UNIVERSITY OF OSLO
Department of informatics

A Case Study on Evaluating State-Based UML Modeling in Software Testing

Master thesis
60 credits

Omar M. Jama

February 2009

Abstract

Model-based software development with UML is becoming a de facto standard in industry. But there is need for more study of actual use of UML in software development companies, especially in ensuring the relevance of research on model-driven development with respect to testability and reliability when using UML.

Software testing is an empirical investigation that is conducted to provide developers and other stakeholders with information about the system under test. The primary purpose of software testing is to detect software failures so that defects may be uncovered and corrected.

Mutation analysis is a fault-based testing technique that uses mutation operators to introduce small changes into a program or specification, producing mutants, and then chooses test cases to distinguish the mutants from the original program. Mutation operators differ in the coverage they get. They also differ in the number of mutants they generate. As a consequence, selecting mutation operators is a major problem whose solution affects the effectiveness and cost of mutation analysis.

This thesis explores software testing as its main theme, especially model-based testing. In this attempt, we present a case study of a major multi-national company with the core business area of development of mechatronics[1] systems with embedded software. This case study is based on experiences of a previous case study conducted at the company which this thesis focuses. I also show in this thesis how mutation analysis can be applied to a test program. .

[1] Mechatronics is (Mechanical and Electronic Engineering) is the combination of mechanical engineering, electronic engineering and computer engineering [10]

Acknowledgement

First of all, I want to express my deepest gratitude to my supervisor Bente Anda for constructive feedback and guidance of great value during the writing process of this thesis.

I am also grateful for Nina Holt for the support offered to me all the way. I am also grateful to the students and employees at the Simula Research Laboratory.

I also give my warmest thanks to my family for their love and support. I thank my beloved wife Adar, my two daughters Hibo and Haweyo, and my son Hanad. I owe this to you all. Thanks for bearing with med during the time of this thesis.

Oslo, February 2009

Table of Contents

1. Introduction

Model-driven software development (MDE), or model-driven engineering, is a software development methodology that focuses on the creation of models that are close to a specific problem area, or domain. It is also meant to increase productivity by maximizing compatibility between systems, simplifying design process, and promoting communication between individuals and teams working on the system. A modelling paradigm for MDE is also considered effective if its models make sense from the point of the user and can serve as a basis of implementing systems. The best known MDE initiative is the Model-Driven Architecture, proposed by the Object Management Group (OMG).

Software development engineering refers to a range of development approaches that are based on the software modelling as the primary form of expression. These models are sometimes constructed to a certain level of detail, and then code is written by hand in a separate step. Sometimes complete models are built including executable actions. Code can be generated from the models, ranging from system skeletons to complete, deployable products. With the introduction of the Unified Modelling Language (UML), discussed in Chapter 2, MDE has become very popular today with a wide body of practitioners and supporting tools.

Model-based testing is a software testing in which test cases are derived in whole or in part from a model that describes some functional aspects of the system under test (SUT) [24]. The model is usually an abstract, partial presentation of the system under test's desired behaviour. The test cases derived from models are functional tests on the same level of abstraction as the model. Theses tests are collectively known as the abstract test suite. The abstract test suite cannot be directly executed against the system under test. This is done by mapping the abstract test cases to concrete test cases suitable for execution. Because test suites are derived from models and not from source code, model-based testing is usually seen as one form of black-box testing.

In OMG's MDA, the model is built before or parallel to the development process of the system under test. The model can also be constructed from completed system. Traditionally the model is created mostly manually, but there are also attempts to create the model automatically, for instance out of source code. One important way to create new models is by model transformation, using languages like ATL, a QVT-like Domain Specific Language (QVT, Query View Transformation – a standard for model transformation defined by Object Management Group).

Software testing is an empirical investigation which is conducted to provide information about the software under test. Testing is performed for evaluating product quality, and for improving it, by identifying defects and problems. Software testing consists of the *dynamic* verification of the behaviour of a program on a *finite* set of cases, suitably *selected* from the usually infinite executions domain, against the *expected* behaviour [26].

Model-based testing is software testing in which test cases are derived in whole or in part from a model that describes some (usually functional) aspects of the system under test. Most model-based testing inherits the complexity of the domain, or more particularly of the related domain models [24]. The focus of this thesis is on the evaluation of two software testing methods: a model-based testing method and another testing method which is not model-based; it is also focused on the requirements related to these two testing methods. The requirements relating to both methods of testing are in the domain of embedded real-time systems.

1.1 Motivation

Model-Driven Software Development (MDD) is a vision of software development where models play an important role as the primary artifacts. It is a new trend in utilizing models in software development. The models are used as primary artifacts in constructing software. This often means that the code is generated from models

The context of this thesis is ABB, which is a global company operating in approximately 100 countries with 100,000 employees. The primary business of ABB is power and automation and it develops software and hardware for each area of its business concern.

Model-based testing has been used for a wide variety of test generation techniques. There are four main approaches which are known as model-based testing. These include generation of test input data from a domain model, generation of test cases from en environment model, generation of test cases with oracles from a behaviour model and generation of test scripts from abstract tests. There are a number of benefits of model-based testing, which include fault detection, reduced testing cost and time gained from test automation [16].

Research into model-based testing, and generally into model-based development can benefit software development at ABB.

1.2 Research Objectives

The previous section provided a brief introduction of the research field of this thesis, which is model-based software development with regard to model-based testing. This thesis is constricting this field in the context of the Application Logic Controller (ALC) system at ABB. This system is designed and developed at ABB with the main function of monitoring and supervising ABB's industrial robots. Reliability of the system and, hence an effective testing, was specially an important issue for the company. Therefore, the main objective of this thesis is how to evaluate model-based development with regard to software testing.

This is further elaborated in a main research objective and a research question.

Research objective: *Explore how much does development of code cost with and without the models.*

Through this research I want to learn about model-driven software development in general, and especially generating tests cases from models. Another research objective is learning about mutant analysis method and its fault-finding power of test suites.

Research question: *How to evaluate model-based testing in the context of embedded and real-time systems.*

1.3 Structure of this Thesis

This thesis is structured as follows: Literature and background is presented in Chapter 2. Software Testing is presemted in Chapter 3. A short introduction to ABB is presented in Chapter 4. The research methods of the thesis are presented in Chapter 5. Evaluation of baseline and model-based testing are presented in Chapter 6. A discussion of the two testing methods is presented in Chapter 7, while in Chapter 8 conclusion is drawn.

2. UML and Model-Driven Development

2.1 What is UML?

The Unified Modelling Language (UML) is a general-purpose visual modelling language for systems. Although UML is most often associated with modelling Object Oriented Software systems, it has a much wider application than this due to its inbuilt extensibility mechanisms [1].

UML was designed to incorporate current best practice in modelling technologies and software engineering. It is important to realize that UML does not give us any kind of modelling methodology. The Unified Process (UP) is a methodology that tells us the workers, activities, and artifacts that we need to utilize, perform, or create in order model a software system [1].

UML is not tied to any specific methodology or life cycle, and indeed it is capable of being used with existing methodologies. UP uses UML as its underlying visual modelling syntax and you can therefore think of UP as being the *preferred* method for UML, as it is the best adapted to it, but UML itself can (and does) provide the visual modelling support for other methods [1].

2.2 History of UML

Prior to 1994, the Object Oriented methods world was a bit of a mess. There were several competing visual modelling languages and methodologies all with their strengths and weaknesses and all with their pros and cons. In terms of visual modelling languages, the clear leaders were Booch (the Booch Method) and Rumbaugh (Object Modelling Technique or OMT), who between them had over half the market. On the methodologies side, Jacobson had by the far the strongest case as although many authors claimed to have a "method", all that many of them actually had was a visual modelling syntax and a collection of more or less useful guidelines [1].

In 1996, the Object Management Group (OMG) produced a request-for-proposal (RFP) fir an OO visual modelling language, and UML was submitted. In 1997, OMG accepted the UML and the first open, industry-standard OO modelling language was born. Since then, all of the competing methods have faded away and UML stands unchallenged as the industry standard OO modelling language.

In 2000, UML 1.4 introduced a significant extension to UML by the addition of action semantics. These describe the behaviour of a set of primitive actions that may implemented by specific action languages. Action semantics plus an action language allow the detailed specification of the behavioural elements of UML models (such as class operations) *directly* in the UML model. This was a very significant development as it made the UML specification computationally complete and thus it became possible to make UML models executable.

In 2005, UML 2.0 specification was finalized. UML is now a very mature modelling language. It had proved its value in thousands of software development projects worldwide. UML 2.0 introduced quite a lot of new visual syntax. Some of this replaces and clarifies existing 1.x syntax, and some of it is completely new and represents new semantics which were added to the language. UML has always provided many options about how a particular model element may be displayed, and not all of those will be supported by every modelling language.

2.3 An Overview of UML

UML has a broad spectrum of usage. It can be used for business modelling, software modelling in all phases of development and for all types of systems, and general modelling of any construction that has both a static structure and dynamic behaviour. In order to achieve these wide-ranging capabilities, the language is defined to be extensive and generic enough to allow for the modelling such diverse systems, avoiding the too specialized and too complex [2].

The overview of UML has the following different parts:

- *Views:* Views show different aspects of the system that modelled. A view is not a graph, but an abstraction consisting of a number of diagrams. Only by defining a number of views, each showing a particular aspect of the system, can a complete picture of the system be constructed. The views also link the modelling language to the method/process chosen for development.

- *Diagrams:* Diagrams are the graphs that describe the contents in a view. UML has nine different diagram types that are used in combination to provide all views of the system.

- *Model elements:* The concepts used in the diagrams are model elements that represent common object-oriented concepts such as classes, objects, and messages, and the relationships among these concepts including association, dependency, and generalization. A model element is used in several different diagrams, but it always has the same meaning and symbol.

- *General mechanisms:* General mechanisms provide extra comments, information, or semantics about a model element; they also provide extension mechanisms to adapt or extend the UML to a specific method/process, organization, or user [2].

2.3.1 Views

Modelling a complex system is an extensive task. Ideally, the entire system is described in a single graph that defines the entire system unambiguously, and is easy to communicate and understand. However, this is usually impossible. A single graph cannot capture all the information needed to describe a system. A system is described with a number of aspects: functional (its static structure and dynamic interactions), non-functional (timing requirements, reliability, deployment, etc), and organizational aspects (work organization, mapping to code modules, etc). Thus a system is described in a number of views, where each view represents a projection of the complete system description, showing a particular aspect of the system.

The different views are:

- *Use-case view:* A view showing functionality of the system as perceived by external actors.

- *Logical view:* A view showing how the functionality is designed inside the system, in terms of the system's static structure and dynamic behaviour.
- *Component view:* A view showing the organization of the code components.
- *Concurrency view:* A view showing concurrency in the system, addressing the problems with communication and synchronization that are present in a concurrent system.
- *Deployment view:* A view showing the deployment of the system into the physical architecture with computers and devices called *nodes.*

Use-Case View

The use-case view describes the functionality the system should deliver, as perceived by external actors. An actor interacts with the system; it can be a user or another system. The user-case view is for customers, designers, developers, and testers; it is described in user-case diagrams and occasionally in activity diagrams. The desired usage of the system is described as a number of use cases in the use-case view, where a use case is a general description of a usage if the system (a function requested) [2].

The use-case view is central, since its contents drive the development if the other views. The final goal of the system is to provide the functionality described in this view – along with some non-functional properties – hence this view affects all the others.

Logical View

The logical view describes how the system functionality is provided. It is mainly for designers and developers. In contrast to use-case view, the logical view looks inside the system. It describes both the static (classes, objects, and relationships) and dynamic collaborations that occur when the objects send messages to each other to provide a given function. Properties such as persistency and concurrency are also defined, as well as the interfaces and the internal structures of classes.

The static structure is described in class and object diagrams. The dynamic modelling is described in state, sequence, collaboration, and activity diagrams.

Component View

The component view is a description of the implementation modules and their dependencies. It is mainly for developers, and consists of the component diagram. The components, which are different types of code modules, are shown with their structure and dependencies. Additional information about the components, such as resource allocation (responsibility for a component), or other administrative information, such as progress report for the development work, can also be added.

Concurrency View

The concurrency view deals with the division of the system into processes and processors. This aspect, which is a non-functional property of the system, allows for efficient resource usage, parallel execution, and the handling of asynchronous events from the environment. Besides dividing the system into concurrently executing threads of control, the view must also deal with communication and synchronization of these threads.

The concurrency view is for developers and integrators of the system, and it consists of dynamic diagrams (state, sequence, collaboration, and activity diagrams) and implementation diagrams (component and deployment diagrams).

Deployment View

The deployment view shows the physical deployment of the system, such as the computers and devices (nodes) and how they connect to each other. The deployment view is for developers, integrators, and is represented by the deployment diagram. This view also includes a mapping that shows how the components are deployed in the physical architecture; for example, which program or objects execute on each respective computer.

2.3.2 Diagrams

The diagrams are the actual graphs that show model element symbols arranged to illustrate a particular part or aspect of the system. A system model typically has several diagrams of each type. A diagram is a part of specific view; and when it is drawn, it is usually allocated to a view. Some diagram types can be part of several views, depending on the contents of the diagram [2].

Use-Case Diagram

A use-case diagram show a number of external actors and their connection to the use cases that system provides (Figure 2). A use case is a description of a functionality (a specific usage of the system) that the system provides. The description of the actual use case is normally done in plain text as a documentation property of the use-case symbol, but it can also be described using an activity diagram. The use cases are described only as viewed externally by the actor (the system behaviour as the user perceives it), and do not describe how the functionality is provided inside the system. Use cases define the functionality requirements of the system [2].

