MODEL-BASED TESTING OF INDUSTRIAL PROTOCOLS TOWARD THE IRC5P CONTROLLER

Universitetet i Stavanger

SVEN GETAZ - 219061

Abstract

This thesis constitutes a partial requirement for the Master of Science degree in Industrial Automation and Signal Processing at the University of Stavanger. The assignment was undertaken during the Spring semester of 2014, and in cooperation with ABB Robotics where I am employed as a Research and Development Engineer in the Embedded Software department.

In summary, the project constituted of attempting to develop an automatic test generation system for the PLC interface toward the Paint Robots that ABB Robotics develop in Bryne. The PLC interface is an important method of control for the robots, and it is vital that as many bugs as possible are uncovered and removed during the development process. Model-based testing was chosen as a method to generate the test cases, as the methods of Model-Based testing have achieved some success in being a state-of-the-art way of developing a vast amount of test-cases with very little code. The modelling tool utilized in this thesis is Spec Explorer, which is a model-based testing tool developed by Microsoft. Based on the language Spec#, a derivative of C#, the tool is intuitive and relatively simple to use.

A simplified model of the System-Under-Test (SUT) was created in Spec Explorer, defined by rules and actions that attempt to mirror the behaviour of the controller to various inputs. Next, an implementation layer was added to directly translate the action calls invoked by the Spec Explorer model, into executable commands to the PLC via an OPC server/client relationship. The inputs from the PLC were then iterated through the controller, which outputs a return code. The return codes from the controller were compared to the expected return code from the model, whereby Spec Explorer would declare the test a success or failure based on whether the result conformed to the expected output.

After succesfully implementing a Model-Based Testing system for the PLC interface, I wondered whether it would be possible to utlize the same model to test an entirely different protocol by simply writing a new adapter layer. Thus I decided develop a Model-Based Testing system for the Robot Web Service interface, which is a feature of the new Robotware 6 software currently in development. As this is new territory, the possibilities of finding bugs in the software was relatively high. This model differed slightly from the previous model, as I also implemented a form of "state-checking" in order to compare a subset of the state between the model and the SUT.

The result of this thesis were two Model-Based Testing systems for both the PLC interface and the Robot Web Service interface toward the robot controller, which allow for automated test case generation for both of these interfaces. Initial testing of the Robot Web Service uncovered many faults, mainly due to differences in the two version of Robotware. After modifying the model in accordance with the differences in versions, several other faults were uncovered due to discrepancies in the return codes between the versions. After having implemented the state-checker, one major bug in the SUT was discovered, and relayed to the Robotics Headquarters in Sweden to patch.

Although the majority of the faults uncovered were due to model faults, I would still claim this study to be a success. Model faults are both easily uncovered and easily rectified and once the model is primed to perfection, it is my opinion that this method will prove to be very useful in uncovering software bugs. This was demonstrated when the bug in the SUT was uncovered in the later stages of testing, once the model had been primed. Model-based testing also proved to be highly customizable, in the sense that by changing the adapter code, one can use the same model to generate test-cases in an entirely different test-environment.

Contents

List of Figures

Listings

Chapter 1

Introduction:

1.1 Motivation:

"It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise - this thing gives out and [it is] then that "Bugs", as such little faults and difficulties are called, show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached."
-Thomas Edison

Software engineering is a tricky business, as it proves to be very easy to get caught up in the details and lose your way in the lines of code. The term "bug" was first coined by Grace Hopper in 1946, when she publicized the cause of a malfunction in an early electromechanical computer.[1] The error was traced back to a moth caught in one of the relays, which gave name to the term "software bug". Generally defined as an inexplicable error or glitch, bugs are a common headache in the field of software engineering. An indication of this is that historically, one metric to evaluate software performance was bugs per line of software code.[2]

This issue is of pivotal importance to the software industry, as software bugs and errors are so prevalent and detrimental that they constitute an estimated annual cost to the US economy in the order of $ 59 billion.[3] Additionally, software testing often consumes between 30 and 60 percent of the overall development effort.[8] Although many software companies already incorporate various automated testing tools, model-based testing pushes the envelope even further by also automating the design of the test cases. A definition of software testing follows, with some important words described below: *Software testing consists of the dynamic verification of the behavior of a program on a finite set of test-cases, suitably selected from the usually infinite executions domain, against the unexpected behavior.*[9]

Dynamics: Specifies that we excite the system with specified input values for the purpose of observing failures in the system. To contrast this, static techniques do not require execution of the program, but base the analysis on walkthroughs, inspections and static analysis tools. One big advantage of dynamic testing, is that the program is executed in a real or simulated environment. By doing this, not only is the design and code being tested, but also the compiler, the libraries, the operating system and network support.

Finite: Exhaustive testing is rarely possible in practice, as the large number of possible inputs coupled with unexpected or erroneous inputs exponentially increase the state-space of the resulting exploration graph. If one takes into account the possible sequence of inputs, the sample space is virtually infinite. Thus, a small sample of test cases must be extracted from the system such that the tests can be executed in a reasonable amount of time.

Selected: Since the set of possible tests is so large, often infinite, the key challenge in software testing is selecting the test-cases that are most likely to expose failures in the system. Different modelling tools often incorporate various test selection algorithms which attempt to automate this challenge.

Expected: After the execution of each test case, one must determine whether the observed behaviour of the system was a failure or not. This is commonly referred to as the oracle problem. The oracle problem is generally solved via manual inspection, but Model-Based testing automates this process, offering efficient and repeatable testing.

Now place yourself, for a second, in the position of a manager in charge of software validation at a large software corporation. When choosing methods of testing the new release, you have the following options: (A)Employ a number of full-time testers to manually design the tests, record the test results and then manually perform the tests each time changes to the system are implemented. (B) Manually design test-cases based on your knowledge of the system in question, use automated test execution tools to run these tests and then rerun them again after every change. (C) Utilize state-of-the-art tools to automatically generate test cases from an abstract model of your system, regenerate updated test-cases each time the system changes, and can report exactly which test-cases failed and why.

Based on several reasons, including economy and efficiency, one could argue that the latter solution is generally the optimal solution. This is one of the main reasons why Model-Based testing is gaining increased recognition in both academia and the industry as a state-of-the-art method of software testing.

1.2 Problem Description:

The industrial robots developed by ABB are used in a variety of processes, including packing, painting, and assembling to name but a few. As an example, a paint robot usually operates in a robot-cell with a conveyor belt, with objects continuously travelling by. A position sensor keeps track of where each work object is on the conveyor belt, indicating whether the work object is ready to be painted. Thus, an external controller is needed in order to provide IO feedback to the robot controller. Different control interfaces are used to perform this task, with the PLC interface being one of the more important interfaces. The PLC interface can be configured according to a several IO protocols, all of which will be discussed in detail in another chapter. Many customers use the PLC interface in a variety of ways, and as such it is important that the operation is robust and efficient.

Today in ABB Robotics, test cases for the PLC interface are defined "the old fashioned way", by manually deciding the most important cases to test and running them through automatic execution. As a method for improving quality on the products, Model-Based testing was suggested as a possible way of making this process more efficient, and seeing whether it was possible to develop Model-Based test case generation for a PLC system. The main challenges involved in this process were first, to develop an abstract, accurate, yet simplified model of the controller PLC interface, within the constraints of Spec

Explorer. Spec Explorer was chosen after some preliminary research had shown it to be rather versatile and intuitive, in addition to being highly customizable.

The Robot Web Service interface to the robot controller is a new feature which is being introduced in Robotware 6, which is currently still in development. It allows for many of the same control features as the PLC interface does, and is based on the RESTful architecture which allows for simple and user-friendly interaction between the client and the controller. As this is a new feature, there is a very high potential for bugs and faults within the underlying structure of this service which ABB hope to uncover by introducing Model-Based Testing as a method of testing.

There are many challenges that need to be addressed with regards to this project. The first task was to develop models that mirrored the behaviour of the respective Systems-Under-Test, which in this case are the PLC interface toward Robotware 5 and the RESTful Web Server interface toward Robotware 6. Although in theory one could use the same model without changes if the logic of the two interfaces where identical, some changes had to be implemented in the different models to account for subtle differences between the two versions of the controller software. This process is documented in chapter 3, "Modelling in Spec Explorer".

Secondly, there was a challenge involved with the processing of the fieldbus IO from the controller into the PLC, and creating a program within the PLC that implements the test execution via Spec Explorer. The way this challenge was tackled is discussed in Chapter 4, "Testing", under the section PLC interface.

The third task was arguably the most challenging, and consisted of implementing layers within Spec Explorer that translated the actions called by the model into executable methods toward the robot-control system. For the PLC interface, this was solved by defining an OPC server in CoDeSys, where relevant global variables incorporated by the PLC were mapped. An OPC client was then created in C#, where upon each action call, the implementation layer wrote the input parameters defined in Spec Explorer directly to the PLC. A byte variable specifying the return code from the controller was then read by Spec Explorer, and compared against the expected SUT behaviour defined by the model to test for conformance.

For the Robot Web Service interface, this was solved by creating a REST client in the adapter layer, which uses the HTTP verbs to PUT and GET commands and their results through the HTTP protocol. All the responses from the Robot Web Service arrive as XML code, which required learning the basic workings of XML and parsing the responses into manipulable code. Also, the controller implements Digest Authentication, which is a method of encryption involving hashing the user-name and password with an MD5 algorithm each session. This had to be incorporated into the REST client in order to start sending and receiving commands. This process is described in Chapter 3.

Once the testing system was up and running, a new challenge presented itself in the form of trying to gain a form of control over the state-space of the resulting exploration graph. Initial exploration of the models revealed a vast state-space that was too vast to cover fully, resulting in the failure to generate test-cases that covered all the possible transitions of the model. This was solved by applying a filter upon the states which the model was "allowed" to be in, in order to gain a method of control over the sequence of executable actions. Another method of control which proved successful was to integrate the Connect() and Disconnect() methods into the model, rather than have them as "hidden" members within the adapter layer. Both the OPC client and the REST client need these methods to initialize certain parameters and establish a connection to the servers, but they are not valid actions per say. However, by "forcing" them to be actions within the model, and assigning a guard to each action requiring a boolean variable representing Connected to be true before allowing the transition to take place, the exploration space of the model now had a natural order in the sequence execution. This will be described in Chapter 4.

Finally, the final challenge involved attempting to automate the process of testing whenever a new build of Robotware was issued from the Robotics Headquarters. This required establishing a connection to the buildserver, subscribing to an event Build-Complete which indicates that the new build is ready to be released, and creating an eventhandler that activates the test upon the BuildComplete event.

1.3 Related Work:

Within academia, there has been a fair amount of research performed on model-based testing and conformance testing. The paper "Model-Based Testing with Labelled Transition

Systems" by J.Tretmans,[19] gives a comprehensive introduction to model-based testing, and provides a more detailed and theoretical approach to the principles behind MBT and Input Output Conformance Testing (IOCO) than will be given in this thesis, which will give a more practical and qualitative theory. However, research toward the industrial application of MBT is not as prolific. One paper of interest was the Doctoral Thesis of Nina Holt[20], who researched the cost-effectiveness of a subset of MBT techniques. This research was in fact performed at the Swedish and Chinese offices of ABB Robotics.

Research related to the application of MBT toward PLC systems was very difficult to come by, although a few research papers were uncovered during a web-search. A thesis by T.Hoeve at the University of Twente[21], used the IOCO principles of MBT to model and generate test cases for a PLC Interlocking System for the Dutch Railway. However, an actual PLC was not used in the thesis as demonstrated by the following quote: "Using a PLC-Interlocking as the SUT is problematic because such a system is very expensive and hard to interface with. Therefore an executable has been created that can be run on a regular PC, which embeds the interlocking logic of a PLC-Interlocking, and adds an interface for JTorX to interact with this logic.".

The Microsoft Research Team have written several papers regarding the theory and principles behind Spec Explorer as a tool for MBT. A paper by a Microsoft Research Team entitled Model-Based Testing of Object-Oriented Reactive Systems [22], is of particular interest as it gives a very theoretically grounded introduction to Spec Explorer along with mathematical definitions of states and the exploration algorithm.

Preliminary research has also been done on Model-Based Testing as applied to RESTful Web Services, as demonstrated by a paper entitled Model-Based Testing of RESTful Web Services Using UML [23], where a UML State Machine model was used to generate test cases for a RESTful web service. In contrast, this thesis will use IOCO principles to generate test cases.

In summary, there is a fair amount of previous research related to Model-Based Testing. However, the testing systems developed in this thesis are unique in the sense that:

- An adapter-layer was created such that the model interfaced with an actual PLC, which was connected up to the SUT.

- IOCO theory was used to generate test-cases for the RESTful Web Service rather than UML.
- This is the first research, to the best of my knowledge, that has been done on Model-Based Testing as applied to an IO-interface of an industrial robot.
- No previous research has been done on trying to integrate Spec Explorer models and test-execution into the software lifecycle, which is a necessity in order for Model-Based Testing systems using Spec Explorer to be successful in industrial environments.

Chapter 2

Theoretical Foundations:

"In theory there is no difference between theory and practice. In practice there is."
Yogi Berra

2.1 Model-Based Testing:

In this section, I will attempt to explain and summarize Model-Based testing as concisely and coherently as possible, using the textbook Practical Model-Based Testing [8] as a reference.

Model-Based testing can generally be divided into four main approaches:

1. Generation of test input data from a domain model: The generation of test input data based on the information about the domains of the input values, where the test generation involves a careful selection and combination of a subset of those values to produce test input data. Being able to automatically generate test inputs is an important feature of this approach, but one is unable to verify whether the test has passed or failed, as no test oracle is created.

2. Generation of test cases from an environment model: This approach is based upon a model which describes the expected environment of the SUT. An example of this could be a statistical model of the expected usage of the SUT.[10] From such a model, automatic

generation of test-cases is still possible. However, as the model does not conform to the expected behaviour of the SUT, it is impossible to predict the output values and thus this approach cannot determine whether a test passes or fails.

3. Generation of test cases with oracles from a behaviour model: Refers to the generation of executable test cases that include a test oracle, where a test oracle can either be expected output values attained by the model or an automated check on the actual output values. In other words, a form of verification of the test-cases are implemented which allows for automatic determination of whether a set of tests pass or fail. However, in order to generate oracles, the model must be of sufficient accuracy to predict the behaviour of the SUT.

4. Generation of test scripts from abstract test cases: A different approach entirely, as it is based on the assumption that we are given an abstract description of a test case, e.g a UML seguence diagram, and proceeds to transform the abstract case into low-level executable test script.

The third approach was the option that was explored in this thesis, for the simple reason that it is the only approach that encompassed the whole test design problem. In essence, it is the "automation of the design of the black-box tests".

The model is thus the heart and soul to each of these approaches to model-based testing, perhaps specifically in the third approach. Below are a couple of definitions of the word model, taken from the American Heritage Dictionary, which emphasize the two most important properties of a good model: [11]

- A small object, usually built to scale, that represents in detail another, often larger object.
- A schematic description of a system, theory, or phenomenon that accounts for its known or inferred properties and may be used for further study of its characteristics.

In other words, the model must be small in relation to the system under test, such that the costs to develop and maintain the model remain low. Yet, it must also be detailed enough such that it accurately reflects the behavioural characteristics of the SUT. Thus, an engineering challenge arises, where one must attempt to gauge which characteristics are required in the model and how much detail is sufficient.

Now that the four main approaches to model-based testing have been mentioned, a step-by-step description of the third approach will follow in an attempt to further familiarize the reader to the process of model-based testing. This process is depicted graphically in figure 2-1. The natural first step to the model-based testing process is to

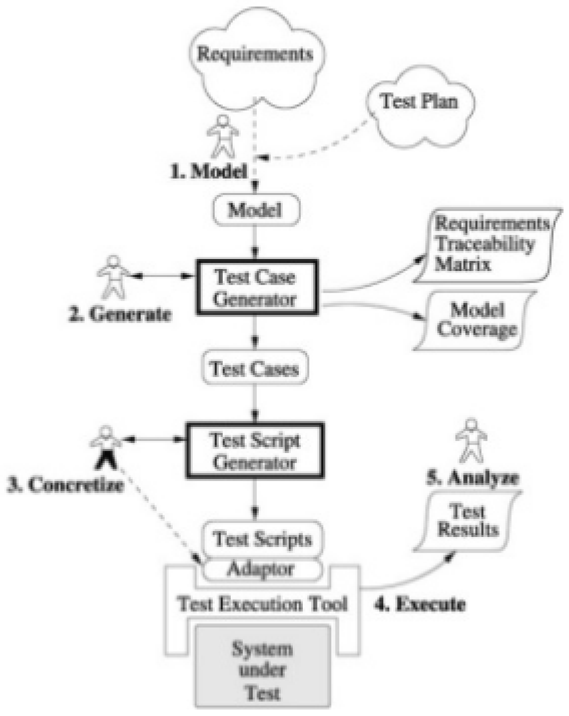


Figure 2.1: The model-based testing process:

develop and write an abstract model of the system that we wish to conduct tests on. We have already briefly mentioned some of the requirements for a good model, and that sometimes these can seem to contradict each other. A general rule when writing the model is: "When in doubt, leave it out", as the model should generally be as simple as possible. Some typical simplification measures used in modelling are as follows: [10]

- Focus primarily on the SUT
- Show only those classes (or subsystems) associated with the SUT and whose values will be needed in the test data

- Include only those operations that you wish to test
- Include only the data fields that are useful for modeling the behaviour of the operations that will be tested
- Replace a complex data field, or a class, by a simple enumeration. This allows you to limit the test data to several carefully chosen example values(one for each value of the enumeration)

One must then determine which notation to use for the model, a decision which is influenced by the modeling tool utilized and the orientation of your system. There are many different notations used in modeling, but for model-based testing purposes, two notations are the most useful: [10] transition-based notation and the pre/post notation. Pre/post notation models a system as a collection of variables, with combinations of these variables serving as the different states the system can fall into. Transition-based modeling focus on describing the transitions between independent states, typically using FSM (Finite State Machines) as a graphical node-and-arc notation. The nodes of the FSM represent the major states of the system, while the arcs represent the actions that map between different states. Spec Explorer actually utilizes a combination of these two forms of notation, which will be discussed later.

The second step in the model-based testing process is to generate abstract test cases from the model. As the model-space is usually infinite, different algorithms and techniques must be implemented in order to generate meaningful test cases. For instance, one can choose to focus on a subset of the entire state-space, choose a model coverage criterion or use algorithms from the mathematical field of combinatorics to simplify the state-space. The output of this step is a test-suite (a collection of test-cases) of abstract tests, which due to the lack of detail employed by the model, are not directly executable. Most tools that are used for model-based testing purposes, include a functionality that allows you to produce a requirement traceability report or a variation of coverage reports. The purpose of these reports are to give you an indication of how well the generated test-suite exercises all the behaviours of the model, for instance what percentage of boolean decisions were performed, coverage statistics for the transitions in the exploration model etc.

As mentioned previously, the model is an abstract representation of the SUT only, and indicates which actions are available for each state given the constraints defined by the model. The third step thus consists of transforming abstract function calls invoked by the model, into executable commands or functions that can be implemented in the SUT. This is generally done by writing an adaptor code that wraps around the SUT and implements each abstract operation in terms of the lower-level SUT details, details which were omitted from the model. One major advantage of this layering approach, abstract and concrete test scripts), is that the abstract test-cases are in general quite independent from the language used to write the tests and the test environment. Thus, one can by modifying or changing the adapter layer, reuse the model to generate test-cases on an entirely different execution environment)

The fourth step consists of executing the concrete tests on the system under test. The test execution can be performed online of offline. Spec Explorer uses an algorithm called On-The-Fly (OTF) testing to execute the generated test-cases online. Online testing is a technique in which each test case is executed as they are generated, where the Model-Based Testing (MBT) tool manages the test execution and the processing of the results. Offline testing is a process in which the model is explored, and test sequences are generated as the exploration graph is traversed from an initial state to an accepting end state. The full collection of test cases is called the test-suite. The MBT tool can generate executable test code from a test suite, which can be executed at any given time. Both of these options were explored in this thesis.

