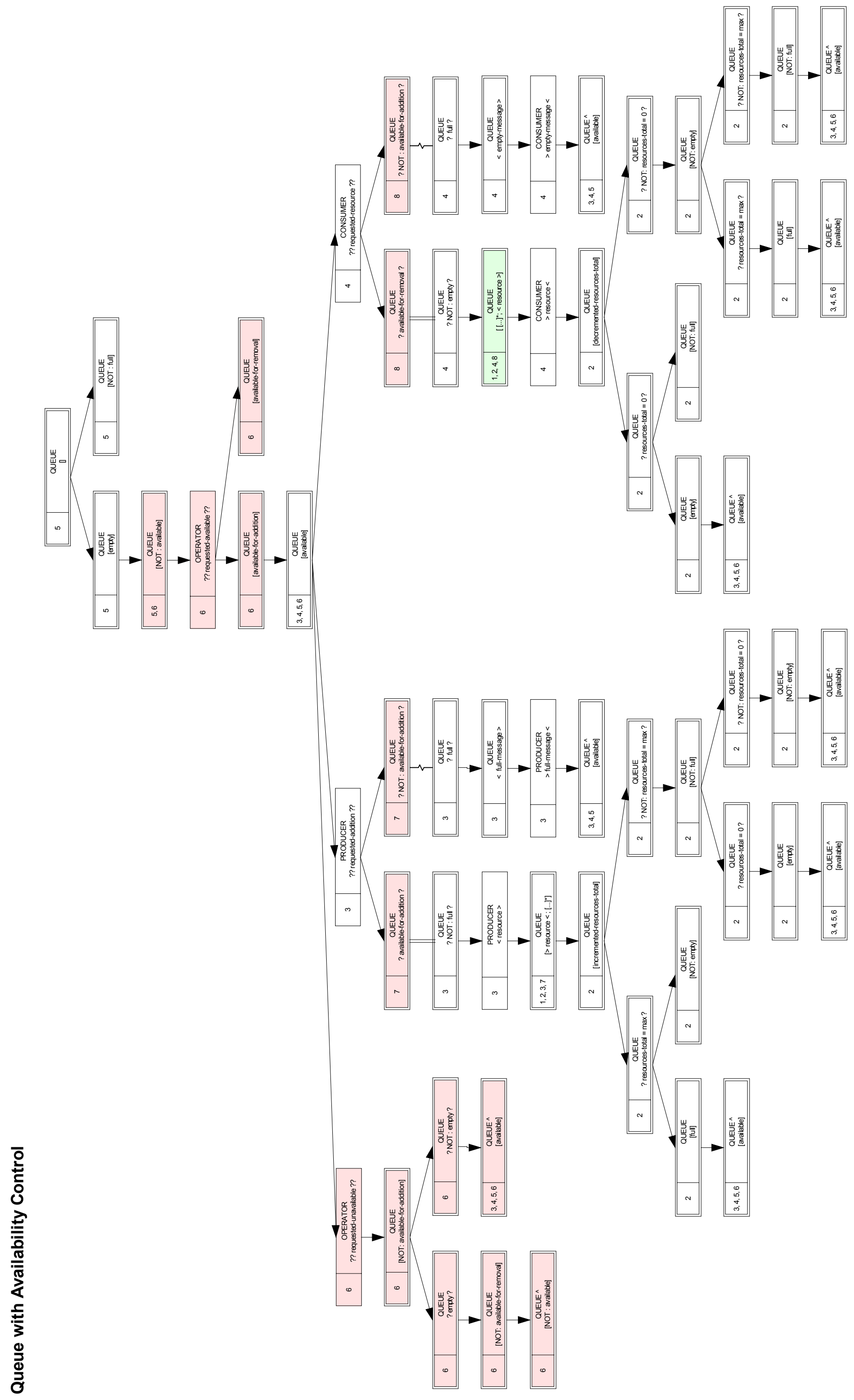


Figure 5.25: Queue Behaviour with External Operator Control Augmentation

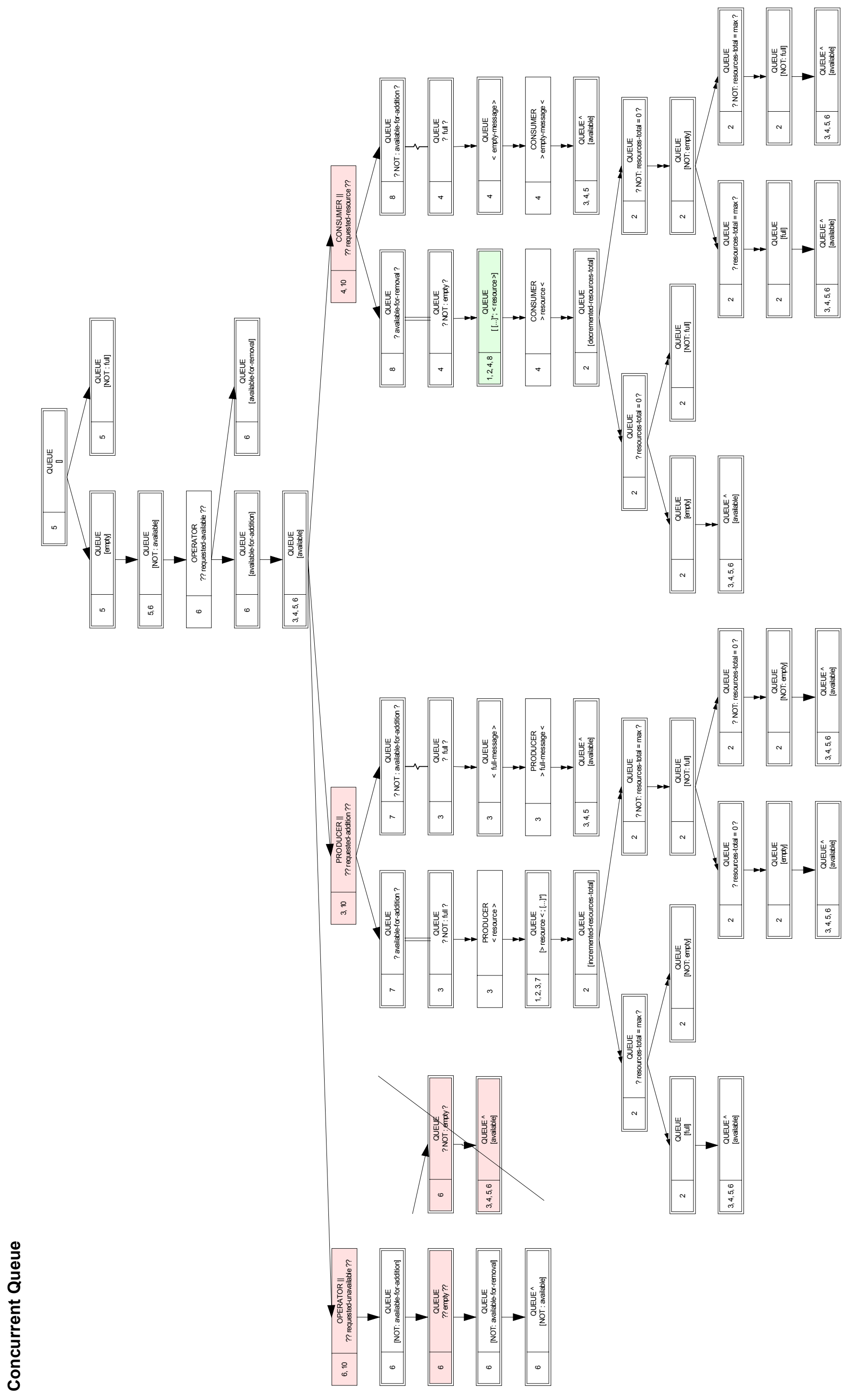


Figure 5.26: Queue Behaviour with Concurrency Augmentation

5.3.3 Set

This section lists all the functional requirements for a set-type unordered collection. These requirements will be used as the basis of the architecture descriptions in shown in Figures 5.27, 5.28, and 5.29. A basic set is augmented with requirements for external operator control of availability, as well as concurrency.

As can be seen the requirements between the Stack from Section 5.3.1, and Queue from Section 5.3.2, and this Set are identical except for FR-01; with how resources are added and removed from the collection, and FR-04; the need to specify the resource to retrieve it.

Basic Set

The following list details the functional requirements for the Basic Set. Please refer to Figure 5.27 for the Behavior Tree description of this architecture.

FR-01 Resources can be added or removed from the set in any particular order. Generally a resource has to be requested by using a unique identifier for it to be retrieved.

FR-02 The set will be able to keep track of the total number of resources currently in its set, knowing when it is full or empty. The maximum size of the set is a pre-defined constant for the system.

FR-03 The set must be not full, for a resource producer to be able to add a new resource to the set.

FR-04 The set must be not empty, and contain the resource requested for a resource consumer to be able to retrieve the resource from the set.

FR-05 The set is initially has no resources, and is able to accept at least one resource.

Set with Availability Control

In addition to the requirements specified for the basic set, these requirements allow specific blocking of the addition and removal processes by an external Operator component. This will allow the set to be gracefully shutdown when requested, by first blocking all new resource additions, and when the set is finally empty, the set will then become unavailable.

The following list details the functional requirements for the Set with Availability Control. Please refer to Figure 5.28 for the Behavior Tree description of this architecture.

FR-06 An Operator must make the set available before additions and removals can commence. When the set is available, the Operator can request that the set shutdown; initially halting any new resource additions, and when the set is empty, no further removals will be allowed, as the set becomes unavailable.

FR-07 The set must be available for additions, before any new resources can be added to the set.

FR-08 The set must be available for removals, before any resources can be removed from the set.

5.3.4 Set with Concurrency

In addition to the requirements specified for the basic set and set with availability control, these requirements make the set's addition, removal, and control processes multi-threaded, with mutual exclusion regions. As a demonstration of availability, this will allow simultaneous additions, and removals by multiple producers and consumers.

The following list details the functional requirements for the Concurrent Set. Please refer to Figure 5.29 for the Behavior Tree description of this architecture.

FR-09 To ensure integrity of data, while a resource is actually being added, or removed, and the set size is being calculated, no concurrent thread may interrupt (preempt) this operation.

FR-10 The addition, removal, and operator shutdown processes must be implemented as concurrent threads of operations, for high availability.

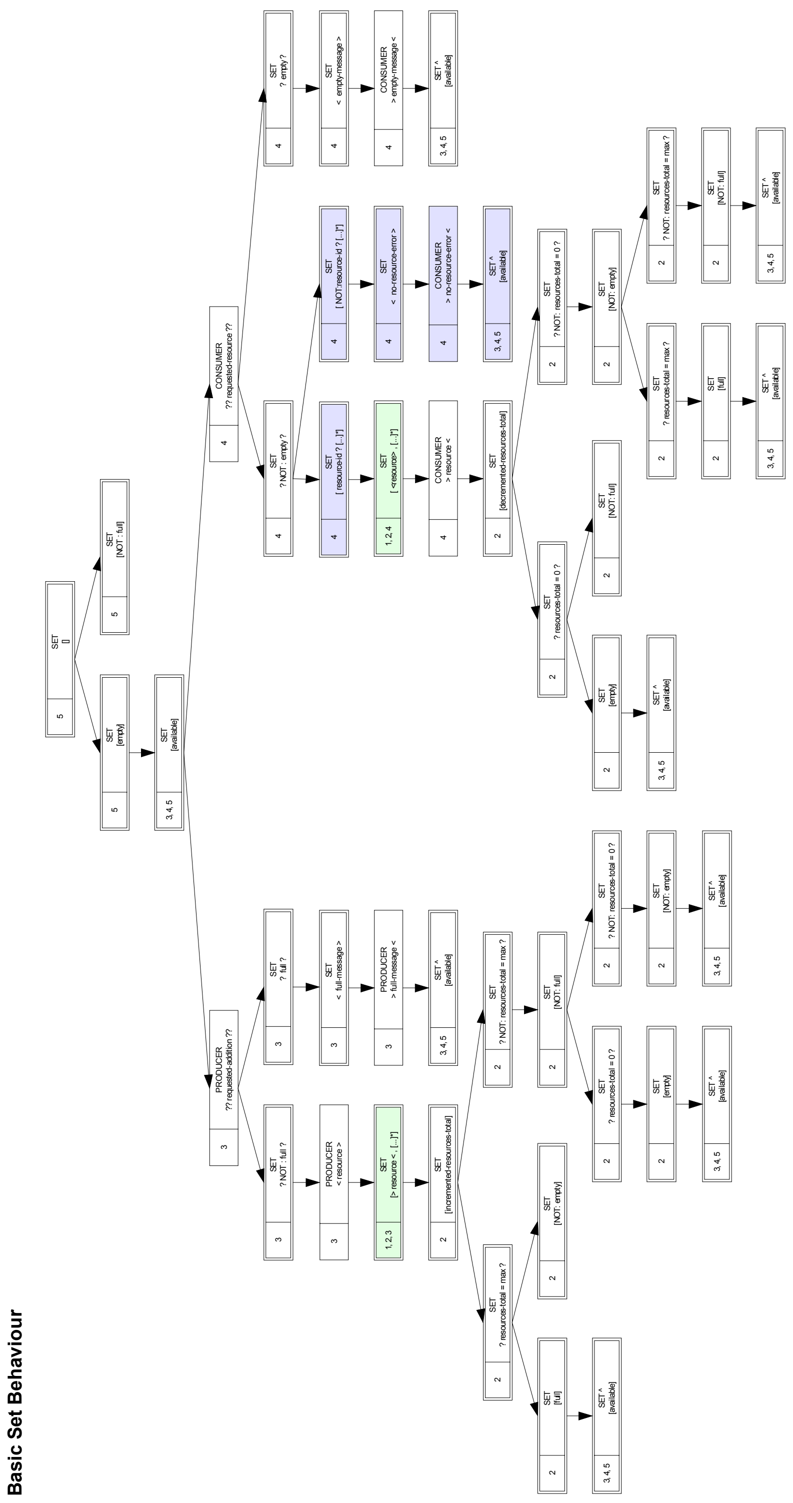


Figure 5.27: Basic Set Behaviour

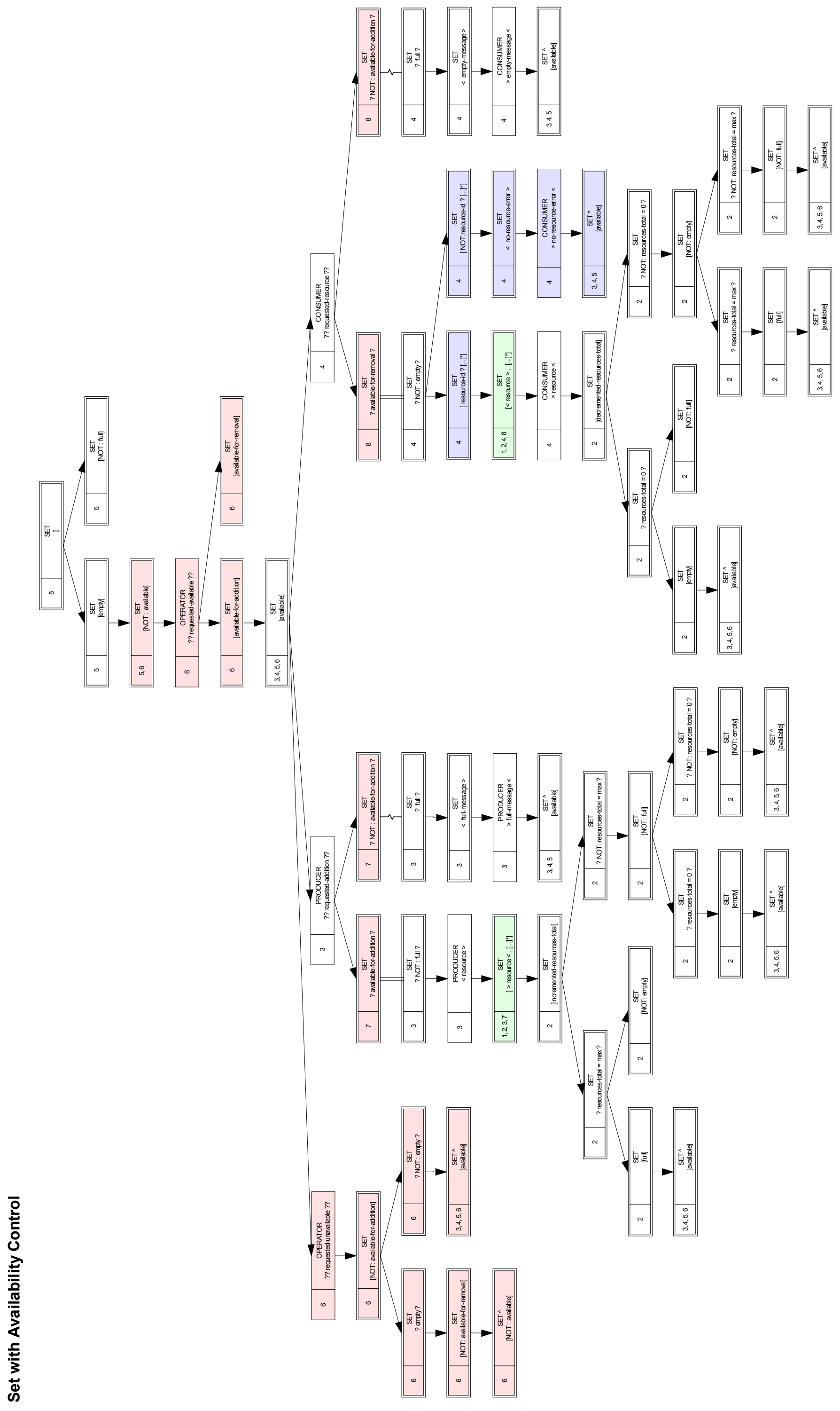


Figure 5.28: Set Behaviour with External Operator Control Augmentation

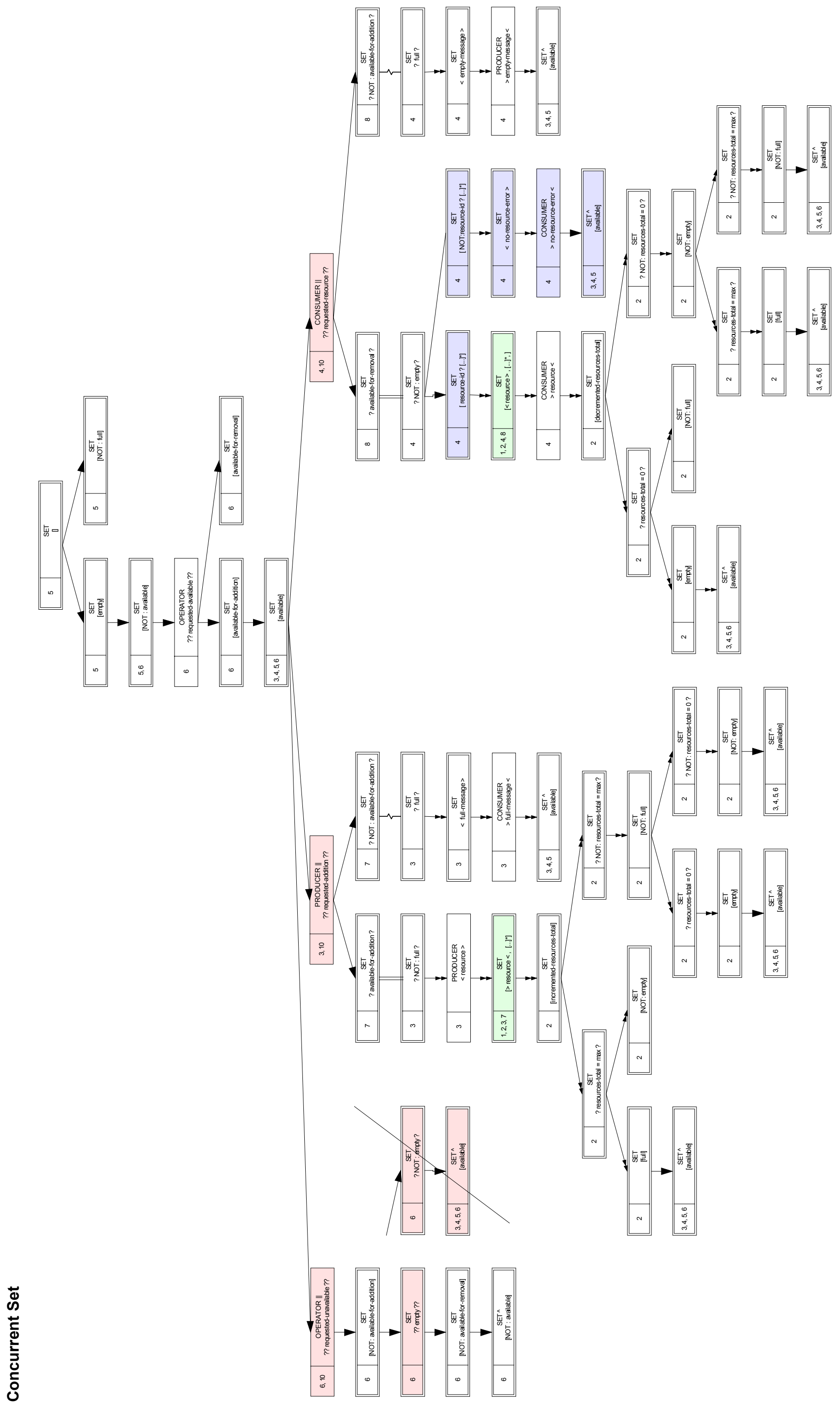


Figure 5.29: Set Behaviour with Concurrency Augmentation

5.3.5 Collection

This section describes a generic collection component that combines the properties of a Stack (Section 5.3.1), Queue (Section 5.3.2), and Set (Section 5.3.3).

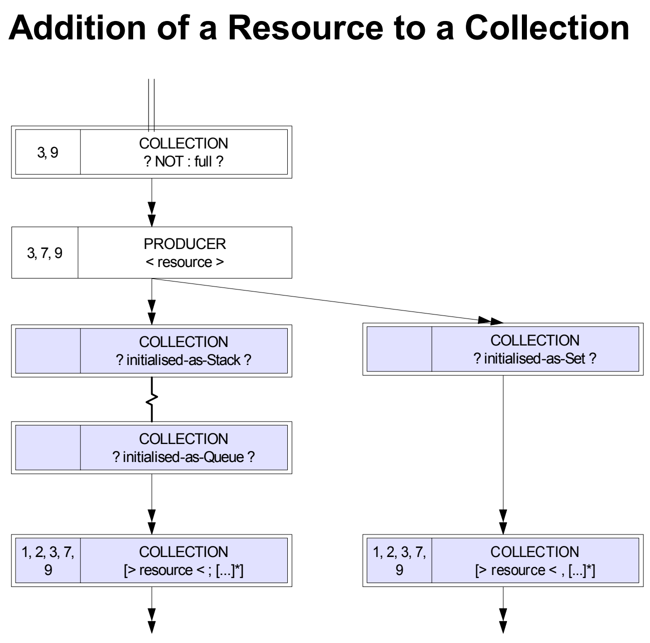


Figure 5.30: Modification of Addition Process to allow for Different Collection Types

By augmenting and comparing the requirements of each of these collection types, a number of conclusions were found.

1. The name of the system; Stack, Queue, Set, or Collection, was the first modification of the system to adapt to the different sets of requirements.