Class Diagram

A class diagram shows the static structure of classes in the system (Figure 3). The classes represent the “things” that are handled in the system. Classes can be related to each other in a number of ways: associated (connected to each other), dependent (one class depends/uses another class), specialized (one class is a specialization of another class), or packaged (grouped together as a unit). All these relationships are in shown in a class diagram along with the internal structure of the classes in terms of attributes and operations. The diagram is considered static in that the structure described is always valid at any point in the system’s life cycle [2].

A system typically has a number of class diagrams – not all classes are inserted into a single class diagram – and a class can participate in several class diagrams.

Object Diagram

An object diagram is a variant of a class diagram and uses almost identical notation. The difference between the two is that object diagram shows a number of object instances of classes, instead of actual classes. An object diagram is thus an example of a class diagram that shows a possible snapshot of the system's execution – what the system could look like at some point in time. The same notation as for class diagram is used, with two exceptions: objects are written with their names underlined and all instances in a relationship.

Object diagrams are not as important as class diagrams, but they can be used to exemplify a complex class diagram by showing what the actual instances and relationships could look like. Object diagrams are also used as part of collaboration diagrams, in which the dynamic collaboration between a set of objects is shown.

State Diagram

A state diagram is typically a complement to the description of a class. It shows all the possible states that objects of the class can have, and which events cause the state to change (Figure 1). An event can be another object that sends a message to it – for example, that a specified time has elapsed – or that some condition has been fulfilled. A change of state is called a *transition.* A transition can also have an action connected to it that specifies what should be done in connection with the state transition.

State diagrams are not drawn for all classes, only for those that have a number of well-defined states and where the behaviour of the class is affected and changed by the different states. State diagrams can also be drawn for the system as a whole.

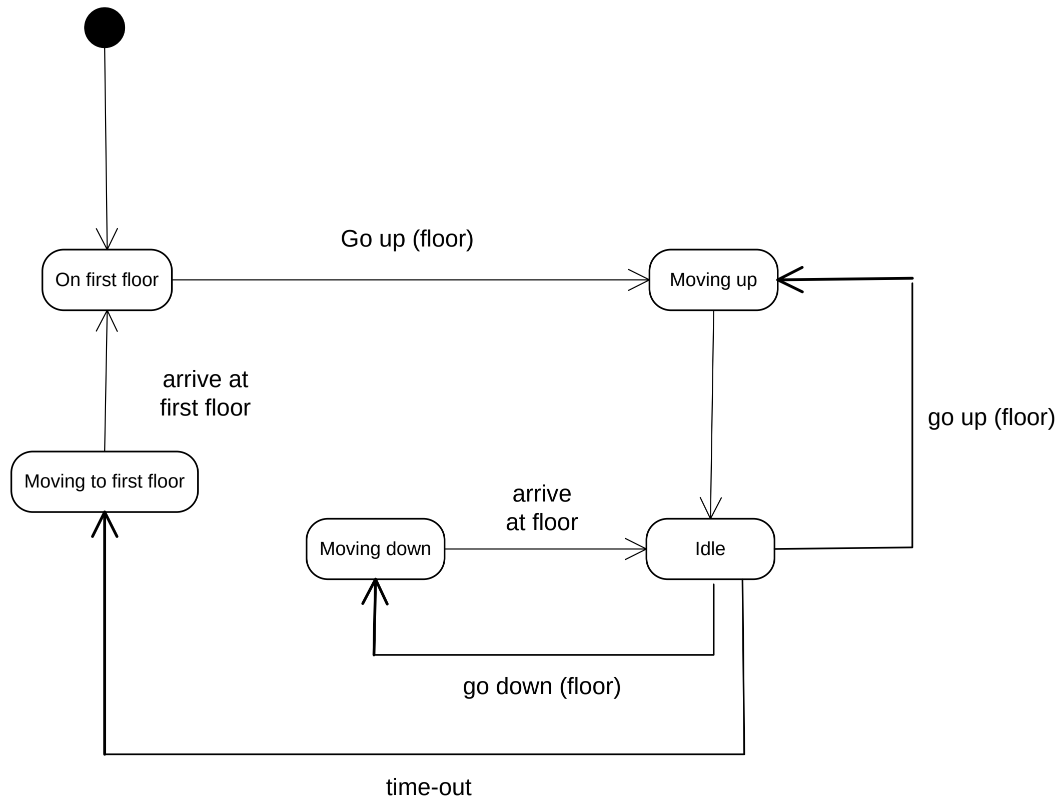

Figure 1 A state diagram of an elevator

Sequence Diagrams

A sequence diagram shows a dynamic collaboration between a number of objects. The important aspect of this diagram is to show a sequence of messages sent between the objects. It also shows an interaction between objects, something that will happen at one specific point in the execution of the system. The diagram consists of a number objects shown with vertical lines. Time passes down in the diagram, and the diagram shows the exchange of messages between the objects as time passes in the sequence or function. Messages are shown as lines with message arrows between the vertical object lines. Time specifications and other comments are added in a script in the margin of the diagram

Collaboration Diagram

A collaboration diagram shows a dynamic collaboration, just like the sequence diagram. It is often a choice between showing collaboration as either a sequence diagram or as a collaboration diagram. In addition to showing the exchange of messages (called the *interaction*), the collaboration diagram shows the objects and their relationships (sometimes referred to as the *context*). Whether you use a sequence diagram or a collaboration diagram can often be decided by: If time or sequence is the most important aspect to emphasize, choose sequence diagram; if the context is important to emphasize, choose collaboration diagram. The interaction between the objects is shown in both diagrams.

The collaboration diagram is drawn as an object diagram, where a number of objects are shown along with their relationships (using the notation in the class/object diagram). Message arrows are drawn between the objects to show the flow of messages between the objects. Labels are placed on the messages, which among other things, show the order in which the messages are sent. It can also show collaborations, iterations, return values, and so on. When familiar with the message label syntax, a developer can read the collaboration and follow the execution flow and exchange of messages. A collaboration diagram can also contain active objects, those that execute concurrently with other active objects.

Activity Diagram

An activity diagram shows a sequential flow of activities. The activity diagram is typically used to describe the activities performed in an operation, though it can also be used to describe other activity flows, such as a use case or an interaction. The activity diagram consists of action states, which contain a specification of an activity to be performed (an action). An action state will leave the state when the action has been performed (a state in a state diagram needs an explicit event before it leaves the state). Thus the control flows among the action states, which are connected to each other. Decisions and conditions, as well as parallel execution of action states, can also be shown in the diagram. The diagram can also contain specifications of messages being sent or received as part of the actions performed.

Component Diagram

A component diagram shows the physical structure of the code in terms of code components. A component can be a source code component, binary component, or an executable component. A component contains information about the logical class or classes it implements, thus creating a mapping from the logical view to the component view. Dependencies between the components are shown, making it easy to analyze how other components are affected by a change in one component. Components can also be shown with any of the interfaces that they expose, such as OLE/COM interfaces; and they can be grouped together in packages. The component diagram is used in practical programming work.

Deployment Diagram

The deployment diagram shows the physical architecture of the hardware and software in the system. You can show the actual computers and device (nodes), along with the connections they have to each other; you can also show the type of connections. Inside the nodes, executable components and objects are allocated to show which software units are executed on which nodes. You can also show dependencies between the components.

The deployment diagram describes the actual architecture of the system. This is far from the functional description of the use-case view. However, with a well-defined model, it is possible to navigate all the way from a node in the physical architecture to its components to the class it implements that objects of the class participate in and finally to a use case. Different views of the system are used to give a coherent description of the system as a whole.

2.3.3 Model Elements

The concepts used in the diagrams are called model elements. A model element is defined with semantics, a formal definition of the elements or the exact meaning of what it represents in unambiguous statements. A model element also has a corresponding view element, which is the graphical representation of the element or the graphical symbol used to represent the element in diagrams. An element can exist in several different types of diagram, but there are rules for which elements can be shown in each type of diagram. Some examples of model elements are class, object, state, node, package, and component.

Some different relationships are:

- *Association:* Connects elements and links instances
- *Generalization:* Also called inheritance, it means that en element can be a specialization of another element.
- *Dependency:* Shows that one element depends in some way on another element.
- *Aggregation:* A form of association in which an element contains other element.

Other model elements besides those described include messages, actions, and stereotypes.

2.3.4 General Mechanisms

UML utilizes some general mechanisms in all diagrams, for additional information in a diagram, typically that which cannot represented using the basic abilities of model elements.

Adornments

Graphical adornment can be attached to the model elements in diagrams. The adornments add semantics to the element. An adornment is, for example, a technique used to separate a type, and its name is displayed in boldface type. When the same element represents an instance of the type, its name is underlined and may specify both the name of the instance as well as the name of the type. A class rectangle, with the name in bold representing a class and the name underlined representing an object, is an example of this. The same goes for nodes, where the node symbol can be either a type in boldface, such as **Printer**, or an instance of a node type, such as <u>John’s HP 5MP-printer</u>. Other adornments are specifications of multiplicity of relationships, where the multiplicity is a number or a range that indicates how many instances of connected types can be involved in the relation. Adornments are written close to the elements to which they add information. All the adornments are described in conjunction with the description of the element that they affect.

Notes

Not everything can be defined in a modelling language, no matter how extensive the language. To enable adding information a model that otherwise cannot be represented, UML provides a notes capability. A note can be placed anywhere in any diagram, and it can contain any type of information. Its information type is a string that is not interrupted by the UML.

The note is typically attached to some element in the diagram with a dashed line that specifies which element is explained or detailed, along with the information in the note.

A note often contains comments or questions from the modeller as a reminder to resolve a dilemma at a later time. Notes can also have stereotypes that describe the note type.

2.4 Use of UML

The UML is used to model systems, the range of which is very broad; many different types of systems can be described with UML. UML can also be used in the different phases in the development of a system, from the requirements specification to the test of a finished system [2].

The goal if UML is to describe any type of system, in terms of object-oriented diagrams. Naturally, the most common use is to create models of software systems, but UML is also used to describe mechanical systems without any software, or the organization of a business. [2].

The different types of systems which can be used to model with UML include:

- *Information systems:* Store, retrieve, transform, and present information to users. Handle large amounts of data with complex relationships, which are stored in relational or object databases.
- *Technical systems:* Handle and control technical equipments such as telecommunications, military systems, or industrial processes. They must handle the special interfaces of the equipment and have less standard software than information systems. Technical systems are often real-time systems.
- *Embedded real-time systems:* Execute on simple hardware embedded in some other equipment such as mobile phone, car, household appliance, etc. This is accomplished through low-level programming that requires real-time support. These systems often lack devices such as display, hard disk, etc.
- *Distributed systems:* Distributed on a number of machines where data is transferred easily from one machine to another. They require synchronized communication mechanisms t ensure data integrity and are often built upon object mechanisms such as CORBA, COM/DCOM, or Java Beans/RMI.
- *System software:* Defines the technical infrastructure that other software uses. Operating systems, databases, and user interfaces perform low-level operations on the hardware, while presenting generic interfaces for other software to use.
- *Business systems:* Describe the goals, the resources (human, computers etc), the rules (laws, business strategies, policies etc), and the actual work in the business (business processes) (B. Nilsson, 1991) [2].

It is important to emphasize that most systems don't fit neatly into one of these categories, but belong to more than one of the system types or as a combination. For example, many of today's information systems have both distributed and real-time requirements. UML has the capability to model all of these system types [2].

2.5 Phases of System Development

There are five phases of system development: requirement analysis, analysis, design, programming, and testing [2].

Requirement Analysis

UML has use cases to capture the requirements of the customer. Through the use-case modelling, the external actors that have interest in the system are modelled along with functionality they require from the system (the use cases). The actors and use cases are modelled with relationships, and have communication associations with each other or are broken down into hierarchies. The actors and use cases are described in a UML use-case diagram. Each use case is described in text, and that specifies the requirements of the customer: What he or she expects of the system without considering how the functionality will be implemented. A requirement analysis can also be done with business processes, not only for software systems.

Analysis

The analysis phase is concerned with the primary abstractions (classes and objects) and mechanisms that are present in the problem domain. The classes that model these are identified, along with their relationships to each other, and described in a UML class diagram. Collaborations between classes to perform the use cases are also described, via any of the dynamic models in UML. In the analysis, only classes that are in the problem domain (real-world concepts) are modelled – not technical classes that define details and solutions in the software system, such as classes for user interface, databases, communication, concurrency, and so on.

Design

In the design phase, the result of the analysis is expanded into a technical solution. New classes are added to provide the technical infrastructure: the user interface, database handling to store objects in a database, communication with other systems, interfacing to devices in the system, and others. The domain problem classes from analysis are “embedded” into this technical infrastructure, making it possible to change both the problem domain and the infrastructure. The design results in detailed specifications for the construction phase.

Programming

In the programming, or construction, phase, the classes from the design phase are converted to actual code in an objected-oriented programming language (using a procedural language is *not* recommended). Depending on the capability of the language used, this can either be a difficult or an easy task. When creating analysis and design models in UML, it is best to avoid trying to mentally translate the models into code. In the early phases, the models are a means to understand and structure a system; thus, jumping to early conclusions about the code can be counterproductive to creating simple and correct models. The programming is a separate phase during which the models are converted into code.

Test

A system is normally tried in unit tests, integration tests, system tests, and acceptance tests. The unit tests are of individual classes or a group of classes, and are typically performed by the programmer. The integration test integrates components and classes in order to verify that they cooperate as specified. The system test views the system as a "black box" and validates that the system has the end functionality expected by the end user. The acceptance test conducted by the customer to verify that the system satisfies the requirements is similar to the system test. The different test teams use UML diagrams as the basis for their work: unit tests use class diagrams and class specifications, integration test typically use component diagrams and collaboration diagrams, and the system tests implement user-case diagrams to validate that the system behaves as initially defined in these diagrams.

