TUHH
Technische Universität Hamburg-Harburg

Bachelor Thesis

Max Schürenberg

Scalability Analysis of the Simulink Design Verifier on an Avionic System

August 8, 2012

supervised by:
Prof. Dr. Sibylle Schupp

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg

TUHH STS
Software
Technology
Systems

The work on this thesis was done in partnership with
Airbus Operations Ltd. and Airbus Operations GmbH

Abstract

In recent years, the application of model-based requirement engineering has been considered very useful and has found its way into avionic system development and certification. New standards explicitly allow and define the use of model-based design in the development process. So far, simulation and testing is the only way to assure that a model meets its required behaviour. Model checking is another promising approach for that. The Simulink Design Verifier is a new commercial tool that allows the application of model checking within the established Matlab/Simulink development environment. The aim of this thesis is to analyse if the Simulink Design Verifier is applicable on a fuel tank management model for a modern long range aircraft. The focus of the analysis is on the scalability of the tool. The analysis revealed that proving simple properties is possible and scales well, while the automated test case generation can not be applied due to design constructs within the fuel management model as well as bugs in the Simulink Design Verifier.

Contents

1 Introduction

In recent years, the application of model-based requirement engineering has been considered very useful and has found its way into avionic system development and certification[26]. New standards explicitly allow and define the use of model-based design in the development process. Furthermore, the use of formal verification techniques within model based-design is recognized and described by the certification authorities for avionic systems [17].

So far, simulation and testing is the only way to assure that a model meets its requirements. Formal verification and model checking follow a completely different approach for that purpose. Formal verification is based on the insight that programs and systems may be viewed as mathematical objects with predictable behaviour. The verification problem is: Determine whether a given program M satisfies a specification h.

Model checking is derived from formal verification. It provides methods to verify a temporal property on a finite state system via efficient graph search. Model checking has its origins in Harvard in the early 1980's [10]. In previous years, various model checking tools for real-world application have been introduced, such as NuSMV[16], SPIN[24] or the SCADE Design Verifier[19].

With the Simulink Design Verifier (SLDV), Mathworks has released another commercial tool that brings model checking into industrial application. It can verify models that are created with tools from the Matlab suite. The company claims that first attempts to apply the SLDV in an industrial environment were successful and profitable [12]. However, the functionality of the tool is only shown on small models. Recent attempts to apply the SLDV in the automotive and avionic industry either showed poor scaling and strong limitations on the applicability [14][23] or even failed completely [26]. In 2011, Mathworks released a new version. The aim of this thesis is to analyse if the SLDV is applicable on a fuel tank management system for a modern Airbus long range aircraft. The main focus of attention is the scalability of the tool and the usefulness of the gained results.

In section 2, I give a short introduction into model-based design. I sketch the development process, present the tools that are used within Airbus and explain the use of model coverage. The key aspect of this thesis is *scalability*. I explain and define that term in section 3. In section 4 I concentrate on the Simulink Design Verifier, explain model checking and the algorithm behind the SLDV and give a program introduction. The core analysis is outlined in section 5, where I present my strategy to analyse the model and show the problems and limitations I encountered during the analysis. In section 6, I present and discuss the data that I collected during the analysis in terms of scalability. Finally, I end up with a conclusion in section 7, where I summarize the results and give a recommendation for future work on the SLDV.

2 Model-Based Design

In this section, I present the fundamentals of model-based design. I explain why model-based design can facilitate the development process significantly, at which points graphical tools can be used in the system development process at Airbus and where the SLDV is applicable.

2.1 Development Process

Because of the necessity of highly reliable aircraft to ensure passengers safety, the development process of avionic systems is subject to strict regulations. In Europe, the certification process is controlled by the European Aviation Safety Agency (EASA). The definition of the development process i.e. how certain documents and certifications can be achieved, is given by the Radio Technical Commission for Aeronautics (RTCA) and its European pendant, the European Organization for Civil Aviation Equipment (EUROCAE). For system development, these institutions provide the documents ARP4754(RTCA) and ED-79(EUROCAE) for the overall development process. Together with the DO-178B and the new DO-178C which describe the software development and certification process, all policies and guidelines are bundled in the Airbus internal documents ABD0200 and ABD0100. According to this, the system development process at Airbus is based on a sequence of

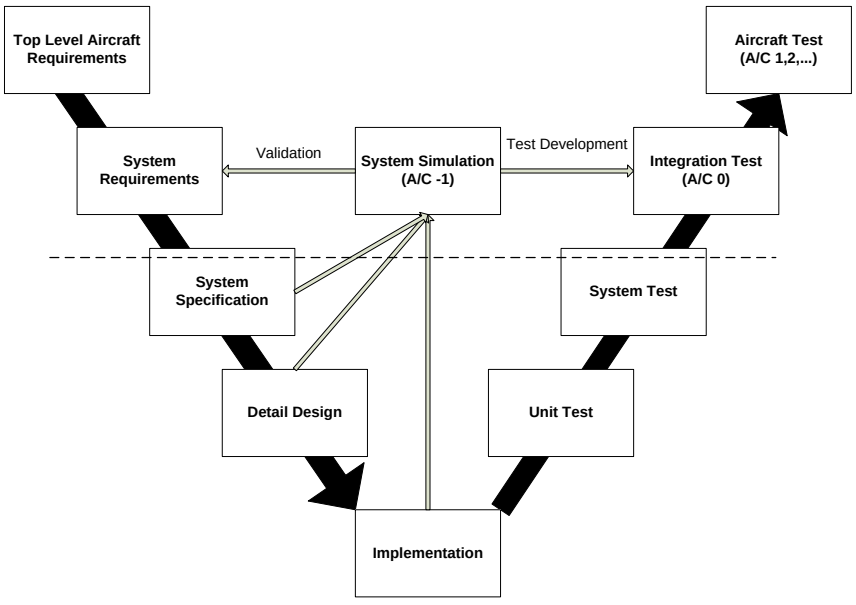

Figure 2.1: A V diagram of system development as in [23]

design and implementation steps. Each of these steps outputs a document that drives the successive step. This workflow is called V model. A diagram of that model can be seen in figure 2.1. The V shape of the diagram illustrates two main aspects of the process.