2. Only the corresponding first requirement of each collection type differed. i.e. how a queue, stack, or set actually stores and retrieves its resources. For the Collection, this functional requirement is rewritten as:

   [FR-01] Resources added to and removed from a collection will follow the process of "first-in, first-out", "first-in, last-out", or requested by using a

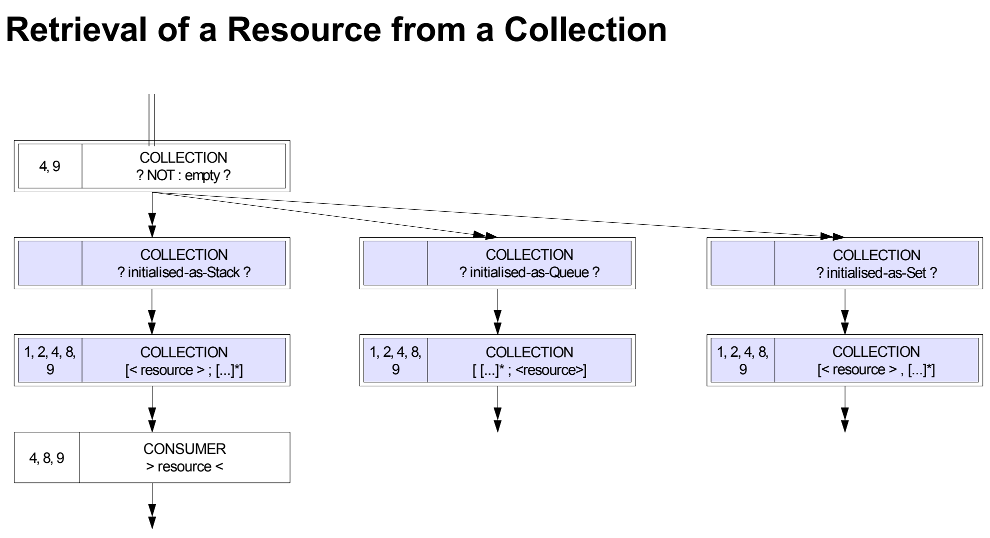


Figure 5.31: Modification of Retrieval Process to allow for Different Collection Types

unique identifier, depending on how the collection is initialised.

This is implemented by added selected flow to the Behavior Tree specifying what addition or retrieval is required by the Collection. See Figures 5.30 and 5.31. This benefits the Reusability of the component at the cost of complexity for the overall Behavior Tree and possibly a performance hit at implementation level.

3. Adding an Operator control process to the system, required additional behaviour in the initialisation sequence and a new control branch for detecting shut-down requests at the same time as additions and retrievals are processed.

4. Adding concurrency to processes is a matter of adding the parallel process operator to the top of each process. Behaviour that related directly to the addition and retrieval of resources, must be mutually exclusive, to other behaviour. i.e. a process in a critical region should not be preempted before it is finished its operations. This is added to the Behavior Tree by changing the

Control Flow arrows to Tight Control Flow double arrows, where required.

To originally maintain linear flow of control through the system, the operator process had to loop back to allow additions and retrievals, before rechecking if the stack was now empty. This was made redundant by using concurrency, where the process could merely be blocked and wait for other processes to finish.

From Section C, adapting the component implementation for concurrency, was be more complex than the architectural modification. Specifically in Java, methods have to be added or changed to use Threads. Overall, by following Sun's current best practice, the architectural model was still essentially valid.

5. Implementation of the Generic Collection component could proceed in two directions:

   (a) Each version of the collection type and augmentation could be packaged into the one component, allowing the developer to choose the optimised version that suits their requirements. This is much like the libraries or APIs that are used regularly by programmers in industry today.

   (b) One system could be implemented containing all the collection types, and augmentations. The cost to the system would be Efficiency, but Reusability (Adaptability) would benefit. This could allow a component to change its internal collection storage type at runtime, not just compile-time.

5.4 Component Adaptation Example

This case study will investigate two real-world allocation systems from an adaptation perspective. The Hospital Bed Allocation System handles the registration and allocation of beds to patients. The requirements and derived architecture for this system is shown in Section 5.4.1. The Carpark System allows for cars to enter, park in a space, and leave an automated car park. Its requirements and architecture are given in Section 5.4.2.

The primary objective for this case study is to compare analogous behaviour between two systems. If the overall functionality of two systems are perceived to be the similar, what specifically is different? If the Behavior Tree methodology tends to produce a similar implementation regardless of the designer, how can two trees differ, and yet produce the same functionality?

Figure 5.32 displays how the two systems will be compared. First of all, requirements will be aligned as much as possible. The Behavior Trees for both systems will then be compared for similarities, such as overall processes. By then using processes of renaming, projection and transformation (Figure 5.33) between the two trees, the intended behaviour of each system can be examined from a different perspective. i.e. If the Carpark system is transformed into a Hospital system, is it a valid Hospital system? How does this differ from the real, original Hospital system?

After this work is reviewed, a generic allocation system will be developed, including components from the Component Development Section 5.3. How easily can the collection component be adapted for use in an allocation system?

Finally, ways of identifying quality attributes within Behavior Trees will be demonstrated. By taking the basic functionality of a system, and adapting it to handle unknown behavior, security and reliability can be built into the system. Performance and efficiency can be added with concurrent processes.

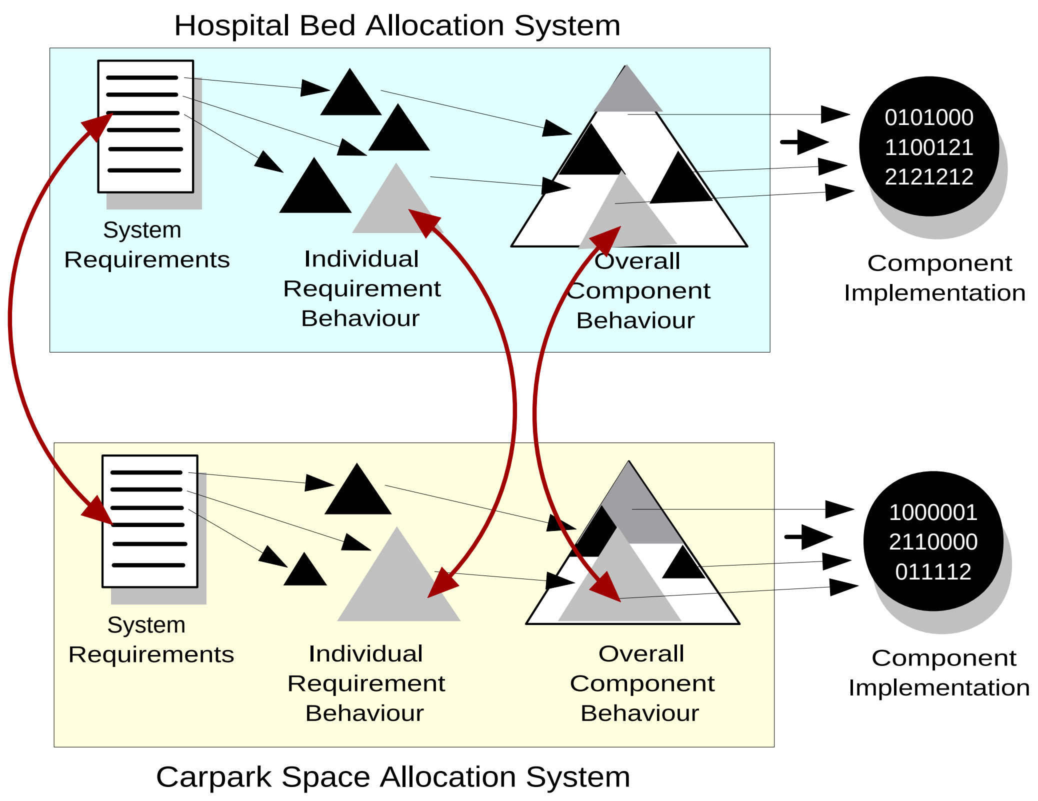

Hospital Bed Allocation System
System Requirements
Individual Requirement Behaviour
Overall Component Behaviour
0101000
1100121
2121212
Component Implementation
System Requirements
Individual Requirement Behaviour
Overall Component Behaviour
1000001
2110000
011112
Component Implementation
Carpark Space Allocation System

Figure 5.32: Analogous Behaviour

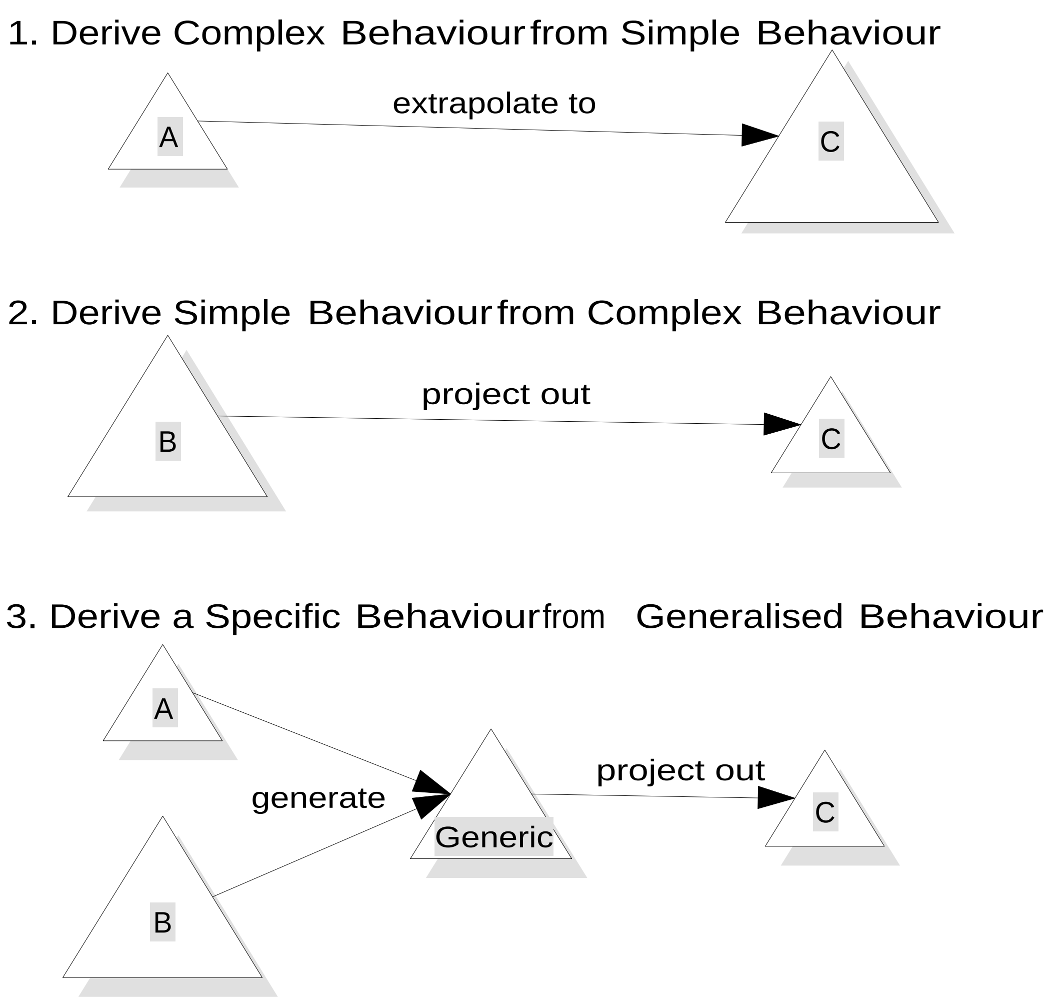

1. Derive Complex Behaviour from Simple Behaviour
A
extrapolate to
C
2. Derive Simple Behaviour from Complex Behaviour
B
project out
C
3. Derive a Specific Behaviour from Generalised Behaviour
A
generate
Generic
project out
C
B

Figure 5.33: Component Architecture Re-Engineering

5.4.1 Hospital Bed Allocation System

Requirements

This section lists all the functional requirements for a Hospital bed allocation system. This system requires that a patient be registered before they are admitted, and cleared-for-departure before checking out.

FR-01 Patients can be admitted to and depart from the Hospital system in any order, but must queue to access individual services.

FR-02 The Hospital keeps a patient database, keeping track of status on registration, admission, bed allocation, and departure clearance. There is a known number of total beds available in the Hospital.

FR-03 A patient can be registered, but not admitted, until a bed suitable for their priority level is available for allocation.

FR-04 A patient cannot check-out from the hospital unless they are actually registered there first. If they have been admitted, they must get prior clearance, before checking-out.

FR-05 A Hospital can be closed for maintenance purposes. When closed no patient details can be processed. The hospital system should have its admission and bed allocation records re-validated before reopening.

Architecture

Figure 5.34 represents the architecture of the system derived from the requirements given in Section 5.4.1. Figure 5.38 shows another version of a Hospital Allocation System transformed from the Carpark Allocation System. The comparison results between these two systems will be provided in Section 5.4.3.

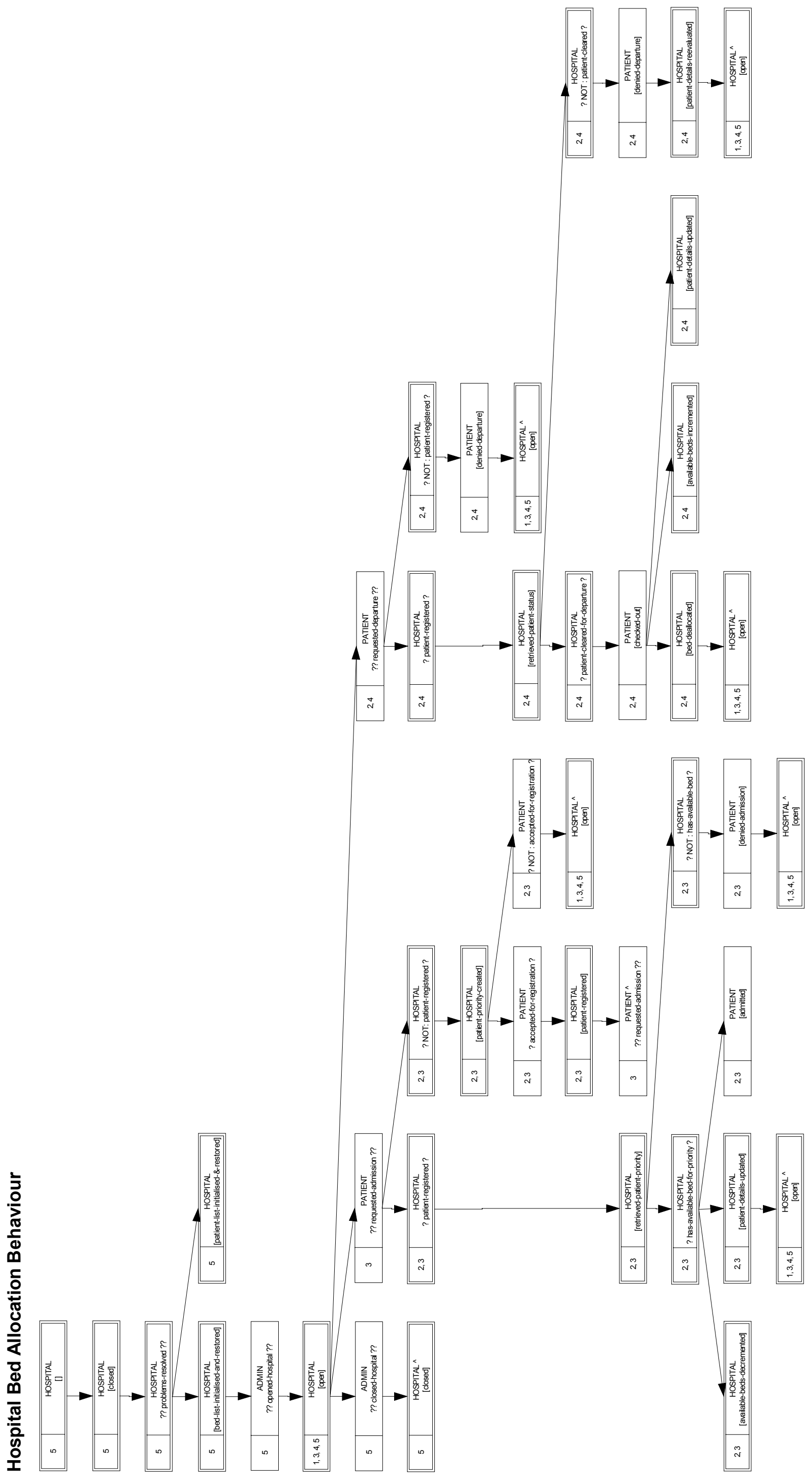


Figure 5.34: Hospital Bed Allocation Behaviour derived from the original requirements

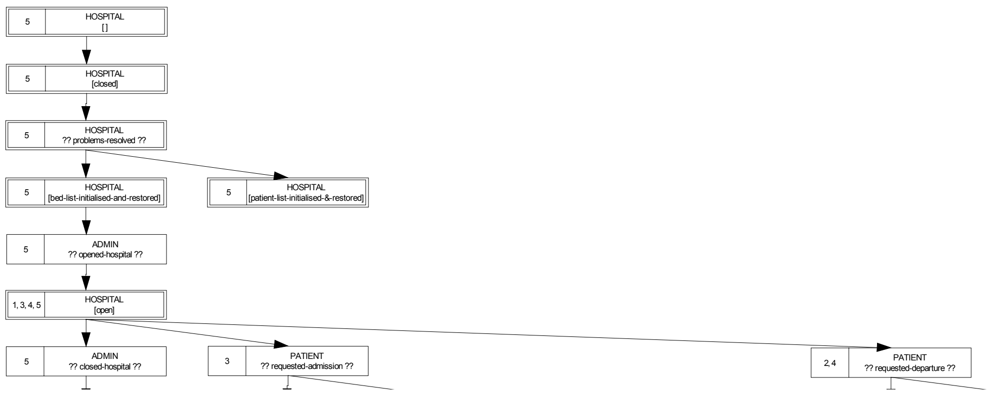


Figure 5.35: Hospital Bed Allocation Initiation Process

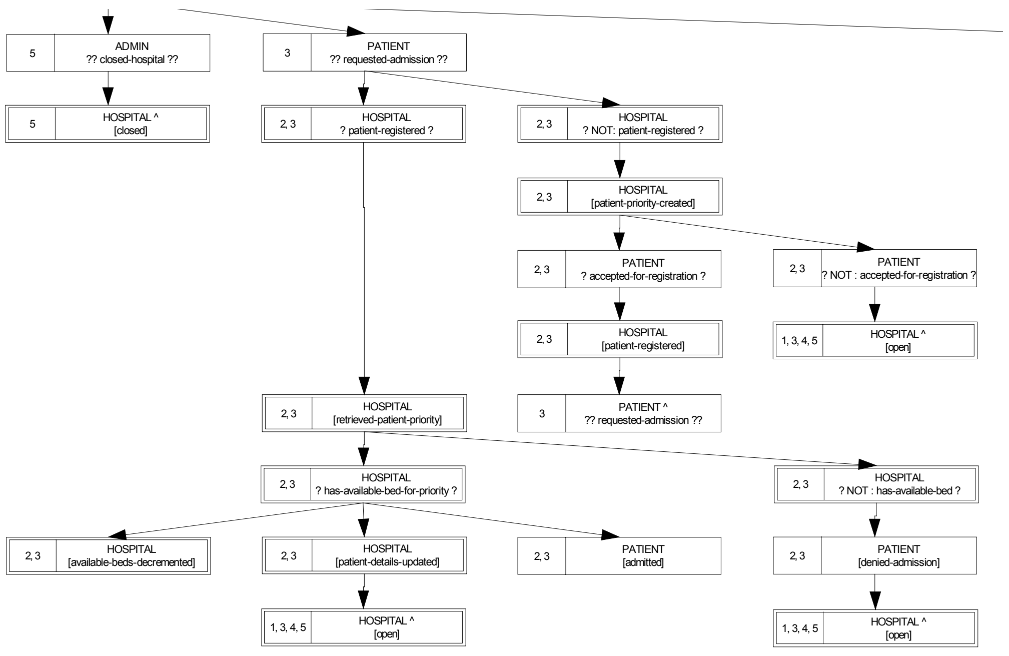


Figure 5.36: Hospital Bed Allocation Admission Process

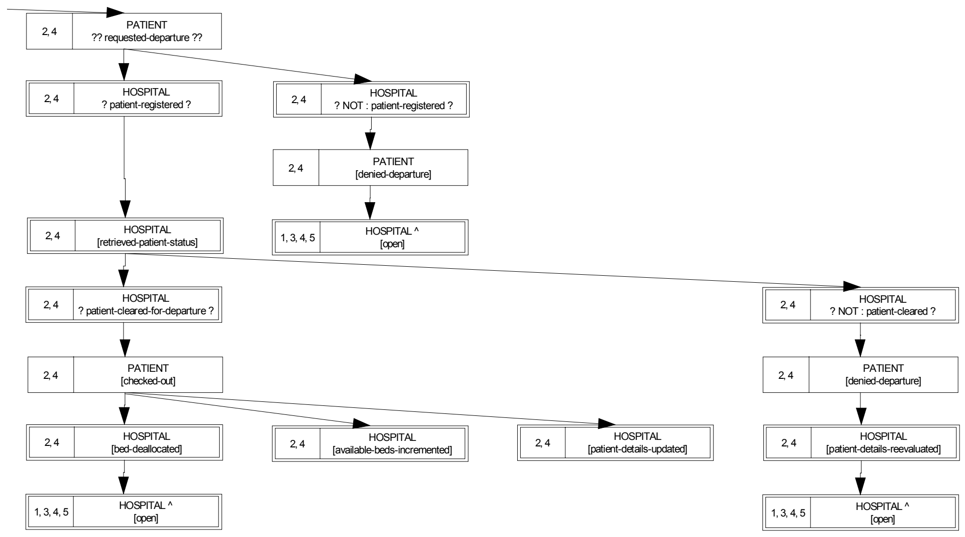


Figure 5.37: Hospital Bed Allocation Departure Process

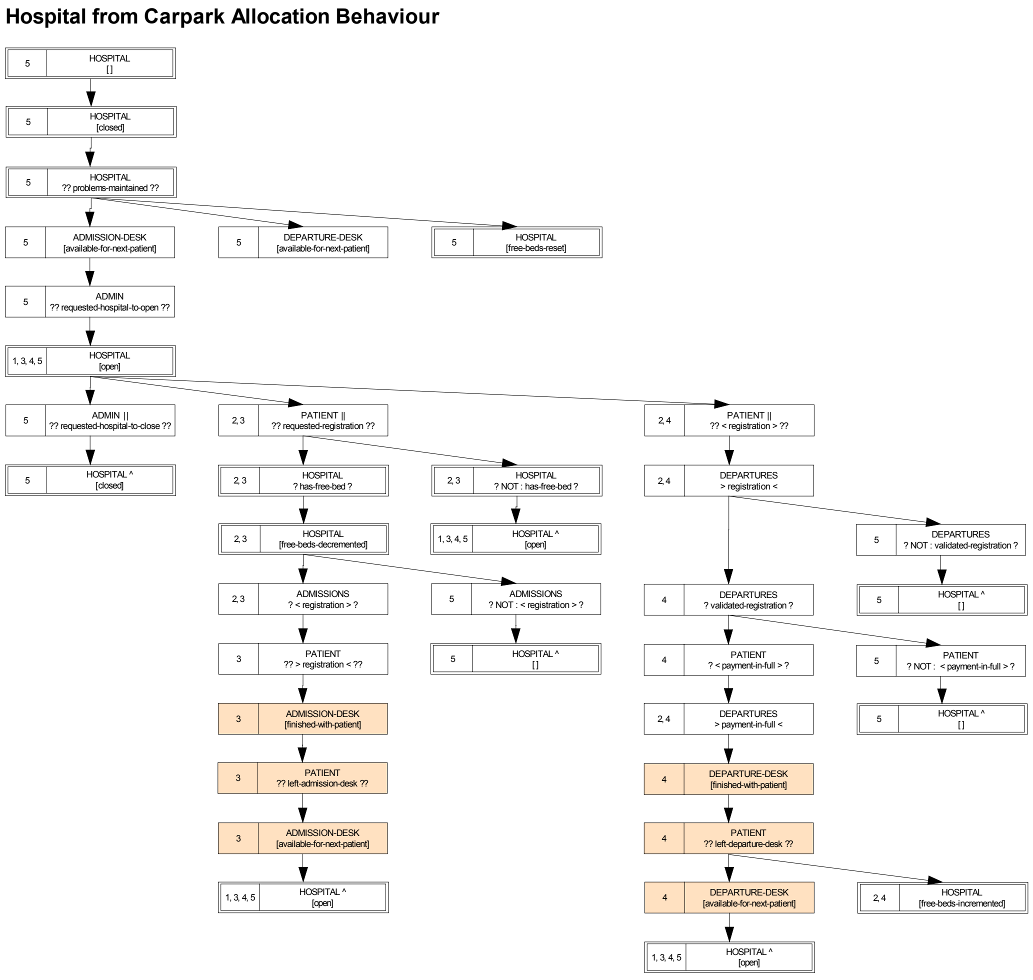

Hospital from Carpark Allocation Behaviour
5 HOSPITAL []
5 HOSPITAL [closed]
5 HOSPITAL ?? problems-maintained ??
5 ADMISSION-DESK [available-for-next-patient]
5 DEPARTURE-DESK [available-for-next-patient]
5 HOSPITAL [free-beds-reset]
5 ADMIN ?? requested-hospital-to-open ??
1, 3, 4, 5 HOSPITAL [open]
5 ADMIN || ?? requested-hospital-to-close ??
5 HOSPITAL ^ [closed]
2, 3 PATIENT || ?? requested-registration ??
2, 3 HOSPITAL ? has-free-bed ?
2, 3 HOSPITAL ? NOT : has-free-bed ?
1, 3, 4, 5 HOSPITAL ^ [open]
2, 3 HOSPITAL [free-beds-decremented]
2, 3 ADMISSIONS ? < registration > ?
5 ADMISSIONS ? NOT : < registration > ?
5 HOSPITAL ^ []
3 PATIENT ?? > registration < ??
3 ADMISSION-DESK [finished-with-patient]
3 PATIENT ?? left-admission-desk ??
3 ADMISSION-DESK [available-for-next-patient]
1, 3, 4, 5 HOSPITAL ^ [open]
2, 4 PATIENT || ?? < registration > ??
2, 4 DEPARTURES > registration <
5 DEPARTURES ? NOT : validated-registration ?
5 HOSPITAL ^ []
4 DEPARTURES ? validated-registration ?
4 PATIENT ? < payment-in-full > ?
5 PATIENT ? NOT : < payment-in-full > ?
5 HOSPITAL ^ []
2, 4 DEPARTURES > payment-in-full <
4 DEPARTURE-DESK [finished-with-patient]
4 PATIENT ?? left-departure-desk ??
4 DEPARTURE-DESK [available-for-next-patient]
2, 4 HOSPITAL [free-beds-incremented]
1, 3, 4, 5 HOSPITAL ^ [open]