2.6 State Diagrams

All systems have a static structure and dynamic behaviour, and the UML provides diagrams to capture and describe both these aspects [2]. Class diagrams are best used t document and express the static structure of a system – the classes, objects, and their relationships. State, sequence, collaboration, and activity diagrams are best used to express the behaviour (the dynamics) of a system, to demonstrate how the objects interact dynamically at different times during the execution of the system.

State diagrams capture the life cycles of objects, subsystems, and systems. They tell the states of an object can have and how events (received messages, time elapsed, errors, and conditions becoming true) affect those states over time. A state diagram should be attached to all classes that have clearly identifiable states and complex behaviour; the diagram specifies the behaviour and how it differs depending on the current state. It also illustrates which events will change the state of the object of the class.

System, subsystems, and components are typically viewed as either transformational or reactive. Transformational components are stateless components; that is, they have no memory that persists between successive invocations of the components [3]. Reactive systems, in contrast, perform an ongoing and often never-ending computation, in which each invocation uses information embedded in the past memory that is required for future. *State*-based models, such as UML statecharts, are one of the most popular methods of describing the behaviour of reactive components.

Finite state machines (FSM) have been used for more that half a century to describe reactive components. Statecharts, being an extension of FSM, continue this tradition and are the most popular language for modelling reactive components [3].

Before state-charts were introduced, FSMs were often considered a good representation for classroom models of reactive components, but in practice, when they are applied to larger problems, the models were cluttered and unreadable [3]. The primary reason for this is that FSMs are flat and sequential and therefore do not scale well when applied to large components. David Harel introduced statecharts in the 1980s as a language for describing complex reactive systems found in contemporary avionics systems. In the next two decades, statecharts were adopted by the Object Modelling Technique (OMT) methodology and later by its descendent, the Unified Modelling Language (UML) standard.

In a nutshell, statecharts extend FSMs with these capabilities: events and conditions; hierarchy (state nesting); concurrency, or state orthogonality; and history states. In addition, statecharts have built-in capabilities for describing their interaction with multiple objects in their environment [3].

The three key elements of statecharts are states, events and transitions:

- State – "a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event" [4];
- Event – "the specification of a noteworthy occurrence that has a location in time and space" [4];
- Transition – the movement from one state to another in response to an event [1].

2.6.1 States

The UML Reference Manual [4] defines a state as "a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for event". The state of an object varies over time, but at any particular point it is determined by:

- The object attribute values;
- The relationships it has to other objects;
- The activities it is performing.

Over time, objects send messages to one another, and these messages are events that may cause changes in object state [1].

All objects have a state; the state is a result of previous activities performed by the object, and is typically determined by the values of its attributes and links to other objects [2]. Examples of object states are:

- The invoice (object) is paid (state).
- The car (object) is standing still (state).
- The engine (object) is running (state).
- Jim (object) is playing the role of a salesman (state).
- Kate (object) is married (state).

An object changes state when something happens, which is called an event; for example, someone pays invoice, starts driving the car, or gets married. There are two dimensions of dynamics: the interaction and the internal state changes. Interactions describe the object's external behaviour and how it interacts with other objects (by sending messages or linking to each other). Internal state changes describe how objects are altering states – for example, the values of its internal attributes. State diagrams are used to show how objects react to events and how they change their internal state; for instance, an invoice changes state from unpaid when someone pays it. When an invoice is created, it enters the state of unpaid (Figure 2) [2].

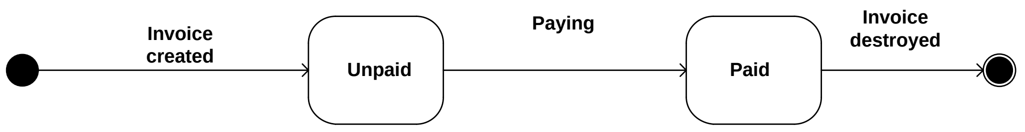

Figure 2 A state diagram for invioces. The solid filled circle indicates the starting point of invoices (object created). The circle surrounding the solid filled circle indicates the end point (object destroyed). The arrow between the states show transitions and the events that cause them.

State diagrams may have a starting point and several end points. A starting point (initial state) is shown as a solid filled circle, and an end point (final state) is shown as a circle surrounding a smaller solid circle. A state in a state diagram is shown as a rectangle with rounded corners. Between the states are state transitions, shown as a line with an arrow from one state to another. The state transitions may be labelled with the event causing the state transition. When an event happens, the transition from one state to another is performed (it is sometimes said that the transition "fires" or that the transition "is triggered").

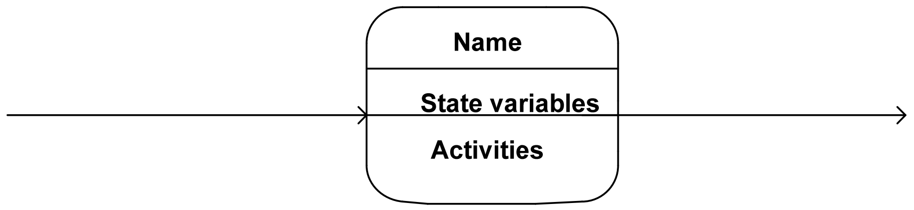

Figure 3 Name, state variable, and activity compartments

A state may contain three kinds of compartments. The first compartment shows the name of the state, for example, idle, paid, and moving. The second compartment is the optional state variable compartment, where attributes (variables) might be listed and assigned. The attributes are those of the class displayed by the state diagram; sometimes, temporary variables are useful in states, for example, counters. The third compartment is the optional activity compartment, where events and actions might be listed. Three standard events may be used in the activity compartment: *entry, exit* and *do*. The entry event can be used to specify actions on the entry of a state; for instance, assigning an attribute or sending a message. The exit event can be used to specify actions on exit from a state. The do event can be used to specify an action performed while in the state; for example, sending a message, waiting, or calculating. These standard events may not used for other purposes. The formal syntax for the activity compartment is:

event-name argument-list '/' action-expression

The event-name can be any event, including the standard events entry, exit, and do. The action-expression tells which action should be performed (e.g. operation calls, incrementing attribute values, etc.). It is also possible to specify arguments to events (entry, exit, and do events do not have any arguments.

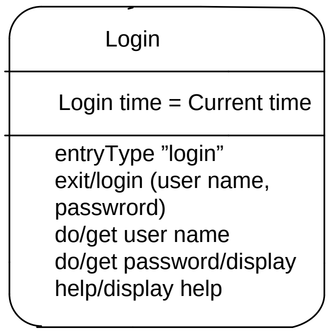

Figure 4 A state called login, where the login time is assigned to the current time, and where a number of actions are performed on entry, exit, and while in the state. The help/display help is a user-defined event and action within the action compartment

A state transition normally has an event attached to it, but it is not necessary. If an event is attached to a state transition, the state transition will be performed when the event occurs. A do-action within a state can be an ongoing process (e.g. waiting, polling, operation control etc) performed while the object is in the given state. A do-action can be interrupted by outside events, meaning that an event on a state transition may interrupt an ongoing internal do-action.

If a state transition does not have an event specified, the attached state will change when the internal actions in the source state are executed (if there are any internal actions such as entry, exit, do, or user-defined actions). Thus, when all the actions in a state are performed, a transition without an event will automatically be triggered.

The formal syntax for specifying a state transition is:

event-signature ' [' guard-condition '] ' ' / ' action-expression ' ^ ' send-clause

where the event-signature syntax as:

event-name ' (' parameter ' ,' , ...') '

and the send-clause syntax is:

destination-expression '.' Destination-event-name ' (' argument ',' ...') '

where the destination-expression is an expression that evaluates an object or a set of objects. Example of event-signatures, guard-conditions, action-expressions, and send-clauses are given next.

Event-Signature

An event-signature, which consists of an event-name and parameters, specifies the event that triggers a transition, along with additional data connected to the event. The parameters are a comma-separated list of parameters with the syntax:

```
    Parameter-name ':' type-expression, Paramater-name ':'
    type-expression ...
```

where the parameter-name is the name of the parameter and the type-expression is the type of the parameter, for example, integer, Boolean, and string. The type-expression may be suppressed, that is, not shown.

Examples of state transitions with event-signature are:

```
draw (f : Figure, c : Colour)
redraw ()
redraw
print (invoice)
```

Guard-Condition

Guard-condition is a Boolean expression placed on a state transition. If the guard-condition is combined with an event-signature, the event must occur *and* the guard-condition must be true for the transition to fire. If only a guar-condition is attached to a state transition, the transition will fire when the condition becomes true. Examples of state transitions with a guard-condition are:

```
[t = 15sec]
[number of invoices > n]
withdrawal (amount) [balance >= amount]
```

Action-Expression

Action-expression is a procedural expression executed when the transition fires. It may be written in terms of operations and attributes within the owing object (the object that owns all of the states) or with parameters within the event-signature. It is possible to have more than one action-expression on a state transition, but they must be delimited with the backward slash (/) character. The action-expressions are executed one by one in the order specified (from left to right). Nested action-expressions or recursive action-expressions are not allowed. It is, however, possible to have a state-transition that contains only an action-expression. Examples of state transitions with an action-expression are (:= is used for assignment):

```
    increase () / n := n + 1 / m := m +1
    add (n) / sum := sum + n
    /flash
```

Send-Clause

The send-clause is a special case of an action. It is an explicit syntax for sending a message during the transition between two states. The syntax consists of a destination-expression and an event-name. The destination-expression should be one that evaluates an object or a set of objects. The event-name is the name of an event meaningful to the destination object (or set of objects). The destination object can also be the object itself.

```
[timer = Time-out] / go down (first floor)
```

can be translated to a send-clause as:

```
[timer = Time-out] ^ self.go down (first floor)
```

Other examples on state transitions with a send-clause are:

```
out_of_paper() ^ indicator.light()
left_mouse_btn_down(location) / colour:=pick_colour(location) ^
pen.set(colour)
```

State diagrams should be easy to communicate and understand (as all models should), but sometimes it is tricky to express complex internal dynamics (the object's internal states and all the state transitions) and at the same time create a model that is easy to communicate. In each situation, the modeller must decide whether to model all internal dynamics as they appear, in detail, or to simplify them to make it easier to understand the model (a simplification could be temporary).

Events

An event is something that happens and that may cause some action. For example, when you press the Play button in your CD player, it starts playing (provided that the CD player is turned on, a CD is loaded, and the CD player is otherwise in order). The event is that you press the Play button, and the action is that it starts playing. When there are well-defined connections between events and actions, it is called *causality.* In software engineering, it is normally modelled causal systems in which events and actions are connected to each other [2]. There are four kinds of events within UML:

- *A condition becoming true:* This is shown as a guard-condition on state transition.
- *Receipt of an explicit signal from another object:* The signal is itself an object. This is shown as an event-signature on state transitions. This type of event is called message.
- *Receipt of a call on an operation by another object (or by the object itself):* This is shown as an event-signature on state transitions. This type of event is also called message.
- *Passage of a designated period of time:* The time is normally calculated after another designated event (often the entry of the current state) or passage of a given amount of time. This is shown as a time-expression on state transitions.

Errors are also events and may be useful to model. The UML does not give any explicit support for error events, but it can be modelled as a stereotyped event; for example:

```
<<error>> out_of_memory
```

It is important to know some basic semantics about events. First, events are triggers that activate state transitions; these events are processed one at a time. If an event potentially can activate more than one state transition, only one of the state transitions will be triggered (which one is undefined). If an event occurs, and the state transition's guard-condition is false, the event will be ignored (the event will not be stored, triggering the transition when the guard-condition later becomes true).

A class can receive or send messages; that is, operation calls or signals. The event-signature for state transitions is used for both. When an operation is called, it executes and produces a result. When a signal object is sent, the receiver catches the object and uses it. Signals are ordinary classes, but used only for sending signals; they represent the unit sent between objects in the system. The signal classes may be stereotyped with the "signal" stereotype, which constrains the semantics of the objects, meaning that only they can be used as signals. It is possible to build signal hierarchies supporting polymorphism, so that if a state transition has en event-signature specifying a specific signal, all the sub-signals will also be receivable by the same specification.

2.6.2 Events

UML defines an event as "the specification of noteworthy occurrence that has a location in time and space." Events trigger transitions in state machines. Events can be shown externally on transitions.

There are four types of event, each of which has different semantics:

- Call event
- Signal event
- Change event
- Time event

Call Events

A call event is a request for a specific operation to be invoked in an instance of the context class. A class event should have the same signature as an operation of the context class of the state machine. Receipt of a call event is a trigger for the operation to execute. As such, a call event is perhaps the simplest type of event.

As an example, we look at a fragment from the state machine of a SimpleBankAccount class. This class is subject to the following business constraints:

- Accounts must always have a balance greater than or equal to zero;
- A withdrawal will rejected if it would take balance below zero.

You can specify a sequence of actions for a call event where each action is separated by semicolon. These actions specify the semantics of the operation, and they can use attributes and operations of the context class. If the operation has a return type, the call event has a return value of that type.

Signal Events

A signal event is a package of information that is sent asynchronously between objects. You model a signal as a stereotyped class that holds the information to be communicated in its attributes. A signal usually does not have any operations because its sole purpose is to carry information.

A signal receipt is indicated by a concave pentagon. It specifies an operation of the context class that accepts the signal as a parameter.

You can also show signal receipts on internal or external transitions by using that standard *eventName/ action* notation.