One is the top-down approach with a narrowing scope of the design steps, where high-level requirements are split into more and more detailed low-level requirements down to concrete design descriptions, followed by a wider scope of testing and integration, where small parts are tested in isolation, integrated into their environment and then tested again for their functionality in conjunction with other system parts. The other main aspect of the V model is the relationship between design and test. Each design step has a corresponding step in the integration and testing phase. Tests can therefore be directly derived from the design document.
The main advantage of such a development process is the requirement traceability. At every stage of design and implementation, all parts, subsystems and source-code snippets of the system and their corresponding high-level requirements can be traced in both directions, which helps to assure a high level of safety and confidentiality. However, it also puts a huge burden on the system design. If a high-level requirement turns out to be incorrect in the corresponding implementation step, the whole process must be done again. NASA published a famous study [8] that shows how the cost of fixing failures increases with the development process stage in which the specific failure was found (see figure 2.2). More than that, Airbus's business strategy is to act as a developer and integrator. Every

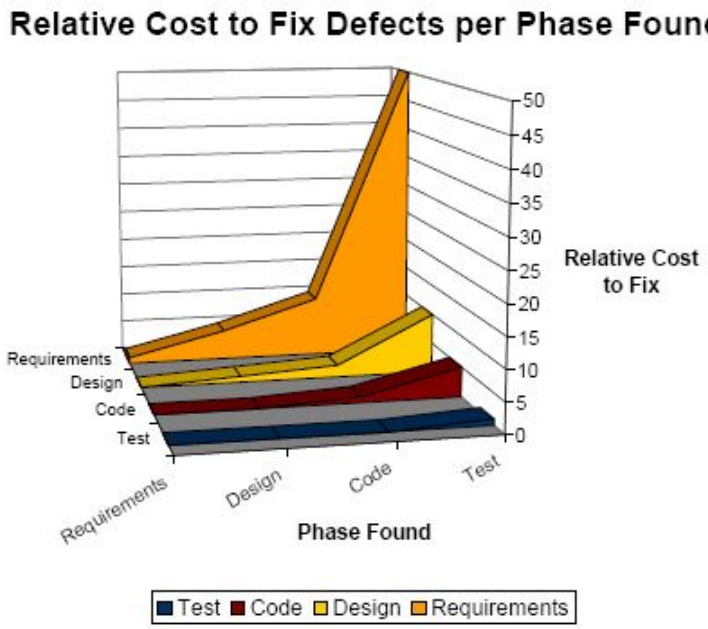

Figure 2.2: NASA study on failure costs as in [8]

fixed implementation of a system or a structure is given to a subcontractor. In figure 2.1, the dotted line marks the interface between the responsibilities of Airbus (above the line) and subcontractors (below the line). In order to prevent misunderstandings at that interface and to ensure that Airbus can not be blamed for failures that occurred on the subcontractor's site, a good system design is critical. It is therefore necessary to ensure in every possible manner that the specification of a system is correct, valid and unambiguous. Model-based design is a powerful aid for that challenge.
A model can help as an executable specification. Textual requirements are often ambiguous. Consider the low-level requirement below:

```
The Left Valve open signal shall be set to true when any of the following
conditions is true:
-The Valve status is NO FAILURE and the Left Tank active signal is
TRUE
```

The problem which arises is that there are separate 'Valve status' signals for the left and right valve, therefore the subcontractor has to interpret the specification. A model would consist of a statechart (see section 2.2.2 on the following page) that clearly describes how the 'Overflow Condition' signal is set.

In the context of this thesis, the most important advantage of a model is the fact that it can also serve as a virtual prototype. That prototype can be simulated and fixed tests can be generated before any real system implementation is carried out. Consider figure 2.1 again to see how system simulation can improve the system development process. In the specification phase, the model is built according to the requirements. Before entering the detail design phase, that model can be executed and tests can be performed to ensure the right functionality according to the requirements and to validate the requirements according to the observed system behaviour. If incorrect requirements are detected, it is only the specification that needs to be adjusted. Furthermore, the test cases that were designed for the model can be used to derive the actual integration tests in later phases of the development process. To sum up, model-based design is a very powerful strategy to avoid failures in early design stages. If it is possible to ensure that the model is completely correct, then failures in the real system would be also unlikely. Model checking is not a replacement, but a powerful addition to simulation and testing. Both are carried out in parallel. In section 4, I point out how model checking differs from testing and how it can help to improve the correctness of a model.