The fifth and final step is to analyze the results of the test executions and employ corrective measures. An investigation is to be performed on the event of a failed test-case, which is a similar procedure to traditional test analysis. Generally, the failures are caused by one of two things, either a fault in the SUT or a fault in the test-case. In model-based testing, a fault in the test-case implies a fault in either the model or the implementation of the model (the adapter layer). Faults can also be caused by errors and misunderstandings regarding the requirement documentation used to develop the model. Typically there are many failures during the initial execution of a test, which are mainly caused by minor errors in the adaptor code and faults in the model. Once the obvious errors are accounted for, more interesting failures occur which require deeper analysis. The more failures that are discovered the better, as the most common failures will be due to model-faults. Rectifying these faults will lead to better models, which will lead to more interesting failures and so on, so it is a highly iterative process.

2.1.1 Input Output Conformance Testing:

Input Output Conformance Testing (ioco)is a particular model based testing theory, which will be the method used in this thesis. A short presentation of **ioco**-theory will follow in this section, presented as a summary of the paper entitled Model-Based Testing with Labeled Transition Systems. [19]. IOCO-theory bases the model, implementation and test generation upon labelled transition systems and uses a formal implementation relation called ioco to define conformance between the SUT and the model specification. Additionally, a particular algorithm is used to generate the test cases, for which there is a completeness theorem (soundness and exhaustiveness).

2.1.2 Formal Testing:

Formal, specification based testing uses different concepts and objects to express the ioco relationship between the model and the SUT. These concepts are defined below.

- Implementation: The implementation, or the SUT, is the system being tested which in this case is a particular subset of the industrial robot controller IRC5P. Generally, an implementation can be a real, physical object (such as a piece of hardware), a software program, an embedded system with software embedded in some physical hardware, or a process control system with sensors and actuators. Since model-based testing generally deals with black-box testing, the SUT is treated as a black-box that exhibits a certain behaviour when excited by inputs, but without knowledge of the internal structure of the system. The aim of the test-process is to verify the *correctness* of the behaviour of the SUT through its interfaces.

- Specification: The *correctness* of the SUT is expressed as a conformance relationship toward the model specification. The model expresses how the implementation should generally behave, and the tests must verify whether this relationship holds for a given set of test-cases. Specifically, the specification is expressed in some language with a formal syntax and semantics. Let the language, and the set of all valid expressions within the language, be denoted by SPEC. A specification *s* is thus an element of this language, represented by s $\in$ SPEC. We wish to verify whether the behaviour of the SUT conforms to s.

- Conformance: In order to verify whether a given SUT conforms to a certain specification s, we need to define what it means for a SUT to conform to s. Thus, a formal definition is required, but this is generally not possible. This is because whereas s is a formal object taken from a formal domain SPEC, a SUT is not a formal object but rather an actual, physical device. Thus, we make the assumption that any implementation can be modelled by some formal object i_{SUT} in a set of models MOD. This assumption is referred to as the *test assumption*, and allows reasoning about the implementations as if they were formal implementations in MOD. An implementation relation, is a relation expressed between the models of implementations and specifications, and is defined by: $\mathbf{imp} \subseteq MOD \times SPEC$

- Testing: As mentioned earlier, the behaviour of a black-box implementation is investigated by performing experiments on it, consisting of supplying stimuli to the implementation and observing its response. Such an experiment, including both the stimuli and the expected response, is called a *test case*, which is expressed as a subset of some set of test cases TEST. Performing such an experiment on an implementation is called *test execution.* The outcome of a test execution can be either successful, if the observed responses correspond to the expected responses, or unsuccessful.

- Conformance testing: Conformance testing thus involves assessing, by means of testing, whether a given implementation conforms to its specification as given by the model. More specifically, the objective is looking for a test suite T_s such that:

  $\forall i \in MOD : i\mathbf{imp}s \longleftrightarrow i\mathbf{passes}T_s$

  A test suite for which this property holds is called complete, and it is a rather stringent requirement for practical testing. In practice, complete test suites for a SUT are often infinite and thus not practically executable. Hence, a weaker requirement for practical test suites is introduced, requiring the test suites to be sound. A sound test suite means all correct implementations, and possibly some incorrect implementations, will pass them. In other words, any failing implementation is non-conforming but **not the other way round**.

- Test generation: The algorithmic generation of test suites from a specification for a given implementation relation is referred to as test generation, and is given by the following definition: $SPEC \rightarrow P(TEST)$, where P(TEST) denotes the set of all subsets of TEST. Such an algorithm is complete (sound and exhaustive) if the generated test suites are complete(sound and exhaustive) for all specifications.

Test generation is one of the main attractors toward model based testing, as it allows the automatic generation of large and demonstrably sound test suites.

- Conclusion: For model based testing using the ioco-theory, we thus need a formal specification language SPEC, a domain of models of implementations MOD, an implementation relation **imp** $\subseteq MOD \times SPEC$ expressing correctness, a test execution procedure **passes** $\subseteq MOD \times TEST$ expressing when a model of an implementation passes a test case, a test generation algorithm $SPEC \to P(TEST)$, and a proof that a model of an implementation passes a generated test suite.

2.2 Paint Protocols:

A brief mention of the PLC interface was included in the motivation section, however, a detailed description of the functionality and architecture of the PLC interface follows, where the theory will be a summary of the PLC Paint Interface manual. [12] The robot controller and the process control system have a set of states, which are given by internal value tags(variables) as well as discrete outputs. The observant reader might then recognize from the previous section, that these kinds of systems are well suited for the pre/post-notation of modeling. These states are designed to enable the robot to perform motion regardless of material change, applicator or color supply type. The robot is controlled through paint commands and input-parameters, which are hidden to the user via graphical user interfaces, but exposed through an I/O protocol. An architectural diagram is shown in 2.2. All versions of Robotware (which is the embedded controller software for the robot) include the job queue function, which is also exposed on all of the PLC interface protocols. The job queue is basically a list of paint programs that are to performed. The main task of the controller will constantly monitor the job queue when not running a paint program, and immediately pop the next job as soon as it is available. "Popping" in this sense, refers to taking the the job at the top of the queue and remove it for processing. It is possible to bypass the job queue, for instance by marking jobs as high-priority or inserting a job at the top of the queue. Several paint commands, such as Execute Program, gives the user the option of running a program before the rest of the queue is serviced.

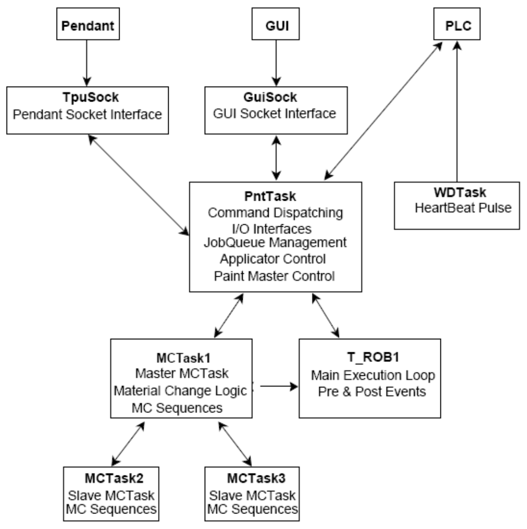
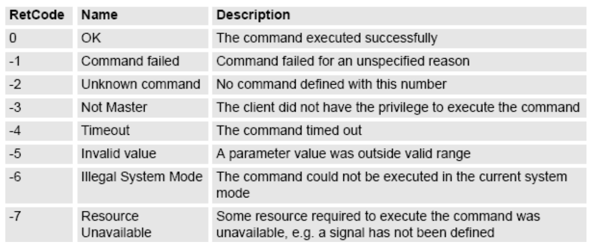


Figure 2.2: Interactions between Rapid tasks and external devices:

2.2.1 Paint Commands:

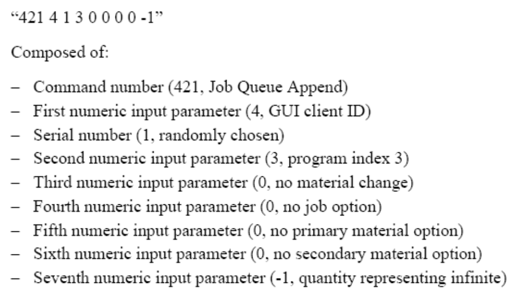
A paint command is a parameterized command which can be sent to the controller from many different clients, for instance a PLC, PC applications and the Teach Pendant Unit (TPU - A graphical unit interface). Each paint command consists of a command number, along with possible input parameters required by the command and a set of corresponding output parameters. A return code is returned by the controller after successfully receiving and executing a command. If this code is non-zero, the error log of the robot controller will contain details on the cause of the error. All the possible return codes are shown in Figure 2-3. On the PLC interface, the value of these return-codes are positive rather than negative. There is a safety feature implemented on the controller,



RetCode	Name	Description
0	OK	The command executed successfully
-1	Command failed	Command failed for an unspecified reason
-2	Unknown command	No command defined with this number
-3	Not Master	The client did not have the privilege to execute the command
-4	Timeout	The command timed out
-5	Invalid value	A parameter value was outside valid range
-6	Illegal System Mode	The command could not be executed in the current system mode
-7	Resource Unavailable	Some resource required to execute the command was unavailable, e.g. a signal has not been defined

Figure 2.3: List of return codes with description:

referred to as the Master function. This feature ensures that only one client can execute certain commands at any given time, and some commands require that the initiator client has master status in order to process the command. These commands will be executable without being master if no other client is master, but blocked for other clients if a given client is master. An example of the structure of a paint command is shown in 2.4. A list of all paint-commands along with a description will be given in the appendix. Currently,


"421 4 1 3 0 0 0 0 -1"

Composed of:

- Command number (421, Job Queue Append)
- First numeric input parameter (4, GUI client ID)
- Serial number (1, randomly chosen)
- Second numeric input parameter (3, program index 3)
- Third numeric input parameter (0, no material change)
- Fourth numeric input parameter (0, no job option)
- Fifth numeric input parameter (0, no primary material option)
- Sixth numeric input parameter (0, no secondary material option)
- Seventh numeric input parameter (-1, quantity representing infinite)

Figure 2.4: Appending a job to the job queue with an infinite quantity:

there are five I/O protocols implemented in the robot controller, allowing us to send paint commands from external devices such as a PLC for instance. An I/O protocol is similar to a standardized configuration, such that both the sender and receiver know what to send and what to expect in response. A list of these protocols will follow, along with the descriptions of each protocol:

Discrete, Compact and Extended Discrete I/O Protocols: The discrete I/O protocl is used to control the robot and send program and material information. It uses 32-bits in and 32-bits out, and input signals are translated to command numbers. The compact I/O protocol is basically a reduced version of the discrete I/O protocol, using 16-bits in and 16-bits out. The extended I/O protocol is thus an extended version of the discrete I/O protocol, using a 16-bit input for indexes(program, material or option). In total, the extended I/O protocol uses 48-bits in and 48-bits out.

Command I/O Protocol and The Extended Command I/O Protocol: The command I/O protocol can execute any valid paint command in the robot controller. The protocol allows up to 10 numeric input parameters and 2 numeric output parameters, and uses 64 bits in and 64 bits out in total. In some of the paint commands, several of the parameters are optional. Take the example given in Figure 2-4, JobQueueAppend, where only the first parameter (program index) is necessary. Also, for PLCs, the client id parameter is not used, such that the parameter list is shifted one position. The extended command I/O protocol is similar to the regular command I/O protocol, but uses 16-bits for each parameter instead, as well as some additional status signals. In total, the extended command protocol uses 80-bits in and 80-bits, which makes it the largest protocol available. For this reason, the extended command protocol is the protocol used in this thesis as a basis for the modeling and testing. The entire Extended I/O Command Protocol is given in the PLC Manual.

2.3 PLC Theory:

PLCs (Programmable Logic Controllers), are computer systems which are commonly used to control and monitor industrial processes. They are highly customizable and stable, and can be tailored to virtually any situation through programming and configuration. They are the most predominant method of control in modern industrial plants, and are likely to remain in the forefront for the foreseeable future.

The control loop is a continuous cycle of the PLC scanning the available inputs, executing the logic in the form of user-created programs and then changing the outputs accordingly. A fourth step is also included in the operation, which involves internal diagnostics and interfacing with network terminals. Figure 3-14 gives a graphical depiction of this process. The time it takes for a PLC to execute one full cycle of operation, namely the four steps depicted in figure 3-14, is called a scan cycle. As the PLC powers up, it executes a *sanity check* to determine whether the hardware is functioning properly and will stop whether any hardware errors are uncovered.[28]

2.3.1 Programming:

One of the main benefits of PLCs is the fact that they are so customizable. Programs can be written in many different languages, and functions or modules written in entirely different languages can be combined effortlessly. Which language to use is highly situational, and largely down to personal preference and program task. There are five different programming languages for PLC as defined by the IEC 61131 international standard for programmable logic controllers:[30] Ladder logic, Stuctured Text, Function Block Diagram, Sequential Function Chart and Instruction list.

Ladder Logic:

Ladder logic evolved into a programming language representing a program by a set of graphical diagrams, which were based on circuit diagrams of relay logic hardware. [31] The name is based upon the fact that the programs written in this format usually resemble ladders, with two vertical rails and horizontal rungs between them. Is very useful for simple control systems, or upgrading outdated relay control systems. Initially, the motivation behind Ladder Logic was to allow technicians to develop software without a formal background or training in software programming. Ladder Logic is considered a *rule-based* language rather than a procedural-based language, with each *rung* in the ladder representing a rule.

Structured Text:

Structured Text is a text-based programming language used to program PLCs. It loosely follows the syntax of C or Basic, and as such facilitates the programming of PLCs by conventional software programmers. The language itself is composed of written statements separated by semicolons, and uses predefined statements and program subroutines to change variables. The types of variables can vary from explicitly defined values, internally stored variables or inputs and outputs. Structured text is not case sensitive, however, it can be useful to make variables lower case and make statements upper case.[32]

Function Block Diagram:

A program written in the FBD language is built up from blocks that take one or more inputs, and return one or more outputs. The function blocks can be implemented in such a way that parts of the outputs of one block can be used as inputs for the next block and so on. The blocks are standardized, but custom programs can be written in any of the five standardized languages and be made into a FBD, examples of which will be given in Chapter 4. Inputs and outputs of function blocks can be mapped to global variables. FBDs are executed from left-to-right and top-to-bottom. Textual representation is not required in FBDs, but rather by "drag and drop" of variables and connecting them up to the inputs and outputs of the graphical representation. [33]

Sequential Function Chart:

Sequential Function Chart is a graphical programming language used for PLCs. This method of programming is particularly useful for processes which can be split into distinct steps, and often used as the Main execution program of the PLC. As such, it is used as the Main execution program for the PLC used in this thesis as the testing phase is divide into distinct steps. The main components of SFCs are as follows:

- Steps with associated actions
- Transitions with associated logic
- Direct links between steps and transitions

Steps can either be active or inactive, and actions are only executed for active steps. A step can be activated for one of two reasons, it is either the initial step as specified by the developer, or it has been activated during a scan-cycle and not been deactivated since. Steps are activated when all steps above the current step is activated, and the transition criteria have been met. When a transition is passed, all steps above are deactivated at once and after all steps below are activated at once.[34] SFC is essentially a parallell language, and steps can have different branches according to the logic of the model. This method of branching is used extensively in the Main program in the PLC used for interacting with the Spec Explorer Model, as different branches account for if the paramToggles are set or not. If in an alternative branch than the "main" branch, the inLongTest boolean variable is set to true and not set to false until the SFC loops back to the initial state waiting for the next input from the model.

Instruction List:

IL is a low-level language used for PLC programming, and closely resembles the assembly language used for programming certain microcontrollers. The variables and function calls are defined by common elements such that different languages can be used in the same program. Program flow is controlled by *jump* statements, or through various subroutines with optional parameters. [35]. It is the only language of the five standardized languages not to be used in the PLC interfacing with the controller in this project.

2.3.2 Object Linking and Embedding for Process Control (OPC)

Object Linking and Embedding for Process Control, is the original name for a standardized specification used to communicate real-time data between different control devices from different manufacturers. OPC has since grown beyond its original OLE implementation, to also include data transportations technologies such as XML, the .NET framework and binary-encoded TCP format. The OPC specification was originally based on the OLE, COM and DCOM technologies developed by Microsoft for the Windows operating system, and defined a standard set of objects, interfaces and methods for use in process control automation applications to facilitate the integration of different forms of control devices. The most common OPC specifiaction is OPC Data Access, which is used to read and write real-time data.

The main strength of OPC servers, is providing a method for different software packages to access and manipulate data from a process control device such as a PLC.[36] To clarify, the OPC server is not a mere subprogram library, but rather an executable program that starts whenever a client/server connection is initiated. Thus, it is able to notify the client when the value or status of the variable changes. Due to the characteristics of DCOM, it is also possible to access OPC servers running on other computers, and furthermore a data source can simultaneously be accessed by multiple clients via OPC. Another advantage OPC gains by utilizing COM, is that different programming languages(C++, Visual Basic, Delphi, Java, C#) can be used to write OPC clients which communicate with the server. However, a major disadvantage is are the CPU resources required. [40]

2.4 Team Foundation Server:

The Team Foundation Server (TFS) is a Microsoft product for the management of source code, product management, automated builds and testing. Thus, it covers the entire *Application Lifecycle Management.*[51]

Chapter 3

Modelling in Spec Explorer:

"The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work"

John Von Neumann

Spec Explorer is a software tool developed by Microsoft, for advanced model-based specification and conformance testing. [4] It supports both online and offline testing, is integrated with the Microsoft .NET architecture, and uses pre/post models that are written in a language called Spec#, which is an extension of C#. Spec# has further use outside the realm of Spec Explorer, and is not just confined to developing models. It is also a general purpose language, designed to to strengthen C# in order to support a more cost-effective way of developing high-quality software. [13] The main extensions added to C# by Spec# are as follows: [14]

- A stronger type of system that can specify which object references must be non-null (for each object type T, there is a corresponding non-null type T!)
- Method specifications using preconditions (requires), postconditions(ensures), frame conditions (modifies) and exception specifications (throws).

- Extensive support for object invariants and class invariants, with fine-grained control over which methods are allowed to break these invariants while updating an object.
- Executable quantifiers and comprehension expressions for data structrues. These make it much easier to specify complex properties of data structures within specifications. For example, the expression Mapd in Members; <d,Seq> creates a new Map object that maps every element in the Members set to an empty sequence.

Models developed in Spec#, which are used by Spec Explorer for test generation, thus adopt many of these Spec# features such as preconditions, object invariants, non-null types, quantifiers etc. Precondition methods are in practice *guards*, determining whether a specific method can be enabled or not. After the preconditions have been stated, the body of the method (function call) can be written as regular C# code, making the process of modeling much easier for most programmers. Spec Explorer uses a theory of *interface automata* to generate tests from Spec# models, in which the test-generation process is somewhat analogous to a game. Playing this game, which proceeds as an interaction between the model and the SUT, requires that methods be annotated as either Actions or Events. Actions are model specific methods, whilst events are SUT specific methods.

3.1 Model-State Initialization:

The state of the model is completely defined by a combination of boolean variables, which represent a subset of the set of output variables from the controller, and a Sequence of Job objects representing the job-queue of the controller. In order to generalize the process of automatically generating test-cases for the controller, a model-state initializer was developed in order to synchronize the state of the model with the state of the SUT. Implementing such a feature also facilitates the automation of the model-based testing process, as each generated test-suite is customized to the actual state of the SUT during test-execution. The model-initialization happens prior to the model-exploration phase and thus requires a new test-suite to be generated upon each test-execution, which is a slight drawback to this feature. This feature was inspired by the information contained in the following source.[41]

As it happens, Spec Explorer cannot explore unmanaged code, which is any code that runs outside of the Common Language Runtime (CLR). In this case, the initialization requires the creation of different COM objects, which are examples of unmanaged code, and thus require a slight 'hack' in order for Spec Explorer to allow execution. This workaround was described in the link above, and involves using the Microsoft.Xrt.Runtime.NativeTypeAttribute namespace, which is an assembly-level attribute indicating that a given type should be treated as native.[42] For the PLC interface, the model-state initialization is executed by placing all the state-variables in the PLC into an OPC server. The adapter layer then places all the state-variables into a StateVariable group, and reads them simultaneously to a boolean array. This array is then called by the model-program during the start of the exploration phase, and the variables are assigned accordingly. For the Robot Web Service, the amount of state-variables has decreased as the level of control allowed through the REST interface has reduced. The model-state initializer for the PLC interface model is shown in 3.1, along with the initializer function in the adapter class in 3.2.