Figure 5.38: Hospital Bed Allocation Behaviour derived from the Carpark Allocation System requirements

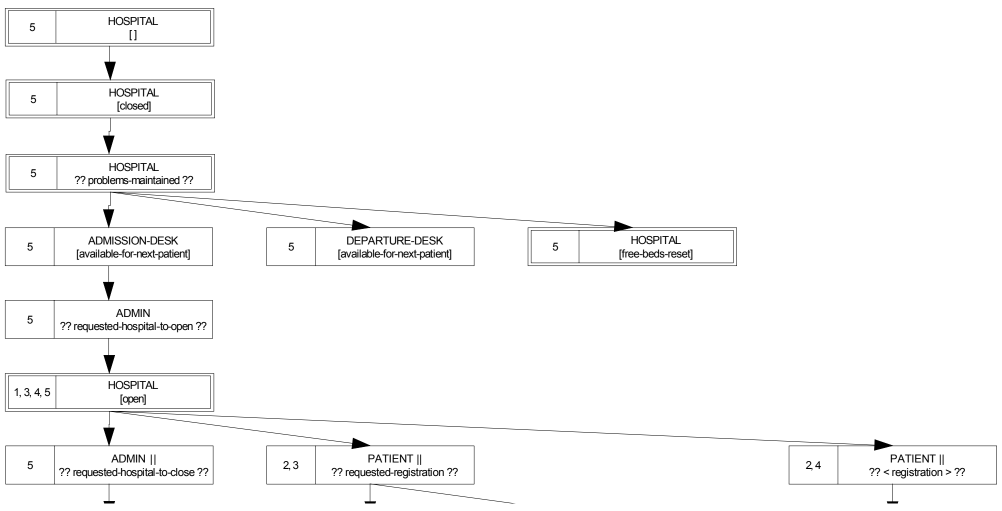


Figure 5.39: Hospital Bed Allocation Initiation Process

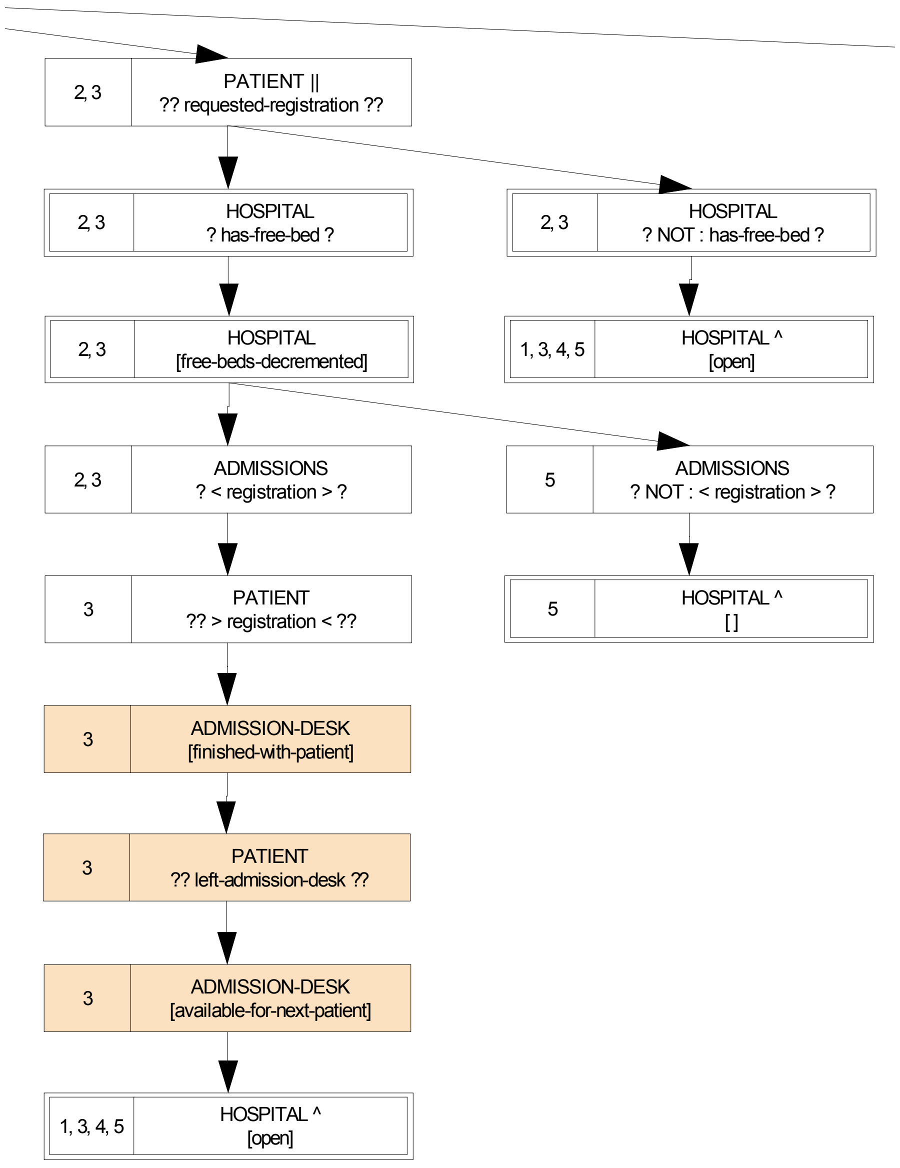


Figure 5.40: Hospital Bed Allocation Admission Process

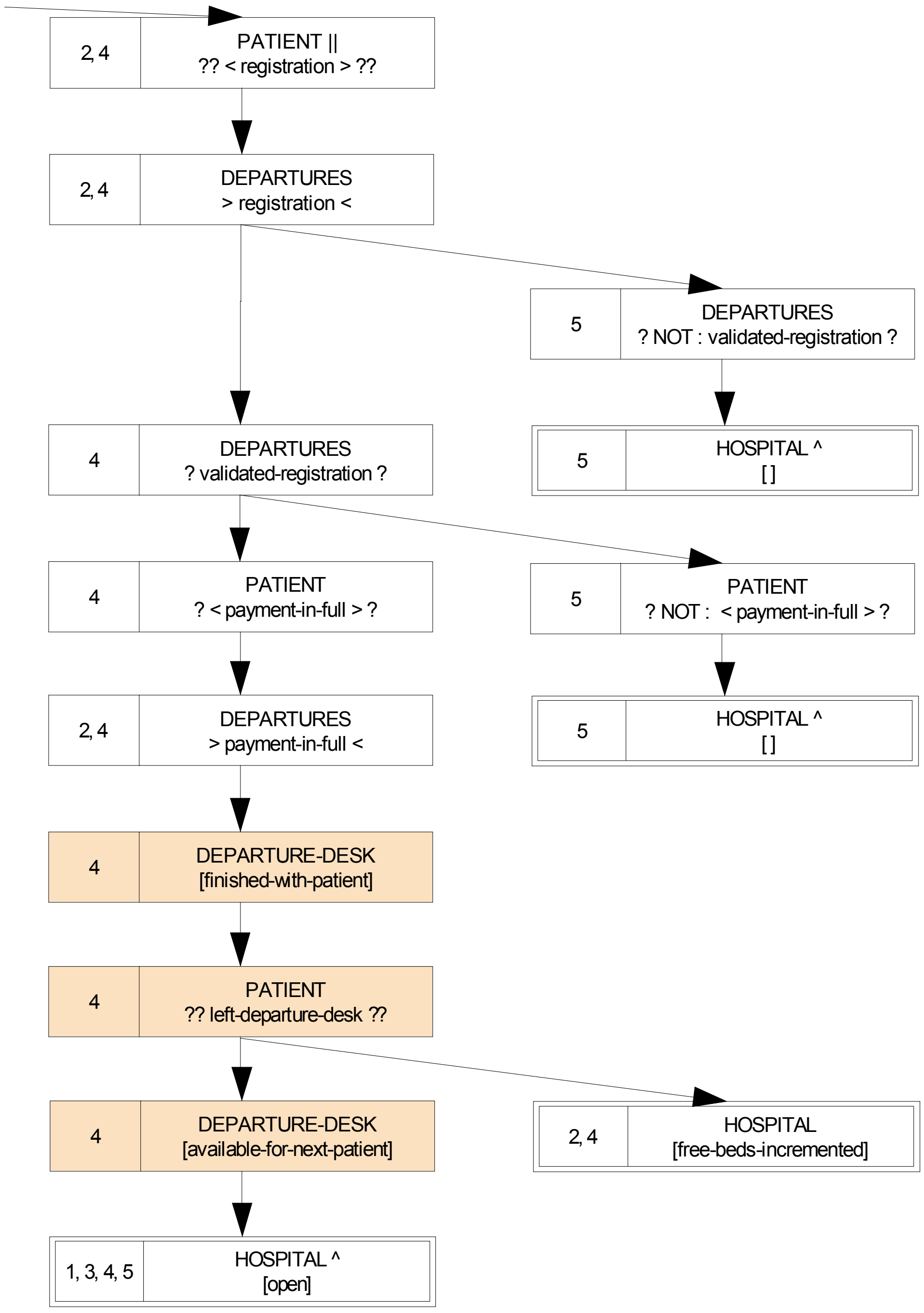


Figure 5.41: Hospital Bed Allocation Departure Process

5.4.2 Carpark Space Allocation System

Requirements

This section lists all the functional requirements for a Carpark space allocation system. There is some specific requirements dealing with the automatic entry and exit systems, with ticket vending and boomgates.

FR-01 Cars must queue up to actually enter and leave the Carpark but generally can leave at their own convenience.

FR-02 The Carpark system keeps track of how many spaces are available and currently used at all times.

FR-03 When a new Car approaches a Carpark entry point, if the Carpark is not full, the driver can press a button requesting a ticket from the Ticket Allocator. When the driver takes the ticket, total number of carspaces are deallocated, the entry boomgate will raise and wait for the Car to pass before lowering, and waiting for another Car's request.

FR-04 When a Car wishes to leave the Carpark, the driver must present the ticket to the Ticket Collector. If the ticket is valid, and the driver provides payment-in-full for the time spent in the Carpark, the exit boomgate will rise, the Car will leave, the boomgate will lower, the total number of available carspaces will be incremented and the next request can be processed.

FR-05 Should the automatic nature of the Carpark system fail, the Carpark should close and be maintained before reopening.

Architecture

Figure 5.42 represents the architecture of the system derived from the requirements given in Section 5.4.2. Figure 5.46 shows another version of a Carpark Allocation System transformed from the Hospital Allocation System. The comparison results between these two systems will be provided in Section 5.4.3.

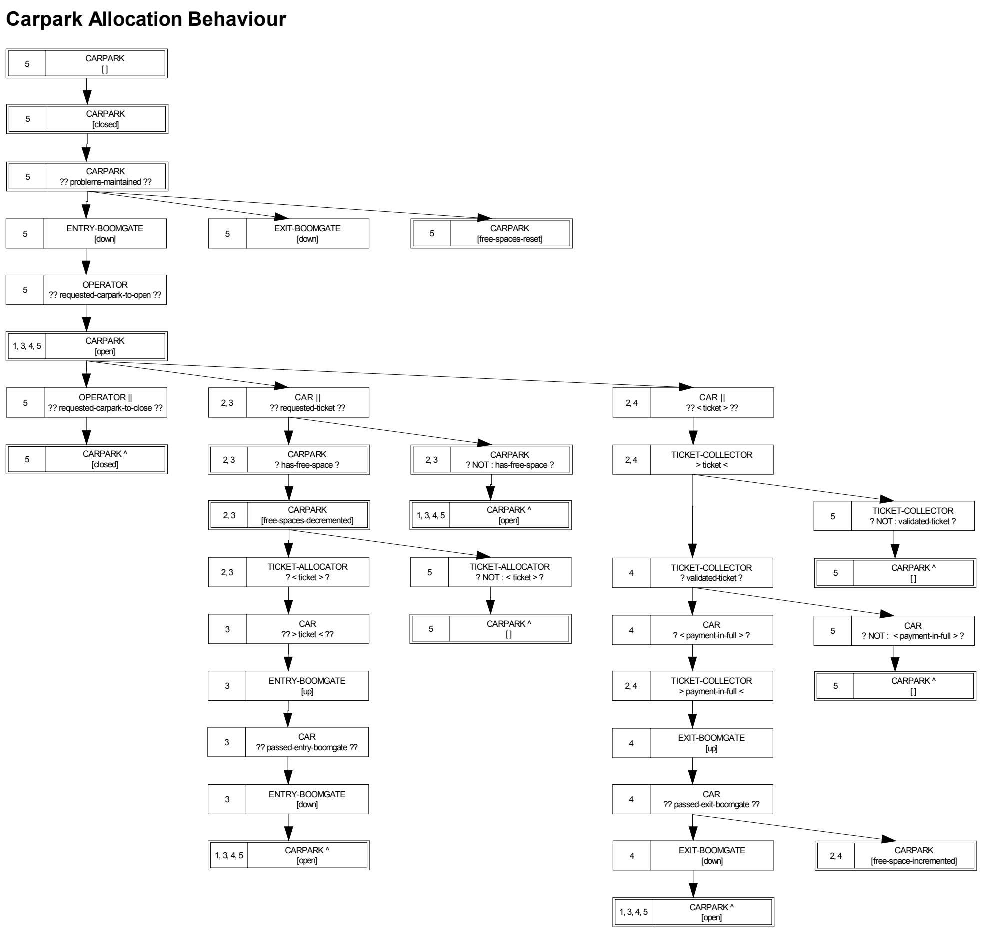

Carpark Allocation Behaviour
5 CARPARK [ ]
5 CARPARK [closed]
5 CARPARK ?? problems-maintained ??
5 ENTRY-BOOMGATE [down]
5 EXIT-BOOMGATE [down]
5 CARPARK [free-spaces-reset]
5 OPERATOR ?? requested-carpark-to-open ??
1, 3, 4, 5 CARPARK [open]
5 OPERATOR || ?? requested-carpark-to-close ??
5 CARPARK ^ [closed]
2, 3 CAR || ?? requested-ticket ??
2, 3 CARPARK ? has-free-space ?
2, 3 CARPARK ? NOT : has-free-space ?
1, 3, 4, 5 CARPARK ^ [open]
2, 3 CARPARK [free-spaces-decremented]
2, 3 TICKET-ALLOCATOR ? < ticket > ?
5 TICKET-ALLOCATOR ? NOT : < ticket > ?
3 CAR ?? > ticket < ??
5 CARPARK ^ [ ]
3 ENTRY-BOOMGATE [up]
3 CAR ?? passed-entry-boomgate ??
3 ENTRY-BOOMGATE [down]
1, 3, 4, 5 CARPARK ^ [open]
2, 4 CAR || ?? < ticket > ??
2, 4 TICKET-COLLECTOR > ticket <
5 TICKET-COLLECTOR ? NOT : validated-ticket ?
5 CARPARK ^ [ ]
4 TICKET-COLLECTOR ? validated-ticket ?
4 CAR ? < payment-in-full > ?
5 CAR ? NOT : < payment-in-full > ?
5 CARPARK ^ [ ]
2, 4 TICKET-COLLECTOR > payment-in-full <
4 EXIT-BOOMGATE [up]
4 CAR ?? passed-exit-boomgate ??
4 EXIT-BOOMGATE [down]
2, 4 CARPARK [free-space-incremented]
1, 3, 4, 5 CARPARK ^ [open]

Figure 5.42: Car Space Allocation Behaviour derived from the original requirements