2.2 Graphical Models

There are several ways to build a model. The system behaviour can be described via pseudo code or graphical modelling tools can be used to illustrate the system architecture. In the field of graphical modelling, two fundamental graphical modelling paradigms can be found: Block diagrams and statecharts. Within Airbus, system models are built with graphical modelling tools, particularly with *Simulink* for the block diagrams and *Stateflow* for the statecharts. Both tools are extensions for the *Matlab* suite, provided by the *MathWorks Inc*[15]. The choice for graphical modelling tools results from the general advantages that graphical representations have over source code. Complex relations between components and signals can be abstracted to easy-to-read blocks and connectors and are therefore easier to understand and maintain. In the following sections, the main functionalities of Simulink and Stateflow are explained based on [13] and [2].

2.2.1 Simulink

A Simulink block diagram consists of system blocks that are connected by lines to represent the dataflow between them. A block is a set of equations which describe the relationship between input and output signals and the state variables. Simulink computes the time-based relationship between signals and state variables in the whole system. Elementary blocks such as controllers, logical functions, sinks and sources can be obtained directly from the library browser. The modelling of more complex equations is possible using Matlab and C-Code. An example from the Airbus Fuel System Modelling Environment (AFSME) is shown in figure 2.3 on the next page. To keep a certain level of clear arrangements,

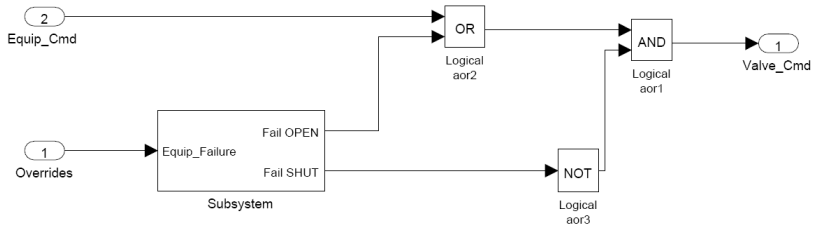

Figure 2.3: Signal Routing with Simulink

blocks can be grouped into subsystems. Therefore, Simulink allows a hierarchical system structure. It is possible to deal with discrete as well as continuous systems. Parameters like start and stop time and number of timesteps for the simulation can be manipulated by the user.

2.2.2 Stateflow

Stateflow is an extension to Simulink. Adding a Stateflow block to a Simulink diagram, enables the user to control the input and output behaviour via statecharts. Statecharts are a variation of the finite state machine visualization after [11]. They can be used to describe how a system with a finite set of states can change from one mode to another. An example statechart from AFSME can be seen in figure 2.4. States can either be exclusive, which

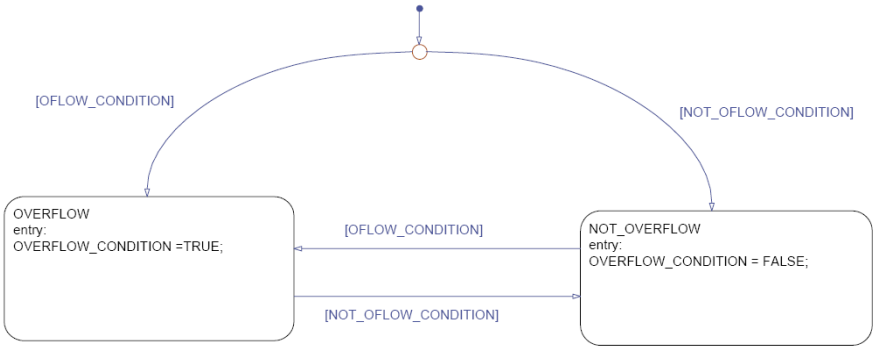

Figure 2.4: Statechart in Stateflow

means that only one state can hold at a time, or parallel. Parallel states are indicated by a dotted frame. A state consists of the following optional entries: *entry, during, exit, on event1, bind.* Events can be passed to these entries, that are than triggered at the specific timestep (e.g. on entry to the state). Transitions are represented by connecting arrows. Unconditional transitions are not labelled and will be executed with a delay of one timestep. Conditional transitions are executed if their corresponding condition holds. In square brackets, an event is defined. If that event holds during program execution, the active state changes to the respective state. With conjunctions and loops, more complex transitions can be modelled.

2.3 Model Coverage

Coverage analysis is a static analysis technique that measures the completeness of testing. It analyses the program execution over time and records which parts of the system or code under test are actually used. This technique is referred to as *white box testing*[20]. There are different types of coverage. In the following, these types are presented using the

```
[1]  d = 1;
[2]  if(a>0 && b>0){
[3]    d = 0;
[4]  } else {
[5]  }
[6]  p = d;
[7]  return d;
```

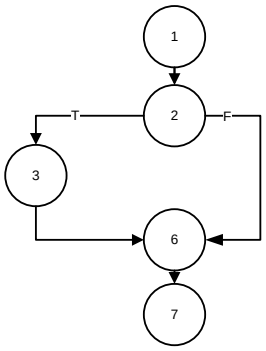

Figure 2.5: Pseudo Code Snippet to illustrate code coverage

example code and control-flow graph in figure 2.5, according to [1].

Statement coverage is the weakest condition. Statement coverage claims that every decision has to be executed at least once, so that every node in the respecting graph is reached. In the example, a test case with a = 1 and b = 1 or any other combination of numbers that leads to a true outcome of the if-statement would satisfy this criteria.

Decision coverage claims that every decision has to take every possible outcome. In the example, two test cases that lead the program to the true branch as well as to the false branch are necessary, e.g. Test Case 1: a=1, b=1 and Test Case 2: a=1, b=0. Looking at the control-flow-graph, decision coverage covers the whole graph. However, it leaves out the problem that can result from logical operations that are used as conditions.