Change Events

A change event is specified as a Boolean expression.The action associated with the event is performed when the value of the Boolean expression transitions from *false* to *true*. All values in the Boolean expression must be constants, globals, or attributes or operations if the context class. From implementation perspective, a change event implies continually testing the Boolean while in the state.

We have modified the SimpleBankAccount state machine so that the manager will be notified if the account *balance* goes greater than or equal to 5000. This notification is so that the manager can alert the customer about other investment options.

Change events are *positive edge triggered.* This means they are triggered each time the value of the Boolean expression changes from *false* to *true.* The Boolean expression must go back to *false* and then transition to *true* again for the change event to be retriggered.

Positive edge triggering is exactly the behaviour we want for this example of SimpleBankAcount. The action *notifyManager()* will only be invoked when the account balance transitions from less than 5000 to 5000 or more. Clearly, if the balance is oscillating rapidly around 5000, the manager will receive multiple notifications. We assume that the manager has some business process in place to handle this.

Time Events

Time events are usually indicated by the keywords *when* and *after*. The keyword *when* specifies a *particular* time at which the event is triggered; *after* keyword specifies a threshold time after which the event is triggered. Examples are *when (data = 07/10/2005)* and *after (3 months)*.

You should always ensure that the units of time (hours, days, months, etc) are recorded on the diagram for each time event. Any symbols in the expression must be constants, globals, or attributes or operations of the context class.

2.6.3 Transitions.

Transitions in behavioural state machines have a simple syntax that may be used for external transitions (shown by an arrow) or internal transitions (nested with a state). Every transition has three optional elements:

1. Zero or more events – these specify external or internal occurrences that can trigger the transition

2. Zero or one guard condition – this is a Boolean expression that must evaluate to *true* before transition can occur. It is placed after the events.

3. Zero or more actions – this is a piece of work associated with the transition, and occurs when the transition fires.

The action may involve variables in the scope of the state machine. For example:

actionPerformed(actionEvent)/ command = actionEvent.getActionCommand()

In this example, actionPerformed(actionEvent) is an event generated by a button press in a Java GUI. On receipt of this event, we execute an action that stores the name of the button in the variable named *command*.

2.6.4 Substates

A state may have nested substates, whereby internally the state has its own substates that can be shown in other state diagrams [2]. The substates may *and-substates* or *or-substates.* An *or-substate* indicates that a state has substates, but only one at a time. For instance, a car may be in the running state, which has two different substates: forward and backward. These are *or-substates* because they may not be true at the same time. The nested substates can be displayed in another state diagram by expanding the running state in the initial state diagram.

On the other hand, the running state can also have many concurrent substates (*and-substates)*: forward and low speed, forward and high speed, backward and low speed, backward and high speed. When a state has *and-substates*, several of them can be true in parallel, indicating that a state can have substates that are both *or-substates* and *and-substates*. *And-substates* are also called concurrent substates, and can be used when modelling the states of concurrent threads.

2.7 UML Methods

UML, as presented in previous sections, is not a methodology but an open, extensible, industry standard visual modelling language which is approved by the Object Management Group (OMG). In this section I will describe a methodology that best compliments UML: the Unified Process (UP).

A software engineering process (SEP), also known as a software development process, defines the "*who", "what", "when",* and "*how"* of developing software. While UML is the visual language part of UML, UP and SEP is the process in which we turn user requirements into software. UML is standardized by the OMG, and the UP is not [2].

To model the “who” of the SEP, UP introduces the concept of the worker. This describes a role played by an individual or team within the project. Each worker may be realized by many individuals or teams, and each individual or team may perform as many different workers.

UP models the “what” as activities and artifacts. Activities are tasks that will be performed by individuals or teams in the project. These individuals or teams will always *adopt specific roles* when they perform certain activities and so, for any activity, UP can tell us the workers (roles) that participate in that activity. Activities may be broken down into finer levels of detail as needed. Artifacts are things that are inputs and outputs to the project – they may be source code, executable programs, standards, documentation, and so on.

UP models the “when” as workflows. These are sequences of related activities that are performed by workers. Workflows may be broken down into one or more workflow details that describe the activities, roles, and artifacts involved in the workflow.

UP is an Iterative and Incremental Process

In order to understand UP, we need to understand iterations. The idea is very simple as experience tells, that small problems are easier to solve than large problems. We therefore break a large software development project down into a number of smaller “mini projects”, which are easier to manage and to complete successfully. Each of these “mini projects” is iteration. Each iteration contains *all* of the elements of a normal software development project, as in the following:

- Planning
- Analysis
- Construction
- Integration and test
- An internal or external release

Each iteration generates a baseline that comprises a *partially complete* version of the final system and any associated project documentation. Baselines build on each other over successive iterations until final finished system is achieved. The difference between two consecutive baselines is known as an increment. This is why UP is known as an iterative and incremental life cycle.

3. Software Testing

Software testing is an empirical investigation conducted to provide stakeholders with information about quality of product or service under test [24]

Regarding a general definition of testing and a detailed definition of software testing in particular, [16] cites IEEE Software Engineering Body of Knowledge as follows (first the general definition of testing and then the detailed definition of software testing):

> Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems.
>
> Software testing consists of the *dynamic* verification of the behaviour of a program on a *finite* set of test cases, suitably *selected* from the usually infinite executions domain, against the *expected* behaviour.

According to the standard IEEE software engineering terminology, a *failure* is an undesired behaviour. Failures are typically observed during the execution of the system being tested. A *fault* is the *cause* of the failure. It is an error in the software, usually caused by human error in the specification, design, or coding process. It is the execution of the faults that causes failures. Once a failure is observed, it can be investigated to find the fault that caused it and is thus that fault is corrected.

3.1 Testing Methods

Software testing methods are traditionally divided into *black box* testing and *white box* testing [24]. These two methods are used to describe the point of view that a test engineer takes when designing test cases.

Black Box Testing

Black box testing treats the software as a black box without any knowledge of internal implementation. It includes *model-based testing, traceability matrix* and *specification-based testing* [24].

Specification-Based Testing

Specification-based testing aims to test the functionality according to the requirements [24]. Therefore the tester inputs data and only sees the output from the test object. This level of testing usually requires thorough test cases to be provided to the tester who then can simply verify that for a given input, the output value (or behaviour), is the same as the expected value specified in the test case.

White Box Testing

White box testing, by contrast to black box testing, is when the tester has access to the internal data structures and algorithms (and the code that implement these) [24].

Types of White Box Testing

The following types of white box testing exists [24]:

- Code Coverage: It creates tests to satisfy some criteria of code coverage
- Mutation Testing methods
- Fault Injection Methods
- Static Testing: White box testing includes all static testing

3.2 Model-Based Testing

There is an increasing interest in model-driven development for object-oriented systems, using the Unified Modelling Language (UML). In addition to being a key resource for designing object-oriented software, models are very useful in testing object-oriented software.

According to [9], there are four main approaches known as model-based testing:

1. Generating of test input data from a domain model
2. Generating of test cases from en environment model
3. Generating of test cases with oracles from a behaviour model
4. Generating of test scripts from abstract test

In the first meaning, when model-based testing is used to mean the generation of test input data, the *model* is the information about the domains of the input values and the test generation involves clever selection and combination of a subset of those values to produce test input values.

The second meaning of model-based testing uses a different kind of model, which describes the expected *environment* of the system under test (SUT). From these environment models it is possible to generate sequences of calls to the SUT. However, like the previous approach, the generated sequences do not specify the expected *outputs* of the SUT. It is not possible to predict the output values because the environment model does not model the behaviour of the SUT. So it is difficult to determine accurately whether a test has passed or failed – crash/no-crash verdict may be all that is possible.

The third meaning of model-based testing is the generation of executable test cases that include *oracle* information, such as the expected output values of the SUT, or some automated check on the actual output values to see if they are correct. This is obviously a more challenging task than just generating test input data or test sequences that call the SUT but do not check the results. To generate tests with oracles, the test generator must know enough about the expected behaviour of the SUT to be able to predict or check the SUT output values. In other words, with this definition of model-based testing, the model must describe the expected *behaviour* of the SUT, such as the relationship between its inputs and outputs. But the advantage of this approach is that it is the only one of the four that addresses the whole test design problem from choosing input values and generating sequences of operation calls to generate executable test cases that include verdict information. This kind of model-based testing is more sophisticated and complex than the other meanings, but it has a greater potential paybacks. It can automate the complete test design process, given a suitable model, and produces complete test sequences that can be transformed into executable test scripts. With this view of model-based testing, model-based testing can be defined as *the automation of the design of black-box tests.* The difference form usual black-box testing is that rather than manually writing tests based on the requirements documentation, a *model* of the expected SUT behaviour can instead be created, which captures some of the requirements. Then the model-based testing tools are used to automatically generate tests from that model.

The fourth meaning of model-based testing is quite different: it assumes that we are given a very abstract description of a test case, such as a UML sequence diagram or a sequence of a high-level procedure calls, and it focuses on transforming that abstract test case into a low-level test script that is executable. With this approach, the model is the information about the structure and API (Application Programming Interface) of the SUT and the details of how to transform a high-level call to executable test script.

3.3 Mutation Analysis Process

Mutation analysis introduces faults into software by creating many versions of the software, each containing one fault [22]. Test cases are then used to execute these faulty programs with the goal of distinguishing the faulty programs from the original program. These faulty versions of the original programs are called *mutants* of the original, and a mutant is *killed* by distinguishing the output of the mutant from that of the original program.

The mutation testing process begins with the selection of mutation operators. Mutation operators are related to the set of fault classes, which are analyzed in [23]. The set of fault classes analyzed include:

- Variable Reference Fault (VRF) – replace a Boolean variable x by another variable y, $x \neq y$.
- Variable Negation Fault (VNF) – replace a Boolean x by x.
- Expression Negation Fault (ENF) – replace a Boolean expression p by p.
- Missing Condition Fault (MCF) – a failure to check preconditions.

Each fault class has a corresponding mutation operator. Applying a mutation operator gives rise to a fault in that class. The mutation operator which is most used in the two tests is Relational Operator Replacement (RRO). With this mutation operator, a relational operator ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$) is replaced by any other relational operator, except its opposite. Another mutation operator chosen was Logical Operator Replacement (LRO), which replaces a logical operator (&, |) by another logical operator.

The goal of software testing is obviously to detect faults in the SUT [16]. Hence, the main reason for the selection of the mutation operators is based on the selection criterion chosen for these tests, which is fault-based criterion.

The user of mutation testing adds test cases (generated manually or automatically) to the mutation system and checks the output of the program on each test case to see if it is correct. If the output is incorrect, a fault has been found and the program must be modified and the process restarted. If the output is correct, that test case is executed against each live mutant. If the output of a mutant differs from that if the original program on the same test case, the mutant is assumed to be incorrect and it is killed [16].

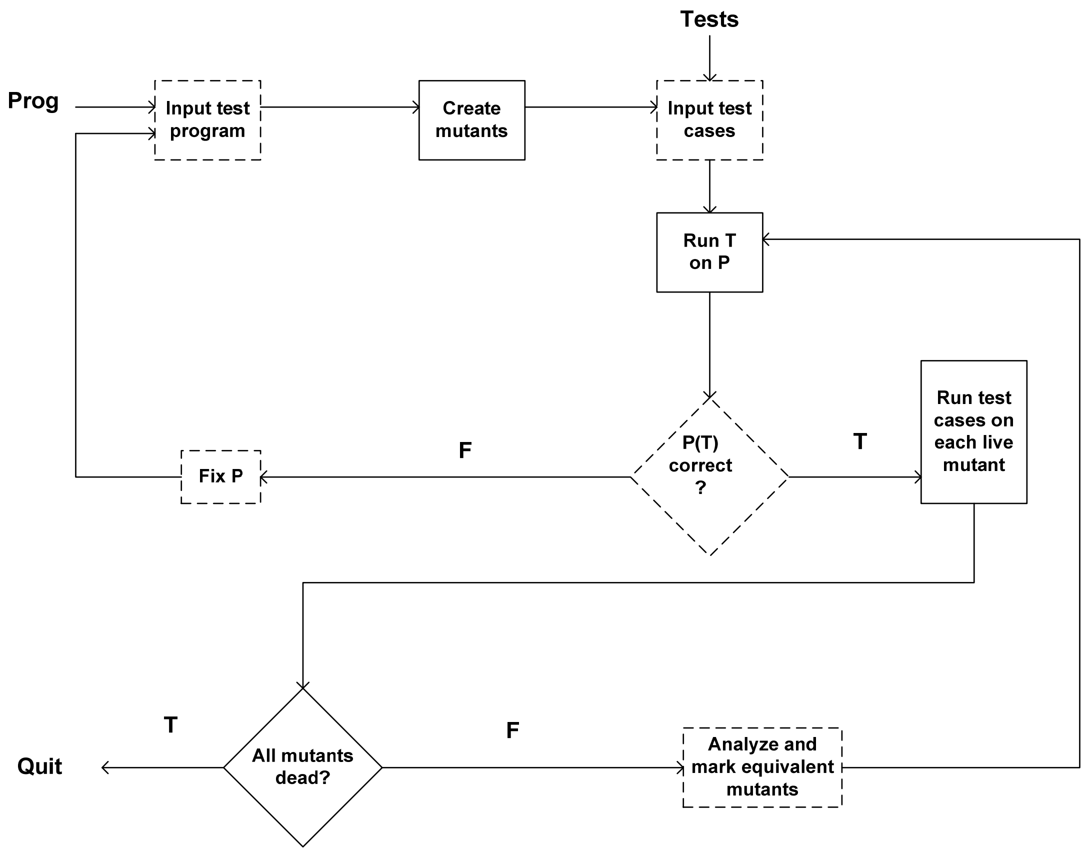

Figure 5: Traditional Mutation Testing Process

Mutation analysis provides a test *criterion*, rather than a test *process*. A *testing criterion* is a rule or collection of rules that imposes requirements on a set of test cases. Test engineers measure the extent to which a criterion is satisfied in terms of *coverage*; a set of test cases achieves 100% coverage if it completely satisfies the criterion. Coverage is measured in terms of the requirements that are imposed. *Test requirements* are specific things that must be satisfied or covered.