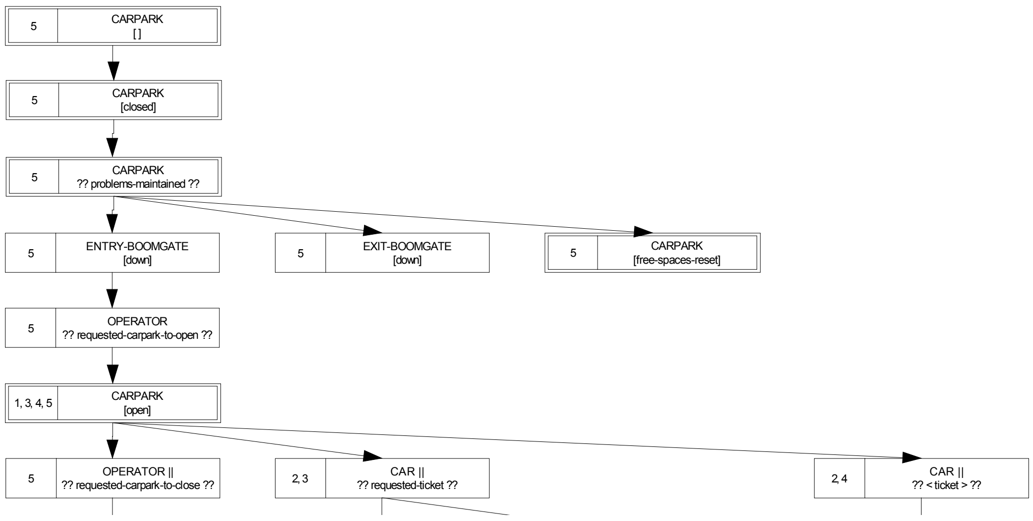


Figure 5.43: Car Space Allocation Initiation Process

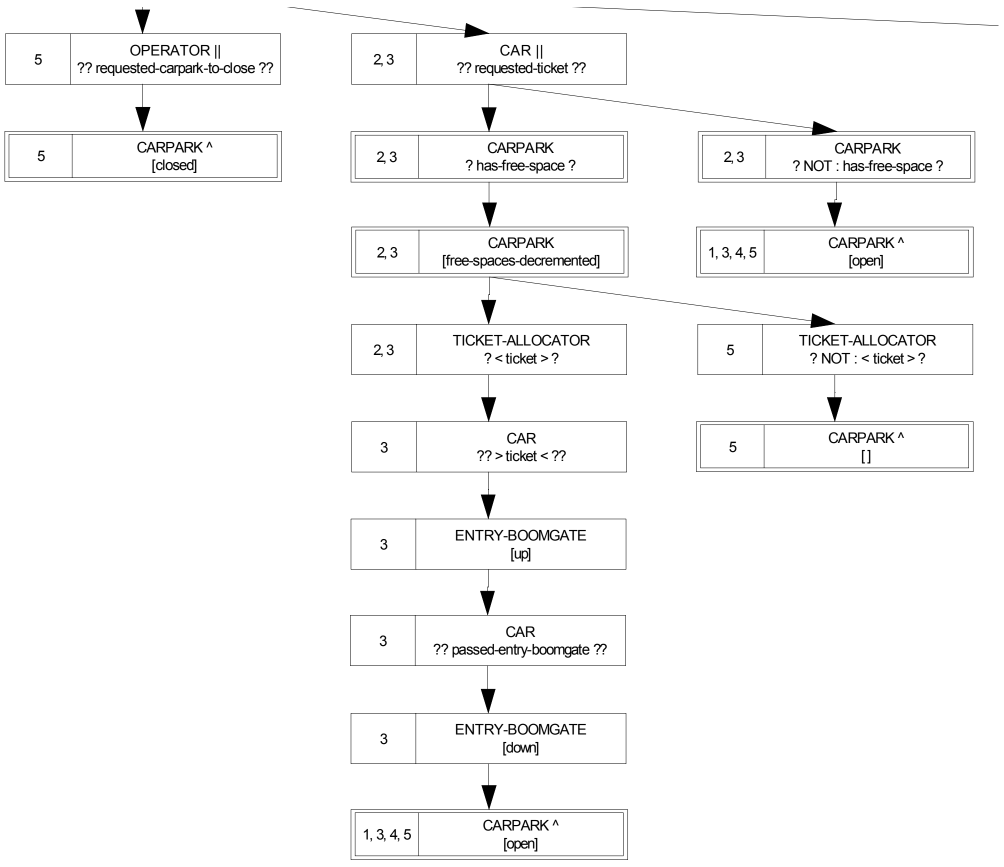


Figure 5.44: Car Space Allocation Entry Process

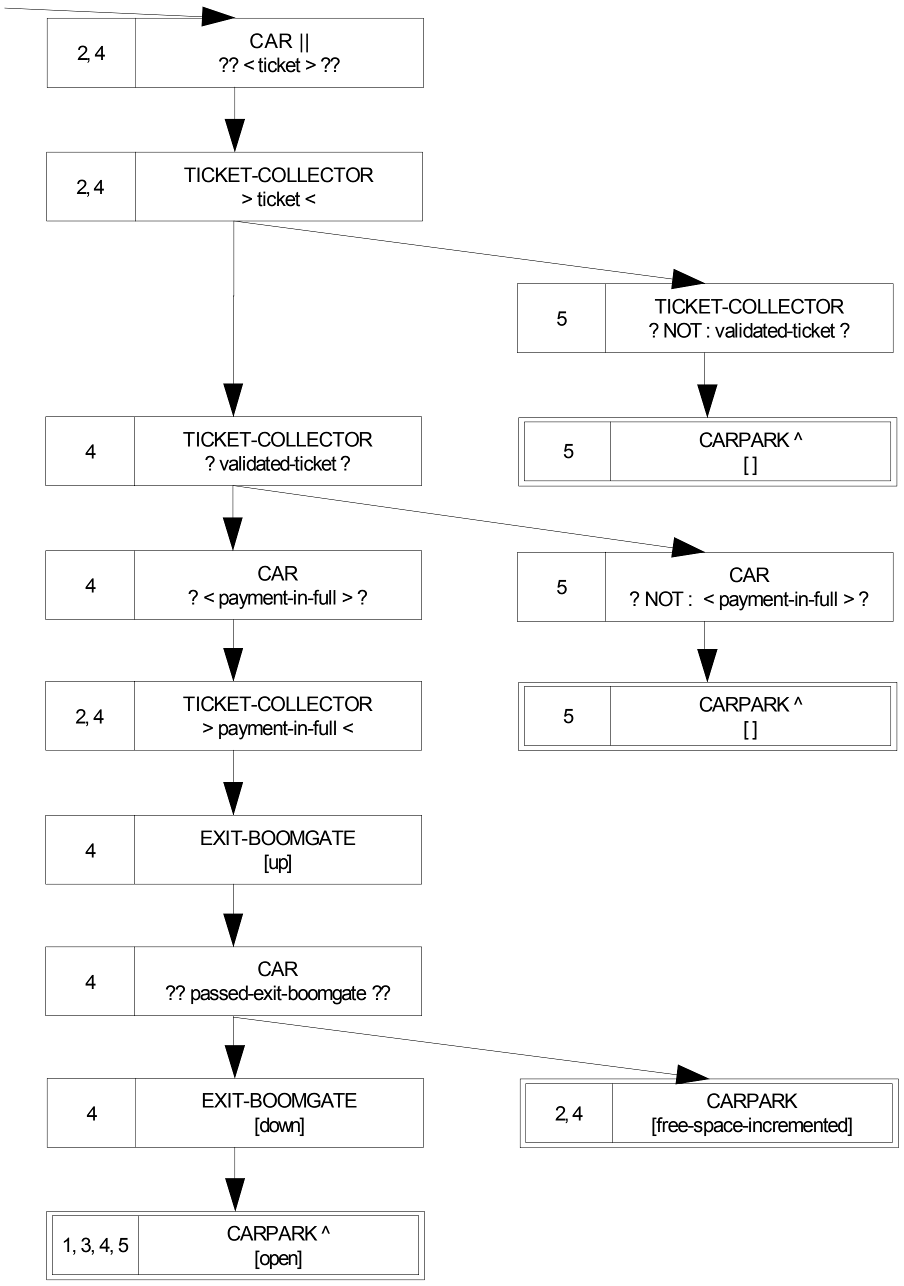


Figure 5.45: Car Space Allocation Exit Process

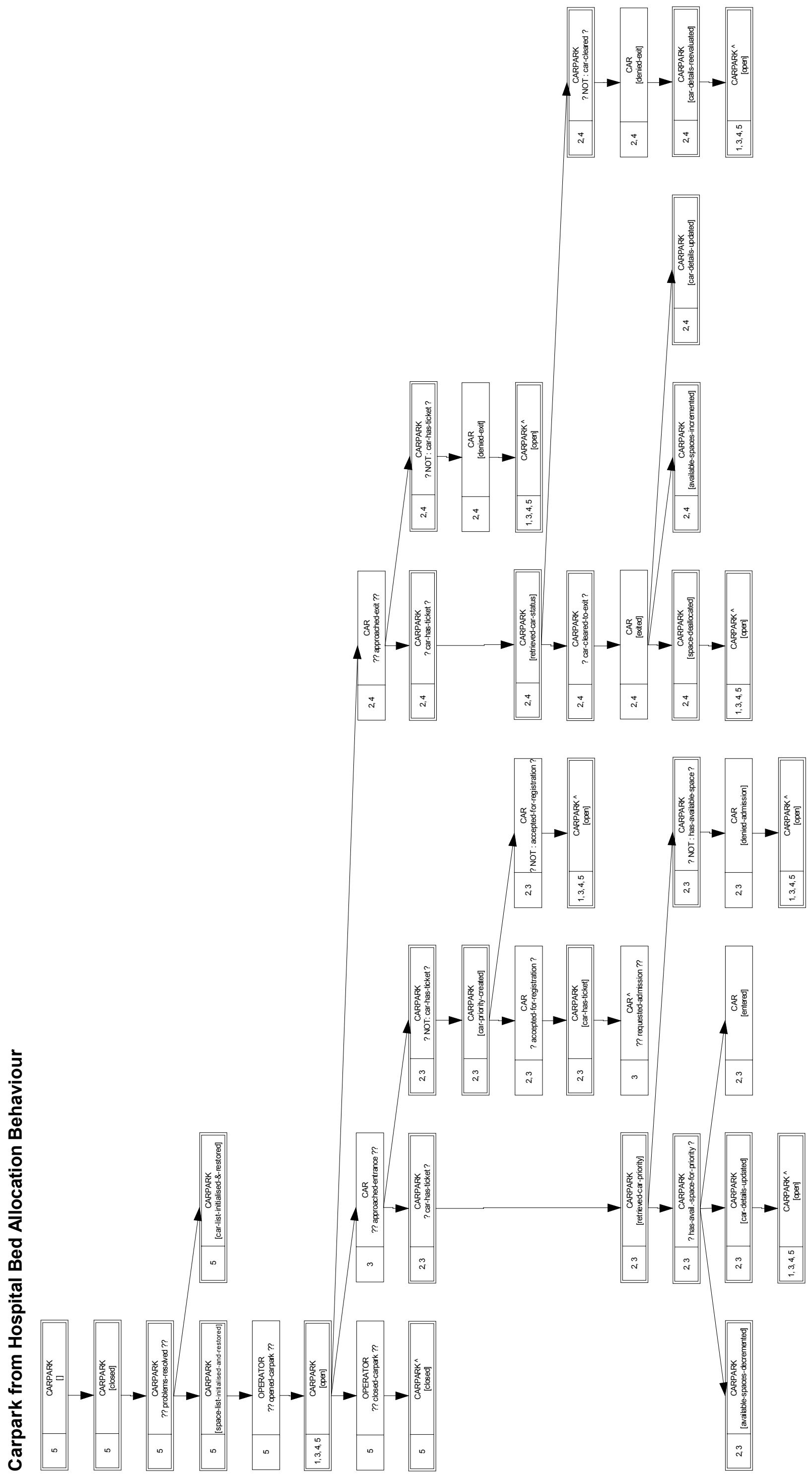

Carpark from Hospital Bed Allocation Behaviour
CARPARK []
CARPARK [closed]
CARPARK ?? problems-resolved ??
CARPARK [space-list-initialised-and-restored]
CARPARK [car-list-initialised-&-restored]
OPERATOR ?? opened-carpark ??
CARPARK [open]
OPERATOR ?? closed-carpark ??
CARPARK ^ [closed]
CAR ?? approached-entrance ??
CARPARK ? car-has-ticket ?
CARPARK ? NOT : car-has-ticket ?
CARPARK [car-priority-created]
CAR ? accepted-for-registration ?
CAR ? NOT : accepted-for-registration ?
CARPARK [car-has-ticket]
CAR ^ ?? requested-admission ??
CARPARK [retrieved-car-priority]
CARPARK ? has-avail.-space-for-priority ?
CARPARK ? NOT : has-available-space ?
CAR [denied-admission]
CAR [entered]
CARPARK [car-details-updated]
CARPARK [available-spaces-decremented]
CAR ?? approached-exit ??
CAR [denied-exit]
CARPARK [retrieved-car-status]
CARPARK ? car-cleared-to-exit ?
CARPARK ? NOT : car-cleared ?
CAR [exited]
CARPARK [space-deallocated]
CARPARK [available-spaces-incremented]
CARPARK [car-details-reevaluated]
CARPARK ^ [open]

Figure 5.46: Car Space Allocation Behaviour derived from the Hospital Bed Allocation System requirements

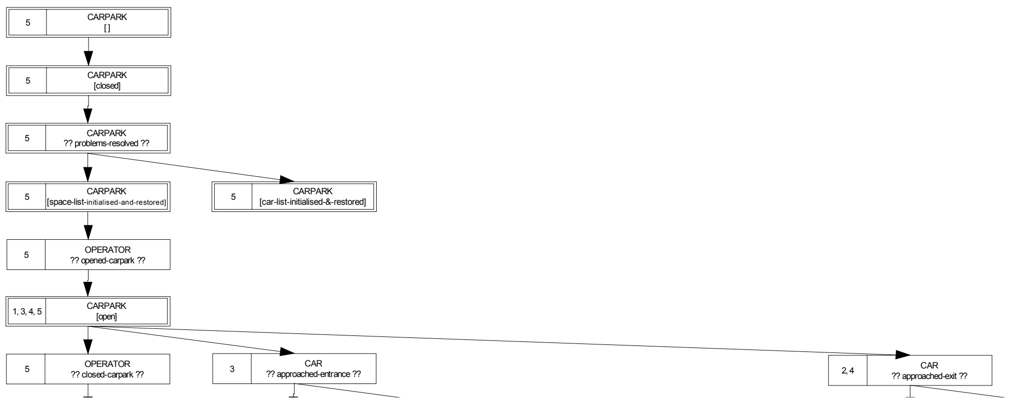


Figure 5.47: Car Space Allocation Initiation Process

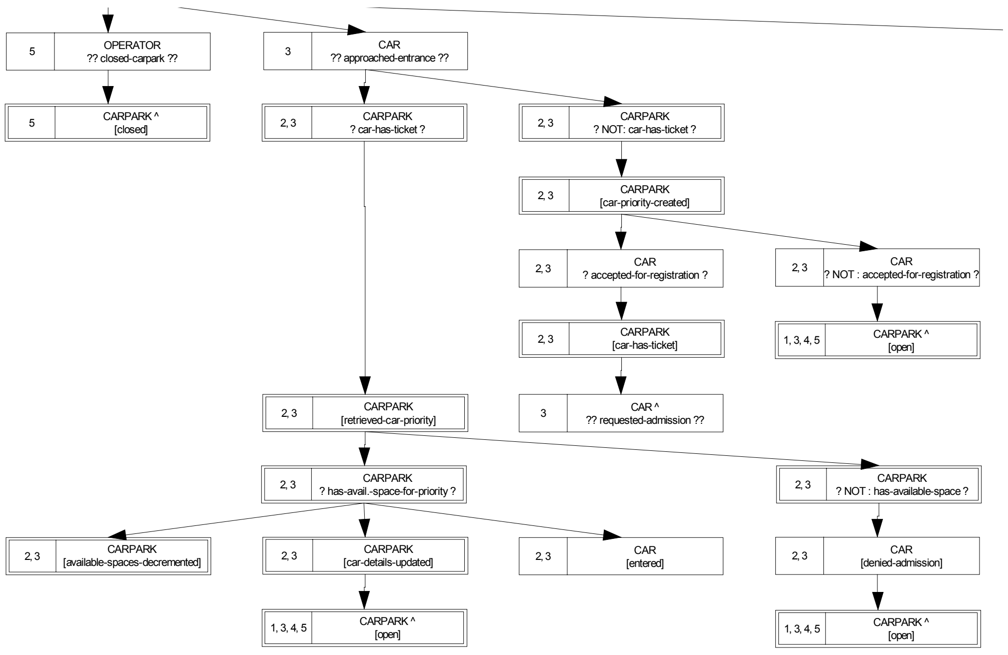


Figure 5.48: Car Space Allocation Exit Process

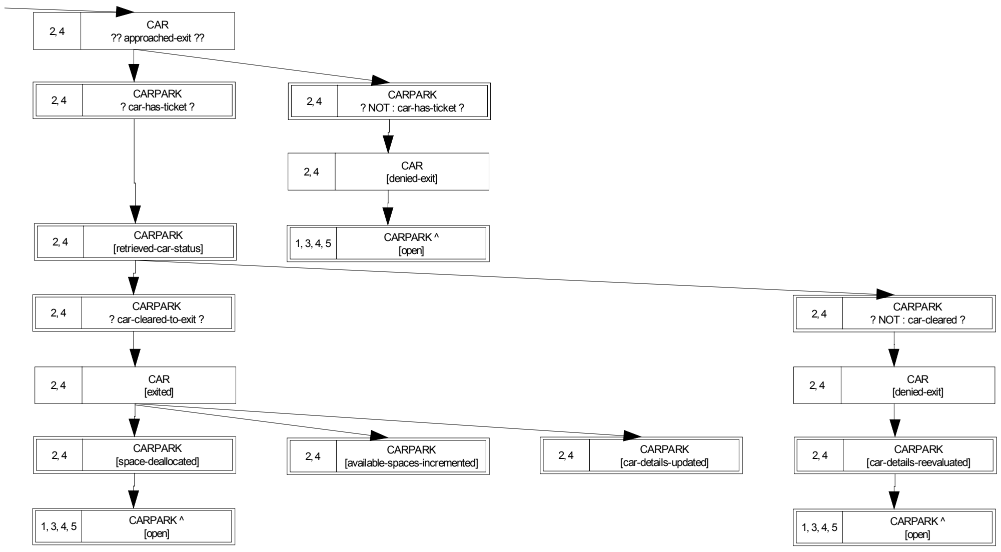


Figure 5.49: Car Space Allocation Entry Process

5.4.3 Translation Results

As a case study for allocation systems, a carpark car-to-space architecture, Figure 5.42 is compared with a hospital patient-to-bed architecture, Figure 5.34. Each architecture was translated into the alternate architecture, Figures 5.38 and 5.46.

The primary methodology for translation was by the careful replacement of component and component-state names with similes that were more meaningful in their new context, as detailed in Table 5.1.

By following this procedure it was found that the carpark and hospital systems were indeed specific instances of a generic allocation model. The overall process followed an identical sequence of events.

What was more interesting, was the differences between the two architectures. Granularity and composition techniques were used in each instance to allow more detailed descriptions of complex processes, and the breakdown of generic components into specialised sub-components.

Granularity relates to the hiding of insignificant information or details by abstraction. The ticket vending process for the carpark is a complex subsystem involving time and status tracking, paper feeding and printing. For the purposes of the allocation system, only the external interfaces to the ticket vendor are important. The internal workings are assumed.

For the hospital system, the registration process is far more complicated. A patient will have to be registered with the hospital and a preliminary determination of status and priority made, before a bed can be allocated. Hence it is important for this registration process to be described in more detail than the carpark system. If the carpark was to account for different types of carspace allocations with pay-by-month, early bird rates, and permanent spaces, the car registration and tracking system would need to be implemented to a similar extent as the hospital system.

The other issue is how the system is composed. For a particular section of functionality or behaviour, it was found that even though the overall process result may be the same, the actions taken to get to that result would differ. The process of admission to the carpark involves approaching a ticket-allocator box, taking the ticket, and entering when the boomgate is up. The semantics of the boomgate are to allow access only after registration. The hospital may have the admissions desk as a barrier-to-entry, but its workings are different to the physical ticket machine and boomgate of the carpark. However the combinations of the two carpark subsystems produce the same result.

Table 5.1: Component and Component-State names and their Equivalents used for the Example Architecture Translations

Allocation Systems		
Hospital	Carpark	Generic
Components		
hospital	carpark	container
admin	operator	operator
patient	car	client
bed	carspace	resource
registration	ticket	index
admissions	ticket-allocator	addition-point
departures	ticket-collector	removal-point
admissions	entry-boomgate	addition-point
departures	exit-boomgate	removal-point
payment	payment	usage-cost
Component-States		
available	opened	available
unavailable	closed	unavailable
finished-processing	boomgate-up	finished-processing
available	boomgate-down	available
registered	has-ticket	is-valid-index
admission	entry	addition
departure	exit	removal
requested-admission	approached-entrance	has-resource
requested-departure	approached-exit	requested-resource
cleared-for-departure	okay-to-exit	resource-available
bed-list	space-list	resource-list
patient-list	car-list	client-list
checked-out	departed	resource-removed

5.4.4 Component Adaptation

Figure 5.50 shows the initial attempt at adapting the Set Collection System for use as a Hospital Bed Allocation System. By changing the names of components such as SET to HOSPITAL and PRODUCER to ADMISSIONS, analogous behaviour has been created.

There are issues however with this adaptation. The Admissions Process highlighted in Figure 5.52 shows the way beds are allocated to patients. The Patient outputs a bed, and the Hospital adds this bed to its list of beds. The Hospital should be providing the bed to the Patient. This is the fundamental difference between a Collection System such as a Set and an Allocation System. Collections store new resources, and Allocation Systems track used resources.

To adapt the Set System for use in the Allocation case-studies, this storage functionality will have to be directly modified, or allowed for in the wrapper code. See Figures 5.54, 5.55, and 5.56 for the modifications made.

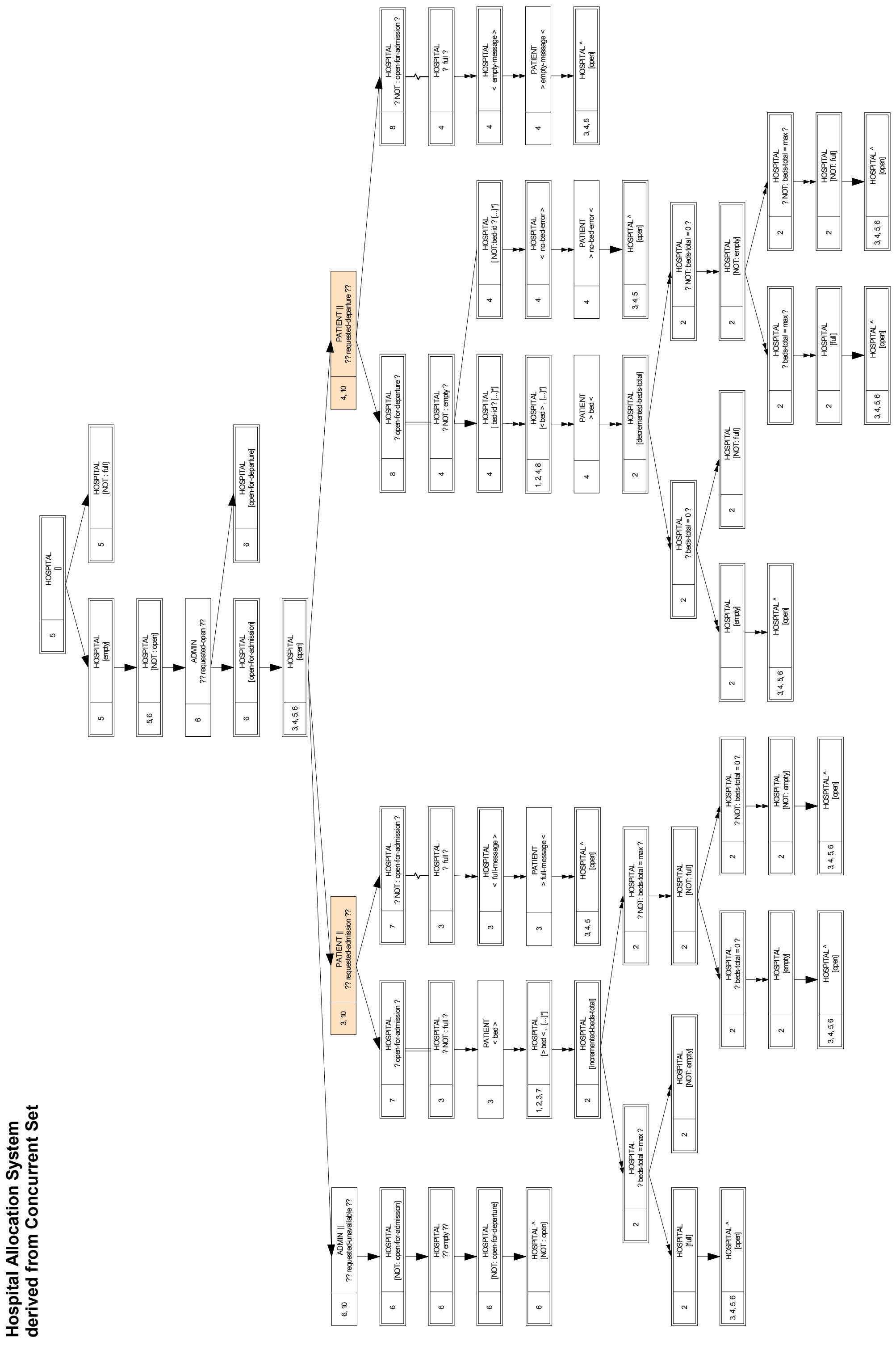


Figure 5.50: Hospital Bed Allocation Behaviour derived from the Set Collection requirements