Condition coverage considers that problem. Condition coverage claims that every condition in a decision executes at least once with a true and a false value. In the example, Test Case 1: a=1, b=1, Test Case 2: a=1, b=0 and Test Case 3: a=0, b=0 would satisfy this criteria.

MCDC stands for modified condition-decision coverage and is the most strict type of coverage. It was originally developed at Boeing [4]. It claims all rules from decision and condition coverage and adds the claim that every single condition has to be tested separately for its affects on the program execution while the other inputs are held constant. In the example, three test cases would be necessary. Test Case 1: a=1, b=1, Test Case 2: a=1, b=0, Test Case 3: a=0, b=1.

To a certain extent, model coverage is more complicated than code coverage due to a larger range of possible control flow constructs. However, in the context of the thesis, I will just deal with statecharts, so I will only need to consider the characteristics of coverage in state diagrams. In state diagrams, transitions are the central components that drive the control flow. In Stateflow, conditional transitions are executed in a specific order that is predicted by the developer. Unconditional transitions are executed by default after every other transition has been tested false. Transitions in Stateflow are equal to statements in source code. That means in order to reach decision coverage, every conditional transition

has to be tested as true (the transition will be taken) and as false (another transition will be taken or the state will hold).
On a small example, test cases that achieve a certain level of coverage can easily be found heuristically. When it comes to bigger models, that process becomes hard and time consuming. In section 4.3, I will show how the SLDV can also help to facilitate test case generation.

3 Scalability

The term *scalability* is frequently used in computer science. It can be found in highly different domains such as parallel computing, software processes, model checking and communication protocols. However, a strict definition of its characteristics is still missing, while several papers on that subject are trying to present a more systematic way of defining scalability [9] [5]. Therefore it is necessary to have a look at how scalability is valued in different domains and to clearly define scalability in the context of this thesis.

3.1 Use and Definitions of Scalability

In general terms, a scalable system is a system that handles an increasing amount of work gracefully and/or can be enlarged to handle that work. Speaking of a system that does not scale means that the resource-costs (memory usage, computation time, human-machine interacting) of coping with increasing work would rise exponentially or that the system can not even handle this increasing amount of work. Increasing resource-costs or even the need for replacement of the software almost automatically lead to increasing financial costs. Therefore, scalability is often a requirement in the design process of systems and software. In university lectures on operating systems, scalability is mentioned as one of the key quality characteristics [27].
[5] tries to specify scalability into four different subsections: *Load scalability* for graceful performance under heavy and light load, *space scalability* for a sublinear relation between number of supported objects and memory consumption, *space-time scalability* for graceful performance with increasing number of supported items and *structural scalability* for a system architecture that does not constrain growth in the number of supported objects, e.g. through finite a address space. However, the given definitions are still very vague and are not universally applicable for the different contexts in the field of computer science. Still, the given approach requires an individual definition of scalability.
[9] considers the fact that a universal definition of scalability can hardly be given and proposes a framework for the characterization and analysis in each specific context. Here, scalability is defined as:

> a quality of software systems characterized by the causal impact that scaling aspects of the system environment and design have on certain measured system qualities as these aspects are varied over expected operational ranges.

The system stakeholder has the responsibility to define scalability variables from the following cluster: *Scaling dimensions* for the independent variables of the system that can actually be manipulated for the scalability analysis, *non-scaling variables* for the characteristics of the application domain or the machine that are fixed and *dependent variables* for the variables that are affected by changes of the independent variables.

3.2 Defining Scalability in the Context of the Thesis

I consider the proposed framework in [9] as to be very useful, so I will use it to define scalability in the context of the thesis. I define the following variables:

Scaling dimensions: number of states in the statecharts, number of transitions in the statecharts, number of verification objectives

The scaling dimensions are chosen due to the analysis strategy picked in section 5.1. In a first part, the number of verification objectives is fixed and the numbers of statecharts and transitions are constantly increased. In a second round, when the model has reached its original size, the number of verification objectives is then increased.

Non-scaling variables: CPU, system memory

The analysis is performed on a single machine with the resources of a typical engineering workplace. The aim of the thesis is to prove if the SLDV is applicable in the 'daily work' of system development, therefore it has to consider the average equipment available. Further work could also consider the analysis on different machines. The exact data of the machine is:

- Intel Xeon CPU W3520 @ 2.67 GHz quad-core
- 3.23 GB RAM
- Windows XP Professional Version 2002 Service Pack 3

Dependent variables: overall computation time, analysis time, CPU-usage, memory usage

The overall computation time is the time that the Matlab process stays idle. That includes the compilation, translation, verification and report generation. Most of these parameters are not specific to the SLDV, however they are of interest for a feasibility analysis. The analysis time is the actual time the SLDV claims to need for the analysis. The CPU-usage is of interest to see if and how the CPU requirements increase with a growing model. Same applies for the memory usage. To reach a good scalability, these parameters are expected to show linear behaviour in relation to the scaling dimensions.