When a program is submitted to a mutation system, the system first creates many mutated versions of the program. A mutation operator is the rule that is applied to a program to create mutants. Typical mutation operators, for example, replace each operand by other syntactically legal operand, or modify expressions by replacing operators and inserting new operators, or delete entire statements. The solid boxes represent steps that are automated, and the dashed boxes represent steps that are done manually.

Next, test cases are supplied to the system to serve as inputs to the program. Each test case is executed on the original program and the tester verifies that the output is correct. If it is correct, the test cases are executed on each mutant program. If the output of a mutant program differs from the original (correct) output, the mutant is marked as being dead. Dead mutants are not executed against subsequent test cases.

Once all test cases have been executed, a *mutation score* is computed. The mutation score is the ratio of the dead mutants over the total number of non-equivalent mutants. Thus a tester's goal is to raise the mutation score to 1.00, indicating that all mutants have been detected. A test set that kills all the mutants is said to be *adequate* relative to the mutants. A test data set is called mutation-adequate if its mutation score is 100%.

4. ABB

In this section, I'll give brief background introduction about ABB since it is the case study of this thesis.

According to [10], ABB – formerly Asea Brown Bovery, is a multinational corporation headquartered in Zurich, Switzerland, operating mainly in the power and automation technology areas. ABB is also one of largest engineering companies as well as one of the largest conglomerate companies in the world. ABB has operations in around 100 countries, with approximately 112 000 employees (2007).

ABB is active in many sectors with its core business in power and automation technology. Here is summery of the main division of ABB.

Power Products

The division incorporates ABB's manufacturing network for transformers, switchgear, circuit breakers, cables, and associated high voltage and medium voltage equipment such as digital protective relays. Power Products are the key components to transmit and distribute electricity. It also offers all the services needed to ensure products' performance and extend their lifespan. The division is subdivided into three business units – High Voltage Products, Medium Voltage Products and Transformers.

Power Systems

Power Systems offers ready-to-use systems and services for power transmissions and distribution grids, and for power plants. Electrical substations and substation automation systems are key areas. Additional highlights include flexible alternating current transmissions systems (FACTS), high-voltage direct current (HVDC) systems and network management systems. In power generation, Power Systems offers the instrumentation, control and electrification of power plants. The division is subdivided into four business units – Grid Systems, Substation Management, and Power Generation.

Automation Products

It provides products to improve customers' productivity, including drives, electric motors and generators, low voltage products, instrumentation and analytical and power electronics. The customer profile of this division includes a wide range of industry and utility operations, plus commercial and residential buildings.

Process Automation

The main focus of this ABB business is to provide customers with integrated solutions for control, plant optimization, and industry-specific application knowledge. The industries served include oil and gas, power, chemicals and pharmaceuticals, pulp and paper, metals and minerals, marine and turbo-charging.

Robotics

ABB has one of the world’s largest installed bases of industrial robots also providing robot software, peripheral equipment and modular manufacturing cells. ABB’s robots are provided for tasks such as welding, assembly, painting and finishing, picking, packing, palletizing and machine tending. Key markets include automotive, plastics, metal fabrication, foundry, packaging, material handling, and food and beverage industries. In 2006 ABB’s global robotics headquarters moved to Shanghai, China and in 2007 ABB added manufacturing unit for industrial robotics in China serving local market.

4.1 Use of UML at ABB

UML and UML-based development methods have become de facto standards in industry, and there are many claims for the positive effects of modelling object-oriented systems using methods based on UML. The paper [5] presents a case study in ABB, which is conducted to identify benefits, difficulties and their causes when a UML-based development is introduced.

According to the paper [5] UML-based development improved traceability, communication, design, documentation and testing. But these improvements were not as great as they could have been because of difficulties with the use of UML, particularly in 1) choice of diagram to use in specific situations, 2) the interfaces between different models 3) the level of detail in the models. And the result showed that these difficulties were caused by project decisions with respect to the reverse engineering of legacy code, distribution of requirements to teams, and choice of modelling tools.

Before the project reported in the paper [5], there was little common streamlining of software development in ABB, according to the paper; a large number of different methods, programming languages and software tools were used. ABB hoped that the introduction of UML-based development would lead to improvements in requirements handling and traceability, improved design of code, fewer defects in the product and reduced overall costs of development.

ABB has an existing methodological framework into which UML-based development was introduced. The overall ABB development follows closely a traditional way of development, and the method is used together with the ABB Gate model for projects. The goal of the ABB UML method, according to [5], was cover the lower part of the V-model, from requirements analysis to functional testing. The first time the method was used emphasis was mostly on analysis and design. The ABB UML model was developed internally. It was based not on any particular method for UML-based developed; it is also generic and thus had no particular relevance to the specifics of software development in ABB.

According to the paper [5], the ABB UML method also prescribes the use of use cases, sequence diagrams, deployment diagrams and class diagrams. The use of state chart diagrams and activity diagrams is optional. The paper also reports that, when it comes to the choice of diagrams, there are problems with their choice of diagram to apply. It is reported that there was too much focus on use cases and sequence diagrams in the ABB UML method, and the interviewees thought that activity diagrams could be more useful early in analysis when few actual objects have been identified.

When it comes to interfaces, [5] reports that the interviewees experienced large problems with the interfaces, especially between models describing different, but interacting, parts of the system.

4.2 Evaluating the Use of UML at ABB

This section describes experience with UML in the application of software development at ABB.

A large proportion of software development in industry is enhancement of existing systems. ABB, which is a large multi-national company operating in many countries decided to attempt to improve the company's software development projects through the use of model-driven development with UML. The aim of the study, which is presented in [6], was to improve, among other things, communication between stakeholders in the projects; the design and documentation of the system as well as the testing procedures. A UML-based development method was developed and applied in a large development project that comprised approximately 230 people. It was decided to evaluate whether the experience of applying UML-based development was positive from the prospective of the individual practitioners and the project managers.

According to [6], the results showed that adopting the UML diagrams that included use cases diagrams, sequence diagrams, and class diagrams in the legacy environments – where it was necessary to model exiting code and functionality as well as to introduce functional enhancements, produced more challenges; it also turned out to be more costly as a consequence. The use of other UML diagrams like activity diagram and state diagram was optional, and such diagrams were used by few developers, and hence were not subject in the study.

Identifying and documenting use cases, in particular, were much more difficult in legacy development than in development from scratch. The use of UML diagrams also yielded fewer benefits in legacy development, although there were some positive effects also in legacy development, in particular from applying sequence and class diagrams. The most positive benefits were obtained with respect to using UML diagrams as input to testing.

The results presented in [6] confirmed that there is potential for model-driven development with UML, also in legacy development. The results also showed that there is a need for more methodological support on constructing and applying UML diagrams in legacy development, and in particular, steps should be taken to improve methodological support for a) reverse engineering from code to UML model, and b) the application of UML in design of enhancements of legacy code.

The positive resulted with respect to UML-based testing reported in the paper has led ABB to an increased focus on how the UML-models can be used in a more systematic way to further improve the company's testing procedures.

Another major study on the use of UML, which also was conducted at ABB and cited in [6], reports experiences from introducing UML-based development in a large safety-critical project. The goal of this project, presented in [5], was to create a new version of a safety-critical process control system based on several existing systems. According to this paper, there were few reported empirical studies on the effects of UML-based development, despite the widespread adoption of UML. Hence, interviews were conducted with 16 managers and developers in the project who, according to the paper, represented different sites, different kinds of development and different roles in the project. The study presented in this paper put forward a number of points in its conclusion. Among them were the following:

The interviewees had obtained several immediate improvements as a consequence of introducing a UML-based development method. These included improved traceability of requirements to code, improved communication within the development teams, improved design of the code, quicker development of test cases and better coverage of these, and a product that was better documented than were earlier products.

There interviewees also stated that there had been difficulties related to obtaining these improvements and also that developments costs had increased due to the adoption of UML. When applying UML, the interviewees had experienced difficulties with choosing an appropriate diagram in specific situation, interfaces between models, and with the level of detail in the models.

The paper concluded that despite the widespread use of UML in industry, that there has been little evaluation of UML-based development in industrial projects.

In another case study on the application of models in software development, two companies – ABB and Ericsson – are presented in the paper [16]. The focus of the paper is the adoption of Model-driven software development in those two companies, but since this thesis is about ABB, we will look at the results concerning ABB. The study at the two companies was done in two distinct periods of time. The study of ABB was done at an earlier time than that of Ericsson. However, at each company the study was done in the following steps:

1. Focus group workshops and interviews
2. Questionnaires

During the interviews and focus group meeting, the list of requirements, factors and potential modelling scenarios were defined as a result of step 1 at ABB, and they were later used in the questionnaires.

The main goal of the study presented in [16] is to provide evidence on how industry approaches the issues related to adopting model-driven development. The results of this study are related to the findings from earlier study, performed at a company and partially at academia and whose context is similar to ABB, and which are cited in the paper; the results of this related study showed that model-driven development did not lead to increase in efficiency, effectiveness, or productivity in the context of software development with a large amount of legacy code.

Combining other experience which [16] cites in its case study, the paper outlines the requirements for making the introduction of MDD easier in large industrial organizations. The paper addresses the main research question in the study: which elements are important in adopting model-driven development in companies developing software for embedded systems with large knowledge base and legacy base? The study then divided the main research question into the following three sub-questions:

RQ1: What are requirements for adopting model-driven development in the company?
RQ2: What factors determine whether model-driven development should be adopted?

RQ3: Which scenario of using model-driven development is the most suitable for the company?

In conclusion, the paper wrote that, despite the promised advantages, the state-of-practice in adopting model-driven development in large industrial projects showed that the technologies behind model-driven development are not yet mature enough for the industry to fully adopt the ideas of using models as the only artifacts in software development. The results from interviews at ABB indicated that at the current state-of-practice in model-driven development, it seems to be unrealistic to use only models in the course of software development. The paper gave two reasons for that: (1) the software development methods are not fitted to use models as the main artifacts in, for example estimations, and (2) the software development environments are not mature enough to support the companies to a sufficient extent.

5. Research Method

The topic of this thesis is a case study at ABB which was conducted as part of an action research project. Both methods will be discussed in the next sections. The reason case study is also chosen as part of action research for the research for this thesis, is because the case study which is conducted at ABB is the primary source of firsthand information in the thesis. The case study method will be addressed in section 5.2.

5.1 Action Research

Action research (AR) is a process of social research where both outsiders and problem owners work together to solve problems and reach common goals. The method seeks not primarily to look for generalizations or to provide theories to be true or false, but concentrate on solving real life problems while creating knowledge. AR promotes and focuses on the interaction between the researcher and the stakeholders, contrary to conventional or critical research. Participation with the problem owners in the specific context is seen as necessary to obtain insight in matters that can not be understood when studying it from a distance.

Modern AR has its heritage from social psychology related research conducted in the 1940s. AR has since its inception developed into a multifaceted research methodology. Unlike conventional social science research that tries to avoid any intervention in the research setting, AR demands intervention. AR does not primarily seek to create general theories and to prove them true or false; AR shifts the focus away from general theories in favour of local relevance. AR is grounded in practical action aimed at solving real life problems while carefully informing theory. To inform theory AR involves a process of deliberate and systematic refection, and generally requires some sort of evidence to be presented to support conclusions. AR is inquiry that is done *by* or *to* or *on* them [12].

AR is, as previously mentioned, a multifaceted methodology which is practised in a number of different ways. According to [27], there are, however, three basic commitments that must be present for a process to be called AR; *research, participation,* and *action*.

In AR the stakeholders in a research setting and the researcher will, usually, all participating. How much stakeholders will participate will vary. In some AR research the stakeholders will be co-researchers, they will participate in the whole research cycle from identifying problems to reflectively identify learning. Many forms of AR, *Participatory Action Research* (PAR) among them, strongly emphasises this. It is important that the researcher and the stakeholders participate on level turf and develop a relationship based on mutual trust. By involving the stakeholders in identifying learning from an AR study, the creation of knowledge is made more democratic, and it is more likely that this learning would be used to inform new actions by stakeholders.

AR is partly based on the belief that action and participation brings understanding. By introducing changes into social process and observe the effect, knowledge about local context can be deduced. AR does not rely on any specific methods for capturing data during the research. The researcher, perhaps in collaboration with stakeholders, can choose whichever methods, qualitative or quantitative, found to be appropriate in the research setting. No matter which methods that eventually are chosen, it is necessary to pay close attention to the quality of the data gathered, and be able to assure the correctness of interpretations made from the data. The researcher has to be able to demonstrate that the interpretations are more likely that the alternative interpretations.

Most conventional research methods gain their rigor by control, standardization, objectivity and the use of numerical and statistical procedures. This sacrifices flexibility during a given experiment – if you change the procedure in mid-stream you don't know what you are doing to the odds that your results occurred by chance.

The most prevalent AR description details a five-phase, cyclical process. The approach first requires the establishment of a client-system infrastructure or research environment. Then, five identifiable phases are iterated [11]:

1. Diagnosing
2. Action planning
3. Action taking
4. Evaluating
5. Specifying

5.1.1 Action Research in Software Engineering

The qualitative research approach is usually used for the investigation of social phenomena, or in other words, situations in which people are involved and different kinds of processes take place. Within this arena, qualitative research is conducted in cases in which what one wishes to learn about environments, situations and processes cannot be retrieved by quantitative data analysis methods. Quantitative data analysis can shed light in many aspects of such situations and may enable us to argue with a certain degree of generalization.