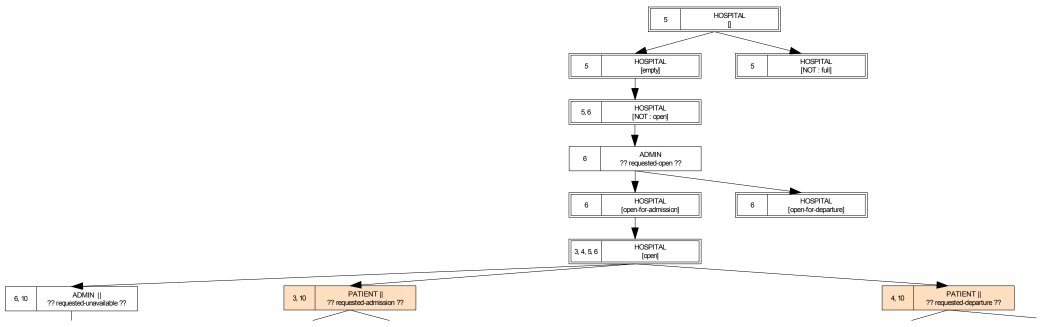


Figure 5.51: Hospital Bed Allocation Initiation Process

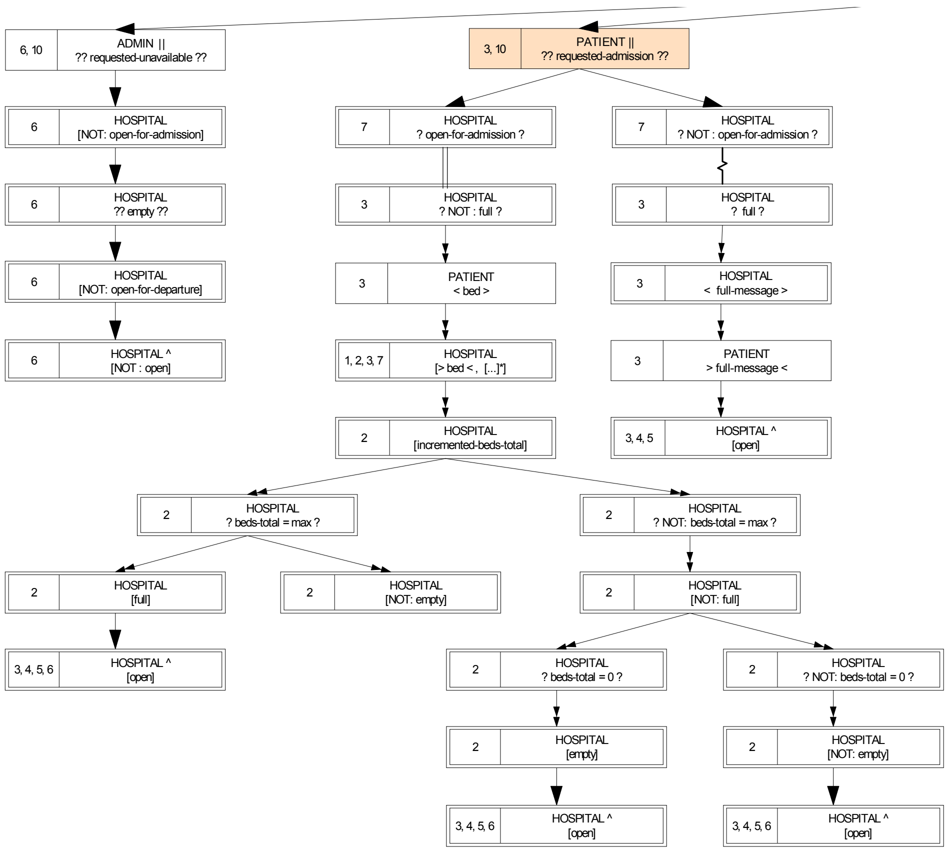


Figure 5.52: Hospital Bed Allocation Admission Process

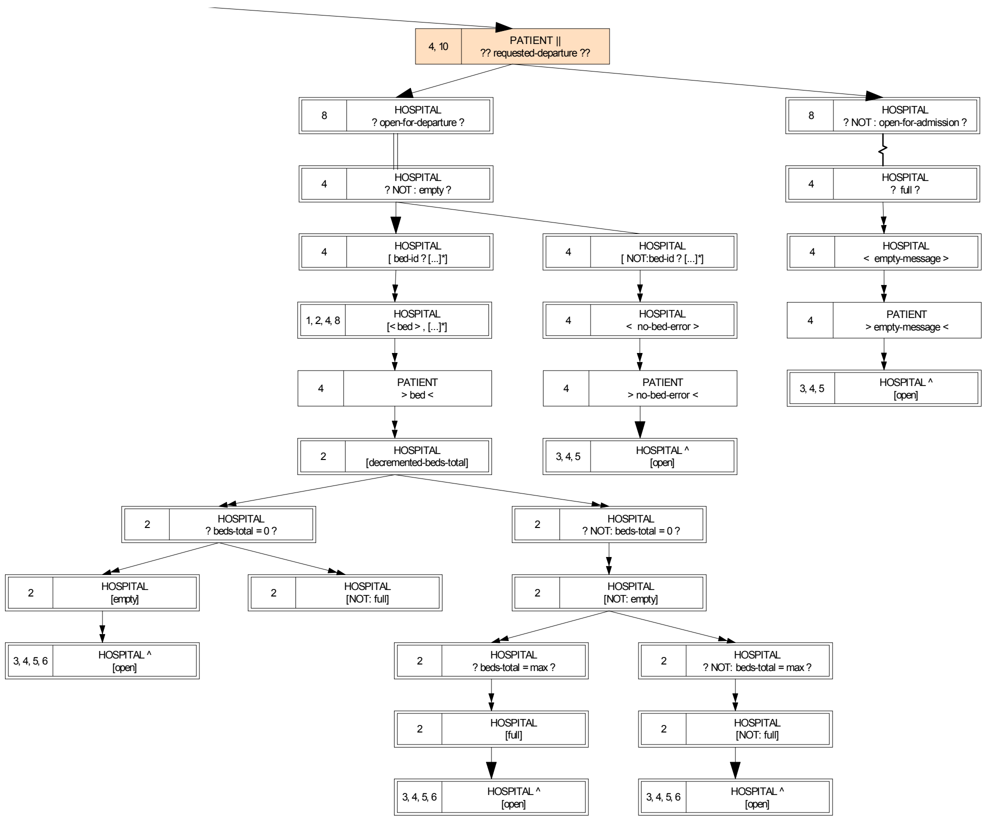


Figure 5.53: Hospital Bed Allocation Departure Process

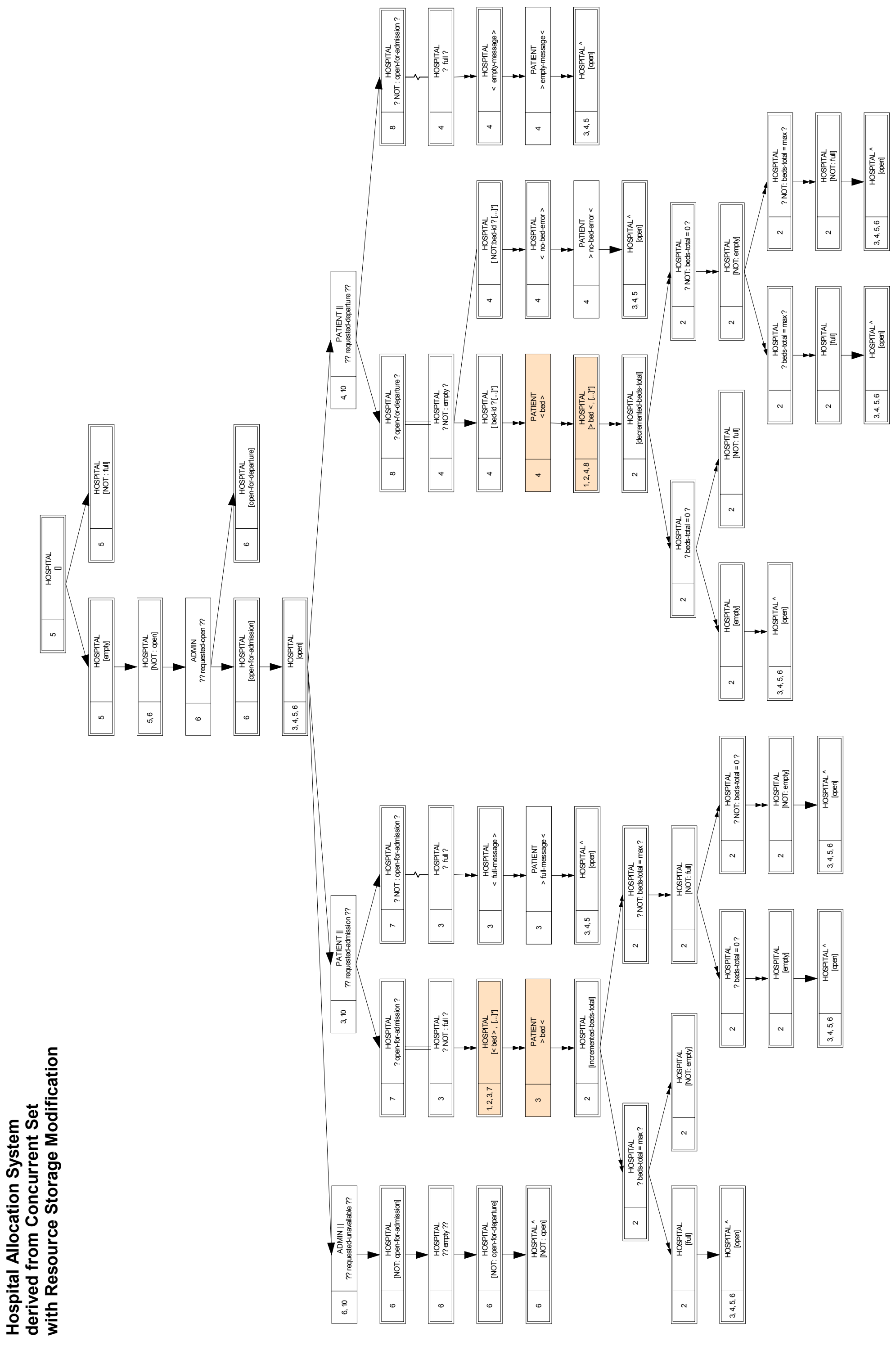

Hospital Allocation System
derived from Concurrent Set
with Resource Storage Modification

Figure 5.54: Hospital Bed Allocation Behaviour derived from the Set Collection requirements

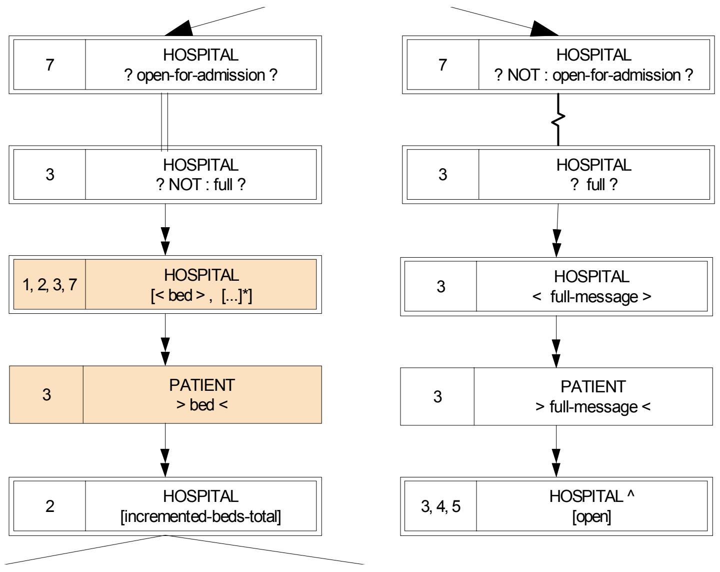


Figure 5.55: Hospital Bed Allocation Admission Process Alterations

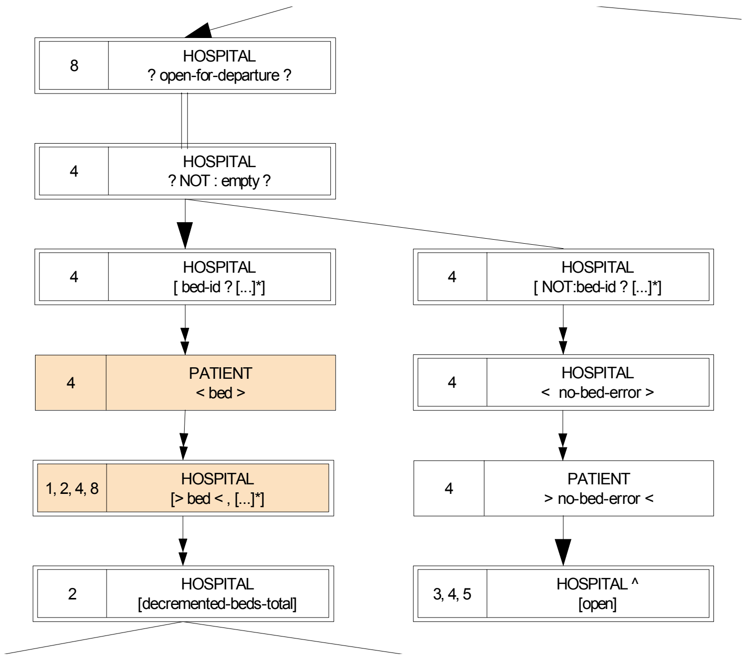


Figure 5.56: Hospital Bed Allocation Departure Process Alterations

5.4.5 Quality Identification

The last task for component re-use is quality identification. This section will undertake to identify some of the quality attributes defined in Section 5.1.3. This will be performed using the Behavior Tree architecture descriptions of the Carpark and Hospital Bed Allocation Systems discussed previously.

There is a point about quality that needs to be made. There is a difference between the quality of a system, and the quality inherent to the model used to represent the system. The discussion that follows generally talks about the quality of the system, but for some items, the quality of the Behavior Tree model is of more importance.

- Availability – The percentage of planned uptime during which system is actually available for use and fully operational. How much a system is usefully available to perform the work which it was designed to do.

  One of the main steps in our Collection and Allocation System case-studies was to add availability control. This was implemented by using a parallel process for an operator to gracefully shutdown and reset the main operations of the system. By implementing this external behavior, if the system undergoes a hardware failure, the operator can restore the system to operational status once the damage has been fixed.

  In terms of a failure of the boomgate or automatic ticket vendor in the Carpark system, the normal operations of the Carpark cease until the Operator restores the system. To implement higher levels of availability, only the processes directly affected by the failure could be shutdown. The exit gate could be closed, but the entrance could still operate correctly. This could introduce longer term issues, with the Carpark filling up. Should the cars be notified upon entry that there may be a problem leaving the Carpark later on?

- Integrity (Security) – The Trustworthiness of the system to be in the right state, with security intact.

As with Availability, it is important that the system can detect when it is operating outside of its intended behavior. Whether this is caused by hardware failure or malicious use, the system must retain its integrity. The primary mechanism for achieving this quality within Behavior Trees is the control flow notations. Equivalent to IF-THEN-ELSE operators, all the ELSE eventualities must be considered and implemented.

For the Carpark, if the Ticket-Vendor isn't operational, is it still possible for a Car to enter the Carpark for free and unnoticed? This case is allowed for by using the Boomgate barrier; a ticket must be taken before the Boomgate will lift, regardless of the operational state of the system.

In the Hospital, there is a problem with integrity. While the hospital is open, there is nothing to stop a person walking in a accessing restricted areas, and data, as well as consuming a bed, without the Allocation system noticing. To build a higher level of integrity into this system, either an area barrier system using ID swipe cards, or random security guard checks could be introduced. This would align the physical system with the software version.

Other techniques for integrity are required with parallel processes accessing the same data, as demonstrated in the case-studies. To maintain integrity of these shared, stored resources, mutual exclusion must be implemented using the Tight Control Flow notation.

- Efficiency – How well the system utilises processor capacity, disk space, memory, or communication bandwidth.

  One method of building efficiency into a Behavior Tree is by using threaded control flow, or concurrent processes. As mentioned previously, integrity of data is maintained using the Tight Control notation. At the cost of some complexity, this is the best way of ensuring availability and efficiency of the system to a large number of external producers and consumers.

The next major step required for adapting the Carpark and Hospital Allocation systems, is to introduce buffers, or queues at each of the major processing points. In reality this means, that cars must queue to enter and exit the Carpark. Without this capability, cars must randomly attempt to be the next to use the entrance, potentially causing some accidents. This is modelled by bottle-necking within the software system.

- Flexibility – How much effort is needed to add new capabilities to the product.

  One of the major tasks of this dissertation was to investigate the adaptability of Behavior Trees. By using techniques such as translation, pruning, integration, and projections, it is shown that any Behavior Tree architecture description will inherently have a high level of flexibility.

- Interoperability – How easily the product can exchange data or services with other systems. The ability to communicate easily with other assets, on different platforms and machines, and be integrated into an application.

  There is high complexity in implementing data communications using JavaBeans or .NET. The Behavior Tree model does not have this problem, because all communications between components are considered to be direct, instantaneous and transparent. Issues for distributed systems, such as time delay and data loss, are only considered at the implementation level at this stage. There is an argument that an architecture description does not need to model these issues. But as this is an important issue for Component-based development, Interoperability needs to be investigated further with more depth.

- Reliability – The degree to which the system does what it is intended to do, as opposed to something else. The probability of software executing without failure for a specified period of time. Percentage of correctly performed operations. The length of time the system runs before revealing a new defect.

This quality attribute is more concerned with the implementation of the system. Behavior Trees tend to be highly reliable, when developed correctly. One of the key benefits of using this methodology is its validation of the user requirements. Its dynamic representation of the behaviour of the system give the designer a good perspective overall. The more issues resolved with requirements at the architectural design stage, will produce a more reliable initial implementation.

- Robustness – The degree to which a system or component continues to function correctly when confronted with invalid input data, defects in connected software or hardware components, or unexpected operating conditions.

  By allowing for the case of hardware failure, in some of the automated systems in the Carpark, and by creating parallel processes for each subcomponent such as the entry and exit gates, robustness can be built into the system. To produce a high level of robustness, separate multiple entry and exit points could be introduced. If one point failed the system could still fully function, while being maintained.

- Usability – How well people are going to be able and be motivated to use the system practically.

  The flow of behaviour through an automated system such as a Carpark must be highly usable. Any issues will effect the traffic flow, and profitability of the system. The Hospital system does not need to be efficient, because versatile humans are part of the system. However to maintain morale, in such a sensitive environment, usability should carefully considered. How many patients are in the queue to being admitted, and why?

- Maintainability – How quickly an unreliable system can be brought to a reliable state.

  By allowing for early identification of hardware and malicious use, the system can go into a maintenance state. This state can be used by the system to validate all internal values, before re-initialising. The best way to introduce maintainability into Behavior Trees, is to allow for situations outside of normal operations. These situations should be modelled using Behaviour Trees just the same as normal functionality. For example, in the Carpark system, if a car breaks down or is unable to pay at one of the exit points, a procedure should be introduced to remove the car, and make sure the resource list is updated before resuming.

- Portability – The ease of moving a system from one environment to another. The effort required to migrate a piece of software from one operating environment to another. The ability to run on different platforms.

  The Behavior Tree model is very portable at an architecture level. It is a representation of the requirements of the system, rather than a binary implementation. By introducing the XML Schema version of the model, use by 3rd-party development software is now possible. Work will commence in the near future in producing a virtual machine version of Behavior Trees. This will allow the model to be run without limitations of operating system, or language.

- Reusability – The extent to which a software component can be used in applications other than the one for which it was initially developed.

  Much of the new work produced in this dissertation, has been the investigation of adaptability or reusability of the components represented by Behavior Trees. It is found that Behavior Tree models are highly reusable. Issues dealing with granularity are discussed in Section 5.4.3. Areas of incompatibility tend to be immediately picked up by the designer. This was demonstrated by adapting the Set Collection to a Hospital Allocation System. It was found that the way resources are stored between the two system is very different. However, once the problem was identified, adapting the system was very simple.

Chapter 6

Conclusions

6.1 Conclusions

The problem defined in Section 3.1 will be answered in the following summary.

1. Architecture determines quality attributes, and very little, must be known about the functionality of the system in order to draw an initial set of quality attribute conclusions. This is a central premise for the scope of this dissertation. Most quality attribute analysis and adaptability work has been produced relative to the software architecture.

2. Quality cannot be built into existing black-box components. These components cannot be modified after production. However, quality can be built into the system that integrates these components.

3. Determining the quality attributes of the system as a whole, requires that all components have an architecture description based on the same model, and have quality attributes specified. This dissertation proposes using XML specification of requirements and the Behavior Tree architecture, to allow for the automatic comparison of attributes between components.

   Certain requirements such as performance, and efficiency relate to the architecture, but more so to the implementation. C would be a better language to implement a performance-critical system, rather than Java. However Java has high Portability. The quality attributes for the architectural design would have to be modified to take these implementation details into account.

4. A developer can only analyse a software component for its quality attributes if:

   (a) the set of requirements used to originally develop the component, as well as the architecture design, and implementation must be complete, correct, consistent. However requirements are rarely complete, correct, consistent. By implementing the requirements within a Behavior Tree, requirements and the overall system behaviour can be validated. Having a single dynamic model of the system's behaviour, helps the designer to identify and correct overall errors and omissions in requirement logic and flow.

   (b) the quality attributes are described and quantified in a standard format, and implemented using a standard methodology. Unfortunately quality standards such ISO 9126, only talk about individual attributes from an abstract, high-level view. They may provide the trade-off relationships between attributes, but measurement or implementation of these attributes is not addressed. the emerging SQuaRE standard proposes a changing viewpoint of quality attributes, depending on the user and stage of development. At this stage it is not certain whether this standard will be released in a timely and usable manner.

5. By generating an initial architecture based only on a component's functional requirements, a component will result which has certain minimum levels of quality attributes. By transforming the structure of this Behavior Tree architecture description using pruning, projection, and augmentation techniques, the quality requirements for the component can be met.

Most of the minimum attribute levels relate to the architecture model used and the functionality as implemented. Behavior Trees have an inherent degree of portability, and reusability. So by generating an architecture description using Behavior Tree, a degree of reusability will already have been achieved. Whether this resusability is compromised or enhanced by the binary implementation of the system, is up to the developer and their choice of programming paradigm and style.

6. The final point deals with the integration and adaptation of existing components in new systems. These conclusions arise from a number of case-studies using the Stack, Queue, and Set Collections, and the Hospital Bed and Carpark Space Allocation Systems.

    - Analogous Behaviour – The Stack, Queue, and Set systems are all types of resource collections. The only thing that varies is the way resources are added or removed from the system. Hence these system are good candidates for being adapted for reuse. By attempting to reuse the Set component in the Hospital Bed Allocation system, it was found that there was a difference in how resources are used in Collections and Allocation Systems. The modification was minor. Overall analogous systems only require minor modifications to make translated behaviour conform to a new use. The validation benefits arising from having a different perspective on the same system, outweigh the modification issues.
    - Granularity – a simple system can be projected out of a more complex analogous system. In the Car Space Allocation System, more detail is placed in the behaviour of the Ticket Vendor and Collector sub-components with Boomgates, due to their automated nature. The Hospital Bed Allocation System, has people operating the Registration Desk, so the behaviour required is more general, and modelled using

only a single component.

- Quality Attributes – to introduce higher levels of efficiency, reliability, performance, maintainability to a system, parallel processes, as opposed to a linear sequence of operations, can be introduced. The functionality of the Behavior Trees is still very similar, except processes for addition, removal, and operation control can be separated, and scaled. An independent security process could be introduced to maintain the integrity of the system.

6.2 Summary of Contributions

This section identifies the contributions of new knowledge that are presented within this dissertation.

1. A detailed analysis of transforming existing Behavior Tree architectures using a different set of requirements is made. Primarily this has demonstrated a way of adapting existing components to allow for changing functional and quality requirements. See Sections 5.4.3 and 5.4.4.

2. Identification of a number of quality attributes within Behavior Tree architecture descriptions has been made, as a part of taking a set of requirements through to an architectural design, and adapting existing components for new situations. See Section 5.4.5.

3. A model for specifying quality attributes and functional requirements using XML is provided. See Section B.

4. An XML schema specification for Behavior Trees has been developed from the existing EBNF specification. See Sections A.3 and A.4.

5. Described in detail the overall process and issues with reusing existing software components in a new system.

6. An review of historical quality attribute specifications with respect to the emerging ISO15000 (SQuaRE) standard has been made.

6.3 Future Research

This section is a brief overview of areas and topics for future research, that are directly related to this dissertation.

- Update quality attribute specifications with best practice. (OOSPICE and ISO 15000 (SQuaRE) will soon be released.) As such, an alignment between product and process quality standards and best practice needs to more fully investigated.

- More detailed research into specific quality attributes, and analysis using Behavior Trees, needs to be made. This dissertation has presented a high-level view of all attributes with respect to case-studies and the Behavior Tree model itself.

- A process for developing components using implementation standards such as JavaBeans or DCOM, from Behavior Tree architecture descriptions needs to be developed.

- Traceability of requirements through architecture and code implementation need to be investigated. This will allow simulation of requirement behaviour and alignment to actual binary run-time code.

- Behavior Tree specification using XML needs to be updated with other current areas of research use. XML will allow transfer of Behavior Trees between various automated testing and analysis tools, so it is imperative that there is one implementation.

- Alignment of work with the new Model Driven Architecture middleware standard from OMG, as well as other open interface standards such as XMI and SOAP need to be investigated.

Chapter 7

Glossary of Terms

This is the definitions list of important terms used in this dissertation.

architectural design is the process of converting a set of requirements into a software architecture that fulfills, or at least facilitates the fulfillment of, the requirements. [12, Bosch(2000)]

Behavior Tree$^{\text{TM}}$ "is a formal tree-like graphical notation that represents the behaviour exhibited by individual or sets of entities, that realise or change states, make decisions, respond to or cause events, and interact by exchanging information and/or passing control." [20, Dromey (2002)]

Behavior Tree component

software architecture The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them. [7, Bass *et al.*(1998)]

software component A software component is a unit of composition with explicitly specified provided, required and configuration interfaces and quality attributes. [12, Bosch(2000)] [58, Weck *et al.*(1998)] additionally requires that components are subject to composition by third parties without adaptation.

software quality The degree to which software possesses a desired combination of attributes (e.g. reliability, interoperability) [16, IEEE 1061(1992)]

software reuse The systematic practice of developing software from a stock of building blocks, so that similarities in requirements and/or architecture between applications can be exploited to achieve substantial benefits in productivity, quality and business performance. [23, Ezan *et al.*(2002)]

system An integrated composite that consists of one or more of the processes, hardware, software, facilities and people, that provides a capability to satisfy a stated need or objective. [17, ISO/IEC 12207(1995)]

Chapter 8

Acronyms

This is the list of acronyms used in this dissertation.

CMM (or SW-CMM) Capability Maturity Model is a method for evaluating the maturity of an organisation's software development and maintenance processes.