Listing 3.1: Initialization in model class

```
static class MyLoader
        {
                public static bool[] Load()
                {
                        bool[] contents =
                            ExtendedCommandIOProtocolModel.Sample.Accumulator.ModelStateInitializer()
                        return contents;
                }
        }
```

Listing 3.2: Initialization in adapter class

```
public static bool[] ModelStateInitializer()
                {
                        try
                        {
                                //stateArray = null;
                                //stateList = null;
                                Console.WriteLine("Attempting to connect to
                                    server (ModelStateInitializer)");
                                theSrv = new OpcServer();
                                Console.WriteLine("Attempting to connect to
                                    server 1(ModelStateInitializer)");
                                theSrv.Connect(serverProgID);
                                Console.WriteLine("Connected(ModelStateInitializer)");
                                Thread.Sleep(waitTime);
                                int CancelID;
                                int[] aE;
                                InitGroup = theSrv.AddGroup("InitGroup", false,
                                    1000);
```

```
itemDefsInit[0] = new OPCItemDef(itemY, true,
    25, VarEnum.VT_EMPTY);
itemDefsInit[1] = new OPCItemDef(itemK, true,
    11, VarEnum.VT_EMPTY);
itemDefsInit[2] = new OPCItemDef(itemL, true,
    12, VarEnum.VT_EMPTY);
itemDefsInit[3] = new OPCItemDef(itemM, true,
    13, VarEnum.VT_EMPTY);
itemDefsInit[4] = new OPCItemDef(itemN, true,
    14, VarEnum.VT_EMPTY);
itemDefsInit[5] = new OPCItemDef(itemO, true,
    15, VarEnum.VT_EMPTY);
itemDefsInit[6] = new OPCItemDef(itemP, true,
    16, VarEnum.VT_EMPTY);
itemDefsInit[7] = new OPCItemDef(itemQ, true,
    17, VarEnum.VT_EMPTY);
itemDefsInit[8] = new OPCItemDef(itemR, true,
    18, VarEnum.VT_EMPTY);
itemDefsInit[9] = new OPCItemDef(itemS, true,
    19, VarEnum.VT_EMPTY);
itemDefsInit[10] = new OPCItemDef(itemT, true,
    20, VarEnum.VT_EMPTY);
itemDefsInit[11] = new OPCItemDef(itemU, true,
    21, VarEnum.VT_EMPTY);
itemDefsInit[12] = new OPCItemDef(itemV, true,
    22, VarEnum.VT_EMPTY);
itemDefsInit[13] = new OPCItemDef(itemW, true,
    23, VarEnum.VT_EMPTY);
itemDefsInit[14] = new OPCItemDef(itemX, true,
    24, VarEnum.VT_EMPTY);
OPCItemResult[] rItmInit;
InitGroup.AddItems(itemDefsInit, out rItmInit);
if (HRESULTS.Failed(rItmInit[0].Error) ||
    HRESULTS.Failed(rItmInit[1].Error) ||
    HRESULTS.Failed(rItmInit[2].Error) ||
    HRESULTS.Failed(rItmInit[3].Error) ||
    HRESULTS.Failed(rItmInit[4].Error)
        || HRESULTS.Failed(rItmInit[5].Error)||
            HRESULTS.Failed(rItmInit[6].Error)
            ||
            HRESULTS.Failed(rItmInit[7].Error)
            ||
            HRESULTS.Failed(rItmInit[8].Error)
            ||
            HRESULTS.Failed(rItmInit[9].Error)
            ||
            HRESULTS.Failed(rItmInit[10].Error)
        || HRESULTS.Failed(rItmInit[11].Error)
            ||
            HRESULTS.Failed(rItmInit[12].Error)
```

```
                                    ||
                                    HRESULTS.Failed(rItmInit[13].Error)
                                    ||
                                    HRESULTS.Failed(rItmInit[14].Error))
                        { Console.WriteLine("OPC Tester: AddItems -
                            some failed"); InitGroup.Remove(true);
                            theSrv.Disconnect(); };
                        Console.WriteLine("ItemDefsInit added to
                            InitGroup!");
                        handlesSrvInit[0] = rItmInit[0].HandleServer;
                        handlesSrvInit[1] = rItmInit[1].HandleServer;
                        handlesSrvInit[2] = rItmInit[2].HandleServer;
                        handlesSrvInit[3] = rItmInit[3].HandleServer;
                        handlesSrvInit[4] = rItmInit[4].HandleServer;
                        handlesSrvInit[5] = rItmInit[5].HandleServer;
                        handlesSrvInit[6] = rItmInit[6].HandleServer;
                        handlesSrvInit[7] = rItmInit[7].HandleServer;
                        handlesSrvInit[8] = rItmInit[8].HandleServer;
                        handlesSrvInit[9] = rItmInit[9].HandleServer;
                        handlesSrvInit[10] = rItmInit[10].HandleServer;
                        handlesSrvInit[11] = rItmInit[11].HandleServer;
                        handlesSrvInit[12] = rItmInit[12].HandleServer;
                        handlesSrvInit[13] = rItmInit[13].HandleServer;
                        handlesSrvInit[14] = rItmInit[14].HandleServer;
                        Console.WriteLine("Handles Created!");
                        InitGroup.SetEnable(true);
                        InitGroup.Active = true;
                        //ReadGroup.DataChanged += new
                            DataChangeEventHandler(ReadGroup_DataChange);
                        InitGroup.ReadCompleted += new
                            ReadCompleteEventHandler(InitGroup_ReadComplete);
                        //InitGroup.DataChanged += new
                            DataChangeEventHandler(InitGroup_DataChange);
                        InitGroup.Read(handlesSrvInit, 55667788, out
                            CancelID, out aE);
                        Thread.Sleep(10000);
                        //stateList.ForEach(Console.WriteLine);
                        stateArray = stateList.ToArray();
                        Console.WriteLine("Model-State initialization
                            array: " + stateArray.ToString());
                        DisconnectInit();
                        return stateArray;
                }
                catch (Exception ex)
                {
                        Console.WriteLine("Connection error: " +
                            ex.ToString());
                        throw ex;
                }
        }
```

3.2 Creating a Model:

This thesis has applied model-based testing to two different protocols, utilizing the same model with relatively minor changes. The model was changed to account for paint-commands that have either been added or removed in the transition from Robotware 5 to Robotware 6, and also due to slight differences in several return codes between the two versions of Robotware. These differences will be discussed in the Results and Conclusion chapter. The model of the controller interface was largely developed through the manual of the PLC interface, however, since this manual was not complete, some of the logic behind the model had to be derived from the source code itself. This is a major flaw, as the source code should in theory be independent from the model. Logically, if the model was developed solely from the source code, it would naturally not lead to any failures. Initially, every output from the Extended Command Protocol were added as variables to the model. As the model increased in complexity, and the level of abstraction was developed, some of the outputs were deemed obsolete and commented out whilst others were 'invented'. Thus, only the necessary outputs needes to accurately model the behaviour of the SUT remained. An overview of every variable used within the model is given in 3.3.

Listing 3.3: Variable initialization in model class

```
static class AccumulatorModelProgram
    {
                public struct StackModelState
                {
                        public Sequence<Job> JobQueue;

                }
        static bool[] stateData = MyLoader.Load();
                static int noOfApplicators = 0;
                public static StackModelState ModelState = new
                    StackModelState() { JobQueue = new Sequence<Job>() };
                    //initialization of JobQueue
                static bool hvEnabled = stateData[1];
                static bool JobInProgress = stateData[2];
                static bool JobPending = stateData[0];
                static bool AutomaticMode = stateData[3];
                static bool AppEnabled = stateData[4];
                static bool isConnected;
                static bool tokenIsHeld = stateData[6];
                static bool MasterGranted = false; //Mastership is held by this
                    client
                static bool MotorOn = stateData[5];
                static int noOfJobs;
                static bool ReadyHighPriority = stateData[7];
                static bool SystemRunning = stateData[8];
                static bool EmergencyStop = stateData[9];
                static bool RobotError = stateData[14];
                static bool GoingToAuto = stateData[13];
                static bool MtrlChangeOn = stateData[10];
                static bool MtrlChangeRunning = stateData[11];
                static bool MtrlSupplyEnabled = stateData[12];
```

Theoretically, there are virtually no restrictions upon which actions can be called to the controller at any given time. There are a few exceptions, namely certain commands which require a prerequisite in order to be successfull. Also, if there are two clients operating on the controller, mastership must be taken into account which will invoke further restrictions. However, if only one client is used, there are very few restrictions placed on the sequence of commands. Naturally, this lead to an explosion in the state space which required the need for further "guards" in the model, for the sole purpose of limiting the state space. As such, mastership was implemented in the model in order to limit the state space and to create a "natural" sequence for the modelling of actions. The modelling of the JobQueueInsert(422) command is depicted in 3.4.

Listing 3.4: Modelled Job Queue Insert Command

```
[Rule(Action = "JobQueueInsert(paintCommand, progNo, materialNumber, jobOpt,
    quantity, forceHighPriority, position)/0")]
                static void JobQueueInsertOK(int paintCommand, int progNo, int
                    materialNumber, int jobOpt, int quantity, int
                    forceHighPriority, int position)
                {
                        Condition.IsTrue(ModelState.JobQueue.Count > 0);
                        Condition.IsTrue(quantity > 0);
                        Condition.IsTrue(position <= ModelState.JobQueue.Count);
                        Condition.IsTrue(ModelState.JobQueue.Count <= 10);
                        Job newjob = new Job { quantity = quantity,
                            programIndex = progNo, materialIndex =
                            materialNumber, programOption = jobOpt,
                            forceHighPriority = forceHighPriority };
                                //JobQueueEmpty = false;
                                        for (int i = 0; i < quantity; i++)
                                        {
                                                position = position + i;
                                                ModelState.JobQueue =
                                                    ModelState.JobQueue.Insert(position,
                                                    newjob);
                                                noOfJobs = noOfJobs + i;
                                        }
                    JobPending = true;
                }
```

In the model of the JobQueueInsert command, one can see how the guards are implemented. If any of the Condition.IsTrue(..) statements are violated, by equating to false rather than true, the action call will not be eligible as a transition to the next state in the exploration graph, essentially ignoring this action call. If all the guards are satisfied, a new Job object is created, which mimics the actual job created in the SUT with the same parameters. Also, a transition will appear in the exploration graph of the model, from a model state with one job less to a model state with one job more. This job is then added to the sequence of jobs, which is a Spec# implementation of a List of jobs, and the boolean variable JobPending is set to true. Additionally, one can see that if all the conditions are satisfied, the model action returns 0 indicating a successfull paint command. Therefore, a Rule must be created for each action return code, requiring altered condition guards.

The configuration for the model in Spec Explorer is the Config.cord file, where all the actions are defined along with the input parameters for each action, and the various switches for exploration and testing. The input parameter configuration for the JobQueueInsert action is demonstrated in 3.5.

Listing 3.5: Config.cord script parameter assignment for job queue insert function

```
action abstract static int Accumulator.JobQueueModify(int paintCommand, int
    progNo, int materialNumber, int jobOpt, int quantity, int
    forceHighPriority, int jobNo)
         where {.
                  Condition.In(paintCommand, 421);
                  Condition.In(progNo, 2);
                  Condition.In(materialNumber, 0);
                  Condition.In(jobOpt, 0);
                  Condition.In(quantity, 1);
                  Condition.In(forceHighPriority, 0);
                  Condition.In(jobNo, 0);

         .};
```

One can configure as many different combinations of input parameters as one wishes, however, caution must be placed on this process as even a slight increase in the combinations of input parameters will cause an exponential increase in the state-space of the exploration graph.

3.2.1 Cord Scripting:

This section will introduce the concept of Cord scripts, and the theory will in essence be a summary of the information contained here.[43]

Cord scripting is the heart of modeling in Spec Explorer, where it is used for configuring model exploration, testing and breaking down the composition of the model into scripting scenarios. In other words, the Cord script maps the model program to a control system (adapter layer) in order to provide a testable model of the SUT. In summary, the Cord script incorporates the following functionalities into the model program:

- "Maps the model to the SUT with action declarations for the methods of interest in the SUT."
- "Adds configurations to the model that further control the modeling, such as setting bounds for model exploration."
- Incorporates the rule methods defined in the model program to model exploration
- Setting the input parameters and combinations to each action declaration.
- Generates a test suite of test cases from the model.

Configuration:

Configurations are used to control model exploration and test generation. One application of such a configuration is to define actions upon which the model should be based. The set of actions represents steps in the model trace or in the execution of an implementation. Actions are generally divided into controlled and observed, or stimuli and response. Included in the configuration are a set of switches and parameters which govern the exploration and testing of the model, an example of which was given in the previous section. While validating a Cord script, Spec Explorer attempts to resolve action declarations to implementation methods, unless they are marked as abstract. Abstract actions do not need to correspond to implementation elements, however, methods or events with the same name need to exist in an implementation or adapter in order to run generated test-cases.[44]

Scenarios and Slicing:

Scenarios are used to limit (known as slicing in Spec Explorer jargon) the exploration space of a model program, such that the exploration graph becomes testable. For instance, a scenario could involve exploring a short sequence of possible actions, before resetting the graph back to the state it was in before the scenario occurred. For our purposes, we could define an arbitrary scenario for our model with the following machine:

machine AddJobs() : Main (JobQueueAppend(_); JobQueueAppend(_); JobQueueClear())*

The scenario above would be a small subset of all the available paint commands, where two jobs would be added to the queue followed by a clearing of the job queue. The scenario uses placeholders instead of concrete input parameters, as well as omitting the result of the call to JobQueueClear. Scenarios can also be combined with other behaviours (machines) by using different Cord operators, such as the parallell composition operator (||) which results in a behaviour containing only the steps that match in each machine. An example of the construction of a parallell composition behaviour is demonstrated in 3.6 [44].

Listing 3.6: Parallell Composition Behaviour Construct:

```
machine SlicedModelProgram() : Main
{
   AddJobs() || ModelProgram() // Parallel composition.
}
```

Operators are essential to understanding the Cord scripting language, however, a detailed explanation of each operator will not be given in this thesis. Detailed explanations can be found here.[46]

3.2.2 Protocols and Interfaces:

OPC Client:

As mentioned previously, the PLC interfaces with a PC through an OPC-client/server relationship. Thus, in order for the Spec Explorer model to execute actions within the PLC, an OPC client had to be created in the adapter-layer in Spec Explorer. The OPC server requires the client to create item groups, where one places the items one wishes to manipulate. For our purposes, three groups are required: Write, Read and Param. Write and Read are self-explanatory, but Param was created out of necessity discovered through trial and error. As has been demonstrated, the Extended Command I/O Protocol supports a maximum of 10 input arguments, however, only 3 parameters can be entered "at a time". The way to add more than 3 parameters, is to first write the initial 3 parameter values to the PLC variables ix_inParameter1, ix_inParameter2 and ix_inParameter3, which will represent the 8th, 9th and 10th input arguments respectively. Then, the variable ix_ParamToggle2 must be set, causing the controller to assign the 3 parameters to the final 3 parameters. The process is then repeated, by assigning 3 new parameters and toggling ix_ParamToggle1 and finally assigning the 3 last parameters and toggling CommandToggle. The Param item group in the OPC client, thus contains the variables ix_ParamToggle1 and ix_ParamToggle2, facilitating the execution of commands with multiple parameters. An excerpt showing the parameter initialization is depicted in 3.7

Listing 3.7: Parameter Initialization Adapter OPC Interface:

```
static OPCItemDef[] itemDefsWrite = new OPCItemDef[5];
                static OPCItemDef[] itemDefsRead = new OPCItemDef[4];
                static OPCItemDef[] itemDefsParam = new OPCItemDef[2];
                static OPCItemDef[] itemDefsInit = new OPCItemDef[15];
                static int[] handlesSrvWrite = new int[5] { 0, 0, 0, 0, 0 };
                static int[] handlesSrvRead = new int[4] { 0, 0, 0, 0 };
                static int[] handlesSrvParam = new int[2] { 0, 0 };
                static int[] handlesSrvInit = new int[15]
                    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
                static List<bool> stateList = new List<bool>();
                static bool[] stateArray = new bool[15];
                static int output;

#region WriteGroupItems
                const string itemA = "PLC1..ix_PaintCommand";
                                            // fully qualified ID of a VT_I4
                    item
                const string itemB = "PLC1..StartLoadCommand";
                const string itemD = "PLC1..ix_InParameter1";
                const string itemF = "PLC1..ix_InParameter2";
                const string itemG = "PLC1..ix_InParameter3";
#endregion
#region ReadGroupItems
                const string itemC = "PLC1..DSQC88_W003";       // fully
                    qualified ID of a VT_R8 item
                const string itemE = "PLC1..InTest";
                const string itemJ = "PLC1..InLongTest";
                const string itemZ = "PLC1..DSQC88_W004";
#endregion
#region ParamGroupItems
```

```
                const string itemH = "PLC1..ix_ParamToggle1";
                const string itemI = "PLC1..ix_ParamToggle2";
#endregion
#region StateInitGroupItems
                const string itemY = "PLC1..JobPending";
            const string itemK = "PLC1..HVEnabled";
                const string itemL = "PLC1..JobInProgress";
            const string itemM = "PLC1..AutomaticMode";
                const string itemN = "PLC1..AppEnabled";
            const string itemO = "PLC1..MotorOn";
                const string itemP = "PLC1..MasterGranted";
                const string itemQ = "PLC1..ReadyHighPriority";
                const string itemR = "PLC1..SystemRunning";
                const string itemS = "PLC1..EmergencyStop";
                const string itemT = "PLC1..MtrlChangeOn";
                const string itemU = "PLC1..MtrlChRunning";
                const string itemV = "PLC1..MtrlSupplyEnable";
            const string itemW = "PLC1..GoingToAuto";
                const string itemX = "PLC1..RobotError";
#endregion
                const int waitTime = 400;
                static int returnCode;
                static bool inTest = true;
                static bool inLongTest = true;
                static bool initiateComplete = false;
```

The meaning of these parameters will be discussed in Chapter 4 - Testing, but suffice it to say, they are required for the Sequential Function Chart program on the PLC, which controls which parameters are routed to the controller. The parameters inTest and inLong test function as locks for the action calls, such that whilst a test is being performed on the PLC no other commands can be issued. A long test boolean is required for commands with many parameters, and which require parameter toggles.

Each action call has a condition that requires the "isConnected" boolean to be true, which represents that the OPC client has been setup and communication with the PLC can commence. While naturally, the Initialize function is not a paint command in and of itself, it is required as an action in order to set up the adapter layer to execute the commands. Item handles are used as integer 'pointers' with which to reference the items. The Initialize function also defines event handlers, which handle OPC events such as DataChanged, ReadComplete and WriteComplete. A depiction of the DataChanged event handler is viewed in 3.8, in which relevant variables are assigned.

Listing 3.8: EventHandler of datachanged event for read only PLC variables:

```
public static void ReadGroup_DataChange(object sender, DataChangeEventArgs e)
```

```
                {
                        foreach (OPCItemState s in e.sts)
                        {
                                if (HRESULTS.Succeeded(s.Error))
                                {
                                        if (s.HandleClient == 5)
                                        {
                                                inTest =
                                                    Convert.ToBoolean(s.DataValue);
                                        }
                                        else if (s.HandleClient == 3)
                                        {
                                                returnCode =
                                                    Convert.ToInt32(s.DataValue);
                                        }
                                        else if (s.HandleClient == 10)
                                        {
                                                inLongTest =
                                                    Convert.ToBoolean(s.DataValue);
                                        }
                                        else if (s.HandleClient == 27)
                                        {
                                                output =
                                                    Convert.ToInt32(s.DataValue);
                                        }
                                }
                                else
                                        Console.WriteLine(" ih={0}
                                            ERROR=0x{1:x} !", s.HandleClient,
                                            s.Error);
                        }
                }
```

An example of a action call from the OPC client for JobQueueAppend, which is a command requiring many parameters, is depicted in 3.9.