In action research, the researcher attempts to solve a real-world problem while simultaneously studying the experience of solving the problem [28]. While most empirical research methods attempt to observe the world as it currently exist, action researcher aim to intervene in the studied situations for the explicit purpose of improving the situation. Action research has been pioneered in the fields such as education, where major changes in educational strategies cannot be studied without implementing them, and where implementation implies a long term commitment, because the effects may take years to emerge. It has also been adopted in information science, where organizational change can sometimes require a long time to have am impact.

A precondition for action research is to have a *problem owner* willing to collaborate to both identify a problem, and engage in an effort to solve it. In action research, the problem owners become collaborators in the research.

5.2 Case Studies

In this section, what a case study is will be given a short description, because as mentioned earlier, a case study at ABB is the primary source of firsthand information in this thesis.

A case study is an in-depth exploration of one situation, and common qualitative method used in the social sciences [17].

This explanation often needs to have a certain time span, as a snapshot of a situation at a particular moment can not capture the process of change. In a case study the researchers devote themselves to the specific situation, and the reward is a richness of data, obtained by the multiple means. A single case study can be hard to use for generalizations, but by finding other similar studies this can be addressed by developing stronger relationships for certain relationships [20].

A case study is often exploratory in nature and is a good fit for studies of little known domains. Case studies have been criticised for being a poor basis for generalizations, especially formalistic generalizations. Case studies are, however, good for naturalistic generalization. By basing assumptions on a number of case studies and by doing longitudinal case studies, stronger and stronger evidence for assumptions can be given. When the problem domain is better understood other kind of research methods line surveys and experiments can be used.

The distinctive need for case studies arises out of desire to understand complex social phenomena. In brief, the case study method allows investigators to retain holistic and meaningful characteristics of real-life events – such as individual life cycles, organizational and managerial processes, neighbourhood change, international relations, and the maturing of industries [18].

Yin also says that case study as a research method is favoured when there is a “how” or “why” question and when the relevant behaviours cannot be manipulated [19]. The contribution of case studies is through analytical generalization (e.g. generalization through theory, surface similarity, ruling out of irrelevancies, rather than statistical generalization (e.g. random sampling), where theories are explained and generalized, although the motive of a case study may also be simple presentation of individual cases.

5.3 Research Approach in this Thesis

When I started to work on this thesis I didn’t have clear understanding of embedded systems in general, or of ALC system in particular. I also did not have much knowledge in action research, especially when applied in the field of software engineering. My motivation of this thesis was to increase my understanding of model-driven software development.

It could be argued that a great deal of software engineering research is actually action research in disguise. Certainly, many new ideas in software engineering were originally developed by trying them out on real development projects.

In action research it is best to be flexible in the choice of methods used for data collection, and questions that demand answers constantly appear when going through the research cycle.

AR is a process of social research where both outsiders and problem owners work together to solve problems and reach common goals. I deem the ALC project to be fertile ground for conducting actions research at ABB. As AR promotes and focuses on the interaction between the researcher and the stakeholders, my research approach in this thesis is through participation with the problem owners (in this case, ABB) in the specific context of the ALC project. In this way it is necessary to obtain an insight in this project which otherwise would not be possible to conduct such a research when studying from distance.

6. Evaluation of Baseline and State-Based Testing

6.1 About ALC

The aim of the Application Logic Controller (ALC) Project was to enhance the safety-critical behaviour of ABB's industrial machines and develop a new version of a safety-critical system for supervising such machines

.

The main function of the ALC is to *supervise the status* of all safety related components interacting with the machine and to *initiate a stop* of the machine in a safe way:

- When one of these components requests a stop, or
- If a fault is detected.

The ALC functionally exists as a module which is integrated with other modules or components. The other main components which are integrated with it include an interface unit called MUX and also with other operator control devices which often are handheld devices. The MUX is the main, interface through which the ALC receives such input signals as which operating modes (e.g. Manual, Full Speed or Auto) to control the robot, and as well as other signals are sent back by the ALC the MUX, like for example speed mode and error messages. The handheld device used by the operator to control the robot is called the teach pendant unit (TPU).

6.1.1 The System Interfaces and Functions

As described earlier, the ALC communicates with the outside world with the MUX. The MUX thus constitutes how all the different safety relevant inputs are logically combined to generate outputs that control safe starting and stopping of the robots. The MUX communicates with several components, e.g. the main computer and robot devices. The MUX provides the necessary information to the ALC. The ALC then reacts upon the information and controls the enabling/disabling of the robot. The information about the detailed status of the robot (errors, mode, speed, etc) is called the system status. This is communicated between the ALC and the MUX, and reported to the main computer by the MUX.

The system consists of the following functions representing the various system statuses:

- **Name:** safeGuard
  Description: The safeguard is derived from several stop signals (generated by the MUX based on a logic OR of the different stop reasons), and indicates when the robot is not allowed to move.
  Type: Boolean (true = = activated, false = = deactivated)
- **Name:** speed
  Description: The speed in which the robot is operated. The speed signal shall be full when the mode selector switch selects Automatic or Manual Full Speed mode, and low when the mode selector switch selects manual mode.
  Type: Boolean (true = = full speed, false = = low speed)
- **Name:** mode
  Description: The robot can be operated in the three modes: Manual, Manual Full Speed, or Auto
  Type: Enumerator [manual, manualFullspeed, auato]
- **Name:** enableDevice

Description: This is a signal from the handheld device (TPU) with two functions: It is used to 1) confirm mode switches to manual full speed and auto, and 2) to enable the robot to move in manual mode (the enabling input is ignored in auto mode).
Type: Boolean (true = = activated, false = = deactivated)

- **Name:** blockDriveEnable
  Description: This variable is *set by the mode* in which the robot is controlled. It decides whether the robot is allowed to be operated or not.
  Type: Boolean (true = = activated, false = = deactivated)
- **Name:** driveEnable
  Description: This variable controls the motor current drive relays. If the ALC detects an error or gets an external command from the MUX to stop the robot or if one or more of the safe inputs to the ALC is set to a value that shall cause a robot to stop, it sets the drive enable signal false to stop the robot.
  Type: Boolean (true = = activated, false = = deactivated)
- **Name:** haltType
  Description: softHalt or hardHalt
  Type: Enumerator (softHalt, hardHalt)
- **Name:** softStop
  Description: A softStop makes sure the robot is stopped gently during 1 second. A softStop wears less on the motor than immediate stops. The softStop is generated by the MUX based upon configuration and current state of the other stop-singals.
  Type: Boolean (true == activated, false = = deactivated)
- **Name:** modeSoftStop
  Description: If modeSoftStop is set to true while the robot is active, then the robot is allowed to be stopped by a softStop.
  Type: Boolean (true = = activated, false = = deactivated)

6.1.2 Concurrency

There are two concurrent aspects of the ALC:

- The **operating mode** of the robot: Manual, Manual Full Speed or Auto. The mode decides whether the robot is controlled automatically (at full speed) by the computer or manually by the operator (at full or low speed).
- The **drive-enable status** of the robot: Init, Enabled, Disabled or Halted. The drive enable status decides whether the robot can be operated or not.

6.1.3 Dependencies

The robot is always in a mode, regardless of the drive enable status. However, the drive enable status some times depends on the mode of the robot (i.e. the mode is independent of the drive enable status, whereas the drive enable status depends on the mode).

6.2 Overview of Two Software Development Methods

This section will present a description of two different approaches to software development. One method focuses on UML models, while the other one is based on functional programming. The latter method is as close as possible to the traditional method used in developing software at ABB, and is therefore considered as a baseline with which to compare model-based development. Both development methods consist of three development phases: a design phase, a code phase (or implementation phase) and a test phase. Each of these phases has, in both development methods, activities with input and output artifacts.

One system, the ALC Controller module, was developed using these two different development methods, so that the participants in the ALC project would be able to make comparisons between the two methods.

The main theme of this thesis is therefore the evaluation of UML-based software development in general, and in particular how tests can be generated from models. I started working on the ALC system in the summer of 2007 and partially wrote the initial ALC code. Two versions of code were developed: a) A version developed with UML state charts by ABB b) A version developed in a procedural approach, which is similar to the method employed by ABB in its code development. My contribution was to the latter version.

6.2.1 Baseline Development

The baseline development method in ABB is based on functions-oriented programming and consists of functional units organized as blocks. These blocks, which are also called functional blocks, handle the inputs and the outputs of the ALC system. The functional blocks represent state models for the functionality of changing between the modes that the machine can operate in.

The baseline method is based on the mere decomposition of the different behaviour of the system into functional units, and is consequently based on functional programming. To develop the ALC module using the baseline development method (shown on the right portion of Figure 6), we started with an input artifact consisting of the requirements specification of the ALC system in the *Design* phase. The main activity in the *Design* phase is then to manually derive a design specification from the requirements specification. The major output artifact of the *Design* phase is a procedural design specification before the next phase starts. After this artifact is produced it is then readied for the next phase.

Following the *Design* phase is the *Code* or implementation phase. This phase starts with the main activity of the implementation of the specifications produced in the preceding *Design* phase. This is done in a function-oriented programming method. The resulting code is a functional code. This code which is produced in this way is also called a baseline code and is the major artifact in the *Code* phase.

The next phase, which is the *Test* phase, the main activity is the use of the baseline code which was implemented in the *Code* phase. Since this phase is a test phase, a baseline test suite is produced using the baseline code.

In the *Test* phase, the testing activities are conducted in two ways: The first one is by the creation of baseline test suite and, the second one by the creation of test driver, which is conducted later.

In the first case, either of three different output artifacts can be used to create a baseline test suite: a) with ABB's existing requirements specification of the ALC b) or, with procedural design specification implemented in the *Design* phase c) or, with the baseline code implemented in the *Code* phase. All of these artifacts can directly be used to implement a baseline test suite, where a baseline test suite is produced. This test suite is then used to create a test driver, where it is run and evaluated. The baseline test results are then obtained.

In the second case, the second way the test code is implemented is by using the baseline code, which was implemented in the *Code* phase, to create a test driver in the *Test* phase. This test drive is then run and evaluated, where baseline test results are obtained.

The code produced in the *Code* phase is a functional code, meaning that the code is written in a functional programming model. As the name suggests, functional programming is so called because it consists entirely of functions [13]. The main program itself is written as a function which receives the program's input as its argument and delivers the program's output as its result. Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitive.

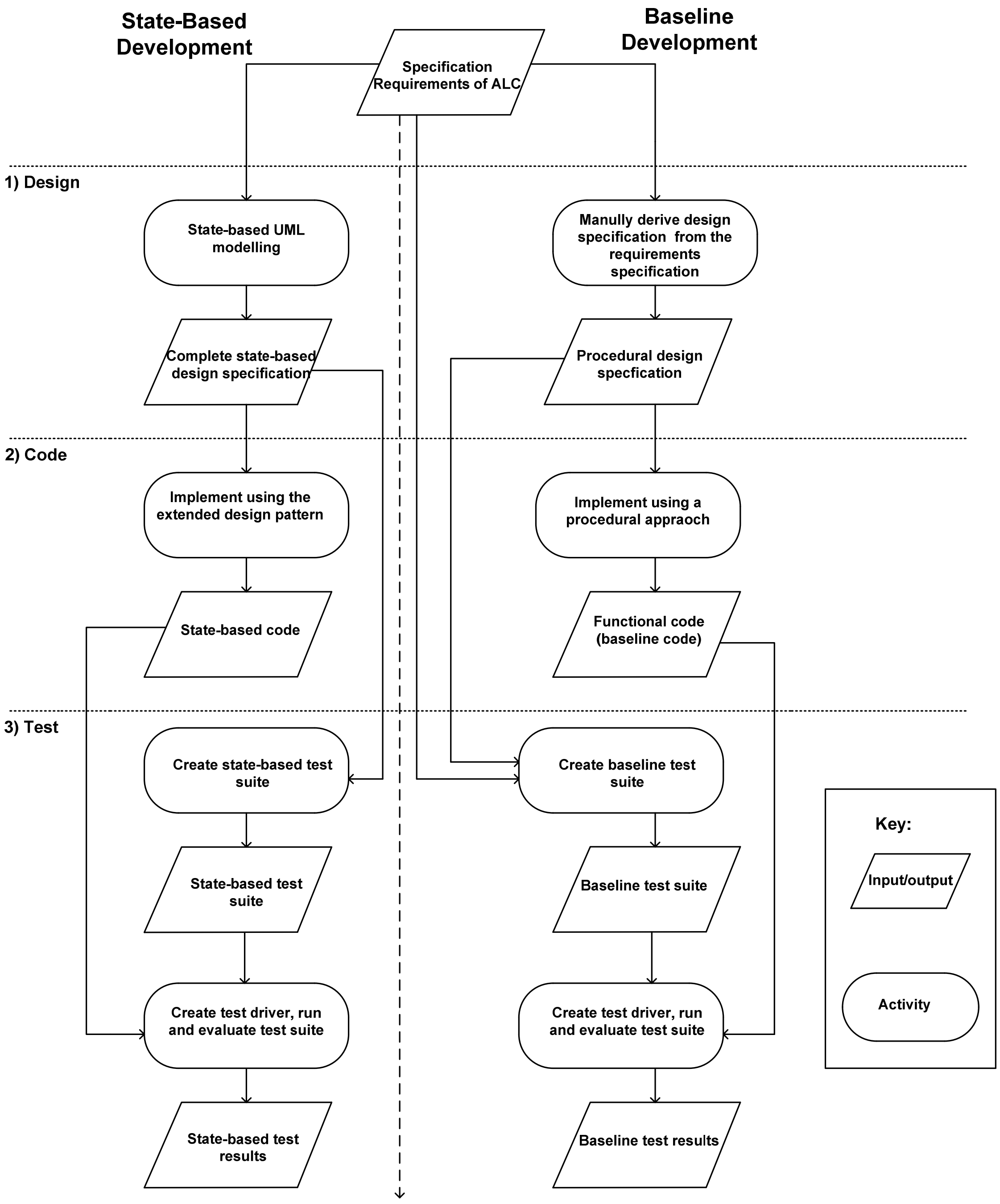
State-Based Development
Specification Requirements of ALC
Baseline Development
1) Design
State-based UML modelling
Manully derive design specification from the requirements specification
Complete state-based design specification
Procedural design specfication
2) Code
Implement using the extended design pattern
Implement using a procedural appraoch
State-based code
Functional code (baseline code)
3) Test
Create state-based test suite
Create baseline test suite
State-based test suite
Baseline test suite
Create test driver, run and evaluate test suite
Create test driver, run and evaluate test suite
State-based test results
Baseline test results
Key:
Input/output
Activity

Figure 6: Two Software Development Models

6.2.2 State-Based Development

Before I describe state-based development, I'll briefly describe an object-oriented analogy with state-chart hierarchy.