CORBA Common Object Request Broker Architecture - OMG middleware framework standard for components distributed across hetergeneous networks. [23, Ezan *et al.*(2002)]

OMG Object Management Group

UML Unified Modelling Language - Standard method-independent language from OMG for specifying, visualising, constructing and documenting artifacts of software systems. [23, Ezan *et al.*(2002)]

OO Object-Oriented - An approach to software development based on the identification and classification of entities in the problem domain and their behaviours. [23, Ezan *et al.*(2002)]

CASE Computer-Aided Software Engineering - The use of software tools to support the software process. [23, Ezan *et al.*(2002)]

TQM Total Quality Management

COCOMO Constructive Cost Modelling

MDA Model Driven Architecture

XML eXtensible Markup Language - A meta-language in which markup languages for different classes of documents can be defined. [23, Ezan *et al.*(2002)]

XMI XML Metadata Interchange

SOAP Simple Object Access Protocol

MOF Meta Object Facility

PIM Platform Independent Model

PSM Platform-Specific Model

QAW Quality Attribute Workshop

CBSE Component-Based Software Engineering

QoS Quality-of-Service

COM Component Object Model - Microsoft framework standard for components running on a single machine. COM+ (also known as DCOM) is an extension for distributed computing. [23, Ezan *et al.*(2002)]

DCOM Distributed COM - Microsoft middleware for components distributed across networks of Windows platforms. Provides distribution and transaction management services. [23, Ezan *et al.*(2002)]

SGML Standard Generalised Markup Language

Bibliography

[1] D. E. Abel and T. P. Rout. Defining and specifying the quality attributes of software products. *Australian Computer Journal*, 25(3):105–112, Aug 1993. 4.2.6, 4.2.6, 4.2.6

[2] Hedley Apperly, Grady Booch, and Bill Councill. The near-term future of component-based software engineering. In *Component-Based Software Engineering: Putting the Pieces Together*, chapter 42. Addison-Wesley, Boston, 2001. 4.7.1

[3] Architecture Board ORMSC. Model driven architecture (MDA). Technical report, Object Management Group, Jul 2001. `http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01` (Accessed October 2002). 4.6.2

[4] Felix Bachmann, Len Bass, and Mark Klein. Illuminating the fundamental contributors to software architecture quality. Technical Report Technical Report CMU/SEI-2002-TR-025, Carnegie Mellon Software Engineering Institute, October 2002. `http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr025.pdf` Accessed October 2002. 2.2

[5] Mario R. Barbacci, Robert Ellison, Judith A. Stafford, Charles B. Weinstock, and William G. Wood. Quality attribute workshops. Technical Report Technical Report CMU/SEI-2001-TR-010, Carnegie Mellon Software Engineering Institute, May 2001. `http://seir.sei.cmu.edu/01tr010.pdf` Accessed March 2002. 4.7.4

[6] Mario R. Barbacci, Mark H. Klein, and Weinstock Charles B. Principles for evaluating the quality attributes of a software architecture. Technical Report Technical Report CMU/SEI-96-TR036, Carnegie Mellon Software Engineering Institute, May 1997. http://seir.sei.cmu.edu/tr036.96.pdf Accessed March 2002. 4.4, 4.7.4

[7] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. The SEI Series in Software Engineering. Addison Wesley Longman, Reading, Massachusetts, first edition, 1998. 2.2, 3.2, 4.1.1, 4.2.2, 4.2.2, 4.2.4, 4.5.2, 4.5.3, 4.5.4, 4.5.5, 4.5.5, 5, 7

[8] Len Bass, Mark Klein, and Felix Bachmann. Quality attribute design primitives. Technical Report Technical Note CMU/SEI-96-TR-017, Carnegie Mellon Software Engineering Institute, Dec 2000. http://seir.sei.cmu.edu/00tn017.pdf Accessed March 2002. 3.2, 4.2.7

[9] John Bergey, Mario Barbacci, and William Wood. Using quality attribute workshops to evaluate architectural design approaches in a major system acquisition: A case study. Technical Report Technical Note CMU/SEI-2000-TN-010, Carnegie Mellon Software Engineering Institute, Jul 2000. http://seir.sei.cmu.edu/00tn010.pdf Accessed March 2002. 3.2, 3.3, 4.7.4

[10] B. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, N.J., 1981. 3.2

[11] B. W. Boehm, J. R. Brown, M. Lipow, G. J. MacLeod, and M. J. Merritt. *Characteristics of Software Quality*. Elsevier North-Holland, New York, first edition, 1978. 4.1.3

[12] Jan Bosch. *Design and Use of Software Architectures*. Addison-Wesley, Harlow, England, first edition, 2000. 2.2, 3.2, 3.3, 4.5.7, 4.5.7, 4.5.7, 7

[13] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (SOAP) 1.1. W3c note, World Wide Web Consortium, May 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508 (Accessed October 2002). 4.6.2

[14] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0 (second edition). W3c recommendation, World Wide Web Consortium, Oct 2000. http://www.w3.org/TR/REC-xml (Accessed October 2002). 4.6.2

[15] David Carney. Assembling large systems from COTS components: Opportunities, cautions and complexities. Technical report, Carnegie Mellon Software Engineering Institute, Jun 1997. 5.2

[16] Software Engineering Technical Committee. Standard for a software quality metrics methodology. Standard 1061, The Institute of Electrical and Electronics Engineers, New York, 1992. 7

[17] Software Engineering Technical Committee. Standard for information technology software life cycle processes. Standard 12207.0, International Organization for Standardization/International Electrotechnical Commission, New York, 1995. 4.4, 5.1.1, 5.1.1, 7

[18] Michael S. Deutsch and Ronald R. Willis. *Software Quality Engineering: A Total Technical and Management Approach*. Prentice Hall, Englewood Cliffs, N.J., first edition, 1988. 4.1.3, 4.2.6, 4.2.6, 4.2.6

[19] R. G. Dromey. Genetic software engineering. Technical report, Software Quality Institute, Jan 2001. http://www.sqi.gu.edu.au/gse/papers/GSE-IEEE-Final.pdf (Accessed April, 2002). 4.6.1

[20] R. G. Dromey. EBNF for behaviour tree notation. Technical report, Software Quality Institute, Jul 2002. http://www.sqi.gu.edu.au/gse/papers/EBNF-Behaviour-Tree.pdf (Accessed July, 2002). 7, A, A.1

[21] R. G. Dromey. Genes, jigsaw puzzles and software engineering. Technical report, Software Quality Institute, Jun 2002. http://www.sqi.gu.edu.au/gse/papers/Genes-Jigsaws-SE.pdf (Accessed July, 2002). 4.6.1

[22] R. Geoff Dromey. A model for software product quality. In Robin B. Hunter and Richard H. Thayer, editors, *Software Process Improvement*, pages 146–462. IEEE Computer Society Press, United States of America, 1995. http://www.sqi.gu.edu.au/publications/Model_prod_qual.ps (Accessed April, 2002). 4.1.3, 4.1.3, 4.1.3

[23] Michel Ezran, Maurizio Morisio, and Colin Tully. *Practical Software Reuse*. Practitioner Series. Springer-Verlag, London, first edition, 2002. 7, 8

[24] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. ITP, London, UK, second edition, 1997. 4.3

[25] Donald G. Firesmith and Brian Henderson-Sellers. The OPEN process framework - an overview. Final Draft, May 2001. 4.7

[26] Tom Gilb. *Principles of Software Engineering Management*. Addison-Wesley Publishing Company, Reading, Massachusetts, first edition, 1988. 4.2, 4.2.1, 4.3, 5

[27] Object Management Group. OMG-XML metadata interchange (XMI) specification, v1.2. Formal specification, Object Management Group, Jan 2002. http://www.omg.org/cgi-bin/doc?formal/02-01-01.pdf (Accessed October 2002). 4.6.2

[28] Kenneth A. Lambert and Martin Osborne. *Java: A Framework for Program Design and Data Structures*. Brooks/Cole, Australia, first edition, 2000. 5.3

[29] P. Lawlis et al. pages 21–25, Sep 1995. 4.4

[30] Carma MacClure. *Software Reuse: A Standards-Based Guide*. IEEE Computer Society, Los Alamitos, California, 2001. 4.4

[31] M. D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Software Engineering, Report on a Conference Sponsored by the NATO Science Committee*, 1969. 3.3, 4.5.7

[32] Andreas Polze. Building blocks for achieving quality of service for commercial off-the-shelf (COTS) middleware. Technical Report Technical Note CMU/SEI-99-TR001, Carnegie Mellon Software Engineering Institute, May 1999. `http://seir.sei.cmu.edu/99tr001.pdf` Accessed March 2002. 4.7.5

[33] William A. Ruh, Francis X. Maginnis, and William J. Brown. *Enterprise Application Integration*. Wiley Tech Brief Series. John Wiley and Sons, Inc, New York, first edition, 2000. 4.5.6

[34] Robert C. Seacord, David Mundie, and Somjai Boonsiri. K-BACEE: A knowledge-based automated component ensemble evaluation tool. Technical Report Technical Note CMU/SEI-2000-TN-015, Carnegie Mellon Software Engineering Institute, Dec 2000. `http://seir.sei.cmu.edu/00tn015.pdf` Accessed March 2002. 3.2, 4.7.6

[35] Secretariat. Software product evaluation – quality characteristics and guidelines for their use. Technical Standard ISO 9126, International Organisation for Standardisation, Geneva, 1991. 4.1.3, 4.1.3, 4.2.6, 4.2.6, 5

[36] JSTC1/SC7 Secretariat. 9126-10: Software engineering software quality general overview, reference models and guide to software product requirements and evaluation (square). Working Document N2596, ISO/IEC JTC1/SC7, Apr 2002. `http://www.sqi.gu.edu.au/`

sc7-mirror/private_files/07n2596.pdf (accessed March 2002). 4.2.5, 5, 5.1.2

[37] JSTC1/SC7 Secretariat. Iso/iec 9126-30: Software engineering software product quality requirements and evaluation part 30: quality metrics metrics reference model and guide. Working Document N2582, ISO/IEC JTC1/SC7, Jan 2002. http://www.sqi.gu.edu.au/sc7-mirror/private_files/07n2582.pdf (accessed March 2002). 4.2.5

[38] JSTC1/SC7 Secretariat. Wd 12119 software engineering software product evaluation requirements for quality of commercial off the shelf software product (cots) and instructions for testing. Working Document N2598, ISO/IEC JTC1/SC7, Mar 2002. http://www.sqi.gu.edu.au/sc7-mirror/private_files/07n2598.zip (accessed March 2002). 4.7.3

[39] Mary Shaw. Truth vs knowledge: The difference between what a component does and what we know it does, Mar 1996. 4.7.5

[40] Jon Siegel and OMG Staff Strategy Group. Developing in OMG's model-driven architecture. White paper, Object Management Group, Nov 2001. ftp://ftp.omg.org/pub/docs/omg/01-12-01.pdf (Accessed October 2002). 4.6.2

[41] Richard Soley and OMG Staff Strategy Group. Model driven architecture. White paper, Object Management Group, Nov 2000. ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf (Accessed October 2002). 4.6.2

[42] Alistair Sutcliffe. *The Domain Theory; Patterns for Knowledge and Software Reuse*. Lawrence Erlbaum Associates, Mahwah, New Jersey, first edition, 2002. 4.5.1

[43] Will Tracz. COTS myths and other lessons learned in component-based software development. In *Component-Based Software Engineering: Putting the Pieces Together*, chapter 6. Addison-Wesley, Boston, 2001. 4.7.2

[44] J. Voas, G. McGraw, and A. Ghosh. Gluing together software components: How good is your glue? In *Proceedings of Pacific Northwest Software Quality Conference*, Oct 1996. http://www.cigital.com/papers/download/pnsqc96.pdf (Accessed April 2002). 4.7.5

[45] J. Voas and K. Miller. Improving the software development process using testability research, 1992. http://www.cigital.com/papers/download/issre92.pdf (Accessed April 2002). 4.7.5

[46] J. Voas and K. Miller. Software testability: The new verification. *IEEE Software*, May 1995. http://www.cigital.com/papers/download/ieeesoftware95.pdf (Accessed April 2002). 4.7.5

[47] J. Voas and J. Payne. Cots software failures: can anything be done? In *Proceedings of the First IEEE Workshop on Application Specific Software Engineering and Technology (ASSET 98), Dallas*, Mar 1998. http://www.cigital.com/papers/download/ASSET98.ps (Accessed April 2002). 4.7.5

[48] Jeffrey Voas. Mitigating the potential for damage caused by cots and third-party software failures. In *Proceedings of AQUIS 98, Venice*, Apr 1988. http://www.cigital.com/papers/download/ven.ps (Accessed April, 2002). 4.7.5

[49] Jeffrey Voas. Testing software for characteristics other than correctness: Safety, failure tolerance, and security, 1996. http://www.cigital.com/papers/download/ictcs96.pdf (Accessed April, 2002). 4.7.5

[50] Jeffrey Voas. Tolerant software interfaces: Can cots-based systems be trusted without them? In *Proceedings of the 15th International Conference on Computer Safety, Reliability, and Security (SAFECOMP 96), Vienna*, Oct 1996. http://www.cigital.com/papers/download/safecomp96.pdf (Accessed April, 2002). 4.7.5

[51] Jeffrey Voas. Using assertions to make untestable software more testable. *Software Quality Professional*, 1997. `http://www.cigital.com/papers/download/ESEC97.ps` (Accessed April, 2002). 4.7.5

[52] Jeffrey Voas. Cots: The economical choice? *IEEE Software (Manager Column)*, Mar 1998. `http://www.cigital.com/papers/download/ieee_softwaremanager.ps` (Accessed April, 2002). 4.7.5

[53] Jeffrey Voas. Defensive approaches to testing systems that contain cots and third-party functionality. In *Proceedings of 15th International Conference and Exposition on Testing Computer System*, Jun 1998. `http://www.cigital.com/papers/download/ictcs98.ps` (Accessed April, 2002). 4.7.5

[54] Jeffrey Voas. Upgrading software maintenance for components. *IEEE Software*, Jul 1998. `http://www.cigital.com/papers/download/maintain.ps` (Accessed April, 2002). 4.7.5

[55] Jeffrey Voas. Software malleability: Were losing it! In *Proceedings of the 2nd Annual Systems Engineering and Supportability Conference*, September 1999, San Diego, CA. 1999. `http://www.cigital.com/papers/download/ndia99.pdf` (Accessed April, 2002). 4.7.5

[56] Jeffrey Voas. This decades eight greatest myths about software quality. *IEEE Software*, Jul 1999. `http://www.cigital.com/papers/download/qt4.ps` (Accessed April, 2002). 4.1.2, 1, 4.7.5

[57] Jeffrey Voas. Third-party usage profiling: A model for optimizing the mass-marketed software industry. *IEEE Software*, 2000. `http://www.cigital.com/papers/download/product-line.pdf` (Accessed April, 2002). 4.7.5

[58] W. Weck, J. Bosch, and C. Szyperski. In *Proceedings of the Third International Workshop on Component Oriented Programming (WCOP'98)*, num-

ber 10 in General Publications. Turku Center for Computer Science, TUCS, Sep 1998. 7

[59] Karl Wiegers. Karl wiegers describes 10 requirements traps to avoid. *Software Development*, 4(5), May 1996. `http://processimpact.com/articles/reqtraps.pdf` Accessed June 2002. 2.1

[60] Karl Wiegers. *Software Requirements*. Microsoft Press, Redmon, Washington, first edition, 1999. 4.2.3

Appendix A

Behavior Tree Architecture Description

The purpose of this appendix is to show the current form and notation of Behavior Tree$^{\mathrm{TM}}$ specifications from [20, Dromey (2002)].

A Behavior Tree "is a formal tree-like graphical notation that represents the behaviour exhibited by individual or sets of entities, that realise or change states, make decisions, respond to or cause events, and interact by exchanging information and/or passing control." [20, Dromey (2002)].

As introduced in Section 4.6.1, Behavior Trees will be used for the representation of the software architecture of a software component or system. Its advantages include:

- It allows a clear, consistent and complete methodology for implementing a set of software requirements as a software architecture.
- Transformation, by augmentation and pruning, of the architecture can be easily achieved with changing requirements, or the application of quality attributes.
- A standard XML Schema form of the EBNF version of Behavior Trees can be directly created, providing a way for developers to use third-party tools

to automatically examine a software architecture. Future development will include tools for compiling and running Behavior Trees, independent of implementation.

A.1 Behavior Tree Notation - Graphical Form

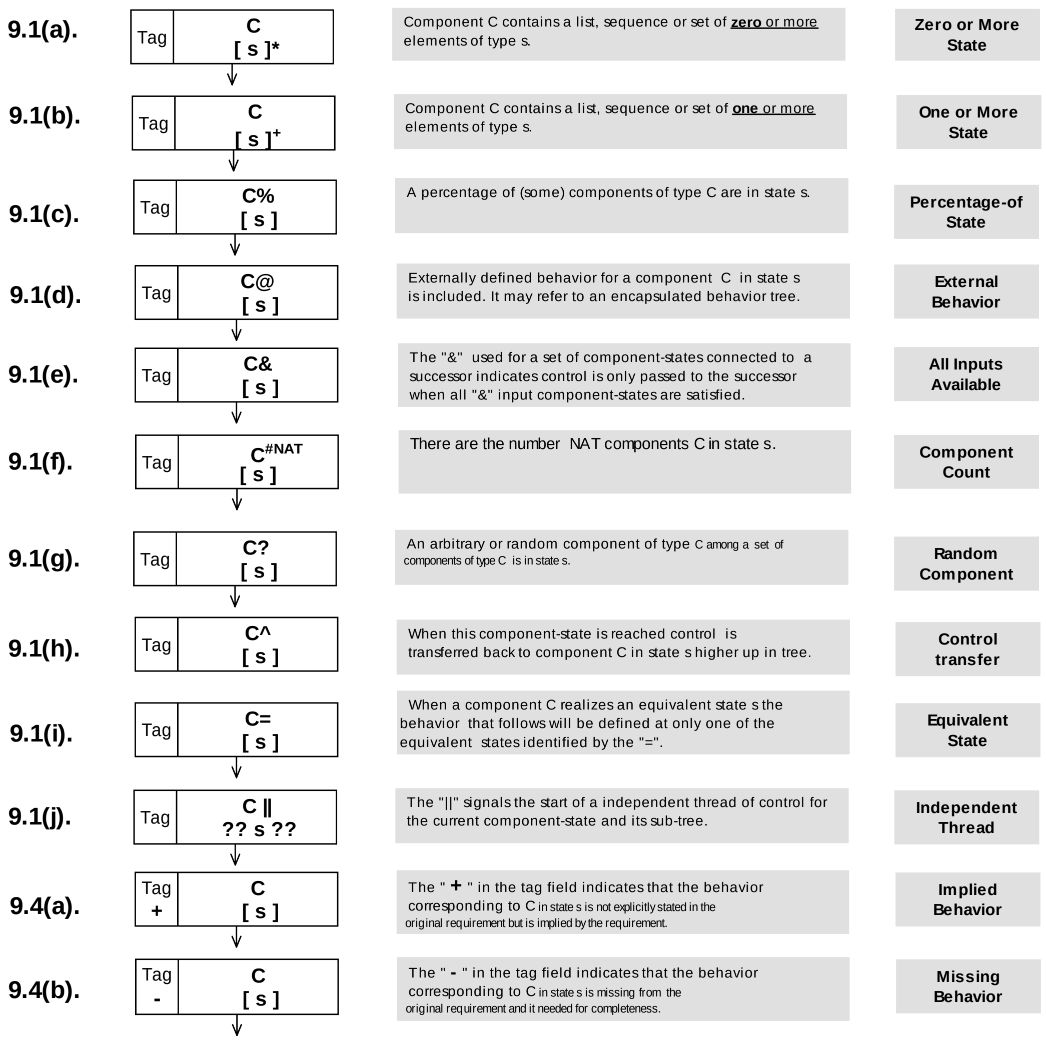
This graphical version of the elements that make up Behavior Trees is referenced from the specification document [20, Dromey (2002)]. This should be used as a guide to understanding the Behavior Tree notation used in the later examples.

1.1(a)	Tag | **C** **[s]**	Component **C** realizes *state* **s** then passes control to its output	**State Realization**
1.1(b)	Tag | **C** **[s]**	System Component **C** realizes the system *state* **s** then passes control to its output.	**System State Realization**
1.1(c)	Tag | **C** **? b ?**	If component C is in state b it passes control to its output. This notation is used to model decisions/conditions. If b does not hold flow of control through this component-state is blocked	**Decision**
1.1(d)	Tag | **C** **? NOT : b ?**	If component **C** is in *state* " **NOT : b**" it passes control to its output. This notation is used to model decisions/conditions. Only <u>one</u> condition from a set emanating from a node is true.	**Negation**
1.1(e)	Tag | **C** **?? b ??**	When and if component **C** realizes *state* " **b**" it passes control to its output. This notation is used to model events. Only <u>one</u> out of a set of alternative events can ever be true at one time.	**Event**
1.1(f)	Tag | **C** **< s >**	Component **C** outputs the *state*, *data* or *attribute* value "s" to the component(s) connected to its output.	**Output**
1.1(g)	Tag | **C** **> s <**	Component **C** inputs the *state*, *data* or *attribute* value "s" **from** the component connected to its input then passes control to the component connected to its output..	**Input**
1.1(h)	Tag | **C** **[a := exp]**	Component **C** realizes the state that its *attribute* "**a**" is assigned the value of the *expression*, *property* or *state* "**exp**". It then passes control to the component connected to its output.	**Attribute Assignment**
1.2(a)	Tag | **C** **?? < s > ??**	When and if component **C** generates an output event the component outputs the *state*, *data* or *attribute* value "s" to the component(s) connected to its output.	**Output Event**
1.2(b)	Tag | **C** **?? > s < ??**	When and if component **C** receives an input event the component transfers control to the component(s) connected to its output.	**Input Event**
1.2(c)	Tag | **C** **<> s <>**	When and if component **C** receives an input "**s**" the component transfers "**s**" to the component(s) connected to its output.	**Input/ Output**
1.2(d)	Tag | **C** **? ELSE ?**	If all other conditions associated with component-states emanating from a given node of control are false then control is passed through the "ELSE" branch to its children.	**Default Condition**
1.2(e)	Tag | **C** **? ERROR ?**	If all other conditions associated with component-states emanating from a given node of control are false then control is passed through the "ERROR" branch to its children.	**Error Condition**
1.3(a)	Tag | **C** **[s1 | s2 | ...| sn]**	A component **C** can have associated with it one or more non-overlapping set of states that are used to define its behavior. Only one state in a set exists at any one time.	**State-set Definition**
1.3(b)	Tag | **C** **{ s }**	Component C in state s is a projected component state. The bracketing indicates that a parent or higher component in the projection environment involved a decision or event.	**Choice Projection**