Listing 3.9: Adapter layer action call for JobQueueAppend:

```
public static int JobQueueAppend(int paintCommand, int progNo, int
    materialNumber, int jobOpt, int quantity, int forceHighPriority, int
    jobOverride)
{
  int CancelID;
  int[] aE;
  Thread.Sleep(waitTime);
  while (true)
        {
         ReadCompleteEventHandler(ReadGroup_ReadComplete);
         ReadGroup.Read(handlesSrvRead, 55667788, out CancelID, out aE);
             if (inLongTest == false)
         {
                  object[] itemValues = new object[5];
                  object[] itemValuesParam = new object[2];
    //Write to Input Parameters and trigger main loop in PLC, all
        inputs are now defined
                  itemValues[0] = (int)0;
                  itemValues[1] = (bool)false;
                  itemValues[2] = (int)forceHighPriority;
                  itemValues[3] = (int)jobOverride;
                  itemValues[4] = 0;
                  itemValuesParam[0] = (bool)false;
                  itemValuesParam[1] = (bool)true; //Triggers alternative
                      loop in PLC, ix_ParamToggle2 = true;
                  WriteGroup.Write(handlesSrvWrite, itemValues, 99887766,
                      out CancelID, out aE);
                  ParamGroup.Write(handlesSrvParam, itemValuesParam,
                      99887766, out CancelID, out aE);
                  Thread.Sleep(waitTime);
                  //Write to Input Parameters and trigger second
                      alternative loop in PLC
                  itemValues[0] = (int)0;
                  itemValues[1] = (bool)false;
                  itemValues[2] = (int)0;
                  itemValues[3] = (int)0;
                  itemValues[4] = (int)quantity;
                  itemValuesParam[0] = (bool)true; //Triggers alternative
                      loop in PLC, ix_ParamToggle1 = true;
                  itemValuesParam[1] = (bool)false;
                  WriteGroup.Write(handlesSrvWrite, itemValues, 99887766,
                      out CancelID, out aE);
                  ParamGroup.Write(handlesSrvParam, itemValuesParam,
                      99887766, out CancelID, out aE);
                  Thread.Sleep(waitTime);
                   //Write to Input Parameters and trigger main loop in
                       PLC, all inputs are now defined
                  itemValues[0] = (int)paintCommand;
                  itemValues[1] = (bool)true; //Triggers main loop in PLC
                  itemValues[2] = (int)progNo;
                  itemValues[3] = (int)materialNumber;
                  itemValues[4] = (int)jobOpt;
                  //WriteGroup.WriteCompleted += new
                      WriteCompleteEventHandler(WriteGroup_WriteComplete);
                  WriteGroup.Write(handlesSrvWrite, itemValues, 99887766,
                      out CancelID, out aE);
                  Thread.Sleep(waitTime);
                  ReadGroup.Read(handlesSrvRead, 55667788, out CancelID,
                      out aE);
                  return returnCode;
        }
        else
```

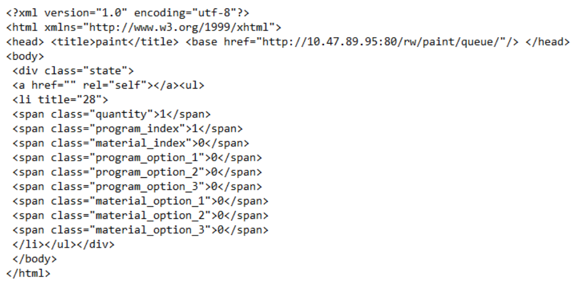
```
                {
                        Thread.Sleep(10);
                }
        }
}
```

Robot Web Service:

Robot Web Services is an interface toward the controller which is designed upon the REST(Representational State Transfer) principles. The fundamental architecture and principles of REST will not be discussed in detail in this thesis. However, Roy Fielding, the father of Representational State Transfer, gives a very good description of REST in Chapter 5 of his Doctoral Thesis. [16]. Basically, REST can be described as an architectural style which utilizes the HTTP protocol. REST is not a standardized protocol as of yet, and thus the developer maintains a certain degree of freedom when setting up and configuring a RESTful Web Service. A discussion of the paint specific aspect of the Robot Web Service will follow.

Robot Web Services uses HTTP as the application protocol, which is in concordance with the principles of REST. The HTTP protocol relies on two main communication parameters, namely URLs and verbs. URLS identify a particular resource, or perhaps more specifically 'points' to a particular resource. An example of a URL used for paint robots is the following, http://[local_host]/rw/paint/command, which points to the location where paint-commands are to be entered. The inputs and command numbers for the REST interface are identical to the PLC interface described in an earlier section. In RESTful architecture, URLs identify a resource which can be read, written, deleted or updated. These actions are executed by the HTTP verbs, which will be discussed in a later paragraph. A URL can also have query parameters, which are defined by a '?'. For instance, in order to successfully use the command URL shown above, one would have to append the query parameter such that the URL becomes, http://[local_host]/rw/paint/command?action=send". There are a number of resources located under /rw/paint, including: queue, command, commandresult, mc-engine, cycle, and ips. A snippet of the XML result from a GET request applied to the queue resource is shown in 3.1. The result was obtained through network sniffing using Wireshark.

The HTTP verbs are the actions that can be executed on each resource, and each resource in the RESTful service must implement at least one of the following verbs:



```
<?xml version="1.0" encoding="utf-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head> <title>paint</title> <base href="http://10.47.89.95:80/rw/paint/queue/"/> </head>
<body>
 <div class="state">
 <a href="" rel="self"></a><ul>
 <li title="28">
 <span class="quantity">1</span>
 <span class="program_index">1</span>
 <span class="material_index">0</span>
 <span class="program_option_1">0</span>
 <span class="program_option_2">0</span>
 <span class="program_option_3">0</span>
 <span class="material_option_1">0</span>
 <span class="material_option_2">0</span>
 <span class="material_option_3">0</span>
 </li></ul></div>
 </body>
</html>
```

Figure 3.1: GET-Queue XML response:

GET - The GET request is used to retrieve information on the resources, and is implemented by all the resources located under Paint except for command. The response to a GET request is an XML structure containing information of the state of the resource.

PUT - The PUT request is used to create or update the state of the resource, and is currently only supported by the command resource.

POST - POST requests are used to update a resource.

DELETE - A delete request erases all data stored in a resource.

Robot Web Services thus exposes a set of web APIs that can be consumed by any HTTP client using any programming language. The APIs return data either as XML or JSON which can be parsed using standard XML/JSON parsers. To use the REST APIs, the client application should make a HTTP request and parse the response. 3.13 depicts an example of a GET-request on the CommandResult resource, which is used in the adapter layer of Spec Explorer to retrieve the return code of the command action. The function used for parsing the XML response is shown in 3.10.

Listing 3.10: Parse XML response for serial:

```
public static void ParseResultForSerial(HttpWebResponse res)
{
        String Xml;
         using (StreamReader reader = new StreamReader(res.GetResponseStream()))
         {
                Xml = reader.ReadToEnd();

         }
        try
        {
                var doc = XDocument.Parse(Xml);
                XNamespace xmlNamespace = "http://www.w3.org/1999/xhtml";
                var spans = doc.Descendants(xmlNamespace + "span");
                foreach (var item in spans)
                {
                        int value = int.Parse(item.Value);
                        serial = value;
                }
        }
        catch (System.Exception ex)
        {
                Console.WriteLine(ex.ToString());
        }
}
```

The GET-request toward the CommandResult resource required appending a parameter, serial, to the URL. This is due to the fact that for each paint-command applied to the controller through the rest-interface, the controller automatically assigns a serial which is included in the XML response to the PUT-request. This serial is then sent with the GET-request, in order to apply the correct result for the appropriate action. The wrapper for the JobQueueAppend method called by the model is shown in 3.11. The function used to execute a paint-command on the controller via the REST interface is shown in 3.12, and the function used to extract the result of the command is shown in 3.12. The string formdata in the ExecuteAction function constitutes the body of the HTTP request, containing all the parameters required for the action.

Listing 3.11: Action called by the model for JobQueueAppend(421) command:

```
public static int JobQueueAppend(int paintCommand, int progNo, int
    materialNumber, int mtrlOpt1, int progOpt1, int mtrlOpt2, int quantity, int
    progOpt2, int progOpt3, int mtrlOpt3)
{
         ExecuteAction(paintCommand, progNo, materialNumber, progOpt1,
             progOpt2, progOpt3, mtrlOpt1, mtrlOpt2, mtrlOpt3, quantity, 0);
         CommandResult();
         return returnCode;
}
```

Listing 3.12: Execution of Paint Command via REST interface:

```
static void ExecuteAction(int command, int arg1, int arg2, int arg3, int arg4,
    int arg5, int arg6, int arg7, int arg8, int arg9)
{
 string server = "http://10.47.89.95";
 string path = "/rw/paint/command?action=send";
 Thread.Sleep(300);
 HttpWebCommunication _webCom = null;
_webCom = new HttpWebCommunication("");
 string formdata = _webCom.createFormData("number", Convert.ToString(command));
 formdata = _webCom.addToFormData(formdata, "arg1", Convert.ToString(arg1));
 formdata = _webCom.addToFormData(formdata, "arg2", Convert.ToString(arg2));
 formdata = _webCom.addToFormData(formdata, "arg3", Convert.ToString(arg3));
 formdata = _webCom.addToFormData(formdata, "arg4", Convert.ToString(arg4));
 formdata = _webCom.addToFormData(formdata, "arg5", Convert.ToString(arg5));
 formdata = _webCom.addToFormData(formdata, "arg6", Convert.ToString(arg6));
 formdata = _webCom.addToFormData(formdata, "arg7", Convert.ToString(arg7));
 formdata = _webCom.addToFormData(formdata, "arg8", Convert.ToString(arg8));
 Uri baseUri = new Uri(server);
 Uri pathAndQuery = new Uri(baseUri, path);
 DigestHttpWebRequest request = new DigestHttpWebRequest(m_user, m_password);
 request.Method = "PUT";
 request.PostData = Encoding.ASCII.GetBytes(formdata);
 HttpWebResponse result = null;
try
        {
          CookieContainer cc = new CookieContainer();
          cc.Add(baseUri, m_abbCookie);
          result = request.GetResponse(pathAndQuery, cc, m_connectionGroup);
          ParseResultForSerial(result);
        }
catch (Exception ex)
        {
        Console.WriteLine("Error:" + ex.ToString());
         if (result != null && result.ContentLength > 0)
        {
         Console.WriteLine(result.StatusCode.ToString());
        }
        DisposeHttpResponse(result);
    }
 }
```

Listing 3.13: Extraction of CommandResult/output:

```
static void CommandResult()
{
        String Xml;
        HttpWebCommunication _webCom = null;
        _webCom = new HttpWebCommunication("");
        string server = "http://10.47.89.95";
        string path = "/rw/paint/commandresult?serial=" + serial;
        Uri baseUri = new Uri(server);
        Uri pathAndQuery = new Uri(baseUri, path);
        DigestHttpWebRequest request = new DigestHttpWebRequest(m_user,
            m_password);
```

```
request.Method = "GET";
request.ContentType = "application/x-www-form-urlencoded;
    charset=utf-8";
HttpWebResponse result = null;
try
{
        CookieContainer cc = new CookieContainer();
        cc.Add(baseUri, m_abbCookie);
        result = request.GetResponse(pathAndQuery, cc,
            m_connectionGroup);
}
catch (Exception ex)
{
        if (result != null && result.ContentLength > 0)
        {
                Console.WriteLine(result.StatusCode.ToString());
        }
}
using (StreamReader reader = new
    StreamReader(result.GetResponseStream()))
{
        Xml = reader.ReadToEnd();
}
try
{
        var doc = XDocument.Parse(Xml);
        XNamespace xmlNamespace = "http://www.w3.org/1999/xhtml";
        var spans = doc.Descendants(xmlNamespace + "span");
        int count = 0;
        foreach (var item in spans)
        {
                Console.WriteLine("Item name in spans of Command
                    Result: " + item.Name);
                if(count == 1)
                {
                        int value = int.Parse(item.Value);
                        returnCode = value;
                }
                else if(count == 2)
                {
                        try
                        {
                                int value = int.Parse(item.Value);
                                outParam = value;
                        }
                        catch (Exception e)
                        {
                                outString = item.Value;
                        }
                }
```

```
                    count++;
                }
                result.Close();
            }
            catch (System.Exception ex)
            {
                Console.WriteLine(ex.ToString());
            }
        }
```

All clients that wish to consume the Robot Web Service must provide authentication in the form of a username and password. Robot Web Services use Digest authentication as a standard method of authentication, which is one of the standard methods a web server can utilize to encrypt credentials with a web browser. It applies a hash function to a password before sending it over the network, which is safer than basic access authentication, which sends plaintext. Technically, digest authentication is an application of MD5 cryptographic hashing with usage of nonce values to prevent replay attacks.[17] When an application connects and does the first HTTP request, the HTTP server on the robot controller responds with a HTTP error Unauthorized (401). When an initial connection is made, and the first HTTP request is sent to the service, the HTTP server on the robot controller responds with a HTTP 401 - Unauthorized Error. This is considered an authentication challenge, and the requester must then send its credentials in order to pass the authentication challenge.

Upon successfull authentication, the server sends a cookie along with the response, which must be utilized to maintain a session. An example of a Digest Request object is shown in the function ExecuteAction, but the entire code behind Digest Authentication will not be discussed here. One useful aspect of the Robot Web Service is subscription, where one subscribes on changes to various resources. Change events are sent on a websocket connection.[18] Other requests to the subscription server are done via the HTTP connection. A client should first create a subscription by specifying the resources of interest. Once the subscription is created, the client can establish a web socket connection to receive updates on the subscribed resources. Unsubscribing or updating the subscription resource list is done via HTTP request.

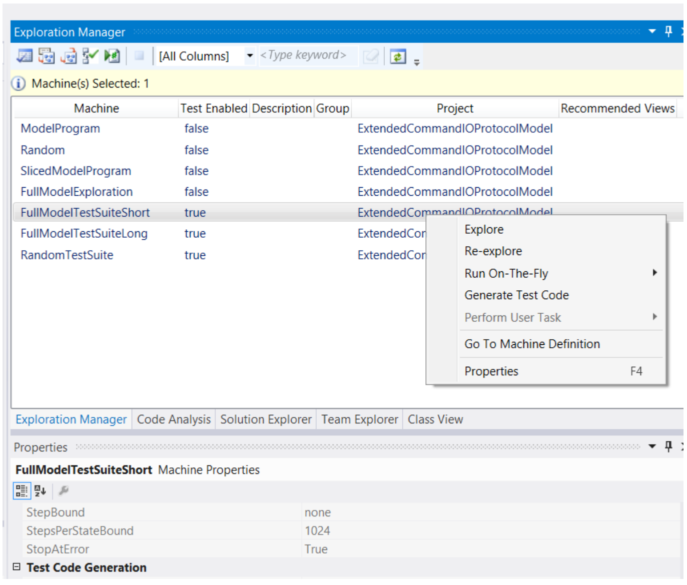
Priority for each resource can be specified during subscription. Resource priorities can be either 0 or 1. Priority 1 means "High" and Priority 0 means "Normal". Events for High priority resources are sent to the client immediately while events for Normal priority resources are sent after a delay. Subscription was used in the adapter layer in order to implement a "state-checker" function in Spec Explorer, namely a function to check whether the state of the model conformed to the state of the SUT more explicitly than a mere return call. Thus, everytime a function that altered the jobqueue was performed, an event was sent to the Spec Explorer test log to compare the states. An excerpt of the subscription function is depicted in 3.14.

Listing 3.14: JobQueue subscription:

```
public static void Subscribe()
{
        string resource = "/rw/paint/queue";
        Uri uri = new Uri(hostname + "/subscription/" + m_subscriptionGrup);
        string postData = "resources=1&1=" + resource + "&1-p=1";
        DigestHttpWebRequest request = new DigestHttpWebRequest(m_user,
            m_password);
        request.Method = WebRequestMethods.Http.Put;
        request.ContentType = "application/x-www-form-urlencoded;
            charset=utf-8";
        request.PostData = Encoding.ASCII.GetBytes(postData);
        HttpWebResponse result = null;
        try
        {
                CookieContainer cc = new CookieContainer();
                cc.Add(uri, m_abbCookie);
                result = request.GetResponse(uri, cc, m_connectionGroup);
                m_subscribedObjects.Add(resource, m_subscriptionGrup);
                if (result.StatusCode != HttpStatusCode.OK)
                {
                        Console.WriteLine("Error add subscription to group");
                }
                else
                {
                        Console.WriteLine("Subscription added");
                }
        }
        catch (Exception ex)
        {
                Console.WriteLine("Error subscribe" + ex.ToString());
                if (result != null && result.ContentLength > 0)
                {
                        Console.WriteLine(result.StatusCode.ToString());
                }
        }
}
```

3.3 Exploring the Model:

Once the model is created, one has to build the exploration graph from the actions specified in the model and the parameters specified as the input arguments to the action calls defined in the Parameter Combination section of the configuration file. Spec Explorer relies on a configuration script with the suffix .cord to initalize all the parameters and to define Machines, which in theory function as Finite State Machine Graphs which govern the exploration of the model. A snippet of the Exploration Manager window is displayed in 3.15, which contains every Machine defined in the configuration script along with the different exploration options.



Machine	Test Enabled	Description	Group	Project	Recommended Views
ModelProgram	false			ExtendedCommandIOProtocolModel	
Random	false			ExtendedCommandIOProtocolModel	
SlicedModelProgram	false			ExtendedCommandIOProtocolModel	
FullModelExploration	false			ExtendedCommandIOProtocolModel	
FullModelTestSuiteShort	true			ExtendedCommandIOProtocolModel	
FullModelTestSuiteLong	true			ExtendedCor	
RandomTestSuite	true			ExtendedCor	

StepBound	none
StepsPerStateBound	1024
StopAtError	True

Figure 3.2: Exploration Manager Window of Spec Explorer:

Listing 3.15: Definition of machines in the Config.cord script:

```
machine FullModelExploration() : Main where ForExploration = true
{
    construct accepting paths where NonDeterministicPathRemoval =
        "RemoveFullPath" for
    construct bounded exploration where PathDepth = 10 for
    ModelProgram;
}


machine FullModelTestSuiteShort() : Main where ForExploration = true,
    TestEnabled = true, OnTheFlyMinimumPathDepth = 5, OnTheFlySaveState = true,
    StopAtError = true
```

```
{
    construct test cases where strategy = "ShortTests" for
        FullModelExploration()
}
machine FullModelTestSuiteLong() : Main where ForExploration = true,
    TestEnabled = true
{

    construct test cases where strategy = "LongTests" for FullModelExploration()
}
```

The definition of several Machines used during model exploration is shown in 3.15, along with some important switches. Switches are flags which Spec Explorer used to trigger certain parameters and exploration options. A definition of the AcceptingPathsConstruct follows:[39]

AcceptingPathsConstruct ::= construct accepting paths [where NonDeterministicPathRemoval = "RemoveFullPath" | "RemoveNonAcceptingChoices"] for Behavior .

The inclusion of this construct in our Machine definition, effectively removes all paths from the resulting exploration that do not end in accepting states as defined in the model. Such paths result in errors when tests are generated, and removing them facilitates the execution of the graph-traversal algorithm. This removal process is referred to as *slicing the model*, or in other words removing a useless subset of the entire exploration state-space.

The NonDeterministicPathRemoval switch removes paths that are not guaranteed to reach accepting states, in the sense that the generated test cases never choose such paths. Only paths guaranteed to reach an accepting end state remain. Thus, triggering this switch effectively limits the state space.

One other switch worth mentioning within the Cord syntax is EnableExplorationCleanup, which removes all duplicated states from the exploration graph. A duplicated state in this context, is a state which contains the exact same combinations of variables and future transitions as any other state. Utilizing this feature cleans up the exploration graph, naturally also reducing redundancy in the process. The number of test-cases that are generated is reduced by a respectable amount, with the added trade-off of drastically increasing the time needed to finish exploring the machine and render the resulting exploration graph. The clean-up takes place after exploration has completed. [48]

3.3.1 Requirement Coverage:

The power of Model-Based Testing lies in the generation of test-cases, for instance generating a test-suite which provides transition coverage for each possible step in the exploration graph of the model. However, full transition coverage is not always possible, thus it is often useful to define a minimum set of requirements that must be satisfied in order for a test to pass. This is achieved using the RequirementCoverageConstruct in Spec Explorer, which is defined below:

RequirementCoverageConstruct ::= construct requirement coverage [where [Strategy=Full | Selective] [,RequirementsToCover=RequirementList] [,MinimumRequirementCount=Number] [for Behavior]] .