4 Simulink Design Verifier

The Simulink Design Verifier is an extension to Simulink. It uses formal verification methods to assist developers in verification and validation of their Simulink and Stateflow models. The SLDV was introduced in Matlab 2007 and renewed under the release 2.0 in Matlab 2011.
The use of formal verification in model-based design is a qualitatively new approach to detect errors. Testing has a severe limitation on ensuring a program's correctness, which is posed by a quote of Edsger W. Dijkstra in 1969:

> Testing shows the presence, not the absence of bugs. [6, p. 16]

The strategy of testing is to trigger the system with very specific inputs and to compare the results from that simulation with the results that were expected. A test case therefore covers a concrete possible execution trace of the program. The results of a test case allow the judgement over the behaviour of the system for exactly these inputs, but they can not be used for any further predictions, which is meant by the quote stated above. If a correct test case finds a failure, the program is obviously faulty, but the fact that a test case does not find a failure does not mean necessarily that the system is correct. More than that, a failing test can also be the result of incorrect test cases and expected results, which puts an additional downside on the confidentiality of testing. To gain a certain level of confidence in the system's functionality, as many test cases as necessary should be written in order to cover every possible execution path of the system.
Model checking follows a completely different approach. Instead of simulating the system, it abstracts the system into a deterministic finite state machine and verifies that abstraction against properties that are given in temporal logic. The problem is then to prove that the property holds within the defined temporal range.
The SLDV is a 'push-button tool', therefore an understanding of the underlying algorithm is not necessary for its application. The SLDV as well as the underlying proof engine, the Prover Plug-In by Prover Technology AB [18], are commercially distributed tools, so details on implementation and functionality are not available to the public. Nevertheless, I will give an overview over model checking with the SLDV and allow a closer look at the underlying algorithm.

4.1 Model Checking

The process of model checking consists of three main steps, which are illustrated in figure 4.1 on the following page and explained below based on [7]:

Modelling The first step is to transfer the model into a formal representation that can be used by the model checking tool. The structure widely used in literature about model checking is called a *Kripke structure*. It consists of a set of states, a set of transitions and a labelling function that relates each state to a set of properties that holds in that state. The representation of the Kripke structure in the latter model checking algorithm is crucial to its performance when it comes to larger state spaces. In *explicit* model checking algorithms, the Kripke structure is represented by a labelled, direct graph. When it comes to systems with many concurrent parts, the number

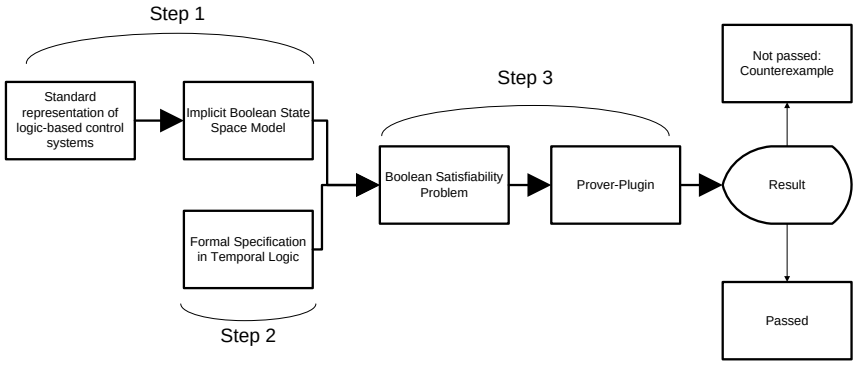

Figure 4.1: Steps Within Model Checking

of states in that transition graph gets too large to handle. With *symbolic* model checking, a more compact, boolean representation of the model is used. The symbolic representation captures loops and regularity in the state space graph. The ordinary symbolic representation is a *Binary Decision Diagram (BDD)*. The underlying proof engine within the SLDV uses an alternative to BDDs, however, due to the proprietary nature of the tool, it is not known how the abstraction is done.

Specification The next step is to inform the model checker about the properties that the model has to satisfy. This is done in *temporal logic*, a logical formalism that allows to describe the ordering of events in time without explicitly mentioning exact times. There are several different types of temporal logic. Linear Time Logic (LTL) is a temporal logic that allows to describe events along a single computation path, so for all possible paths from a state. Computation Tree Logic (CTL) is a branching-time temporal logic; here, it is possible to express properties for all paths or for only one possible path from a state. Finally, CTL* combines LTL and CTL.

Verification The model checking problem is introduced: Given the model M and the specification f in temporal logic, prove if M satisfies f at all time. In the underlying engine, the model and the specification are merged propositional logic and the model checking problem turns into a Satisfiability Problem (SAT): Search for a combination of parameters that can evaluate the formula to true. The model checking algorithm returns a counterexample if the given property could not be satisfied.

4.2 Stålmarck's Proof Procedure

It is known from the documentation of the SLDV, that the Prover Plug-In is based on Stålmarck's proof procedure which was patented in 1992. In the following, the concepts of that proof procedure are presented following a tutorial which was released by Gunnar Stålmarck and Mary Sheeran in 2000 [21][1].
Stålmarck's proof procedure is a tautology checker for propositional logic. The method introduces *triplets*, a special representation for formulas in propositional logic.