One of the cornerstones of object-oriented programming (OOP) is the concept of class inheritance, which allows you to define new classes of objects in terms of existing classes, and consequently enables you to construct hierarchically layered taxonomies of classes [3]. In a state-chart hierarchy with a superstate and a sub-state, if the sub-state is completely empty, with no internal structure (no transitions and no reactions). If such a state becomes active, it will automatically forward all events to the superstate. Therefore, the behaviour of such a sub-state will be externally indistinguishable from the superstate – the empty state *inherits* the exact behaviour from its superstate. This is analogous to an empty subclass, which does not declare any methods and attributes. An instance of such a subclass is, in every respect, equivalent to an instance of its super-class. This child class is indistinguishable from the parent class because everything is inherited exactly.

In a class taxonomy, classes lower in the hierarchy are more specialised than their ancestors; and classes higher in the hierarchy are generalization of their descendents. The same holds true in state hierarchies [3].

Behavioural inheritance is also one of the cornerstone concepts in implementing statecharts, in the same way that abstraction, inheritance, and polymorphism are cornerstone concepts in OOP [3].

To develop the ALC module in the state-based development model (shown on the left portion of Figure 6), as in the baseline development model discussed in the previous section, the state-based model is started with ABB's existing requirements specification of the ALC. The first activity in the *Design* phase is state-based UML modelling. From this state-based UML modelling, a complete state-based design specification is then generated as its main output artifact.

In the *Code* phase, the design specification produced in the *Design* phase is implemented by using the extended state-design pattern. A stated-based code is thus obtained in this way.

In the *Test* phase, the testing activities are conducted in two ways. These activities include the creation of state-based test suite and the creation of test driver.

In the first case, state-based test suite is implemented by using the state-based design specification, which was obtained i the *Design* phase with UML-modelling.

In the second case, state-based test results are obtained from the state-based code, which was implemented in the *Code* phase.

In a model-based testing, a model is a depiction of its behaviour [28]. Behaviour can be described in terms of the input sequence accepted by the system, the actions, conditions, and output logic, or the flow of data through the application's modules and routines.

Statecharts are an extension of finite state machines that specifically address modelling of complex or real-time systems [28]. They also provide for concurrent state machines. In addition, the structure of state-charts involves external conditions that affect whether a transition takes place from a particular state, which in many situations can reduce the size of the model being created.

Fundamental task of model-based testing is to understand the system under test (SUT). Forming a mental representation of the functionality is a prerequisite to building models [21].

Building the Model

In building the model, state-based testers in general define high-level state abstractions and then refine these abstractions into an actual state space. State abstractions are based on inputs and information about the applicability of each input and the behaviour that the input elicits [21].

The general procedure that underlies many model-based testing (MBT) methodologies is:

- A list of inputs is made
- For each input the situations are documented in which the input can be applied by users and, conversely, the situations in which users are prevented from applying the input. These situations are referred to as *input applicability constraints.*
- For each input the situations are documented in which the input causes different behaviours (or outputs) to be generated depending on the context in which the input is applied. These situations are referred to as *input behaviour constraints.*

Both input applicability constraints and input behaviour constraints describe scenarios that need to be tested. The fact that MBT forces a tester to explicitly think these situations through thoroughly is one of its benefits.

Generating Tests from Models

The difficulty of generating tests from a model depends on the nature of the model. Models that are useful for testing usually possess properties that make test generation effortless. For some models, all that is required is to go through combinations of conditions described in the model. In the case of finite state machines, it is as simple as implementing an algorithm that randomly traverses the state transition diagram. The sequences of arc labels along the generated paths are, by definition, tests. For examples, in the state transition in the diagram below, the sequence of inputs “a, b, d, e, f, i, j, k” qualifies as a test of the represented system [21].

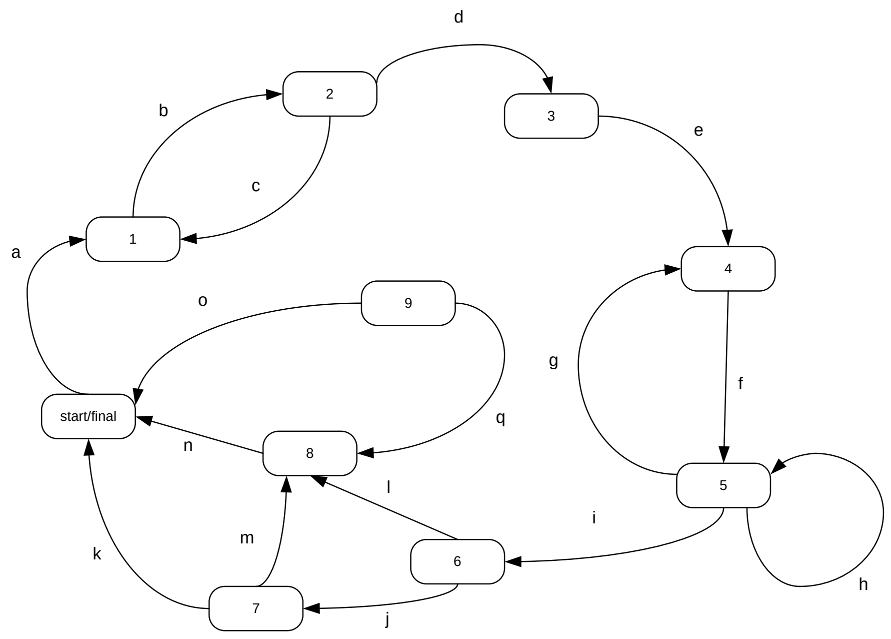

Figure 7 State machine with test paths

There are a variety of constraints on what constitutes a path to meet the criteria for tests. Examples include having the path start and end in the starting state, restricting the number of loops and cycles in a path, and restricting the states that a path can visit.

6.3 Test Plan for the Evaluation

In this section I will describe two test plans for two testing processes that are performed for the two development methods. In the first method, the testing is performed in a traditional testing process. This method is also called functional testing and often involves manual testing. The other method of testing is model-based testing, which is a method used to automate a detailed design of the test cases and their generation.

The test selection criterion for both tests is the fault-based criteria. This criterion chosen for these tests is based on the mutant analysis approach, which is a method based on the use of syntactic errors (e.g. errors in the use of relational or arithmetic operators, incorrect variable references, etc). In the following sections the two test methods will be described.

6.3.1 The Baseline (Functional) Testing

The test plan for the functional testing gives a high-level overview of the testing objectives, such as which aspects of the system under test (SUT) should be tested, what kinds of test strategies should be used, how often testing should be performed, and how much testing will be done [9].

The Baseline Test Plan

A major step in this test plan is designing the test cases from the system requirements. It is also important to take into consideration the high-level test objectives and policies. For the baseline test plan, the test size for the baseline test plan consists of the main functions in the ALC system, where 15 of the functions were selected for testing.

As mentioned earlier the selected criterion for the test is fault-based criterion. The goal is to detect faults in the system under test. The technique used in this testing is Mutation Analysis, a technique which introduces faults into the code by creating many versions of the software, each containing one or more faults. Test cases will then be used to execute the faulty programs from the original ALC system.

Another component of the test plan is the test design. This can be done manually, based on the informal requirements document. The output of this is a human-readable document that describes the desired test cases. These test cases are used to execute the mutants.

The next step is the execution of these test cases, which can be done manually. For each test, steps are then followed and since the system under test (SUT) is an embedded application, meaning that it is often not possible to interact with directly, a *test execution environment* may be used to allow the tester to enter input and observe outputs. The SUT output should then be compared with the expected output, and the test verdict should be recorded.

Test Plan Objectives

This Test Plan for the baseline testing supports the following objectives:

- Define the activities required for and conduct the baseline testing
- Communicate to all responsible parties the baseline testing strategies
- Communicate to all responsible parties the various Dependencies and Risks

The Baseline Test Scope

The scope of the baseline testing includes the following:

- The examination of the baseline code consisting of fifteen functions
- The examination of the aspects of the code: does it do what it is supposed to do and do what it needs to do
- The effectiveness of the code in general

The Baseline Test Coverage

In order to measure how well the baseline code is exercised by the test case script, the following coverage criteria will be used:

- Function coverage: Has each function in the code been executed?
- Statement coverage: Has each line of the source code been executed?

- Decision coverage (also known as Branch coverage): Has each control structure (such as an *if* statement) evaluated both to true and false?
- Condition coverage: Has the Boolean sub-expression evaluated both to both true and false (this does not need necessarily imply decision coverage)?
- Path coverage: Has every possible route through a given part of the baseline code been executed?
- Entry/exit coverage: Has every possible call and return of the function been executed?

Since the system under test, in this case the ALC system, is a safety-critical application, it is required to demonstrate that the testing achieves 100% of some form of baseline code coverage.

The Baseline Test Strategy

Test strategy is the process of analyzing a code item to detect the differences between existing conditions and required conditions and to evaluate the features of the code item [25]. The test strategy of the baseline testing mainly consists of mutation testing.

The testing strategy plan for the baseline code is the instantiation of mutation testing, where program mutants of the original baseline code are created. These mutants are created by applying a number of mutation operators. Many different faults can occur in software and accordingly, it can be mutated in a variety of ways and the selection of mutation operators depends on the fault class, as described in [23].

The plan for the mutation analysis includes the selection of fifteen functions in total. The functions selected for the analysis include the following:

```
SetBlockEnable()
SetSafeGuard()
SetSoftGuard()
SetEnableDrive()
SetHalt()
SetModeSoftStop()
SetEnableStatus()
SetModeStatus()
ActivateRobot()
SoftStop()
DisableRobot()
HaltRobot()
SetModeAuto()
SetModeManual()
SetModeManualFullSpeed()
```

The mutant operators chosen for this mutation testing are mainly arithmetic operators and relational operators. The selected mutation operators are limited to simple changes to the original program on the basis of the *coupling effect,* which says that complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect most complex faults.

6.3.2 Model-Based Testing

Model-Based Test Plan

The first step of model-based testing is to write an *abstract model* of the system that is to be tested. This is called *abstract* because it is smaller and simpler than the system under test (SUT). This step is important because we can decide on the level of *abstraction* that is which aspects of the SUT to include in the model and which aspects to emit. The model will be used to define the state-related behaviour of existing classes of the system. These classes are usually defined by a UML class diagram, so they already have data fields, methods (functions) and associations with one another. The transitions of the state machine can read and update these data fields.. The sequence of inputs in generating these tests from models is similar to the one shown in Figure 7.

The second step in the model-based testing is to generate abstract tests from the model. It should then be decided which selection criteria to choose. As in the baseline testing, the test selection criterion for this testing is fault-based criteria. The method used for this test is therefore the Mutation Analysis method mentioned earlier.

The Model-Based Test Scope

The UML-models in the model-based testing are modelled in state-chart consisting of two main superstates, namely an operating `Mode` superstate and `Drive Enable` superstate, which models the status of the robot. The Robot Safety Functional Block is in either of these two states.

The `Mode` superstate consists of three states: `Auto` state, `Manual` state and `Manual Full` state. The `Mode` decides whether the robot is controlled automatically (at full speed) by the computer or manually by the operator (at full or low speed).
The `Drive Enable` superstate consists of theses states: `Enabled`, which has the two internal states `Active` and `SoftStop`; and `Disabled` and `Halted`.

The model is usually an abstract, partial presentation of the system under test's desired behaviour. The test cases derived from this model are functional tests on the same level as the model [9].

The Model-Based Test Strategy

To apply Mutation Analysis in the context of state-charts, we create a set of mutation operators. This is done based on the error classes as defined by [30] and trying to guarantee minimal testing requirements, e.g. covering all transitions; we then create a set of mutation operators for the state-charts. These mutation operators include the following:

- Arc-missing
- Wrong-starting-state
- Event-missing
- Event-changed
- Event-extra
- State-extra
- Output-exchanged

- Output-missing
- Output-extra

Based on this set of operators, we apply Mutation Analysis to the state-chart depicted in Figure 8. Due to confidentiality constraint, this statechart is not complete.