This construct reduces a finite exploration result to the sub-graph necessary to cover a set of requirements. This construct is used to produce a graph that covers all reachable requirements in a model, as declared both in Requirement.Capture and in Requirement.AssumeCaptured statements.[47]

Regarding the strategy switch, Spec Explorer supports the following strategies:

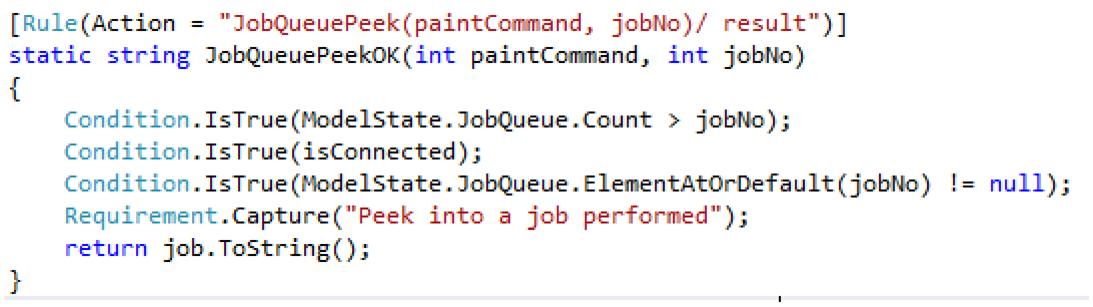
- Strategy set to Full, implies that all *requirement-covering transitions* are retained. This is also the default switch.
- Strategy set to Selective, maintains the requirement coverage from the original exploration, potentially removing duplicated coverage requirements.

An optional argument RequirementsToCover can also be specified, where the RequirementList is a string containing a list of comma-separated requirement IDs. If this option is not specified, then all the requirements in the model are covered.

Thus in order to make the test-suites more manageable, and to demonstrate proof of concept of the requirement coverage technique, a requirement coverage exploration graph and test-suite was created. For each interface, each of the modelled paint command actions was defined as a requirement. An example of a modelled paint-command (JobQueuePeek - allowing the user to extract information about a certain job in the queue) is depicted in 3.16

Listing 3.16: Requirement Capture in JobQueuePeek function:

```
static string JobQueuePeekOK(int paintCommand, int jobNo)
{
        Condition.IsTrue(ModelState.JobQueue.Count > jobNo);
        Condition.IsTrue(isConnected);
        Condition.IsTrue(ModelState.JobQueue.ElementAtOrDefault(jobNo) != null);
    Job job = ModelState.JobQueue[jobNo];
    Requirement.Capture("Peek into a job performed");
    return job.ToString();
}
```



```
[Rule(Action = "JobQueuePeek(paintCommand, jobNo)/ result")]
static string JobQueuePeekOK(int paintCommand, int jobNo)
{
    Condition.IsTrue(ModelState.JobQueue.Count > jobNo);
    Condition.IsTrue(isConnected);
    Condition.IsTrue(ModelState.JobQueue.ElementAtOrDefault(jobNo) != null);
    Requirement.Capture("Peek into a job performed");
    return job.ToString();
}
```

Figure 3.3: Example of requirement capture in model:

In other words, the requirement coverage ensures that all actions are executed at least once in the least number of transitions.

3.3.2 Robot Web Service Model Exploration:

When Spec Explorer has completed the exploration process, the resulting exploration graph is saved in the following format: "MachineName".seexlp. As these are rather large and unruly, they will not be displayed here, but they are provided on the accompanying CD. However, some key results will be presented here and can be compared with the requirement coverage graph.

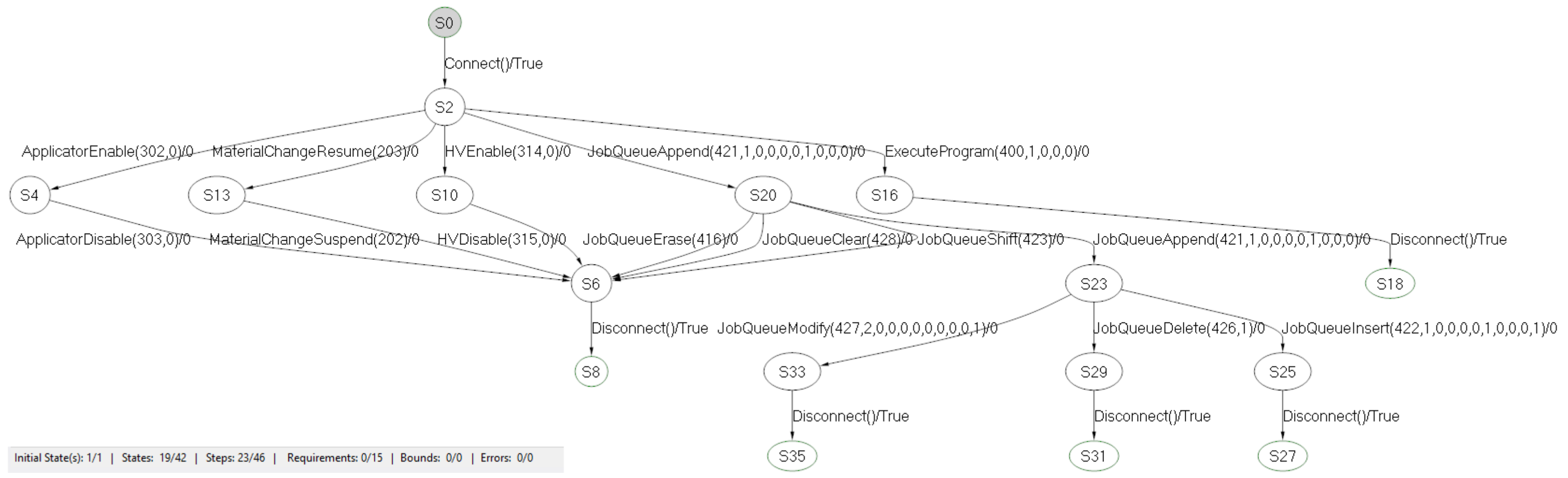
The FullModelExplorationRest.seexlp file, is the exploration graph generated by "stepping through" the model based on the input parameters in the cord script. It provides full transition coverage for the specified inputs, and consists of 295 states and 416 transitions. During the graph clean-up algorithm, which Spec Explorer implements during the final graph exploration process and consists of removing duplicate states and transitions, the number of states has been reduced to 102 and the number of transitions has been reduced to 223.

The RequirementExplorationRest.seexpl file, is the exploration graph generated by using the requirement coverage switch with the strategy set to Selective. This ensures that each requirement is performed at least once. All 15 requirements where covered, with 42 states and 46 transitions, reduced to 19 and 23 respectively using the clean-up algorithm, as depicted in 3.4.

3.3.3 PLC Interface Model Exploration:

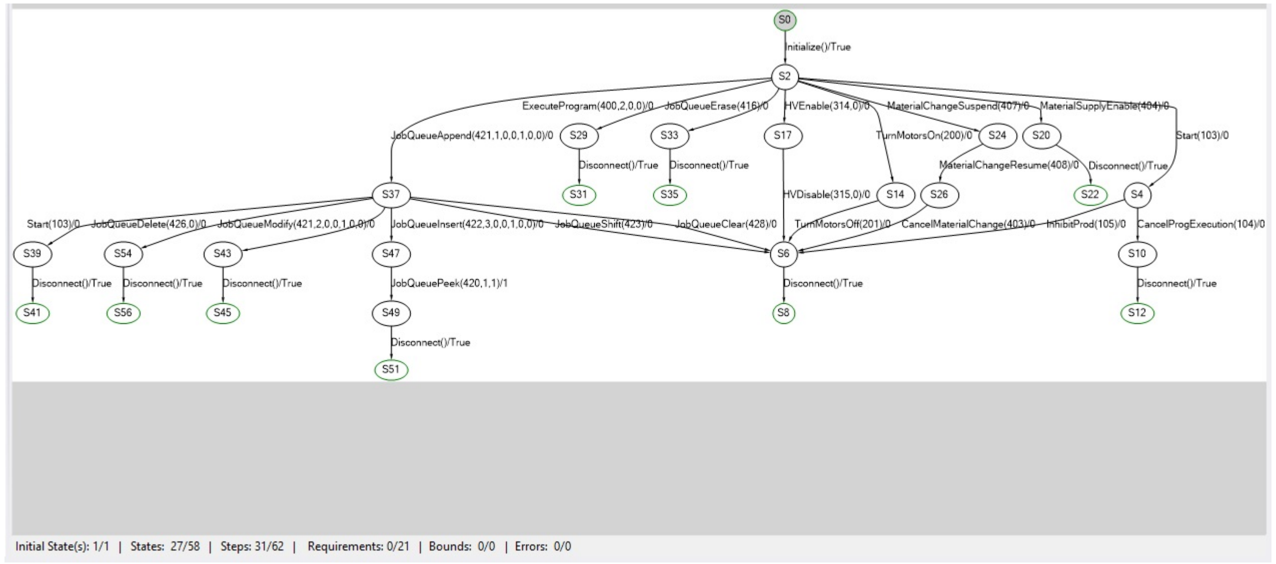
The FullModelExplorationPLC.seexlp file, is the exploration graph generated by "stepping through" the model based on the input parameters in the cord script. It provides full transition coverage for the specified inputs, and consists of 868 states and 1134 transition. During the graph clean-up algorithm, which Spec Explorer implements during the final graph exploration process and consists of removing duplicate states and transitions, the number of states has been reduced to 325 and the number of transitions has been reduced to 591.

The RequirementExplorationPLC.seexpl file, is the exploration graph generated by using the requirement coverage switch with the strategy set to Selective. This ensures that each requirement is performed at least once. All 21 requirements where covered, with 58 states and 62 transitions, reduced to 27 and 31 respectively using the clean-up algorithm, as depicted in 3.5.


S0
Connect()/True
S2
ApplicatorEnable(302,0)/0
MaterialChangeResume(203)/0
HVEnable(314,0)/0
JobQueueAppend(421,1,0,0,0,0,1,0,0,0)/0
ExecuteProgram(400,1,0,0,0)/0
S4
S13
S10
S20
S16
ApplicatorDisable(303,0)/0
MaterialChangeSuspend(202)/0
HVDisable(315,0)/0
JobQueueErase(416)/0
JobQueueClear(428)/0
JobQueueShift(423)/0
JobQueueAppend(421,1,0,0,0,0,1,0,0,0)/0
Disconnect()/True
S6
S23
S18
Disconnect()/True
JobQueueModify(427,2,0,0,0,0,0,0,0,0,1)/0
JobQueueDelete(426,1)/0
JobQueueInsert(422,1,0,0,0,0,1,0,0,0,1)/0
S8
S33
S29
S25
Disconnect()/True
Disconnect()/True
Disconnect()/True
S35
S31
S27

```
Initial State(s): 1/1  |  States: 19/42  |  Steps: 23/46  |  Requirements: 0/15  |  Bounds: 0/0  |  Errors: 0/0
```

Figure 3.4: RequirementCoverage Exploration Graph Rest:



Initial State(s): 1/1 | States: 27/58 | Steps: 31/62 | Requirements: 0/21 | Bounds: 0/0 | Errors: 0/0

Figure 3.5: RequirementCoverage Exploration Graph PLC:

Chapter 4

Testing:

"Begin at the beginning," the King said, very gravely, "and go on till you come to the end: then stop."
- Lewis Carroll, Alice in Wonderland

4.1 On-The-Fly Testing Algorithm:

Naturally, as is typically the case with model-based testing, the state-space of the model often becomes too vast to generate test-code that provides full transition-coverage for the model. Therefore, a great deal of focus and attention was placed on the online testing algorithm that is integrated into Spec Explorer, known as On-The-Fly Testing. Wolfram Schulte et al. give an in-depth analysis of OTF-testing here [15], but a brief summary will follow:

The test generation starts from the initial state of the model, which is defined during the model-state initialization process. Online testing implies that the test generation and exploration happens simultaneously, meaning that the algorithm is traversing the exploration graph and executing each cases as it goes along. Lewis Carroll summarizes the behaviour of this algorithm indirectly, with the elegant quote referred to at the start of this chapter, namely that the algorithm will stop when it reaches an accepting end state. Such a state can freely be specified during the design of the model. Each time the algorithm reaches a state, Spec Explorer first waits for a state-dependent timeout period to see whether an observable event arrives from the SUT. If such an event arrives, and is successfully intercepted by Spec Explorer, the algorithm chooses the transitions that correspond to the behaviour of the SUT. However, should no such event arrive before the timeout, the algorithm proceeds to execute one of the controllable methods (actions) in the model. The execution of the method requires that all of the preconditions (guards) for that method are satisfied. It then sends the corresponding event to the SUT, and validates the response to see whether the result *conforms* to the expected result. Once the algorithm reaches an end state, it will loop back to the initial state and explore another possible path within the model state-space and so on.

4.2 PLC Test Interface:

The PLC requires some form of software with which to interface with the robot controller. As has been stated previously, the robot controller inputs a set of I/Os to the PLC. The Robotware version running on the testrack was 5.15.3034, and includes the Extended Command I/O Procotol as an installation IO option. A Profinet fieldbus adapter was used in the main computer, a setup which is shown in 4.1 The PLC also has a Profinet IO communication module (CM579-PNIO), which allows the PLC to communicate with remote Profinet IOs. An illustration of this communication module is given in 4.2.

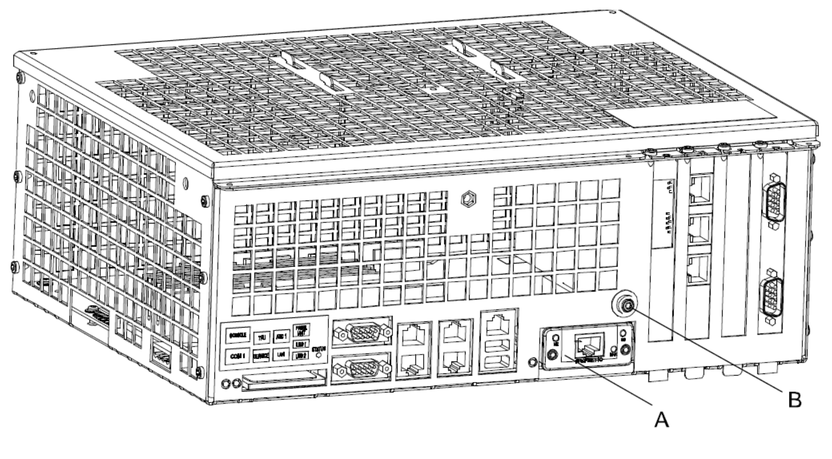
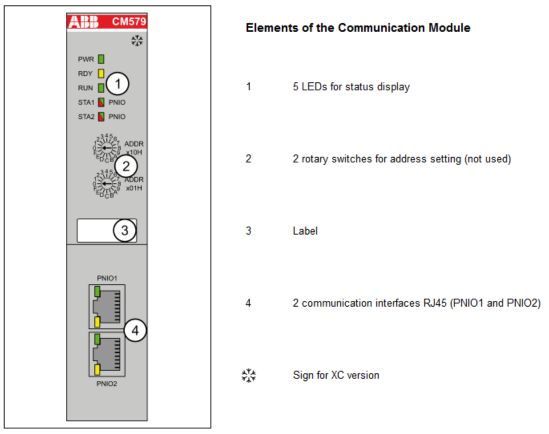


Figure 4.1: Main Computer with Profinet Fieldbus Adapter (A) added:

The controller has a predefined setting for the Profinet IO protocol, where the inputs and outputs come in as four distinct UDINTs (Unsigned Digital Integer), where each UDINT

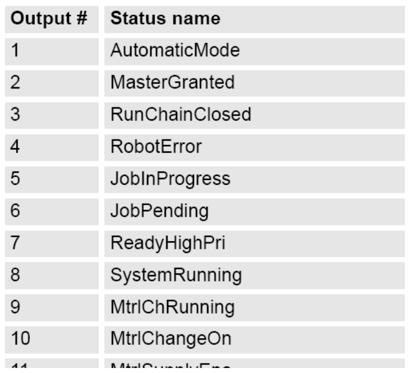


Elements of the Communication Module

1	5 LEDs for status display
2	2 rotary switches for address setting (not used)
3	Label
4	2 communication interfaces RJ45 (PNIO1 and PNIO2)
❊	Sign for XC version

Figure 4.2: Profinet Communication Module:

is 32 bits in size. Some of these UDINTs will thus be obsolete, as the Extended Command Protocol only requires 80 bits for the inputs and outputs to effectively communicate all relevant information. The first UDINT, which is the input to the PLC and output from the controller, consists of 32 boolean variables which represent the state of the controller. An extract of the entire output table for the Extended Command Protocol is given in 4.3



Output #	Status name
1	AutomaticMode
2	MasterGranted
3	RunChainClosed
4	RobotError
5	JobInProgress
6	JobPending
7	ReadyHighPri
8	SystemRunning
9	MtrlChRunning
10	MtrlChangeOn
[illegible]	[illegible]

Figure 4.3: Selected outputs from the Extended Command Protocol:

A natural starting point was then to create a program that extracted the boolean variables from the UDINT, a process which eventually resulted in two separate programs. The first program was a Function Block program written in a combination of ladder logic and the standardized Function Blocks within CoDeSys, and was used to break each UDINT down into four eight-bit bytes. An excerpt of this program is shown in 4.4. The second program involved extracting each boolean variable from the bytes and assigning them

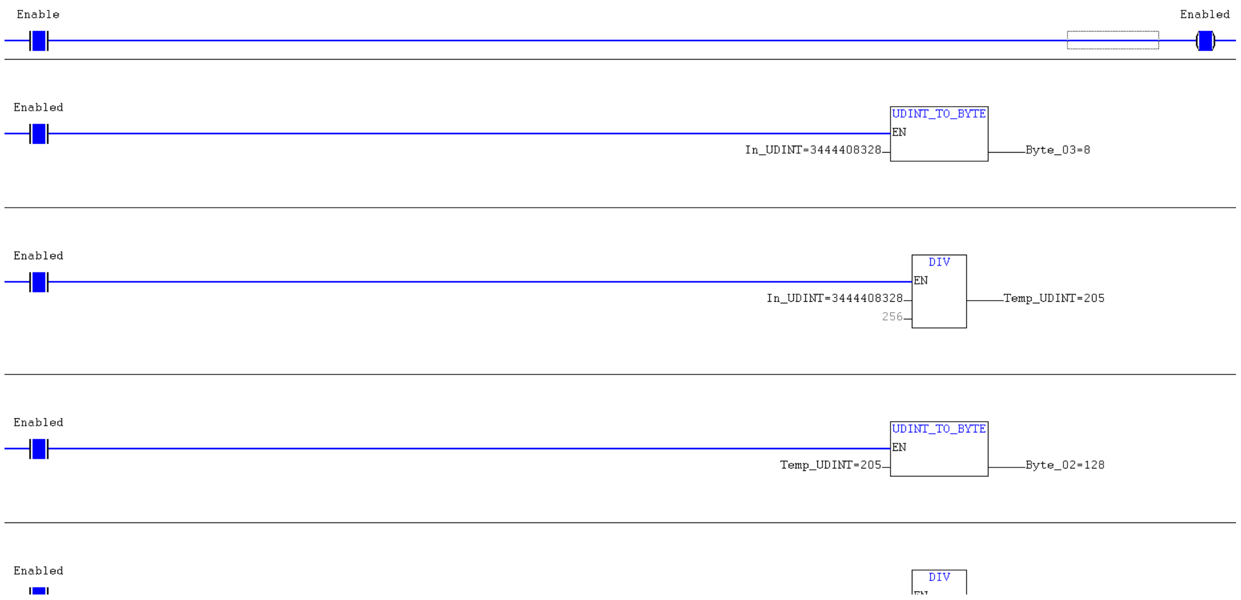


Figure 4.4: UDINT to BYTE program:

to a variable. This program was a Function Block program written in ladder logic, and assigned each bit to a either a local variable or a global variable depending on whether the variable was used elsewhere in the PLC or not. A depiction of this program is given in 4.5.