Triplets Triplets can be defined as the following: Let x, y and z be variables. A triplet is a 3-Tupel (x, y, z) which is an abbreviation for the logical proposition $x \leftrightarrow (y \rightarrow z)$.
In conjunction with the logical operator $FALSE$, represented by a 0, triplets are a function-

[1] The explanation of the algorithm is based on the given paper, therefore it is not repeatedly referenced in this subsection

ally complete set [28] because they contain the implication and all other logical operators can be derived:

$$\begin{aligned} \neg A \quad &\text{to} \quad A \to FALSE \\ A \vee B \quad &\text{to} \quad \neg A \to B \\ A \wedge B \quad &\text{to} \quad \neg(A \to \neg B) \end{aligned}$$

Therefore, any formula in propositional logic can be expressed in a connective set of triplets. For example, the formula $(A \wedge B) \to C$ can be transformed into $\neg(A \to \neg B) \to C$. The triplet representation is the following:

$$\begin{aligned} (b', b, 0) \quad & \neg B \\ (d, a, b') \quad & A \to \neg B \\ (d', d, 0) \quad & \neg(A \to \neg B) \\ (f, d', c) \quad & \neg(A \to \neg B) \to C \end{aligned}$$

Here, a, b and c are representations for the variables A, B and C, while d, a', b' and f represent subformulas to serve as connectives between triplets. The variable f contains the whole formula. To prove if the formula is valid, one assumes it to be false and tries to find a contradiction. A contradiction is a *terminal triplet*, i.e. a triplet that is contradictory in itself. The terminal triplets and the reasons why they are contradictory are given below:

$(0, y, 1)$	it must be possible to conclude something true
$(1, 1, 0)$	something true can not result in something false
$(0, 0, x)$	something false can result in anything

To drive a set of triplets into a set that contains a terminal triplet, Stålmarck presents the application of *simple rules* or a branching rule, called *dilemma rule*.

Simple rules Simple rules can be applied to triplets that matches their pattern. They are used to derive new information about variables in the whole set of triplets. The simple rules are simply derived from the truth table of the implication, e.g. if a set contains the triplet $(x, 0, z)$, the variable x must always be true because something false can result in anything. x can therefore be replaced by 1. All simple rules are listed below.

$$\text{Notation:} \quad \frac{\text{triplet}}{\text{variable/substitution}}$$

$$\text{Rule 1:} \quad \frac{(0, y, z)}{y/1 \quad z/0} \qquad \text{Rule 2:} \quad \frac{(x, y, 1)}{x/1}$$

$$\text{Rule 3:} \quad \frac{(x, 0, z)}{x/1} \qquad \text{Rule 4:} \quad \frac{(x, 1, z)}{x/z}$$

$$\text{Rule 5:} \quad \frac{(x, y, 0)}{x/\neg y} \qquad \text{Rule 6:} \quad \frac{(x, x, z)}{x/1}$$

$$\text{Rule 7:} \quad \frac{(x, y, y)}{x/1}$$

Take for example the logical proposition $a \to (b \to a)$ and its triplet representation $(i, b, a), (j, a, i)$. As mentioned earlier, the entry step is to claim that the whole formula is

not valid, therefore to set $j = 0$. The resulting triplet $(0, a, i)$ matches the pattern of rule 1, so in the whole set of triplets the substitutions $(a/1, i/0)$ can be applied. The resulting set of triplets is $(0, b, 1), (0, 1, 0)$. The first of these triplets is a terminal triplet. Thus, $a \to (b \to a)$ is valid and always true.
Of course, not all sets of triplets can be solved with simple rules. With the *dilemma rule*, Stålmarck introduces a branching method based on the relation of two subformulas. To understand the dilemma rule, formula relations need to be understood.

Formula relations If X is a formula, then $S(X)$ is a set which contains all subformulas of X and their complements. A formula relation $\sim$ is an equivalence relation between the subformulas A and B on $S(X)$ with the constraint that if $A \sim B$, then the same applies for the complements of A and B, $A' \sim B'$. Thus, A and B are from the same equivalence class and have the same truth value. $R(A \equiv B)$ denotes the smallest formula relation that contains R and somehow relates A and B. Important formula relations are X^+, which puts every element of $S(X)$ into its own equivalence class. $X^+(X \equiv \top)$ would then be the starting point when trying to disprove X. Rules can be applied to add or to remove information to the relation. For example, the knowledge that $A \wedge B \equiv \top$ allows to set $A \equiv \top$ and $B \equiv \top$. Using rules, one can merge equivalence classes inside formula relations. When an element and its complement are detected within the same equivalence class, the relation is proved to be false.

Dilemma rule The dilemma rule uses the above information to distinguish between cases when applying simple rules does not lead to results. The formula is shown below:

$$\frac{\begin{array}{c} R \\ \hline \begin{array}{cc} R(A \equiv B) & R(A \equiv \neg B) \\ \text{(derivation)} & \text{(derivation)} \\ R_1 & R_2 \end{array} \end{array}}{R_1 \sqcap R_2} \tag{4.1}$$

The relation R is a set of triplets. A and B have to be from different equivalence classes, in fact B is from a space of $\{TRUE, FALSE\}$. The second line in equation (4.1) leads to two new Dilemma derivations R_1 and R_2. Each of these derivations is analysed separately until two elements F and $\neg F$ are found in the same equivalence class, so until a contradiction is found. The respective derivation is then expected to be $\bot$. The information gained from the analysis of the two separate derivations is then merged, so the operation $R_1 \sqcap R_2$ can have the following results:

- R_1 if $R_2 = \bot$. R_2 leads to a terminal triplet, therefore R_1 is followed.
- R_2 if $R_1 = \bot$ analogous.
- Finish if $R_1 = \bot$ and $R_2 = \bot$. Both derivations lead to a terminal triplet, therefore the whole set is contradictory and so the underlying logical proposition can be proven valid.
- $R_1 \cap R_2$ if neither R_1 nor R_2 lead to a terminal triplet.