Figure A.1: Graphical Behavior Tree Notation - Part 1

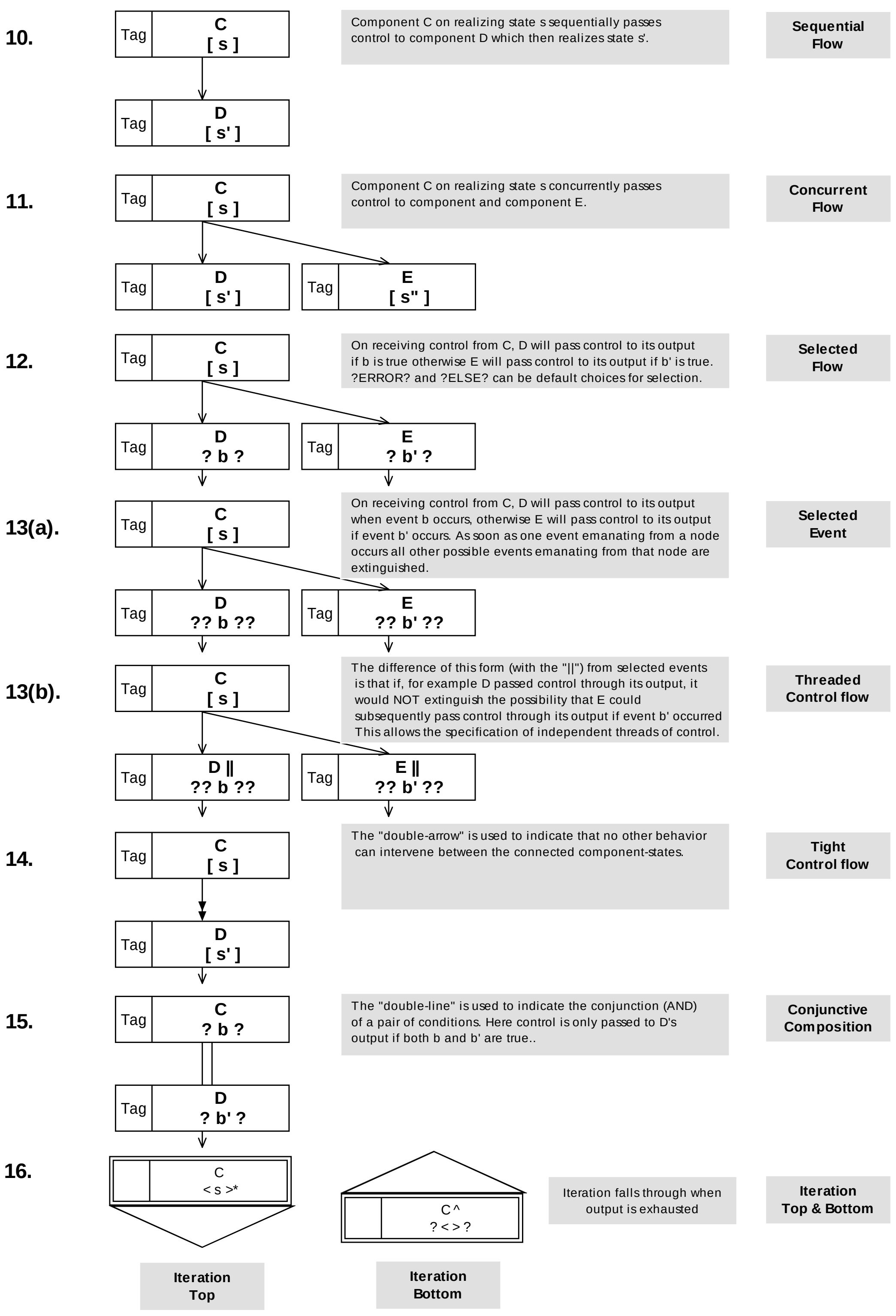
1.4(a).	Tag	C < b> ; [...]*	This says remove b from the front of the list of b's contained within C and output it. An element can also be output from them end of a list. The [...] refers to a list of implied b's.	**List-output Delete element**
1.4(b).	Tag	C [> b< ; [...]*]	This says input b and add it to the front of the list of b's contained within C. An element can also be input and added to the end of the list.	**List-input Add element**
1.4(c).	Tag	C [b , [...]*]	The [...] refers to an implied set that assumes the type of the accompanying element that is associated. Here "b" is added to a set of b's in C	**Set- Add element**
1.4(d).	Tag	C [b ; [...]*]	The [...] refers to an implied collection (list, sequence or set) that assumes the type of the accompanying element that is associated. Here "b" is prepended to a sequence of b's in C	**Sequence Prepend**
1.4(e).	Tag	C [[...]* ; b]	The [...] refers to an implied collection (list, sequence or set) that assumes the type of the accompanying element that is associated. Here "b" is appended to a sequence of b's in C	**Sequence Append**
1.5(a).	Tag	C [b ? [...]*]	This says pass control to the component(s) connected to C's output if "b" is in the collection of b's.	**Collection Search**
1.5(b).	Tag	C [b × [...]*]	This says delete b from the collection of b's contained within C then pass control to C's output.	**Delete Element**
2(a).	Tag	C' :: C" [s]	The relationship between component **C'** and component **C"** has realized the state **s**	**Relation State**
2(b).	Tag	C' : C" [s]	Component **C'** which is also known as **C"** or **C'** which is of type **C"** has realised state **s**	**Alias State**
3(a).	Tag	C [C'[s']]	Component **C** has realized the state that C' is in state s' **and component C' has realized state s'**	**Composite State**
3(b).	Tag	C [[s'] C']	Component **C** has realized the state that the state s' **has been realized by component C' . the component C' has in turn realised the state s'**	**Composite State**
6.1(a).	Tag	C [ ]	Component **C** is in an uninitialized state or all members of a sequence or set associated with C have been processed or component C has had its states erased or is a new component.	**Uninitialized State**
6.1(b).	Tag	C [×]	Dynamic component **C** is removed from the current context.	**Remove Component**
8.1(a).	Tag	C [s / q]	Component C is in state s qualified or restricted to those with the property q. It is also possible for C to be directly qualified	**Remove Component**
8.1(b).	Tag	C \\ e [s]	The set of components of type C, excluding e, are in the state s. Exclusion is a form of qualification.	**Exclusion**

Figure A.2: Graphical Behavior Tree Notation - Part 2



9.1(a).	Tag | C $[s]^*$	Component C contains a list, sequence or set of **zero** or more elements of type s.	**Zero or More State**
9.1(b).	Tag | C $[s]^+$	Component C contains a list, sequence or set of **one** or more elements of type s.	**One or More State**
9.1(c).	Tag | C% [s]	A percentage of (some) components of type C are in state s.	**Percentage-of State**
9.1(d).	Tag | C@ [s]	Externally defined behavior for a component C in state s is included. It may refer to an encapsulated behavior tree.	**External Behavior**
9.1(e).	Tag | C& [s]	The "&" used for a set of component-states connected to a successor indicates control is only passed to the successor when all "&" input component-states are satisfied.	**All Inputs Available**
9.1(f).	Tag | $C^{\#NAT}$ [s]	There are the number NAT components C in state s.	**Component Count**
9.1(g).	Tag | C? [s]	An arbitrary or random component of type C among a set of components of type C is in state s.	**Random Component**
9.1(h).	Tag | C^ [s]	When this component-state is reached control is transferred back to component C in state s higher up in tree.	**Control transfer**
9.1(i).	Tag | C= [s]	When a component C realizes an equivalent state s the behavior that follows will be defined at only one of the equivalent states identified by the "=".	**Equivalent State**
9.1(j).	Tag | C ‖ ?? s ??	The "||" signals the start of a independent thread of control for the current component-state and its sub-tree.	**Independent Thread**
9.4(a).	Tag + | C [s]	The " **+** " in the tag field indicates that the behavior corresponding to C in state s is not explicitly stated in the original requirement but is implied by the requirement.	**Implied Behavior**
9.4(b).	Tag - | C [s]	The " **-** " in the tag field indicates that the behavior corresponding to C in state s is missing from the original requirement and it needed for completeness.	**Missing Behavior**

Figure A.3: Graphical Behavior Tree Notation - Part 3



10. Component C on realizing state s sequentially passes control to component D which then realizes state s'. — **Sequential Flow**

11. Component C on realizing state s concurrently passes control to component and component E. — **Concurrent Flow**

12. On receiving control from C, D will pass control to its output if b is true otherwise E will pass control to its output if b' is true. ?ERROR? and ?ELSE? can be default choices for selection. — **Selected Flow**

13(a). On receiving control from C, D will pass control to its output when event b occurs, otherwise E will pass control to its output if event b' occurs. As soon as one event emanating from a node occurs all other possible events emanating from that node are extinguished. — **Selected Event**

13(b). The difference of this form (with the "||") from selected events is that if, for example D passed control through its output, it would NOT extinguish the possibility that E could subsequently pass control through its output if event b' occurred This allows the specification of independent threads of control. — **Threaded Control flow**

14. The "double-arrow" is used to indicate that no other behavior can intervene between the connected component-states. — **Tight Control flow**

15. The "double-line" is used to indicate the conjunction (AND) of a pair of conditions. Here control is only passed to D's output if both b and b' are true.. — **Conjunctive Composition**

16. Iteration falls through when output is exhausted — **Iteration Top & Bottom**

Figure A.4: Graphical Behavior Tree Notation - Part 4

A.2 Behavior Tree Notation - Textual Form

This section shows the current specification for Behavior Trees in Extended Backus-Naur Form (EBNF). BNF grammar rules were first used in the description of the Algol60 language. EBNF allows for repetition and optional operations within BNF.

EBNF allows the rules for how a Behavior Tree can be structured to be explicitly expressed in a standard form. However it is not easy for people to read, and is difficult to implement within a language, especially if modifications are required. Section A.3 will use this structure to create a software-friendly version using the new standard XML Schema specification form.

```
1. Behavior Tree Tree ::= Node(C,s) | Node(C,s) ; (Tree, Tree,  ,
   Tree ) | Empty Nodes and Component-States Node(C,s) ::= C{s} |
   C{s}//Tag | [C{s}] | [C{s}//Tag]

2. C{s} ::= C OP[s]OP1 | C?b? | C?NOT:b? | C??b?? | C<s>OP1 | C>s<
   OP1 | C[a:=exp]OP1

3. C{s} ::= C??<s>??OP1 | C??>s<??OP1 | C<>s<>OP1 | C?ELSE? |
   C?ERROR?

4. C{s}::= C[ {[s | s |  | s]}+ ] | C{s}

5. C{s} ::= C<s>;[]OP1 | C[s;[]OP1 ] | C[[]OP1;s] | C[s,[]OP1]

6. C{s} ::= C[s?[s]OP1] | C[s[s]OP1]

7. C ::= C::C | C:C

8. s ::= State | C[s] | [s]C |  | Empty

9. b ::= s | Boolean_Exp

10. Boolean_Exp ::= exp

11. exp ::= Relational_Exp | Identifier | function | Qualifier |
    [] | [????]

12. Relational_Exp ::= Simple_Exp Relational_Operator Simple_Exp

13. a , State, s, C, C, C ::= Identifier |
    Identifier / Qualifier | Identifier \\ Qualifier

14. State ::= Identifier.{ Digit } + | Qualifier

15. Qualifier ::= Identifier {  Identifier }* | Boolean_Exp |
    # NAT | Empty
```

```
16. OP ::= * | + | % | @ | & | # NAT | ? | ^ | = | || | Empty

17. OP1 ::= * | + | % | # NAT | ? | Empty

18. Tag ::= Number Status | Identifier Status | Status | Empty

19. Status ::= + | - | Empty

20. Alpha ::= A | B |  | Z | a | b |  | z

21. Digit ::= 0 | 1 | | 9

22. Identifier ::= Identifier & Identifier |
    Identifier | Identifier

23. Identifier ::= Alpha { Alpha | Digit | _ Alpha | -Alpha |
    _Digit | -Digit } *

24. NAT ::= { Digit } +

25. Number ::= NAT { . NAT }*

26. Empty ::=

27. C[a := State] =C[ State ]

28. C[a := Qualifier] =C[ Qualifier ]
```

A.3 XML Schema Specification for Behavior Trees

This XML file conforming to the XML Schema model defined at:

http://www.w3.org/2001/XMLSchema-instance

is a direct translation of the EBNF form of Behavior Trees specified in Section A.2.

The XML Schema file can be considered as the grammar, or database structure for defining how a Behavior Tree can be composed from its various elements. For any given Behavior Tree to be considered valid, it must conform to these rules, provided in the form of an XML Schema file.

The advantage of using XML Schema, is that it is an emerging, implementation-independent standard for describing complex structures. Unlike the previous Data-Type-Description (DTD) files used with XML, explicit specification of numeric, and string types can be made; limiting them to ranges or certain patterns.

An actual Behavior Tree (see the example in Section A.4) can then merely specify the data values required, and refer to this unchanging Schema file for the structure it is conforming to.

This Schema file is directly aligned to the proper specification of Behavior Trees and is used by various other research projects at this time, namely a Behavior Tree simulator, and diagramming tool that both read and write Behavior Trees as XML files. As such this Schema will be undergo revisions, to remove errors and conform to current best practice.

```
<?xml version="1.0"?> <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema">

   <xs:element name="tree">
      <xs:complexType>
         <xs:choice>
            <xs:sequence>
               <xs:element ref="node" />
               <xs:sequence minOccurs="0" maxOccurs="1">
                  <xs:element ref="semicolon" />
                  <xs:element minOccurs="1" maxOccurs="unbounded" ref="tree" />
               </xs:sequence>
            </xs:sequence>
            <xs:element ref="empty" />
         </xs:choice>
      </xs:complexType>
   </xs:element>

   <xs:element name="node">
      <xs:complexType>
         <xs:choice>
            <xs:sequence>
               <xs:element ref="cs" />
               <xs:sequence minOccurs="0" maxOccurs="1">
                  <xs:element ref="double_forward_slash" />
                  <xs:element ref="tag" />
               </xs:sequence>
            </xs:sequence>
            <xs:sequence>
               <xs:element ref="left_brace" />
               <xs:element ref="cs" />
               <xs:sequence minOccurs="0" maxOccurs="1">
                  <xs:element ref="double_forward_slash" />
                  <xs:element ref="tag" />
               </xs:sequence>
               <xs:element ref="right_brace" />
            </xs:sequence>
         </xs:choice>
      </xs:complexType>
   </xs:element>

   <xs:element name="cs">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="c" />
            <xs:choice>
               <xs:sequence>
                  <xs:element ref="op" />
                  <xs:element ref="left_brace" />
                  <xs:element ref="s" />
                  <xs:element ref="right_brace" />
                  <xs:element ref="op1" />
               </xs:sequence>
               <xs:sequence>
                  <xs:element ref="double_query" />
                  <xs:choice>
                     <xs:sequence>
                        <xs:element ref="b" />
                        <xs:element ref="double_query" />
                     </xs:sequence>
                     <xs:sequence>
                        <xs:element ref="smaller_than" />
                        <xs:element ref="s" />
                        <xs:element ref="greater_than" />
                        <xs:element ref="double_query" />
                        <xs:element ref="op1" />
                     </xs:sequence>
                     <xs:sequence>
                        <xs:element ref="greater_than" />
                        <xs:element ref="s" />
                        <xs:element ref="smaller_than" />
                        <xs:element ref="double_query" />
                        <xs:element ref="op1" />
                     </xs:sequence>
                  </xs:choice>
               </xs:sequence>
               <xs:sequence>
                  <xs:element ref="query" />
                  <xs:choice>
                     <xs:sequence>
                        <xs:element ref="not" />
                        <xs:element ref="colon" />
                        <xs:element ref="b" />
                     </xs:sequence>
                     <xs:element ref="else" />
                     <xs:element ref="error" />
                     <xs:element ref="b" />
                  </xs:choice>
                  <xs:element ref="query" />
               </xs:sequence>
               <xs:sequence>
                  <xs:element ref="smaller_than" />
                  <xs:choice>
                     <xs:sequence>
                        <xs:element ref="greater_than" />
                        <xs:element ref="s" />
                        <xs:element ref="smaller_than" />
                        <xs:element ref="greater_than" />
                        <xs:element ref="op1" />
                     </xs:sequence>
                     <xs:sequence>
                        <xs:element ref="s" />
                        <xs:element ref="greater_than" />
                        <xs:choice>
                           <xs:sequence>
                              <xs:element ref="semicolon" />
                              <xs:element ref="left_brace" />
                              <xs:element ref="sequence" />
                              <xs:element ref="right_brace" />
                              <xs:element ref="op1" />
                           </xs:sequence>
                           <xs:element ref="op1" />
                        </xs:choice>
                     </xs:sequence>
                  </xs:choice>
               </xs:sequence>
               <xs:sequence>
                  <xs:element ref="greater_than" />
                  <xs:element ref="s" />
                  <xs:element ref="smaller_than" />
                  <xs:element ref="op1" />
               </xs:sequence>
```

```
<xs:sequence>
   <xs:element ref="left_curly" />
   <xs:element ref="s" />
   <xs:element ref="right_curly" />
</xs:sequence>
<xs:sequence>
   <xs:element ref="left_brace" />
   <xs:choice>
      <xs:sequence>
         <xs:element ref="a" />
         <xs:element ref="assign" />
         <xs:element ref="exp" />
         <xs:element ref="right_brace" />
         <xs:element ref="op1" />
      </xs:sequence>
      <xs:sequence>
         <xs:element ref="left_brace" />
         <xs:choice>
            <xs:sequence>
               <xs:element ref="sequence" />
               <xs:element ref="right_brace" />
               <xs:element ref="op1" />
               <xs:element ref="semicolon" />
               <xs:element ref="s" />
            </xs:sequence>
            <xs:sequence>
               <xs:element ref="s_" />
               <xs:element ref="right_brace" />
               <xs:sequence minOccurs="0" maxOccurs="unbounded">
                  <xs:element ref="left_brace" />
                  <xs:element ref="s_" />
                  <xs:element ref="right_brace" />
               </xs:sequence>
            </xs:sequence>
         </xs:choice>
         <xs:element ref="right_brace" />
      </xs:sequence>
      <xs:sequence>
         <xs:element ref="s" />
         <xs:choice>
            <xs:element ref="semicolon" />
            <xs:element ref="comma" />
         </xs:choice>
         <xs:element ref="left_brace" />
         <xs:element ref="sequence" />
         <xs:element ref="right_brace" />
         <xs:element ref="op1" />
         <xs:element ref="right_brace" />
      </xs:sequence>
      <xs:sequence>
         <xs:element ref="s_" />
         <xs:choice>
            <xs:element ref="query" />
            <xs:element ref="cross" />
```

```
                     </xs:choice>
                     <xs:element ref="left_brace" />
                     <xs:element ref="s" />
                     <xs:element ref="right_brace" />
                     <xs:element ref="op1" />
                     <xs:element ref="right_brace" />
                  </xs:sequence>
               </xs:choice>
            </xs:sequence>
         </xs:choice>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="s">
   <xs:complexType>
      <xs:choice>
         <xs:element ref="state" />
         <xs:sequence>
            <xs:element ref="c_" />
            <xs:element ref="left_brace" />
            <xs:element ref="s_" />
            <xs:element ref="right_brace" />
         </xs:sequence>
         <xs:sequence>
            <xs:element ref="left_brace" />
            <xs:element ref="s_" />
            <xs:element ref="right_brace" />
            <xs:element ref="c_" />
         </xs:sequence>
         <xs:element ref="empty" />
      </xs:choice>
   </xs:complexType>
</xs:element>

<xs:element name="b">
   <xs:complexType>
      <xs:choice>
         <xs:element ref="s" />
         <xs:element ref="boolean_exp" />
      </xs:choice>
   </xs:complexType>
</xs:element>

<xs:element name="boolean_exp">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="exp" />
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="exp">
   <xs:complexType>
```

```
      <xs:choice>
         <xs:element ref="relational_exp" />
         <xs:element ref="identifier" />
         <xs:element ref="function" />
         <xs:element ref="qualifier" />
         <xs:sequence>
            <xs:element ref="left_brace" />
            <xs:choice>
               <xs:element ref="right_brace" />
               <xs:sequence>
                  <xs:element ref="cross" />
                  <xs:element ref="right_brace" />
               </xs:sequence>
            </xs:choice>
         </xs:sequence>
      </xs:choice>
   </xs:complexType>
</xs:element>

<xs:element name="relational_exp">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="simple_exp" />
         <xs:element ref="relational_operator" />
         <xs:element ref="simple_exp" />
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="a">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="identifier" />
         <xs:sequence minOccurs="0" maxOccurs="1">
            <xs:choice>
               <xs:element ref="forward_slash" />
               <xs:element ref="double_backslash" />
            </xs:choice>
            <xs:element ref="qualifier" />
         </xs:sequence>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="state">
   <xs:complexType>
      <xs:choice>
         <xs:element ref="qualifier" />
         <xs:sequence>
            <xs:element ref="identifier" />
            <xs:choice>
               <xs:sequence>
                  <xs:element ref="forward_slash" />
                  <xs:element ref="qualifier" />
               </xs:sequence>
               <xs:sequence>
                  <xs:element ref="double_backslash" />
                  <xs:element ref="qualifier" />
               </xs:sequence>
               <xs:sequence>
                  <xs:element ref="dot" />
                  <xs:element minOccurs="1" maxOccurs="unbounded" ref="digit" />
               </xs:sequence>
            </xs:choice>
         </xs:sequence>
      </xs:choice>
   </xs:complexType>
</xs:element>

<xs:element name="c">
   <xs:complexType>
      <xs:choice>
         <xs:sequence>
            <xs:element ref="c_" />
            <xs:choice>
               <xs:element ref="double_colon" />
               <xs:element ref="colon" />
            </xs:choice>
            <xs:element ref="c__" />
         </xs:sequence>
         <xs:sequence>
            <xs:element ref="identifier" />
            <xs:choice minOccurs="0" maxOccurs="1">
               <xs:element ref="forward_slash" />
               <xs:element ref="double_backslash" />
            </xs:choice>
            <xs:element ref="qualifier" />
         </xs:sequence>
      </xs:choice>
   </xs:complexType>
</xs:element>

<xs:element name="s_">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="identifier" />
         <xs:sequence minOccurs="0" maxOccurs="1">
            <xs:choice>
               <xs:element ref="forward_slash" />
               <xs:element ref="double_backslash" />
            </xs:choice>
            <xs:element ref="qualifier" />
         </xs:sequence>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="c_">
```

```
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="identifier" />
        <xs:sequence minOccurs="0" maxOccurs="1">
          <xs:choice>
            <xs:element ref="forward_slash" />
            <xs:element ref="double_backslash" />
          </xs:choice>
          <xs:element ref="qualifier" />
        </xs:sequence>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="c__">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="identifier" />
        <xs:sequence minOccurs="0" maxOccurs="1">
          <xs:choice>
            <xs:element ref="forward_slash" />
            <xs:element ref="double_backslash" />
          </xs:choice>
          <xs:element ref="qualifier" />
        </xs:sequence>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="qualifier">
    <xs:complexType>
      <xs:choice>
        <xs:sequence>
          <xs:element ref="identifier" />
          <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="space" />
            <xs:element ref="identifier" />
          </xs:sequence>
        </xs:sequence>
        <xs:element ref="boolean_exp" />
        <xs:sequence>
          <xs:element ref="hash" />
          <xs:element ref="nat" />
        </xs:sequence>
        <xs:element ref="empty" />
      </xs:choice>
    </xs:complexType>
  </xs:element>

  <xs:element name="op">
    <xs:complexType>
      <xs:choice>
        <xs:element ref="asterisk" />
        <xs:element ref="plus" />
```

```
         <xs:element ref="percent" />
         <xs:element ref="at" />
         <xs:element ref="ampersand" />
         <xs:sequence>
            <xs:element ref="hash" />
            <xs:element ref="nat" />
         </xs:sequence>
         <xs:element ref="query" />
         <xs:element ref="hat" />
         <xs:element ref="equals" />
         <xs:element ref="or" />
         <xs:element ref="empty" />
      </xs:choice>
   </xs:complexType>
</xs:element>

<xs:element name="op1">
   <xs:complexType>
      <xs:choice>
         <xs:element ref="asterisk" />
         <xs:element ref="plus" />
         <xs:element ref="percent" />
         <xs:sequence>
            <xs:element ref="hash" />
            <xs:element ref="nat" />
         </xs:sequence>
         <xs:element ref="query" />
         <xs:element ref="empty" />
      </xs:choice>
   </xs:complexType>
</xs:element>

<xs:element name="tag">
   <xs:complexType>
      <xs:choice>
         <xs:sequence>
            <xs:element ref="number" />
            <xs:element ref="status" />
            <xs:element ref="descriptor" />
         </xs:sequence>
         <xs:sequence>
            <xs:element ref="identifier" />
            <xs:element ref="status" />
            <xs:element ref="descriptor" />
         </xs:sequence>
         <xs:sequence>
            <xs:element ref="status" />
            <xs:element ref="descriptor" />
         </xs:sequence>
         <xs:element ref="empty" />
      </xs:choice>
   </xs:complexType>
</xs:element>
```

```
<xs:element name="status">
   <xs:complexType>
      <xs:choice>
         <xs:element ref="plus" />
         <xs:element ref="dash" />
         <xs:element ref="empty" />
      </xs:choice>
   </xs:complexType>
</xs:element>

<xs:element name="alpha">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="[a-zA-Z]{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>

<xs:element name="digit">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="[0-9]{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>

<xs:element name="identifier">
   <xs:complexType>
      <xs:choice>
         <xs:sequence>
            <xs:element ref="identifier" />
            <xs:choice>
               <xs:element ref="ampersand" />
               <xs:element ref="bar" />
            </xs:choice>
            <xs:element ref="identifier" />
         </xs:sequence>
         <xs:sequence>
            <xs:element ref="alpha" />
            <xs:choice minOccurs="0" maxOccurs="unbounded">
               <xs:element ref="alpha" />
               <xs:element ref="digit" />
               <xs:sequence>
                  <xs:element ref="underline" />
                  <xs:element ref="alpha" />
               </xs:sequence>
               <xs:sequence>
                  <xs:element ref="dash" />
                  <xs:element ref="alpha" />
               </xs:sequence>
               <xs:sequence>
                  <xs:element ref="underline" />
                  <xs:element ref="digit" />
               </xs:sequence>
               <xs:sequence>
                  <xs:element ref="dash" />
                  <xs:element ref="digit" />
               </xs:sequence>
            </xs:choice>
         </xs:sequence>
      </xs:choice>
   </xs:complexType>
</xs:element>

<xs:element name="nat">
   <xs:complexType>
      <xs:sequence>
         <xs:element minOccurs="1" maxOccurs="unbounded" ref="digit" />
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="number">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="nat" />
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="dot" />
            <xs:element ref="nat" />
         </xs:sequence>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="empty">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="\s{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>

<xs:element name="descriptor" type="xs:string" />
<xs:element name="relation" type="xs:string" />
<xs:element name="arithmetic_exp" type="xs:string" />
<xs:element name="simple_exp" type="xs:string" />
<xs:element name="relational_operator" type="xs:string" />
<xs:element name="function" type="xs:string" />

<xs:element name="left_brace">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="\[{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="right_brace">
   <xs:simpleType>
```

```
      <xs:restriction base="xs:string">
         <xs:pattern value="\]{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="left_curly">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="\{{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="right_curly">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="\}{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="query">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="\?{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="double_query">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="\?{2}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="hash">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="#{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="else">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="(else){1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="error">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="(error){1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
```

```
<xs:element name="smaller_than">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="&lt;{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="greater_than">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="&gt;{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="not">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="(NOT){1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="assign">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="(:=){1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="forward_slash">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="/{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="double_backslash">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="\\{2}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="double_forward_slash">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="/{2}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="qualify">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value=":{2}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="underline">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="_{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="dash">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="-{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="asterisk">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="\*{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="plus">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="\+{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="percent">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="%{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="bar">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="|{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="or">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="|{2}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="hat">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="^{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="equals">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="={1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="at">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="@{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="ampersand">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="&amp;{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="colon">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value=":{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="double_colon">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value=":{2}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="semicolon">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value=";{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="dot">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value=".{1}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
<xs:element name="cross">
```

```
      <xs:simpleType>
         <xs:restriction base="xs:string">
            <xs:pattern value="X{1}"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:element>
   <xs:element name="sequence">
      <xs:simpleType>
         <xs:restriction base="xs:string">
            <xs:pattern value=".{3}"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:element>
   <xs:element name="comma">
      <xs:simpleType>
         <xs:restriction base="xs:string">
            <xs:pattern value=",{1}"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:element>
   <xs:element name="space">
      <xs:simpleType>
         <xs:restriction base="xs:string">
            <xs:pattern value=" {1}"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:element>
</xs:schema>
```