With all the subfunctions now in position, the main program of the PLC could be written which would execute the function calls to the controller. As this process would be highly iterative and step-based, the Sequential Function Chart seemed like an excellent choice for the main program of the PLC. An initial attempt at a main program for the PLC was written in Structured Text, and implemented a primitive state-machine. However, this was abandoned in favour of the SFC program. A snippet of the main execution program of the PLC is given in 4.7 and is summarized as follows: (From here on out, the 'inputs and outputs' are referring to the inputs and outputs of the controller.)

- 1) First the PLC initializes all the input variables (which are the "writeable" inputs to the PLC from the Spec Explorer adapter layer), and executes the functions mentioned above allowing the outputs to be read and shown on the CoDeSys program. The external agent, in this case the Spec Explorer OPC client, must now write to all the input parameters and trigger the StartLoadCommand once

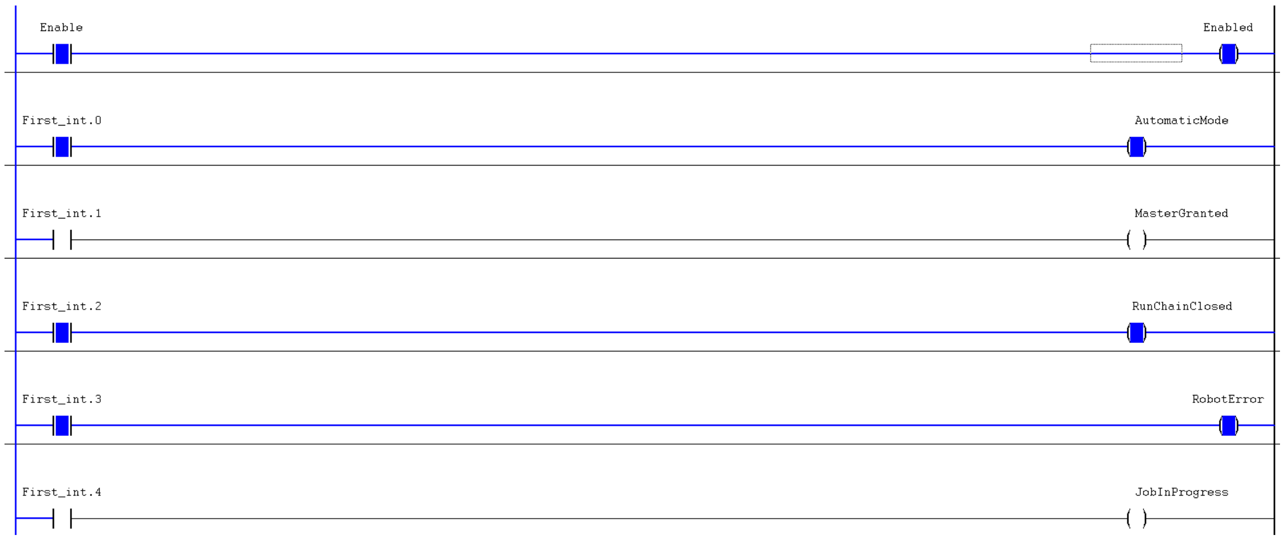
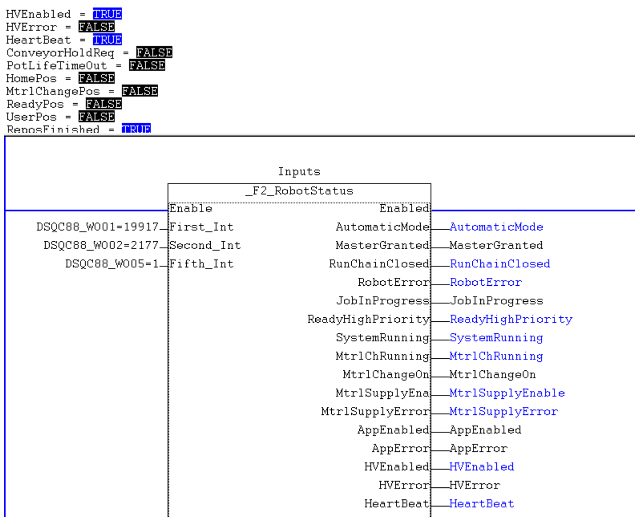


Figure 4.5: Extraction of boolean bits from a byte:

the parameters have been loaded. The StartLoadCommand acts as a transition trigger, allowing the transition to the next state when the value of that parameter is TRUE.

- 2)There are several alternative branches from the initial branch, one for ix_ParamToggle1 and one for ix_ParamToggle2, depending on whether there are several input parameters or not. Thus, if Spec Explorer wanted to execute the JobQueueAppend function (which has nine different input parameters), the C# OPC client would first write the three last inputs, trigger ix_ParamToggle2 (which acts as a trigger to the alternative step), and load the parameters into the controller. The ParamAck output is used as a transition trigger, indicating that the controller has successfully retrieved and stored the parameters. Upon ParamAck the state machine will loop back to the initial state, where the process repeats for ix_ParamToggle1. Finally, the final parameters are written to and StartLoadCommand is set to TRUE, triggering the main branch.
- 3) The main branch loads outputs, sets/and resets the CommandToggle bit (which is the last bit in the Command word), and waits for the CommandAck bit to go High (indicating a successfull command). Each read and write action step is followed by a Timer step, with a wait-time equivalent to at least one PLC cycle, giving the PLC time to stabilize. There are a number of alternative branches, where the state machine loops back to the CommandToggle step if the CommandAck bit



```
HVEnabled = TRUE
HVError = FALSE
HeartBeat = TRUE
ConveyorHoldReq = FALSE
PotLifeTimeOut = FALSE
HomePos = FALSE
MtrlChangePos = FALSE
ReadyPos = FALSE
UserPos = FALSE
ReposFinished = TRUE

                                     Inputs
                              _F2_RobotStatus
                        Enable                      Enabled
DSQC88_W001=19917       First_Int             AutomaticMode  AutomaticMode
DSQC88_W002=2177        Second_Int            MasterGranted  MasterGranted
DSQC88_W005=1           Fifth_Int             RunChainClosed RunChainClosed
                                                 RobotError  RobotError
                                              JobInProgress  JobInProgress
                                          ReadyHighPriority  ReadyHighPriority
                                              SystemRunning  SystemRunning
                                              MtrlChRunning  MtrlChRunning
                                              MtrlChangeOn   MtrlChangeOn
                                              MtrlSupplyEna  MtrlSupplyEnable
                                            MtrlSupplyError  MtrlSupplyError
                                                 AppEnabled  AppEnabled
                                                   AppError  AppError
                                                  HVEnabled  HVEnabled
                                                    HVError  HVError
                                                  HeartBeat  HeartBeat
```

Figure 4.6: Depiction of the running program described above, with assigned inputs and outputs

is not set, and assigns the error value to a global error variable upon a RobotError or CommandError output.

Loading the inputs to the controller required writing a function that transformed INTs to UDINTS, or 16-bit to 32-bit. The resulting program was a Function Block program written in Structured Text, which is depicted in 4.9.

4.3 OPC Server Configuration:

The OPC server used is the OPC server package that was included in the ABB Automation Builder software, and is delivered as a part of CoDeSys. A diagram of the CoDeSys OPC server architecture is shown in 4.11. When a project is loaded from the CoDeSys programming system to the controller, one can also simultaneously generate a **symbol file** (*.sym or *.sdb) and store it in the gateway. The symbol contains items, which are data objects that correspond exactly to one variable within the controller program. It is through these items that manipulation of variables within the controller can occur. The OPC server requests the content of the symbol file from the gateway server, and creates an item list from it. Since the contents of this list are determined by the variables assigned in the controller, they cannot be influenced by any external OPC client.

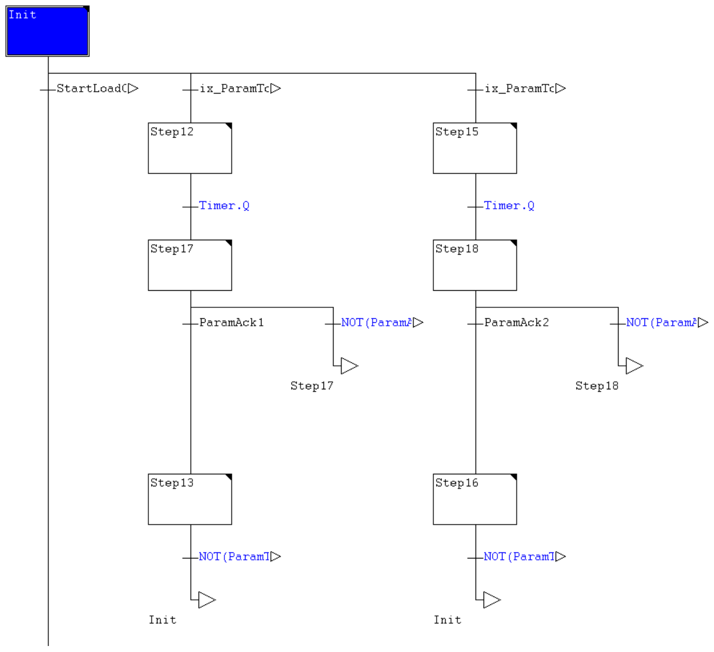
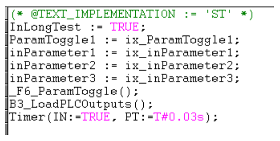


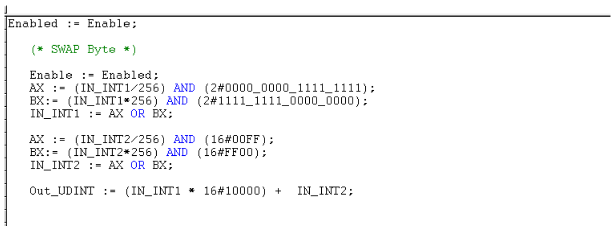
Figure 4.7: Snippet of main PLC program written in SFC:



```
(* @TEXT_IMPLEMENTATION := 'ST' *)
InLongTest := TRUE;
ParamToggle1 := ix_ParamToggle1;
inParameter1 := ix_inParameter1;
inParameter2 := ix_inParameter2;
inParameter3 := ix_inParameter3;
_F6_ParamToggle();
B3_LoadPLCOutputs();
Timer(IN:=TRUE, PT:=T#0.03s);
```

Figure 4.8: Example of the ST defining a step, in this case Step 12:

The OPC server always reads the last loaded symbol file for the project via the gateway. All items within the OPC symbol list are updated each scan cycle with the value of the controller variables, with read and write access times for these variables in the order of approximately 1 millisecond. According to ABB documentation, the maximum amount of items in a symbol list that can be handled 'smoothly' is approximately 15 000 items, corresponding to a symbol file size of approximately 1.5 MB. The OPC server also supports grouping of data, and distinguishes between public (allocated by the OPC server) and private (allocated by the OPC client) groups. Grouped data should always be read by the OPC server in a consistent manner, for instance all variables at the same time.[40]



```
Enabled := Enable;

   (* SWAP Byte *)

   Enable := Enabled;
   AX := (IN_INT1/256) AND (2#0000_0000_1111_1111);
   BX:= (IN_INT1*256) AND (2#1111_1111_0000_0000);
   IN_INT1 := AX OR BX;

   AX := (IN_INT2/256) AND (16#00FF);
   BX:= (IN_INT2*256) AND (16#FF00);
   IN_INT2 := AX OR BX;

   Out_UDINT := (IN_INT1 * 16#10000) +  IN_INT2;
```

Figure 4.9: 16-bit to 32-bit transformation:

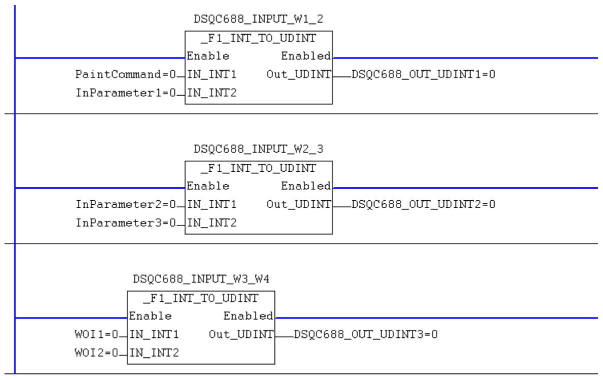


Figure 4.10: Assigning inputs:

4.4 Spec Explorer Configuration:

The generated test-cases are basically the result of the exploration of a Cord machine, with the switch TestEnabled set to true. As briefly mentioned earlier, the test-cases are the set of "unwrapped" paths of the exploration graph, where a path is defined as the trace from the initial state of the model to an accepting end state. The set of all traces is referred to as the test-oracle, or the test-suite. If the model is sufficiently structured and well-behaved, test-cases which offer full transition coverage of the model can be generated. Otherwise, if the model proves to be too unruly, On-The-Fly testing offers an alternative to conventional test generation.

If test-cases can be generated, the output of the test generation process is a C# file containing the test-code. The machine that was selected for test-generation via the Spec Explorer Exploration Manager Window will be displayed on the completion of the process, along with the filename and location of the test code output files. The generated test format is one which can be incorporated into the Microsoft Visual Studio Test Tools framework, which is the general format for many other automated tests for ABB Robotics

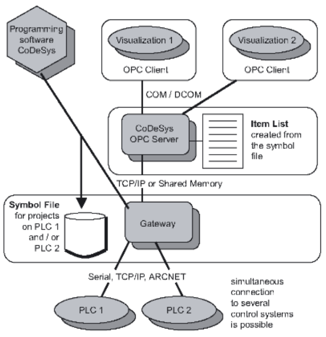

Programming software CoDeSys
Visualization 1
OPC Client
Visualization 2
OPC Client
COM / DCOM
CoDeSys OPC Server
Item List created from the symbol file
TCP/IP or Shared Memory
Symbol File for projects on PLC 1 and / or PLC 2
Gateway
Serial, TCP/IP, ARCNET
PLC 1
PLC 2
simultaneous connection to several control systems is possible

Figure 4.11: Architecture of the CoDeSys OPC server V2.0

and allows for automated testing using the Microsoft TFS API. Generated tests do not require the model program to run, as all inputs and outputs along with the expected return codes are contained within the test code. A screenshot of some generated test-code for the PLC interface is shown in 4.1.

Listing 4.1: Generated and executable test code:

```
#region Test Starting in S1036
       [Microsoft.VisualStudio.TestTools.UnitTesting.TestMethodAttribute()]
       public void FullModelTestSuiteLongS1036() {
           this.Manager.BeginTest("FullModelTestSuiteLongS1036");
           this.Manager.Comment("reaching state \'S1036\'");
           bool temp30;
           this.Manager.Comment("executing step \'call Connect()\'");
           temp30 = RW6ModelRestService.Sample.Adapter.Connect();
           this.Manager.Comment("reaching state \'S1037\'");
           this.Manager.Comment("checking step \'return Connect/True\'");
           TestManagerHelpers.AssertAreEqual<bool>(this.Manager, true, temp30,
               "return of Connect, state S1037");
           this.Manager.Comment("reaching state \'S1038\'");
           int temp31;
           this.Manager.Comment("executing step \'call
               ApplicatorEnable(302,0)\'");
           temp31 = RW6ModelRestService.Sample.Adapter.ApplicatorEnable(302,
               0);
           this.Manager.Comment("reaching state \'S1039\'");
           this.Manager.Comment("checking step \'return ApplicatorEnable/0\'");
           TestManagerHelpers.AssertAreEqual<int>(this.Manager, 0, temp31,
               "return of ApplicatorEnable, state S1039");
           this.Manager.Comment("reaching state \'S1040\'");
           int temp32;
           this.Manager.Comment("executing step \'call
               JobQueueAppend(421,1,0,0,0,0,1,0,0,0)\'");
```

```
        temp32 = RW6ModelRestService.Sample.Adapter.JobQueueAppend(421, 1,
            0, 0, 0, 0, 1, 0, 0, 0);
        this.Manager.Comment("reaching state \'S1041\'");
        this.Manager.Comment("checking step \'return JobQueueAppend/0\'");
        TestManagerHelpers.AssertAreEqual<int>(this.Manager, 0, temp32,
            "return of JobQueueAppend, state S1041");
        this.Manager.Comment("reaching state \'S1042\'");
        int temp33;
        this.Manager.Comment("executing step \'call JobQueueClear(428)\'");
        temp33 = RW6ModelRestService.Sample.Adapter.JobQueueClear(428);
        this.Manager.Comment("reaching state \'S1043\'");
        this.Manager.Comment("checking step \'return JobQueueClear/0\'");
        TestManagerHelpers.AssertAreEqual<int>(this.Manager, 0, temp33,
            "return of JobQueueClear, state S1043");
        FullModelTestSuiteLongS164();
        this.Manager.EndTest();
    }
    #endregion
```

4.4.1 Test Case Construct:

The test case construct is an aspect of the Cord scripting language, and is used to generate test-cases and define certain test strategies. A definition is given below:

TestCasesConstruct ::= construct test cases whereStrategy="ShortTests" | "LongTests" [,AllowUndeterminedCoverage= true | false] [,StopAtAccepting= true | false] [,MinimumPathLength= Number] for Behavior .

The test strategy is an important switch, as it defines which algorithm should be used in the generation of the test-cases. Two different strategies are defined, ShortTests and LongTests. Both strategies ensure that each transition in the exploration graph is covered at least once in at least one test case in the test suite, which results in a test oracle providing full transition coverage of the model. There are, however, some important differences which will be mentioned briefly below:

- **Short Tests** - *"This strategy generates a complete test case as soon as it finds an accepting state with a path that includes at least one transition step that has not already been tested. To do this, it performs a depth-first search of the exploration* *graph, starting from an initial state, adding edges in the path to the current subgraph (called a "test case". When an accepting end state is reached, the current test case is cut and added to the output, and a new test case starts from an initial state. An accepting end state is defined as an accepting state with no outgoing uncovered edges. Observable choice nodes cause the algorithm to explore all outgoing paths in parallell until they are all cut. This strategy has a tendency to generate a larger number of test cases, but shorter ones.)*[50]

- **Long Tests** - This strategy bases itself on a general tour of the exploration graph. Within this tour, it attempts to find the most complete path from an initial state that covers as many transition steps as possible before ending with an accepting state that is as "deep" as possible. Contradictory to the other strategy, LongTests tends to generate fewer, but longer test-cases.

Within the "ShortTests" strategy, two additional switches are offered which provide further control over the generated test cases.

- **StopAtAccepting** - Specifies whether a test case can be cut and stored upon encountering an accepting state, and not only at accepting end states.
- **MinimumPathLength** - This is an integer which specifies that a test case can be cut at an accepting end state, or at an accepting state at least this distance from the initial state.

There is a requirement placed on the machine using this construct, namely that the machine must have a finite number of states. By default, Spec Explorer verifies during graph exploration that the resulting test suite does not rely on undetermined coverage. Although there is a swith that overrides this default behavior, and can be changed by setting the AllowUndeterminedCoverage switch to true.

4.5 Test Suites:

Three test suites were created for each interface, with three different switches: LongTest, ShortTest and RequirementCoverage. This is not generally necessary, but for completeness three test suites were chosen. Each interface and their respective test suites will be summarized below:

4.5.1 Robot Web Service:

The FullModelTestSuiteLongRest.seexlp file, is the test suite generated by "stepping through" the model based on the input parameters in the cord script, with the LongTests switch set to true. It provides full transition coverage for the specified inputs, while attempting to follow the longest possible route from an initial state to an accepting end state within the model exploration graph. It consists of 1106 states and 1071 transitions. During the graph clean-up algorithm, which Spec Explorer implements during the final graph exploration process and consists of removing duplicate states and transitions, the number of states has been reduced to 576 and the number of transitions has been reduced to 541.