Proof hardness and time consumption It is obvious that with depth of open derivations, the number of sets that need to be analysed rises exponentially. Stålmarck introduces the term *hardness*. If the proof system uses only the simple rules, the system is called M_0. If the dilemma rule is used for only one level of branching, it is called M_1, if it uses branching within branching it is M_2 and so on. Thus, a system M_1 has only one open assumption

on a variable when it analyses the derivations. A valid formula is declared as *i-hard* if it is solvable in M_i but not in $M_i + 1$. The complexity for solving a formula in M_i is $O(n^{2i+1})$. Here, n is the size $|A|$ of a formula A which is the number of variable occurrences plus the number of connectives. A proof in M_0 can be done in linear time; Stålmarck's proof method is thus much more prone to proof-hardness than to proof-size. The authors claim that most formulas that occur in the verification of industrial systems are of hardness degree 0 or 1, which would explain why the method is used in the plug-in for the SLDV.

4.3 Product Overview

The SLDV works in three modes: *Property Proving*, *Test Case Generation* and *Design Error Detection* [3]. The latter can be used to detect design errors like integer overflows or division-by-zeroes in the model. The Fuel Quantity Management System (FQMS) is not using particular arithmetic operations, therefore the design error detection mode is not used in this analysis. As mentioned in section 2.3, building exhaustive tests to reach a certain level of coverage is often hard. Collecting coverage information will detect non-tested elements, but it will not give inputs to stimulate these elements to reach full coverage. With the test case generation mode, the tool can automatically generate test cases for a given level of coverage. It analyses the model to detect elements that need to be covered, called objectives, e.g. a conditional transition that needs to be triggered as *TRUE* and *FALSE* adds two objectives. To find a test case that covers a certain objective, the SLDV uses the underlying proof engine by negating the given objective and searching for a counterexample with the proof engine. The SLDV offers decision coverage, condition coverage and MCDC coverage. The result of a test case generation analysis is an overview of the objectives that could be covered, the percentage of coverage that could be achieved and the underlying set of test cases that had been produced. The objectives are classified as *satisfied* if they could be covered by test case, *unsatisfiable* if they denote an infeasible path that can never be reached or *undecided* if the SLDV can not find a test case that satisfies the objective but also can not declare the objective as unsatisfiable.
The property proving mode can be used to check if the model meets the underlying requirements. The requirements can be modelled within Simulink. The proof engine picks the requirement and searches for counterexamples in the model. To model requirements for property proving, the SLDV provides a block library and an own application programming interface (API). I did not use the API in the analysis, so I just present how to work with the library. The main blocks that are used are the *Assumption* and the *Proof Objective* blocks. With a proof objective, one can design a range of values that a specific signal has to hold during program execution. With an assumption, one can restrict input signals to a range of values for the analysis of the respective proof objective. An example of how to build an easy model with assumptions and prove objectives can be seen in figure 4.2. The

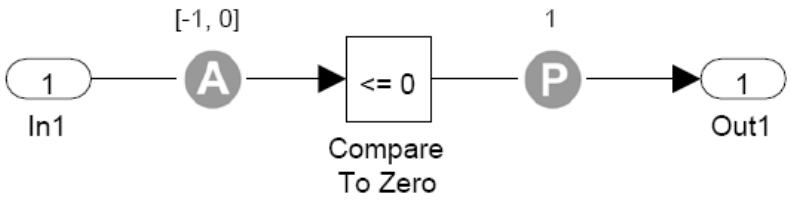

Figure 4.2: Model block with verification components

given example shows how to directly integrate the verification objectives in the model. The *P-block* in the right part of the model is the property block. Within this block, one can define a range of values that the underlying signal has to hold at all time during program execution. In property proving mode, the SLDV searches for violations of all properties that are defined. The *A-block* in the left part of the model is a proof assumption block. This block serves to restrict an input signal to a range of values for property proving. The SLDV keeps the input signal within the defined range when searching for violations for the respective property. Note that the A-block does not influence the signal in the model simulation. The underlying signal can still carry any value during model execution. It is just used when the developer wants to prove properties in a predicted environment. The SLDV would prove the property in the example to *TRUE*. The A-block restricts the input signal to the values -1 or 0. The *Compare-To-Zero-Block* outputs a 1 when its input signal is smaller than or equals 0. Thus, the proof assumption restricts the input signal to a range of values that lead the Compare-To-Zero-Block to always output a 1. The P-block claims that the underlying signal has to be a 1 at all times, therefore this property can be proven valid.
The SLDV also provides a block called *Verification Subsystem*, which contains a reference to the design model. Verification objectives can be built inside the verification subsystem. Other blocks in the SLDV environment are temporal operators, that allow to model and detect signal delays. The result of a property proving analysis is either a simple success message if all properties could be proved to be right, or the counterexample that disproves the requirement. A harness model can be produced to drive the model to the state where the property is violated.

4.4 Possible Problems

[22] mentions several problems and limitations that could influence the analysis:

- The application of formal verification methods is limited to discrete systems. If the model contains continuous blocks like a PID-controller, these blocks have to be replaced.
- Stateflow in general is not capable of handling concurrency. Therefore, the execution order for transitions has to be determined. Even the so called parallel statecharts are not really executed in concurrency. That limits the accuracy of the verification results and also leads to problems with certain model designs (section 5.3).
- In order to avoid an infinite number of possible system states, the software performs several simplifications on data types and structures. That includes the conversion of floating-point to rational number arithmetic and a maximum number of while-loop executions in test case generation mode. That could possibly affect the accuracy of the verification.
- In property proving mode, unbounded loops are not supported. If the model contains for- or while loops with no or a conditional exit statement, the analysis process will stop.