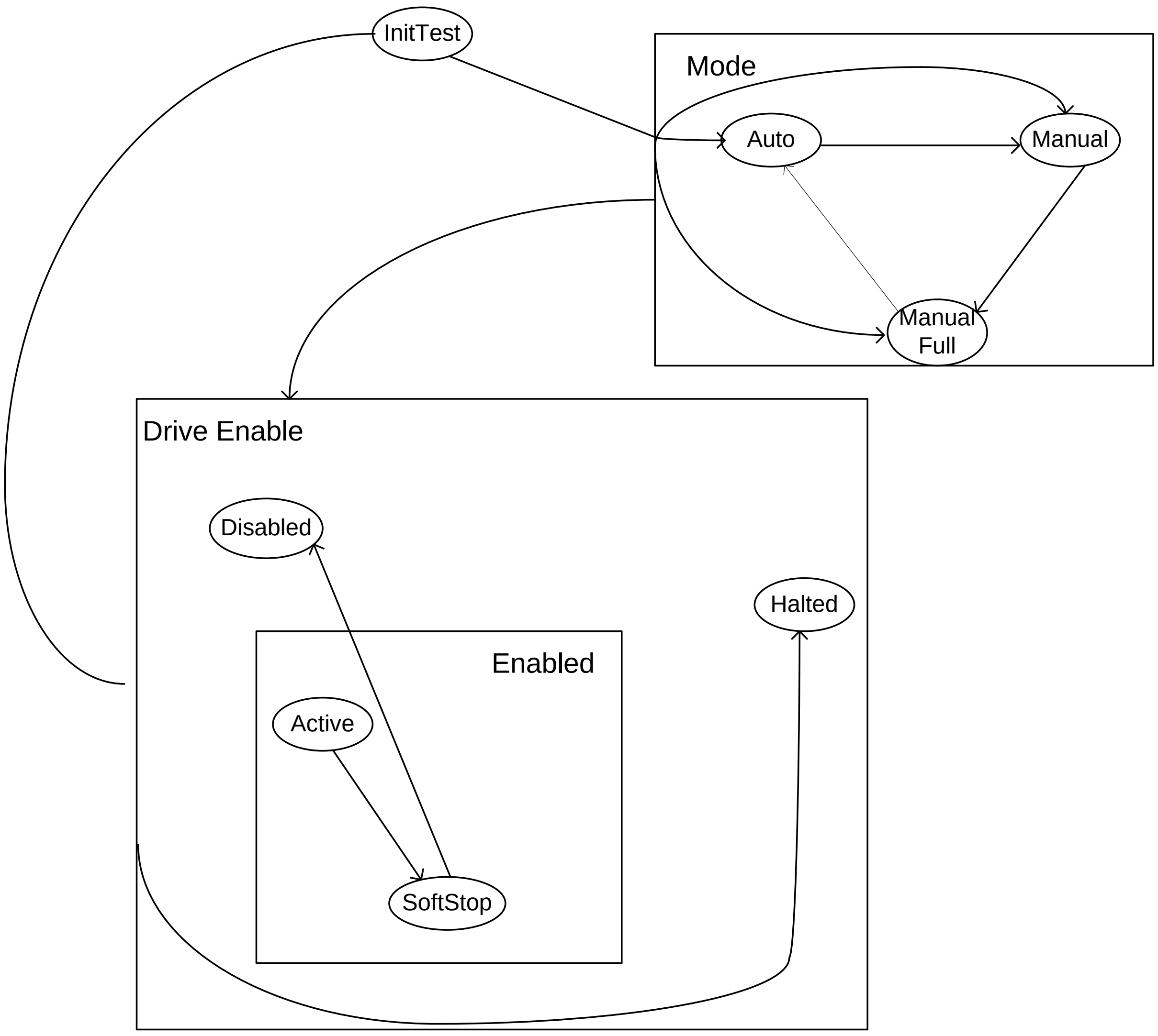

Figure 8: State-chart showing partial depiction of Robot Safety Functional Block

7. Discussion

As described in the introduction of this thesis, my research objective was to explore model-driven software development, especially conduct evaluation of statechart-based UML modelling in software development. In this chapter, I will first discuss using mutation analysis to detect faults, especially its relevance in a software development context as the one at ABB.

7.1 Using Mutation Analysis to Detect Faults

In the standard IEEE terminology, a *failure* is an external and incorrect behaviour of a program, or an incorrect output or a runtime failure. A fault is a group of incorrect statements in the program that causes a failure. Failures in the software are detected when test cases are executed against the original program.

For the test types that were planned in this thesis, we needed to decide whether the output of the program of each of the test case is correct. If the output is correct, the process continues as described in Section 5.3. If, however, the output is incorrect, then a failure has been found and the process stops until the associated fault can be corrected.

One major hindering factor in practical use of mutation analysis is the unacceptable computational expense of generating and running vast numbers of mutant programs against the test cases. The number of mutants generated for a software unit is proportional to the product of the number of data references and the number of data objects [22]. Typically, this is a large number for even small software units. Because each mutant program must be executed against at least one, and potentially many, test cases, mutation analysis requires large amounts of computation. This is shown in Figure 5 in the box labelled "Run test cases on each live mutant".

There are approaches designed to the computational expense mentioned earlier, and they usually follow one of three strategies: *do fewer*, *do smarter*, or *do faster*. The "do fewer" approaches seek ways of running fewer mutant programs without incurring intolerable information loss. This strategy is the one chosen for the tests conducted for this thesis. The "do smarter" approaches seek to distribute the computational expense over several machines or factor the expense executions by retaining state information between runs or seek to avoid complete execution. The "do faster" approaches focus on ways of generating and running each mutant program as quickly as possible.

7.2 Examining the Feasibility of Model-Based Tests

In this section I will address the research question of the thesis. This research question, which was presented in Chapter 1, tries to explore tests which are mainly generated from UML-models:

Research question: *How to evaluate model-based testing in the context of embedded and real-time systems.*

As described in [15] there is a novel collection of testing criteria for UML, which allows software-level tests to be derived from UML statechart diagrams. The paper also writes that UML statecharts provide a solid basis for test generation in a form of that can be easily manipulated. This technique includes coverage criteria that enable highly effective tests to be developed.

According to [15] there is another tool for automatic test data generation that has been developed. This tool, called UMLTest, is integrated in the Rational Rose tool – an object-oriented modelling tool – and is completely automated; this automatically generates tests from UML specifications. Another point which is cited in [1] is from empirical results from using UMLTest, which indicated that highly effective tests can be automatically generated for system level testing.

Model-based testing can be used to automate the process of designing test cases with details. It can also be used to generate test scripts that can execute the tests cases. In model-based testing, a test designer would be able to write an abstract model of the system under test, and then a model-based testing tool can be used to generate a set of test cases from that model.

There are clear advantages of model-based testing over baseline functional testing. The issues the model-based testing can be used to solve include: automation of the design of functional test cases, including generation of the expected results, in order to reduce the design cost and to produce test suites with systematic coverage of the model. It is also used to reduce the maintenance costs of the test suite.

Model-based testing can be used to leverage models in order to support the analysis of requirement defect. Model checking can ensure that properties, like consistency property, are not violated. The models in model-based tests can help refine unclear and poorly defined requirements. When the models are refined, tests can be generated verifying the system under test. Eliminating model defects before coding begins, automating the design of tests, and generating the test drivers or scripts can then result in a more efficient process, significant cost savings, and higher quality code.

It is evident that the model-based testing approach has advantages. These include the following:

- All models can use the same test driver schema to produce test script for the requirements captured in each model.

- When system functionality changes or evolves, the logic in the models change, and all related test are regenerated using the existing test driver schema.

- If the test environment changes, only the test driver schema needs modification. The test drivers associated with each model can be regenerated without any changes to the model.

In model-based testing approach, a test automation method can be used to traverse the logical paths through the program, determining the locations of boundaries and identifying reachability problems, where a particular thread through a model may not be achievable in the program,

Model-based testing can support requirement testability analysis, which can allow developers to iteratively refine and clarify models until they are free from defects. Defect discovery using model-based test automation is effective and more efficient than using only manual methods often used in the baseline testing.

In manual test generation practices, defects are not identified until late in the process, sometimes after the release, when they are most expensive to fix. On the other hand, automating test generation based on models and used in mode-based testing, defects are found earlier in the process and faster. Many defects are found in the requirement phase, before they propagate to later development phases. Defect prevention is most effective during requirement phase when it costs two orders of magnitude less than the coding process.

8. Conclusion

In this chapter I will summarize my research, and attempt to draw conclusion

Unified Modelling Language (UML) is a general-purpose visual language for systems. UML is not only associated with modelling Object Oriented Software, but it has a much wider application due to its inbuilt extensibility as well.

Software testing is an empirical investigation which is conducted to provide stakeholders with information about the quality of the product or service under test. Software testing methods are traditionally divided into two methods: a black box testing method and a white box testing method. In a black box testing, the testing treats the software a black box without any knowledge of internal implementation. In contrast, white box testing is when the tester has access to the internal data structures and algorithms and the code that implement these.

Mutation analysis introduces faults into software by creating many versions of the software, each containing on fault. Test cases are then used to execute these faulty programs with the gaol of distinguishing the faulty programs from the original program. These faulty versions of the original programs are called *mutants* of the original, and a mutant is *killed* by distinguishing the output of the mutant from that of the original program.

In this thesis we have described how to evaluate model-based testing. We have also described two alternative methods for software development: a function-based programming method and model-based development method, where code is generated from models. In order to make the evaluation of the model-based testing, we described how both methods of development could be used to develop the same module. The evaluation was conducted with respect to testability. This was conducted in the context of ABB.

For our evaluation of the two software development methods, we presented two testing plans, one for code developed in a functional programming method, and another for code developed on model-based development method where models are essentially the main artifacts. We also described mutation analysis and the selection criteria for mutation operators to be used in the mutation analysis. For the two testing plans, we presented test strategies for two mutation analyses.

My research into model-driven software development and model-based testing, where two methods of development were described and evaluated, has been rather broad. In order to investigate how effectively tests can be generated from models, there is need for more focused research, which can validate such an evaluation.

9. References

[1] Hans-Erik E., Magnus P., *UML Toolkit*
Wiley, 1998

[2] Jim A., Ila N., *UML 2 AND THE UNIFIED PROCESS*
Practical Object-Oriented Analysis and Design Second Edition
Pearson 2006

[3] Miro S., *Practical Statecharts in C/C++ Quantum Programming for Embedded Systems* CMP Books, 2002

[4] Rambaugh, J., I. Jacobsen, and G. Booch, *The Unified Modelling Language Reference Manual, Second Edition*
Addison Wisley, 2005

[5] Anda, B., Hansen, K., Gullesen, I., Thorsen, H. K., *Experiences from Introducing UML-Based Development in a Large Safety-Critical Project*
Springer Science 2006, pp 558 -576

[6] Anda, B., Hansen, K. *A Case Study on the Application of UML in Legacy Development*
2006 ACM

[7] Abdurazak, A., Offut, J., Baldini, A., *A Controlled Experimental Evaluation of Test Cases Generated from UML Diagrams*
George Mason University, Politecnico di Torino

[8] Dobing, B., Parsons, J. *How UML is Used*
May 2006, Communications of the ACM, pp 109, 110

[9] Utting, M., Legeard: *Practical Model-Based Testing*
Morgan Kaufmann, 2006

[10] Wikipedia 2008
http://en.wikipedia.org/wiki/Asea_Brown_Boveri
About ABB

[11] Baskerville, Richard L. (1999). *Investigating information systems with action research* (http://www.cis.gsu.edu/~rbaskerv/CAIS_2_19/CAIS_2_19.html).

[12] Herr, Kathryn and G. L. Anderson (2005). *The Action Research Disserta-tion: A Guide for Students and Faculty*. Sage Publication

[13] Hughes, J., *Why Functional Programming Matters*
Chalmers Tekniska Høgskole, Gøtenberg 1984

[15] Cruz-Lemus, J., Genero, M., M., Esperanza, Piattini, M. *Evaluating the Effects of Composite States on the Understandability of UML State-chart Diagram*
2005, Spring-Verlag Berlin Heidelberg

[16] Utting, M., Legeard, B. *Practical Model-Based Testing*
2007, Morgan Kaufmann

[17] Conford T., Smithson S. *Project Research in Information Systems – A student's guide*
MacMillan Press, 1996

[18] Yin, R. K., *Case Study Research Design and Methods*
2003, Sage Publications

[19] Holt, N., *A Systematic Review of Case Studies in Software Engineering*
2006, University of Oslo. p 18

[20] Øverland, L. *Global Software Development and Local Capacity Building: A Means for Improving Sustainability in Information Systems Implementations*
2006, University of Oslo. p.37

[21] El-Far, I. K. Whittaker. J., *Model-Based Software Testing*
2001, Florida Institute of Technology

[22] Offutt, A. J., Untch, R. H., *Mutation 2000: Uniting the Orthogonal*
George Mason University, Department of Computer Science at Middle Tennessee University

[23] Black, E. P., Okun, V., Yesha, Y., *Mutation Operators for Specifications*
National Institute of Standards and Technology, and University of Maryland at Baltimore

[24] Wikipedia 2009
http://en.wikipedia.org/wiki/Software_testing

[25] IEEE Standard for Test Documentation
Std 829-1998

[26] Swebok, *Guide to the Software Engineering Body of Knowledge*
IEEE, 2004 Version

[27] Greenwood, D., Levin, M., *Introduction to Action Research: Social Research for Social Change*
1998 Sage Publications

[28] Davidson, R., Martinsons, M., & Kock, N. *Principles of Canonical Action Research*
2004, Information Systems Journal 14(1), pp 65-86

[29] Ibrahim K. E., James, A. W., *Model-Based Software Testing*
2001, Florida Institute of Technology

[30] Choi, T.S. *Testing Software Design Modeled by Finite-State Machines.*
IEEE Transactions on Software Engineering, SE(4(3)), pp. 178-187, 1978