The FullModelTestSuiteShortRest.seexlp file, is the test suite generated by "stepping through" the model based on the input parameters in the cord script, with the ShortTests switch set to true. It provides full transition coverage for the specified inputs, while attempting to follow the shortest possible route from an initial state to an accepting end state within the model exploration graph. It consists of 1170 states and 1135 transitions. During the graph clean-up algorithm, which Spec Explorer implements during the final graph exploration process and consists of removing duplicate states and transitions, the number of states has been reduced to 608 and the number of transitions has been reduced to 573. As expected, Short Tests create more test cases than Long Tests, but each test case consists of fewer action call sequences.

The RequirementTestSuiteRest.seexlp file, is the test suite generated by "stepping through" the model based on the input parameters in the cord script, with the RequirementCoverage switch set to true, and the ShortTests switch set to true. It provides requirementCoverage for the specified inputs, with the RequirementStrategy set to selective ensures that each requirement is fulfilled at least once. It consists of 90 states and 84 transitions. During the graph clean-up algorithm, which Spec Explorer implements during the final graph exploration process and consists of removing duplicate states and transitions, the number of states has been reduced to 48 and the number of transitions has been reduced to 42.

4.5.2 PLC Interface:

The FullModelTestSuiteLongPLC.seexlp file, is the test suite generated by "stepping through" the model based on the input parameters in the cord script, with the LongTests switch set to true. It provides full transition coverage for the specified inputs, while attempting to follow the longest possible route from an initial state to an accepting end state within the model exploration graph. It consists of 5059 states and 4858 transitions. During the graph clean-up algorithm, which Spec Explorer implements during the final graph exploration process and consists of removing duplicate states and transitions, the number of states has been reduced to 2650 and the number of transitions has been reduced to 2449.

The FullModelTestSuiteShortPLC.seexlp file, is the test suite generated by "stepping through" the model based on the input parameters in the cord script, with the ShortTests switch set to true. It provides full transition coverage for the specified inputs, while attempting to follow the shortest possible route from an initial state to an accepting end state within the model exploration graph. It consists of 3277 states and 3155 transitions. During the graph clean-up algorithm, which Spec Explorer implements during the final graph exploration process and consists of removing duplicate states and transitions, the number of states has been reduced to 1716 and the number of transitions has been reduced to 1594. As expected, Short Tests create more test cases than Long Tests, but each test case consists of fewer action call sequences.

The RequirementTestSuitePLC.seexlp file, is the test suite generated by "stepping through" the model based on the input parameters in the cord script, with the RequirementCoverage switch set to true, and the ShortTests switch set to true. It provides requirementCoverage for the specified inputs, with the RequirementStrategy set to selective ensures that each requirement is fulfilled at least once. It consists of 132 states and 122 transitions. During the graph clean-up algorithm, which Spec Explorer implements during the final graph exploration process and consists of removing duplicate states and transitions, the number of states has been reduced to 71 and the number of transitions has been reduced to 61.

4.6 Test Execution and Logging:

The process of test execution differs greatly between OTF-testing and the generation of test-code. As the generated test-code uses the Visual Studio framework for testing, the result is a set of tests cases that are executed sequentially with a pass/fail verdicted upon each completed test-case. Upon completion of the tests, a complete test log is created and stored in the project folder along with the Spec Explorer console log for debugging and detailed analysis of the execution. Tera Term, which is an open-source terminal emulator program, is also useful during test execution to directly log the paint protocol inputs and outputs being sent to and from the controller.

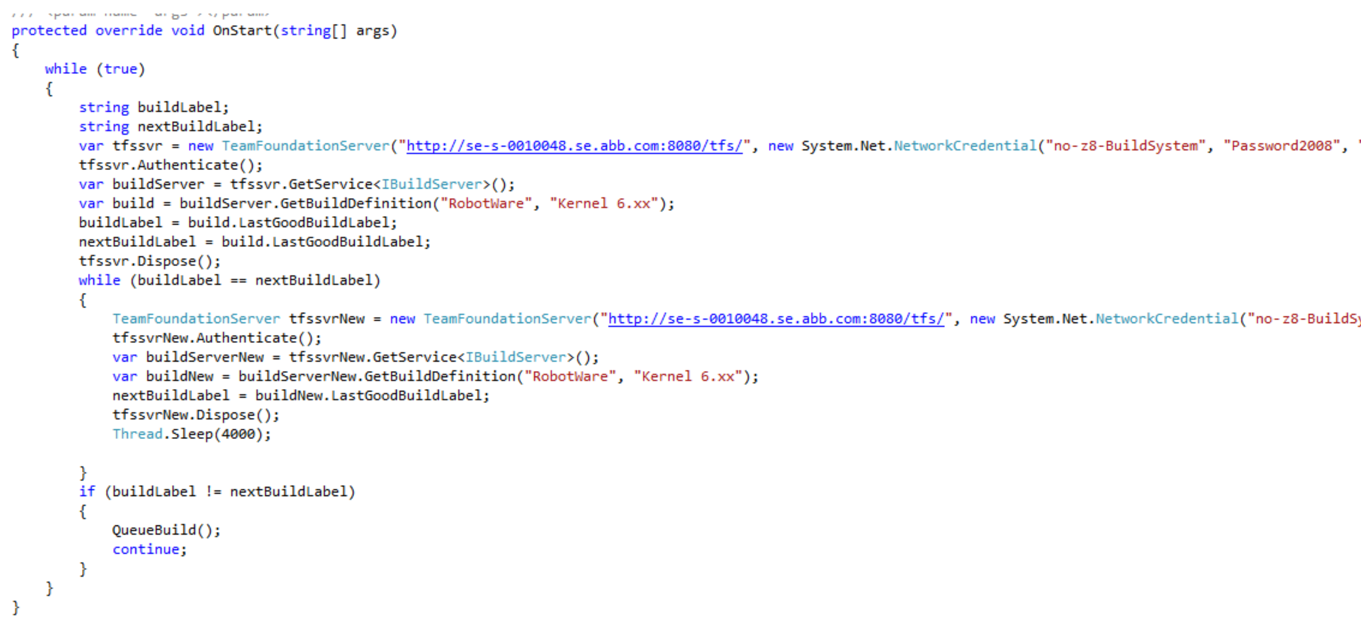
This is not the case for On-The-Fly testing, as when using OTF there are no pre-programmed test cases as such. During On-The-Fly testing, the entire test process is depicted on the Spec Explorer console with a summary of the results being generated at the end of the testing process. Interestingly, upon completion of OTF-testing a detailed log of the executed test cases are exported to the project folder, which contains detailed information of the model state during every step.

4.7 Automatic Test Execution On Complete Build:

The next feature which was needed was a method of generating and executing the different test suites upon the completion of a new build of Robotware. The building of Robotware happens in Sweden, where the developers in Bryne are informed per mail upon completion of a build. A build machine was created in TFS, which basically binds the test cases to a Visual Studio Testing environment and performs the test cases when the build machine is queued.

4.7.1 RW6BuildListener Windows Service:

To summarize, a Windows Service is a program that operates in the background and can be configured to initialize on start-up and run for as long as the computer is running. They require no log in, as they operate in the context of their own dedicated user-accounts. [52] A prototype "listening hook" Windows Service was created using the Team Foundation API, which "listened" for new builds of RobotWare and triggered test-case generation and execution upon detection of a new build. A snippet of the OnStart function is shown in 4.12 The basic functionality of the service is as follows: First, the application logs onto the ABB Robotics Team Foundation Server, and polls the current build label of RW6. Thus, the application enters a continuous loop, where the build label is polled every four seconds and compared to the previous build label. If the build labels differ, the application calls the QueueBuild method. The QueueBuild methods queues the RW6Build machine defined in the Team Foundation Server, and executes the execution of the test-cases.



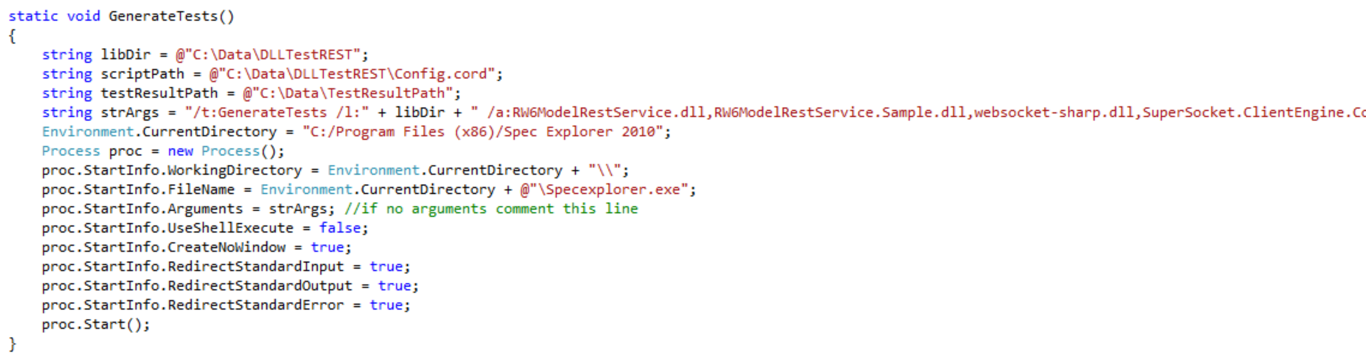
```
/// <param name="args"></param>
protected override void OnStart(string[] args)
{
    while (true)
    {
        string buildLabel;
        string nextBuildLabel;
        var tfssvr = new TeamFoundationServer("http://se-s-0010048.se.abb.com:8080/tfs/", new System.Net.NetworkCredential("no-z8-BuildSystem", "Password2008", 
        tfssvr.Authenticate();
        var buildServer = tfssvr.GetService<IBuildServer>();
        var build = buildServer.GetBuildDefinition("RobotWare", "Kernel 6.xx");
        buildLabel = build.LastGoodBuildLabel;
        nextBuildLabel = build.LastGoodBuildLabel;
        tfssvr.Dispose();
        while (buildLabel == nextBuildLabel)
        {
            TeamFoundationServer tfssvrNew = new TeamFoundationServer("http://se-s-0010048.se.abb.com:8080/tfs/", new System.Net.NetworkCredential("no-z8-BuildSy
            tfssvrNew.Authenticate();
            var buildServerNew = tfssvrNew.GetService<IBuildServer>();
            var buildNew = buildServerNew.GetBuildDefinition("RobotWare", "Kernel 6.xx");
            nextBuildLabel = buildNew.LastGoodBuildLabel;
            tfssvrNew.Dispose();
            Thread.Sleep(4000);

        }
        if (buildLabel != nextBuildLabel)
        {
            QueueBuild();
            continue;
        }
    }
}
```

Figure 4.12: RW6BuildListener Windows Service Snippet:

Extensions:

The RW6BuildListener service is just a prototype, and although it is capable of starting the execution of the generated test suites automatically it is not very useful in it's current state. At present, the service starts the test suite execution when it detects a new build, but it has no way of generating test cases or downloading and installing the new build. However, due to several constraints to the physical robots at the lab, this method is not yet ready for practical implementation. An extension has been developed, which in time will be incorporated into the RW6BuildListener Service allowing for the generation and execution test suites for the newly installed RW version.

This extension implements a method of generating the test suites automatically utilizing SpecExplorer.exe, which is a standalone executable incorporating the full functionality of Spec Explorer, but with a command line interface substituting the GUI. A snippet of this extension is depicted in 4.13, which demonstrates a verified and tested mechanism to generate test cases automatically. The inspiration for this method came from this post. [53]



```
static void GenerateTests()
{
    string libDir = @"C:\Data\DLLTestREST";
    string scriptPath = @"C:\Data\DLLTestREST\Config.cord";
    string testResultPath = @"C:\Data\TestResultPath";
    string strArgs = "/t:GenerateTests /l:" + libDir + " /a:RW6ModelRestService.dll,RW6ModelRestService.Sample.dll,websocket-sharp.dll,SuperSocket.ClientEngine.Co
    Environment.CurrentDirectory = "C:/Program Files (x86)/Spec Explorer 2010";
    Process proc = new Process();
    proc.StartInfo.WorkingDirectory = Environment.CurrentDirectory + "\\";
    proc.StartInfo.FileName = Environment.CurrentDirectory + @"\Specexplorer.exe";
    proc.StartInfo.Arguments = strArgs; //if no arguments comment this line
    proc.StartInfo.UseShellExecute = false;
    proc.StartInfo.CreateNoWindow = true;
    proc.StartInfo.RedirectStandardInput = true;
    proc.StartInfo.RedirectStandardOutput = true;
    proc.StartInfo.RedirectStandardError = true;
    proc.Start();
}
```

Figure 4.13: Future Extension to RW6BuildListener allowing automatic test generation:

Chapter 5

Results and Conclusions:

5.1 Results

The basis for the model-based testing systems developed in this thesis, is the paint protocol toward Robotware which is described briefly in Chapter 2. This thesis ultimately resulted in two model based testing systems, which each facilitate the automatic generation of test-cases based upon a model of the interface. However, the models are somewhat incomplete, as only a subset of all the possible paint commands were incorporated into the model. Several paint commands where determined to be obsolete, as they would never be called in practice by a PLC in an industrial plant. Examples of such obsolete functions include, Spy Log Start/Stop, System Position Teach and System Tool Teach, Execute Program by String etc.

A bullet point summary of the model-based testing framework developed in this thesis will follow;

- An abstract model of the Extended Command PLC Interface toward Robotware 5, giving an abstracted and simplified approximation of the SUT. A control interface toward the Extended Command PLC Interface, allowing for external control of the robot and functioning as an adapter layer (glue) between the model and the SUT.
- An abstract model of the RESTful Robot Web Service Interface toward Robotware 6, giving an abstracted and simplified approximation of the SUT. A control interface toward the RESTful Robot Web Service Interface, allowing for external control of

the robot and functioning as an adapter layer (glue) between the model and the SUT.

- Coord scripts for the two models, which implement the following functionalities: the input parameters for each method in the model, the state machines used for exploration of the model and the generation of the test cases.
- Test suites that conform to the Microsoft Visual Studio Test framework, allowing a relatively easy integration into the existing testing framework for ABB Robotics.
- A listening service that starts automatic execution of the test-suites when a new build is released, with an extension allowing for automatic test case generation (complementing the PLC model feature which provides model state initialization such that each generated test suite is customized to the present SUT state) and execution
- A number of different state machines for test case generation, including: Full Transition Coverage Long Traversal, Full Transition Coverage Short Traversal and Requirement Coverage.

The accompanying CD contains the detailed test logs for the final test run on this project, which can be viewed in Visual Studio. The test logs detail the run of 796 tests combined for the two different models, which consist both Transisition Coverage Long/Short traversal and Requirements Coverage. The CD also holds the exploration graphs and test suite graphs for all the state machines defined in the Coord script. The following section will present the various faults uncovered during the testing phases, detailing how the fault occurred, what was wrong and the actions taken to solve the fault. The faults and errors uncovered have been divided into four distinct and self-explanatory groups, where at least one example from each group was uncovered and will be presented. The four categories are as follows:

- Model faults
- Implementation faults
- Documentation faults
- SUT faults/bugs

5.1.1 Model-Faults:

MasterRequest Error:

Below is a snippet from the Spec Explorer debugging console window, taken during test-execution:

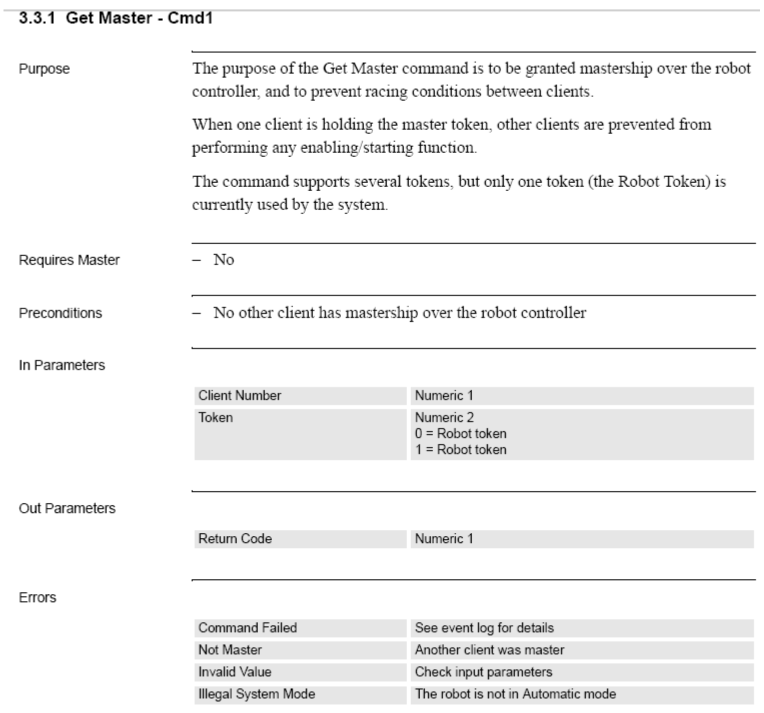
```
reaching state S5.

Conformance failure: none of the potential model steps match implementation step.

The potential model action(s):

return MasterRequest/3

Available return observation queue:

return MasterRequest/0

Event observation queue is empty.
```

This result is relatively self-explanatory. The model called the function MasterRequest and expected the return code 3 from the SUT, however, the actual return code from the SUT was 0. The model code for the MasterRequest function is shown below, along with the description of the function in the documentation:

Listing 5.1: Modelling of MasterRequest:

```
[Rule(Action = "MasterRequest(paintCommand, client)/3")]
                static void GetMasterFailedMaster(int paintCommand, int client)
                {
                        //int retCode;
                        Condition.IsTrue(!RobotError);
                        Condition.IsTrue(tokenIsHeld);  // SHOULD TRIGGER WHEN
                            TOKENISHELD, NAMELY WHEN MASTER IS GRANTED TO PLC
                            OR HELD BY ANOTHER PLC
                        //Condition.IsTrue(paintCommand == 1);
                        RobotError = true;
                        //retCode = NotMaster;
                        //return retCode;
                }
```

The error lies in an ambiguity with the Not Master error, as it was taken to mean that this error would be returned if the model requests Mastership twice. However, close examination shows that this error requires that mastership is granted to a second PLC. The model function was then updated to reflect this change, as demonstrated in 5.2.


3.3.1 Get Master - Cmd1

Purpose	The purpose of the Get Master command is to be granted mastership over the robot controller, and to prevent racing conditions between clients. When one client is holding the master token, other clients are prevented from performing any enabling/starting function. The command supports several tokens, but only one token (the Robot Token) is currently used by the system.
Requires Master	– No
Preconditions	– No other client has mastership over the robot controller

In Parameters

Client Number	Numeric 1
Token	Numeric 2 0 = Robot token 1 = Robot token

Out Parameters

Return Code	Numeric 1

Errors

Command Failed	See event log for details
Not Master	Another client was master
Invalid Value	Check input parameters
Illegal System Mode	The robot is not in Automatic mode

Figure 5.1: Description of MasterRequest in manual:

Listing 5.2: Modelling of MasterRequest Update:

```
[Rule(Action = "MasterRequest(paintCommand, client)/3")]
                 static void GetMasterFailedMaster(int paintCommand, int client)
                 {
                         //int retCode;
                         Condition.IsTrue(!RobotError);
                         Condition.IsTrue(tokenIsHeld && !MasterGranted);
                         //Condition.IsTrue(paintCommand == 1);
                         RobotError = true;
                         //retCode = NotMaster;
                         //return retCode;
                 }
```

This error facilitated the removal of the MasterRequest and MasterRelease actions from the model, which was done in a later revision.