A.4 Example XML Behavior Tree

Software components are generally considered as “black-boxes” for the purposes of its functionality, and only its interface specifications are known. Behavior Trees provide a way analysing the functionality of a component without needing to see the actual source or binary implementation code. So by using the XML version of Behavior Trees, there is the potential for third-party software to automatically analyse the internal behaviour and quality attributes of any conforming component.

This XML file represents a very simple Behavior Tree with only one component. It’s prime purpose is to demonstrate how to create a Behavior Tree in XML that conforms to the XML Schema specification for Behavior Trees given in Section A.3.

```
<?xml version="1.0"?> <tree
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="file:behaviour_tree.xsd">
   <node>
      <cs>
         <c>
            <identifier>
               <alpha>D</alpha>
               <alpha>O</alpha>
               <alpha>O</alpha>
               <alpha>R</alpha>
            </identifier>
         </c>
         <op><hat>^</hat></op>
         <left_brace>[</left_brace>
         <s>
            <state>
               <qualifier>
                  <identifier>
                     <alpha>o</alpha>
                     <alpha>p</alpha>
                     <alpha>e</alpha>
                     <alpha>n</alpha>
                     <alpha>e</alpha>
                     <alpha>d</alpha>
                  </identifier>
               </qualifier>
            </state>
         </s>
         <right_brace>]</right_brace>
         <op1><asterisk>*</asterisk></op1>
      </cs>
   </node>
</tree>
```

Appendix B

Quality Attribute Specification

An example of an attribute specification format table from Section 4.2.1 follows:

Maintainability:

scale = minutes to do simplest repair to software using templates

date (initial delivery to customers) = January next year

test (unit) = at least ten consecutive repair attempts to be done within worst case level for each module

test (system) = at least 50 random, representative system level bugs to be inserted and then repaired within planned level requirement

worst (by initial release date) = 10 minutes

plan (by initial release date) = 5 minutes

plan (by 3 years after initial release date, for online modules only) = 2

record (lab experiment TR23.3 1989) = 10 seconds average

now (old system, last year average) = 30 minutes

see (marketing strategy, Part 2.3.12) = input to this requirement

source (marketing requirement) = MRS Jan 28th 198x, JCP

This table will be encoded using the XML Schema specification in Section B.1. The primary purpose of this section is to demonstrate a standard way of formatting the textual data of requirements specifications. This XML data can then be included with the XML Behavior Tree specification and the binary code of the component, to provide detailed data for testing and certification of the component.

B.1 XML Schema for Quality Attribute Specification

This XML file conforms to the XML Schema model defined at:

http://www.w3.org/2001/XMLSchema-instance

```
<?xml version="1.0"?> <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema">

   <xs:element name="requirements">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="functional" />
            <xs:element ref="nonfunctional" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="functional">
      <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="number" />
            <xs:element ref="description" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="nonfunctional">
      <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="number" />
            <xs:element ref="name" />
            <xs:element ref="scale" />
            <xs:element ref="date" />
            <xs:element ref="test" />
            <xs:element ref="worst" />
            <xs:element ref="plan" />
            <xs:element ref="record" />
            <xs:element ref="now" />
            <xs:element ref="see" />
            <xs:element ref="source" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="name">
      <xs:simpleType>
         <xs:element ref="description" />
      </xs:simpleType>
   </xs:element>

   <xs:element name="scale">
      <xs:simpleType>
         <xs:element ref="description" />
      </xs:simpleType>
   </xs:element>

   <xs:element name="date">
      <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="type" />
            <xs:element ref="description" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="test">
      <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="type" />
            <xs:element ref="description" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="worst">
      <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="type" />
            <xs:element ref="description" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="plan">
      <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="type" />
            <xs:element ref="description" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="record">
      <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="type" />
            <xs:element ref="description" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="now">
      <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="type" />
            <xs:element ref="description" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="see">
      <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="type" />
            <xs:element ref="description" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="source">
      <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="type" />
            <xs:element ref="description" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="number">
      <xs:simpleType>
         <xs:element ref="description" />
      </xs:simpleType>
   </xs:element>

   <xs:element name="type">
      <xs:simpleType>
         <xs:element ref="description" />
      </xs:simpleType>
   </xs:element>

   <xs:element name="description">
      <xs:complexType>
         <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:restriction base="xs:string">
               <xs:pattern value="[a-zA-Z0-9 ,.]{1}"/>
            </xs:restriction>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
```

```
</xs:schema>
```

B.2 Example XML Quality Attribute Specification

This XML file represents a very simple quality attribute specification for only one attribute. It's prime purpose is to demonstrate how to create an overall requirements specification using XML that conforms to the Schema specification given in Section B.1.

```
<?xml version="1.0"?> <requirements
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="file:requirements.xsd">
   <functional>
   </functional>
   <nonfunctional>
      <number>
         <description>QR01</description>
      </number>
      <name>
         <description>Maintainability</description>
      </name>
      <scale>
         <description>minutes to do simplest repair to
          software using templates</description>
      </scale>
      <date>
         <type>
            <description>initial delivery to customers</description>
         </type>
         <description>January next year</description>
      </date>
      <test>
         <type>
            <description>unit</description>
         </type>
         <description>at least ten consecutive repair attempts to be
          done within worst case level for each module</description>
         <type>
            <description>system</description>
         </type>
         <description>at least 50 random, representative system level
          bugs to be inserted and then repaired within planned level
          requirement</description>
      </test>
      <worst>
         <type>
            <description>by initial release date</description>
         </type>
         <description>10 minutes</description>
      </worst>
      <plan>
         <type>
            <description>by initial release date</description>
         </type>
         <description>5 minutes</description>
         <type>
            <description>by 3 years after initial release date, for
             online modules only</description>
         </type>
         <description>2 minutes</description>
      </plan>
      <record>
         <type>
            <description>lab experiment TR23.3 1989</description>
         </type>
         <description>10 seconds average</description>
      </record>
      <now>
         <type>
            <description>old system, last year average</description>
         </type>
         <description>30 minutes</description>
      </now>
      <see>
         <type>
            <description>marketing strategy, Part 2.3.12</description>
         </type>
         <description>input to this requirement</description>
      </see>
      <source>
         <type>
            <description>marketing requirement</description>
         </type>
         <description>MRS Jan 28th 198x, JCP</description>
      </source>
   </nonfunctional>
</requirements>
```

Appendix C

Component Implementation Example using Java

This section shows an initial implementation of the Concurrent Stack, Queue, and Set Components as a generic Collection. It is primarily a prototype, to assist work design work produced in this dissertation. It uses the requirements and architecture descriptions developed from Sections 5.3.1, 5.3.2, 5.3.3, and 5.3.5

Sun's Java Software Development Kit version 1.4.0 was used for the implementation language. It offers development within an Object-Oriented paradigm, operating system portability, and full support for parallel threads and mutual exclusion. Ideally JavaBeans or Enterprise JavaBeans should have been used for their component development capabilities, but due to complexities and time limitations of this dissertation, they where considered out of scope. This should be avenue for future research, however.

The StackTest object is an example of how to use the concurrent stack system. This object was primarily used to test the behaviour and implementation of the system, as well as verify the requirements and architecture design.

Stack Object

```
/*
 * Stack 1.0 01/10/2002
 *
 * Copyright (c)2001 Tony Gillan. All rights reserved.
 */

import java.io.*;
import java.util.*;

/**
 * class Stack represents the basic resource addition and removal
 * processes used to simulate a stack.
 *
 * @version 1.0  1 October 2002
 * @author (c)2002 Tony Gillan.  tony@gridloq.com
 */
public class Stack implements Serializable {
   private static final int MAX_SIZE = 10;

   private LinkedList resources;
   private int resourcesTotal;

   public Stack() {
      this.resources = new LinkedList();
      this.resourcesTotal = 0;
   }

   public synchronized boolean addResource(Object resource) {
      boolean result = false;
      if (resource!=null && resources!=null && !isFull()) {
         resources.addLast(resource);
         resourcesTotal++;
         result = true;
         System.out.println("stack: resource added, " +
                           resourcesTotal + " total.");
      } else {
         System.out.println("stack: resource couldnt be added, " +
                           resourcesTotal + " total.");
      }
      return result;
   }

   public synchronized Object removeResource() {
      Object result = null;
      if (resources!=null && !isEmpty()) {
         result = resources.removeFirst();
         resourcesTotal--;
         System.out.println("stack: resource removed, " +
                           resourcesTotal + " left.");
      } else {
        System.out.println("stack: resource couldnt be removed, " +
                          resourcesTotal + " total.");
      }
      return result;
   }
   public synchronized boolean isFull() {
      return resourcesTotal>=MAX_SIZE;
   }

   public synchronized boolean isEmpty() {
      return resourcesTotal==0;
   }

   public synchronized void reset() {
      resources = new LinkedList();
      resourcesTotal = 0;
   }
}
```

ConcurrentStack Object

```
/*
 * ConcurrentStack 1.0 01/10/2002
 *
 * Copyright (c)2001 Tony Gillan. All rights reserved.
 */

import java.io.*;
import java.util.*;

/**
 * class ConcurrentStack represents a stack that can have concurrent additions
 * and removals made by external Consumer and Producer objects. Additionally
 * an Operator can initialise and shutdown the stack.
 *
 * @version 1.0  1 October 2002
 * @author (c)2002 Tony Gillan.  tony@gridloq.com
 */
public class ConcurrentStack extends Stack implements Runnable {
   private volatile Thread thread;
   private boolean running;

   private boolean available;
   private boolean availableForAddition;
   private boolean availableForRemoval;
   private boolean requestedClose;

   private Object resourceToAdd;
   private Object resourceRetrieved;
   private boolean requestResource;

   public ConcurrentStack() {
      super();
      this.thread = null;
      this.running = false;
      this.available = false;
      this.availableForAddition = false;
      this.availableForRemoval = false;
      this.requestedClose = true;
      this.resourceToAdd = null;
      this.resourceRetrieved = null;
      this.requestResource = false;
      start();
   }

   public void start() {
      if (thread==null) {
         thread = new Thread(this);
         running = true;
         thread.start();
      }
   }

   public void stop() {
      running = false;
      thread = null;
   }

   public void run() {
      Thread thisThread = Thread.currentThread();
      while (thread==thisThread && running) {
         try {
            thisThread.sleep((int)(Math.random() * 100));
         } catch (InterruptedException e){
         }
         if (available) {
            if (requestedClose) {
               if (availableForAddition) {
                  availableForAddition = false;
               }
               if (isEmpty()) {
                  if (availableForRemoval) {
                     availableForRemoval = false;
                     available = false;
                  }
               }
            }
            if (requestResource && resourceRetrieved==null) {
               resourceRetrieved = removeResource();
            }
            if (resourceToAdd!=null) {
               addResource(resourceToAdd);
               resourceToAdd = null;
            }
         }
      }
   }


   public synchronized boolean add(Object resource) {
      boolean result = false;
      if (availableForAddition && resourceToAdd==null) {
         resourceToAdd = resource;
         result = true;
      }
      return result;
   }

   public synchronized Object retrieve() {
      Object result = null;
      if (availableForRemoval) {
         requestResource = true;
         if (resourceRetrieved!=null) {
            requestResource = false;
            result = resourceRetrieved;
            resourceRetrieved = null;
         }
      }
      return result;
   }

   public synchronized void requestClose() {
      requestedClose = true;
   }

   public synchronized void requestInit() {
      if (!available) {
         super.reset();
         resourceToAdd = null;
         resourceRetrieved = null;
         requestedClose = false;
         requestResource = false;
         availableForRemoval = true;
         availableForAddition = true;
         available = true;
      }
   }
}
```

Producer Object

```
/*
 * Producer 1.0 01/10/2002
 *
 * Copyright (c)2001 Tony Gillan. All rights reserved.
 */

import java.io.*;
import java.util.*;

/**
 * class Producer represents a producer of resources to be placed on the
 * ConcurrentStack object.
 *
 * @version 1.0  1 October 2002
 * @author (c)2002 Tony Gillan.  tony@gridloq.com
 */
public class Producer implements Runnable {
   private volatile Thread thread;
   private boolean running;

   private ConcurrentStack stack;
   private static int resourceID = 0;

   public Producer(ConcurrentStack stack) {
      this.stack = stack;
      start();
   }

   public void start() {
      if (thread==null) {
         thread = new Thread(this);
         running = true;
         thread.start();
      }
   }

   public void stop() {
      running = false;
      thread = null;
   }

   public void run() {
      Thread thisThread = Thread.currentThread();
      String resource = "" + (resourceID++);
      while (thread==thisThread && running) {
         try {
            thisThread.sleep((int)(Math.random() * 100));
         } catch (InterruptedException e){
         }
         if (stack.add(resource)) {
            System.out.println("producer: added resource: " + resource);
            resource = "" + (resourceID++);
         } else {
            System.out.println("producer: couldnt add resource: " + resource);
         }
      }
   }
}
```

Consumer Object

```
/*
 * Consumer 1.0 01/10/2002
 *
 * Copyright (c)2001 Tony Gillan. All rights reserved.
 */
```

```
import java.io.*;
import java.util.*;

/**
 * class Consumer represents a consumer of resources provided by the
 * ConcurrentStack object.
 *
 * @version 1.0  1 October 2002
 * @author (c)2002 Tony Gillan.  tony@gridloq.com
 */
public class Consumer implements Runnable {
   private volatile Thread thread;
   private boolean running;

   private ConcurrentStack stack;
   private static int resourceID = 0;

   public Consumer(ConcurrentStack stack) {
      this.stack = stack;
      start();
   }

   public void start() {
      if (thread==null) {
         thread = new Thread(this);
         running = true;
         thread.start();
      }
   }

   public void stop() {
      running = false;
      thread = null;
   }

   public void run() {
      Thread thisThread = Thread.currentThread();
      Object resource = null;
      while (thread==thisThread && running) {
         try {
            thisThread.sleep((int)(Math.random() * 100));
         } catch (InterruptedException e){
         }
         resource = stack.retrieve();
         if (resource!=null && resource instanceof String) {
            System.out.println("consumer: retrieved resource: " + ((String)resource));
         } else {
            System.out.println("consumer: couldnt retrieve resource");
         }
      }
   }
}
```

StackTest Object

```
/*
 * StackTest 1.0 01/10/2002
 *
 * Copyright (c)2001 Tony Gillan. All rights reserved.
 */

import java.io.*; import java.util.*;

/**
 * class StackTest represents a simple test method for the ConcurrentStack
 * object and its dependent concurrent Producer and Consumers objects.
 *
 * @version 1.0  1 October 2002
 * @author (c)2002 Tony Gillan.  tony@gridloq.com
 */
public class StackTest {
   public static void main(String[] args) throws IOException {
      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
      ConcurrentStack stack = new ConcurrentStack();
      Consumer consumer = new Consumer(stack);
      Producer producer = new Producer(stack);
      boolean running = false;
      String command = in.readLine();
      while (true) {
         if (running) {
            System.out.println("operator has requested system close.");
            stack.requestClose();
            running = false;
         } else {
            System.out.println("operator has requested system initialise.");
            stack.requestInit();
            running = true;
         }
         command = in.readLine();
      }
   }
}
```

Trace Log of StackTest Execution

```
producer: couldnt add resource: 0
producer: couldnt add resource: 0
producer: couldnt add resource: 0
consumer: couldnt retrieve resource
producer: couldnt add resource: 0
consumer: couldnt retrieve resource
producer: couldnt add resource: 0
consumer: couldnt retrieve resource
producer: couldnt add resource: 0

operator has requested system initialise.
consumer: couldnt retrieve resource
```

```
stack: resource couldnt be removed, 0 total.
producer: added resource: 0
consumer: couldnt retrieve resource
stack: resource couldnt be removed, 0 total.
stack: resource added, 1 total.
producer: added resource: 1
stack: resource removed, 0 left.
stack: resource added, 1 total.
producer: added resource: 2
consumer: retrieved resource: 0
stack: resource added, 2 total.
consumer: couldnt retrieve resource
producer: added resource: 3
producer: couldnt add resource: 4
stack: resource removed, 1 left.
stack: resource added, 2 total.
consumer: retrieved resource: 1
producer: added resource: 4
stack: resource added, 3 total.
consumer: couldnt retrieve resource
producer: added resource: 5
stack: resource removed, 2 left.
stack: resource added, 3 total.
consumer: retrieved resource: 2
producer: added resource: 6
producer: couldnt add resource: 7
stack: resource added, 4 total.
consumer: couldnt retrieve resource
stack: resource removed, 3 left.
consumer: retrieved resource: 3
consumer: couldnt retrieve resource
consumer: couldnt retrieve resource
producer: added resource: 7
producer: couldnt add resource: 8
stack: resource removed, 2 left.
stack: resource added, 3 total.
consumer: retrieved resource: 4
consumer: couldnt retrieve resource
stack: resource removed, 2 left.
producer: added resource: 8
producer: couldnt add resource: 9
consumer: retrieved resource: 5
stack: resource added, 3 total.
producer: added resource: 9
stack: resource added, 4 total.
producer: added resource: 10
consumer: couldnt retrieve resource
producer: couldnt add resource: 11
stack: resource removed, 3 left.
consumer: retrieved resource: 6
stack: resource added, 4 total.
consumer: couldnt retrieve resource
producer: added resource: 11
stack: resource removed, 3 left.
```

```
stack: resource added, 4 total.
consumer: retrieved resource: 7
producer: added resource: 12
producer: couldnt add resource: 13
stack: resource added, 5 total.

operator has requested system close.
consumer: couldnt retrieve resource
producer: added resource: 13
consumer: couldnt retrieve resource
producer: couldnt add resource: 14
stack: resource removed, 4 left.
stack: resource added, 5 total.
producer: couldnt add resource: 14
consumer: retrieved resource: 8
consumer: couldnt retrieve resource
consumer: couldnt retrieve resource
producer: couldnt add resource: 14
stack: resource removed, 4 left.
consumer: retrieved resource: 9
producer: couldnt add resource: 14
consumer: couldnt retrieve resource
consumer: couldnt retrieve resource
consumer: couldnt retrieve resource
stack: resource removed, 3 left.
consumer: retrieved resource: 10
producer: couldnt add resource: 14
consumer: couldnt retrieve resource
stack: resource removed, 2 left.
consumer: retrieved resource: 11
producer: couldnt add resource: 14
producer: couldnt add resource: 14
consumer: couldnt retrieve resource
producer: couldnt add resource: 14
stack: resource removed, 1 left.
consumer: retrieved resource: 12
producer: couldnt add resource: 14
consumer: couldnt retrieve resource
stack: resource removed, 0 left.
consumer: retrieved resource: 13
producer: couldnt add resource: 14
consumer: couldnt retrieve resource
producer: couldnt add resource: 14
producer: couldnt add resource: 14

operator has requested system initialise.
consumer: couldnt retrieve resource
stack: resource couldnt be removed, 0 total.
stack: resource couldnt be removed, 0 total.
consumer: couldnt retrieve resource
producer: added resource: 14
consumer: couldnt retrieve resource
stack: resource couldnt be removed, 0 total.
stack: resource added, 1 total.
```

```
consumer: couldnt retrieve resource
stack: resource removed, 0 left.
producer: added resource: 15
producer: couldnt add resource: 16
consumer: retrieved resource: 14
stack: resource added, 1 total.
consumer: couldnt retrieve resource
consumer: couldnt retrieve resource
producer: added resource: 16
stack: resource removed, 0 left.
stack: resource added, 1 total.
consumer: retrieved resource: 15
producer: added resource: 17
consumer: couldnt retrieve resource
producer: couldnt add resource: 18
stack: resource removed, 0 left.
stack: resource added, 1 total.
consumer: retrieved resource: 16
producer: added resource: 18
stack: resource added, 2 total.
producer: added resource: 19
producer: couldnt add resource: 20
consumer: couldnt retrieve resource
stack: resource removed, 1 left.
stack: resource added, 2 total.
consumer: retrieved resource: 17
consumer: couldnt retrieve resource
producer: added resource: 20
stack: resource removed, 1 left.
stack: resource added, 2 total.
consumer: retrieved resource: 18
consumer: couldnt retrieve resource
producer: added resource: 21
stack: resource removed, 1 left.
stack: resource added, 2 total.

operator has requested system close.
```