5.1.2 Documentation Faults:

JobQueueShift return code mismatch between RW5 and RW6:

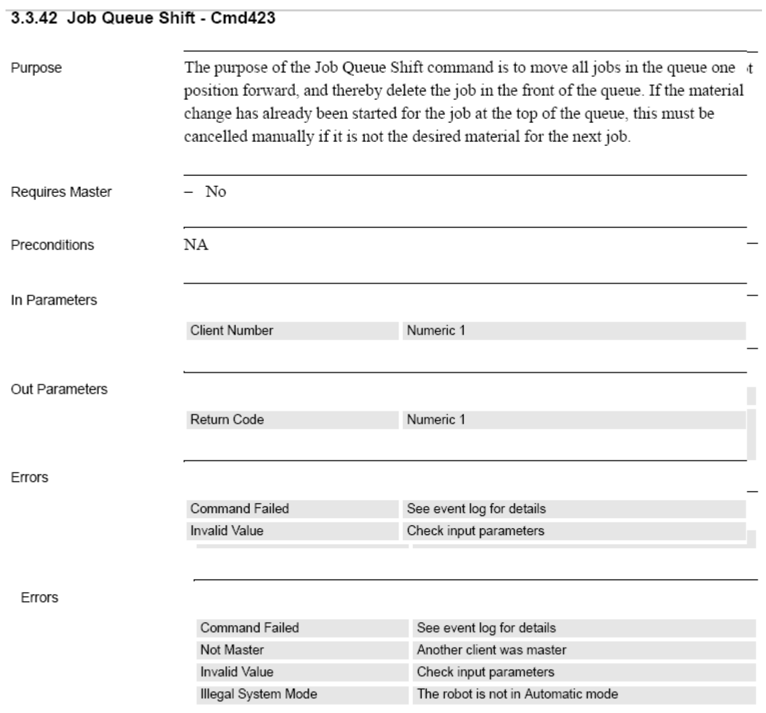
A snippet from the Spec Explorer Console window showing the error is depicted below:

```
reaching state S1.

Conformance failure: none of the potential model steps match implementation step.

The potential model action(s):

return JobQueueShift/-1

Available return observation queue:

return JobQueueShift/-7

Event observation queue is empty.

End executing test

Begin executing test TestCase15

reaching state S0.
```

Listing 5.3: Modelled Function - JobQueueShift Command Failed

```
[Rule(Action = "JobQueueShift(paintCommand)/-1")]
                static void JobQueueShiftFailedResource(int paintCommand)
                {
                        Condition.IsTrue(ModelState.JobQueue.Count < 1);
                        RobotError = true;
                }
```

Whether this error is a documentation fault or a model fault is possibly a matter of interpretation. In the model for RW5, the function was modelled as above and coincided with the response received from the SUT when attempting to shift a JobQueue that was already empty. As shown in 5.2, the manual only indicates -1 Command Failed as a valid response, and does not state that -7 Resource Unavailable is a valid return code for this function. However, it can be argued that -7 Resource Unavailable is the more logical return code, as it gives a better description behind the failure than "Command Failed". Attributed to outdated documentation, as the documentation has yet to be revised for changes implemented in Robotware 6.


3.3.42 Job Queue Shift - Cmd423

Purpose: The purpose of the Job Queue Shift command is to move all jobs in the queue one position forward, and thereby delete the job in the front of the queue. If the material change has already been started for the job at the top of the queue, this must be cancelled manually if it is not the desired material for the next job.

Requires Master: – No

Preconditions: NA

In Parameters

Client Number	Numeric 1

Out Parameters

Return Code	Numeric 1

Errors

Command Failed	See event log for details
Invalid Value	Check input parameters

Errors

Command Failed	See event log for details
Not Master	Another client was master
Invalid Value	Check input parameters
Illegal System Mode	The robot is not in Automatic mode

Figure 5.2: JobQueueShift - PLC manual:

5.1.3 Implementation Faults:

Test cases failing and being caught by following test case:

The Spec Explorer debug log and the Teraterm console log give contradicting information, as one clearly sees that the JobQueueShift command returns -1 on the Teraterm console. Looking at the Spec Explorer log, one sees that the model was expecting 1 as the result but that this result was not picked up until the subsequent action call, where it was misrepresented as the return code for the MasterRequest function. One attempt to solve this problem was to extract the line 'returnCode = 0' (initializing the returnCode) out of an IF-statement in the Initialize function, but the cause of this error lay in the PLC Main program. The InLongTest boolean variable was not set to false in the correct position, meaning that the 'locking' mechanism was not functioning correctly.

```
reaching state S3.
Conformance failure: none of the potential model steps match implementation step.
The potential model action(s):
return JobQueueShift/1
Available return observation queue:
return JobQueueShift/0
Event observation queue is empty.
End executing test
Begin executing test TestCase21
reaching state S0.
executing step 'call MasterRequest(1,1)'
OpcGroup.OnDataChange
OpcGroup.OnReadComplete
OpcGroup.OnWriteComplete
OpcGroup.OnDataChange
reaching state S1.
Conformance failure: none of the potential model steps match implementation step.
The potential model action(s):
return MasterRequest/0
Available return observation queue:
return MasterRequest/1
Event observation queue is empty.
End executing test
```

Figure 5.3: Spec Explorer Log of Error:

```
16:15:46:401181 Eiosig.c,6469:eiosig__signal__changed: Signal CommandToggle, value = 1, time stamp at 1389716146,401169 (s,us)
[Cmd423 :, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0] − 1
16:15:46:649403 Eiosig.c,7018:eiosig__write__signal: Signal CommandAck, write mode = 0, delay = 0, pulses = 0, value = 1
16:15:46:649415 Eiosig.c,6469:eiosig__signal__changed: Signal CommandAck, value = 1,
time stamp at 1389716146,649410 (s,us)
16:15:46:693151 Eiosig.c,6469:eiosig__signal__changed: Signal CommandToggle, value = 0, time stamp at 1389716146,693138 (s,us)
```

Figure 5.4: TeraTerm Console - Controller output via serial port:

5.1.4 System faults:

HttpContextFactory Error:

The fault uncovered here was the main payoff of this thesis, as it uncovered a mildy serious bug that had not been caught in previous tests. Not strictly a paint related bug, but a bug in the underlying architecture of Robotware 6, or more specifically the Robot Web Service Interface, which was uncovered by stressing the paint interface. An excerpt of the Teraterm console output is shown in 5.5 This seems to happen consistently when

```
18:02:33:943011 pntprot__cmd.c,203:pntprot__cmd__append: command append, client , command = 203, args = 0x19055fac,
num__args = 10
18:02:34:513091 pntprot__cmd.c,203:pntprot__cmd__append: command append, client , command = 303, args = 0x19055fac,
num__args = 10
14-02-03 18:02:39 MC0: rapi__resource.cpp 267 HttpMsgContextFactory Error create object
14-02-03 18:27:39 MC0: src \rapi__handler.cpp 484 Error in IPC-RECEIVE
14-02-03 18:27:39 MC0: src \rapi__handler.cpp 315 No response received from service
```

Figure 5.5: TeraTerm Console log of SUT bug:

the subscription group ID exceeds 51, and causes the controller to freeze, requiring a warm-start in order to reset the controller. The error seems to be attributed to the new RobApi 2 functionality, or more specifically related to the subscription service used to subscribe to resources. The bug has been reported to the software department in India, who are currently working on resolving it.

SUT Behavior Not Updated:

The following SUT bug is an error in RW6 that was discovered when the Requirement-Coverage test suite was created for the RobotWebService interface, and occurs when JobQueuePeek is called to verify that a certain job in the model job queue conforms with the actual job in the SUT job queue. The actual test is depicted in 5.7. As demonstrated, the model expects a certain result but fails due to the expected result not conforming to the actual result. One parameter is missing, and the order of the parameters is scrambled. The modelled action for the JobQueuePeek command is shown in 5.4.

Listing 5.4: Modelled Function - JobQueuePeek

```
[Rule(Action = "JobQueuePeek(paintCommand, jobNo)/ result")]
static string JobQueuePeekOK(int paintCommand, int jobNo)
{
        Condition.IsTrue(ModelState.JobQueue.Count > jobNo);
        Condition.IsTrue(isConnected);
        Condition.IsTrue(ModelState.JobQueue.ElementAtOrDefault(jobNo) != null);
        Job job = ModelState.JobQueue[jobNo];
    Requirement.Capture("Peek into a job performed");
    return job.ToString();
}
```

5.2 Conclusion:

Developing a model for any SUT is a very difficult task, exacerbated by the contradicting requirements of both being abstract and detailed. Although the models developed in this thesis was relatively successful, better results could possibly be achieved by splitting the entire model up into smaller models, possibly even developed by different developers, and then reintegrated into one, complete model. As it stands, the current models are sufficient to generate test-cases which provide a decent coverage of the possible paint commands and apply a significant amount of stress to the controller.

One drawback to the models developed in this thesis, is that due to the lack of detail presented in the PLC Manual, most of the model logic was directly based on the source code. Naturally, this should not happen as the model and the source code should be independent. It is logical to assume that due to the fact that the model was, in essence, 'modelled' on the source code less faults would be found as the model directly reflects the operation of the SUT. The implementation of successful model-based testing systems require formal documentation defining the high-level functionality of the SUT, which both the source code and the model are based on.

Although this testing framework is complete and, in theory, ready to be fully integrated into the existing test methodology in ABB Robotics, it must be stressed that what is presented in this thesis is only a working prototype. Some more work is needed befoe the system is production ready, demonstrated by some minor problems in the implementation. An example of such a problem is present in the OPC client which functions as an adapter between the PLC model and the SUT, where the application sometimes crashes during testing for unexplained reasons. Whether the fault ultimately lies within the adapter layer, or within the PLC program itself is uncertain, but this has to be sorted out before the system can be put into production.

One of the more difficult and time-consuming tasks during this project was attempting to make a 'well-behaved' exploration graph, where well-behaved implies that the exploration graph is finite (such that test suites can be generated) and relatively concise (meaning that it does not contain thousands of useless test cases). One of the main solutions to the graph exploration (and subsequent test case generation) was the addition of requirement capturing and thus being able to generate requirement coverage. Thus, only test cases that work towards satisfying distinct requirements are generated, nullifying the generation of vast amounts of obsolete test cases. Ultimately many more requirements should be introduced, ideally with requirements for error states. Theoretically, this would allow one to transition between all the different error messages and test whether the correct error message is given for a certain state. Naturally, this would also require the need for a more detailed model.

5.2.1 Future Work:

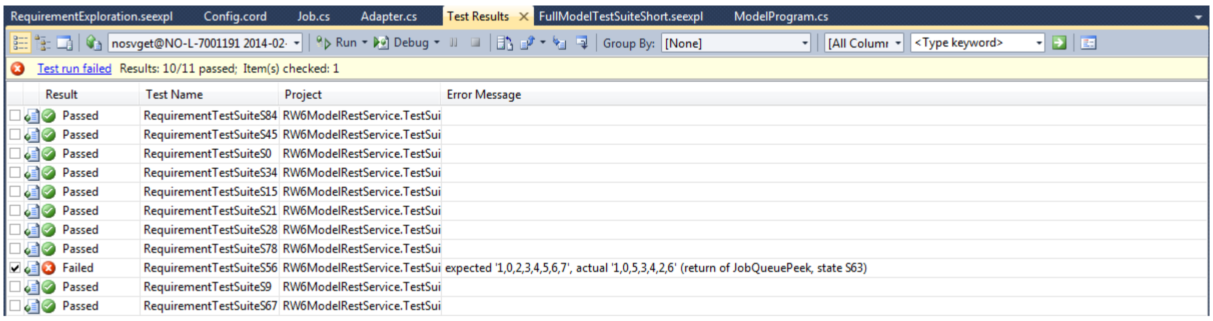
During the end of this project, development had started on providing support for RAPID(the programming language used on the robot programs) commands in the model-based testing framework. This would allow the testing framework developed in this thesis to test further Paint specific functions that are not exposed through any other interface, thus possibly allowing this Model-Based Testing solution to fully replace the existing testing frameworks for Robotware Paint at ABB Robotics.

Several RAPID functions were incorporated into the Adapter layer of the REST interface, with the actions being modelled in the model class. This required creating additional classes in the adapter layer, in order to communicate with the Robotware interface, and automatically generate RAPID programs based on the commands that were to be tested. The generated programs were then uploaded to the controller via FTP, and executed via the REST interface. An eventhandler was developed in order to get events directly from Robotware, in order to process whether the executed RAPID command conformed to the expected result.

Preliminary testing has proven positive, and the plan is to continue to develop this additional support in the future.

```
reaching state S5.
checking step 'return MaterialChangeResume/0'
reaching state S6.
executing step 'call ApplicatorDisable(303,2)'
[REQUESTHEADERS] :
PUT http://10.47.89.95/rw/paint/command?action=send
Host: 10.47.89.95
Cookie: ABBCX=63
Content-Length: 77
[RESPONSEHEADERS]
200 OK
Keep-Alive: timeout=10, max=186
Connection: Keep-Alive
Content-Length: 379
Cache-Control: no-cache="set-cookie"
Content-Type: application/xhtml+xml
Date: MON, 03 FEB 2014 18:02:34 GMT
Set-Cookie: -http-session-=::http.session::a91714d3008301dd6bd966ff1859aed9; path=/;
domain=10.47.89.95
Server: Embedthis-http
——————————————
reaching state S7.
checking step 'return ApplicatorDisable/-7'
reaching state S8.
executing step 'call Disconnect()'
Subscription removed
Websocket closed
reaching state S9.
checking step 'return Disconnect/True'
reaching state S10.
End executing test
Begin executing test TestCase41
reaching state S0.
executing step 'call Connect()'
Subscription group id=51
Websocket opened
Error subscribe
reaching state S1
. checking step 'return Connect/True'
reaching state S2.
executing step 'call MaterialChangeResume(203)'
[REQUESTHEADERS] :
PUT http://10.47.89.95/rw/paint/command?action=send
Host: 10.47.89.95
Cookie: ABBCX=64
Content-Length: 77
——————————————
[RESPONSEHEADERS]
The operation has timed out
Error:System.NullReferenceException: Object reference not set to an instance of an object.
at MSDigestHttpWebRequest.DigestHttpWebRequest.LogResponseError(WebException ex) in
c:\data\visualstudio2010\Projects\RW6ModelRestService\RW6ModelRestService.Sample\digestClient.cs : line327
at MSDigestHttpWebRequest.DigestHttpWebRequest.GetResponse(Uri uri, CookieContainer cookieContainer, String
connectionGroup) in
\data\visualstudio2010\Projects\RW6ModelRestService\RW6ModelRestService.Sample\digestClient.cs : line124
at RW6ModelRestService.Sample.Adapter.ExecuteAction(Int32 command, Int32 arg1, Int32 arg2, Int32 arg3, Int32 arg4, Int32
arg5, Int32 arg6, Int32 arg7, Int32 arg8, Int32 arg9) in
C : \Data\VisualStudio2010\Projects\RW6ModelRestService\RW6ModelRestService.Sample\Adapter.cs : line575
[REQUESTHEADERS] :
GEThttp : //10.47.89.95/rw/paint/commandresult?serial = 107

Content-Type: application/x-www-form-urlencoded; charset=utf-8
```

Figure 5.6: Spec Explorer Debug Log:



Test run failed Results: 10/11 passed; Item(s) checked: 1

	Result	Test Name	Project	Error Message
☐	Passed	RequirementTestSuiteS84	RW6ModelRestService.TestSui	
☐	Passed	RequirementTestSuiteS45	RW6ModelRestService.TestSui	
☐	Passed	RequirementTestSuiteS0	RW6ModelRestService.TestSui	
☐	Passed	RequirementTestSuiteS34	RW6ModelRestService.TestSui	
☐	Passed	RequirementTestSuiteS15	RW6ModelRestService.TestSui	
☐	Passed	RequirementTestSuiteS21	RW6ModelRestService.TestSui	
☐	Passed	RequirementTestSuiteS28	RW6ModelRestService.TestSui	
☐	Passed	RequirementTestSuiteS78	RW6ModelRestService.TestSui	
✓	Failed	RequirementTestSuiteS56	RW6ModelRestService.TestSui	expected '1,0,2,3,4,5,6,7', actual '1,0,5,3,4,2,6' (return of JobQueuePeek, state S63)
☐	Passed	RequirementTestSuiteS9	RW6ModelRestService.TestSui	
☐	Passed	RequirementTestSuiteS67	RW6ModelRestService.TestSui	

Figure 5.7: JobQueuePeek SUT Bug:

Chapter 6

Bibliography

[1] -http://en.wikipedia.org/wiki/Software_bug

[2] - http://en.wikipedia.org/wiki/Software_metric

[3] - http://en.wikipedia.org/wiki/Software_bug

[4] - http://research.microsoft.com/en-us/projects/specexplorer/

[5] - http://blogs.msdn.com/b/specexplorer/archive/2010/04/21/actions-step-and-rules.aspx

[6] - http://blogs.msdn.com/b/specexplorer/archive/2010/04/21/actions-step-and-rules.aspx

[7] - http://blogs.msdn.com/b/specexplorer/archive/2010/04/21/actions-step-and-rules.aspx

[8] - M.Utting & B.Legeard, Practical Model-Based Testing - A Tools Approach, Morgan Kaufmann Publishers, 2007

[9] - IEEE Software Engineering Body of Knowledge, SWEBOK, 2004

[10] - M.Utting & B.Legeard, Practical Model-Based Testing - A Tools Approach, Morgan Kaufmann Publishers, 2007, page 7

[11] - The American Heritage Dictionary of the English Language, Houghton Mifflin, 2000

[12] - Reference Manual : Paint Commands and PLC Interface, ABB Robotics, 2013

[13] - M.Utting & B.Legeard, Practical Model-Based Testing - A Tools Approach, Morgan Kaufmann Publishers, 2007, page 240

[14] - M.Barnett, K.Rustan, M.Leino and W.Schulte. The Spec# programming system: An overview, Proceedings of the International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, 49-69, Springer-Verlag, 2004

[15] - Margus Veanes, Colin Campbell, Wolfram Schulte, and Pushmeet Kohli, On-The-Fly Testing of Reactive Systems, 2005

[16] - http://www.ics.uci.edu/ fielding/pubs/dissertation/rest_arch_style.htm, Chapter from Roy Fielding Doctoral Dissertation

[17] - http://en.wikipedia.org/wiki/Digest_authentication, Digest Authentication

[18] - https://tools.ietf.org/html/rfc6455, Websocket Protocol RFC6455

[19] - J.Tretmans, Model Based Testing With Labeled Transition Systems

[20] - N. Holt, Empirical Evaluations on the Cost-Effectiveness of State-Based Testing - Industrial Case Studies and Extensible Tool, University of Oslo, 2012

[21] - T. Hoeve, Model Based Testing of a PLC Based Interlocking System, University of Twente

[22] - Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer - M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillman, L. Nachmanson, 2007 Microsoft Research

[23] - P. Pinheiro, A. Endo, A. Simao - Model-Based Testing of RESTful Web Services Using UML Protocol State Machines

[24] - http://www.amci.com/tutorials/tutorials-what-is-programmable-logic-controller.asp

[25] - http://www.pacontrol.com/download/plcbook5_0.pdf

[26] - http://webstore.iec.ch/webstore/webstore.nsf/Artnum_PK/47556, IEC Standard Homepage

[27] - http://en.wikipedia.org/wiki/Ladder_logic, Wikipedia Article - Ladder Logic

[28] - http://claymore.engineer.gvsu.edu/ jackh/books/plcs/chapters/plc_st.pdf

[29] - T. Hoeve, Model Based Testing of a PLC Based Interlocking System, University of Twente

[30] - http://en.wikipedia.org/wiki/Sequential_function_chart

[31] - http://en.wikipedia.org/wiki/OLE_for_Process_Control

[32] - http://msdn.microsoft.com/en-us/library/ee620451.aspx

[33] - AC500 Control Builder User Documentation, Chapter 8 - OPC

[34] - http://blogs.msdn.com/b/specexplorer/archive/2011/08/02/how-to-initialize-the-initial-state-from-an-external-source.aspx

[35] - http://msdn.microsoft.com/en-us/library/microsoft.xrt.runtime.nativetypeattribute.aspx

[36] - http://msdn.microsoft.com/en-us/library/ee620419.aspx

[37] - http://msdn.microsoft.com/en-us/library/ee620467.aspx

[38] -http://msdn.microsoft.com/en-us/library/ee620446.aspx

[39] - http://msdn.microsoft.com/en-us/library/ff635843.aspx

[40] - http://msdn.microsoft.com/en-us/library/ee620418.aspx

[41] - http://msdn.microsoft.com/en-us/library/gg521168.aspx

[42] - http://msdn.microsoft.com/en-us/library/ee620422.aspx

[43] - http://msdn.microsoft.com/en-us/library/ee620427.aspx

[44] - http://en.wikipedia.org/wiki/Team_Foundation_Server

[45] - http://en.wikipedia.org/wiki/Windows_service

[46] - http://msdn.microsoft.com/en-us/library/ee620414.aspx