5 Analysing an Aircraft Fuel Management Model with the Simulink Design Verifier

The model on which I performed the scalability analysis is the Fuel Quantity Management System of a modern Airbus long range aircraft. It is the heart of the Airbus Fuel System Modelling Environment (AFSME). While almost all other parts of the AFSME serve to guarantee an exact-as-possible simulation of the system, the fuel management subsystem actually describes the later implementation of the system. Therefore, the FQMS serves as the executable specification mentioned in section 2.1. That is why that model has to be as correct and exact as possible. The FQMS consists only of statecharts, which are considered easier, clearer and less ambiguous than enabled subsystems [23]. Each major aircraft function has an own statechart. Transition conditions are calculated as booleans within Simulink to ensure an easy reading. The key data to the Fuel Quantity Management System is the following:

- 810 Stateflow states
- 175 functions to calculate transition conditions
- 509 junctions that split or merge transitions
- 1754 Stateflow transitions
- 45 input vectors that sum up to 185 input signals
- 28 output vectors that sum up to 321 output signals

The top level of the statecharts (see figure 5.1 on the following page) contains four parallel states. The main state contains all the actual operations of the FQMS in its subcharts. The other states deal with equipment monitoring and general equipment logic. The operations in the main state are exclusively separated by ground operations and in-flight operations. The ground operations subchart contains exclusive states that mainly deal with refuelling and defuelling. The flight operations subchart contains four parallel states, dealing with jettison, engine feed et al. The system hierarchy of the first three levels of the FQMS is shown in figure B.2.

Even if the SLDV is a 'push-button tool' that does not require theoretical background or mathematical operations, the verification of such a big model is not trivial and requires a considerable amount of preliminary considerations, mainly concerning strategies and best practices. The considerations underlying the scalability analysis of this thesis are presented in the following section.

5.1 Strategies to Analyse an Existing Large Model

To effectively analyse the model and to gain useful data to measure the scalability of the SLDV, it is necessary to find a way to start the analysis with only a small part of the model. Step-by-step, the number of states can then be increased. Thereby, the verification objectives should be built in such a way that they do not constrain this step-by-step process and keep their validity. In terms of general strategy, I decided to divide the analysis into

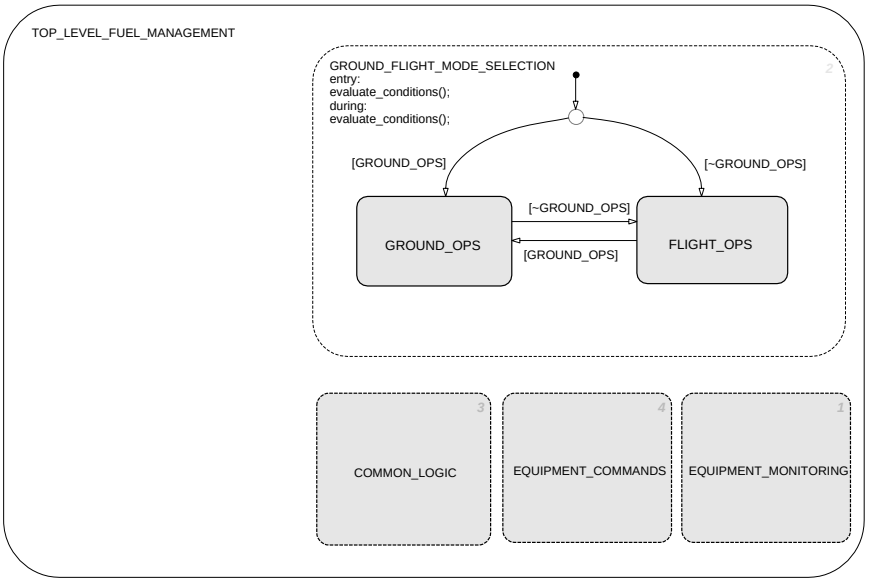

Figure 5.1: Top level of the FQMS statecharts

two main parts. In the first part, I picked the requirements in figure C.1 and C.2 and rebuilt them as verification objectives, which resulted in three properties that needed to be proved. Starting with the small part of the model and adding statecharts with every iteration, I proved these properties in property proving mode and also generated tests for the model in test case generation mode. In the second part, I took the complete model and increased the number of properties. For that, I picked the requirements ID3 to ID10 (see figure C.2 ff.). Together with the requirements in ID1 and ID2, that resulted in 13 properties. The requirements ID4, ID5 and ID10 led to a failure in the SLDV which is explained in section 5.3 on page 32, therefore they were not used later on. Starting with one property, I added two properties in every iteration to see how the number of properties influences the performance of the SLDV. The selection of the requirements is to a certain extent arbitrary. The Sub-System Requirements Document (SSRD) contains approximately 800 requirements, so this analysis can only give a slight insight.

5.1.1 Top-Down Approach or Bottom-Up Approach

To find an approach for the analysis, it has to be considered that the SLDV runs with two modes i.e. *test generation* and *property proving*. Property proving requires that the states that affect the given property are included in the model, otherwise the verification will logically fail. Test generation produces concrete test cases that execute the model in a certain sequence. If states are embedded into superstates after a test generation, that might lead to a different execution order and so the test cases from the previous step get useless. To sum up, the requirements for efficient property proving and test generation are contrary to a certain extent. That has to be considered in the decision for an analysis strategy. Therefore, I suggest the following two approaches:

Bottom-Up approach: The bottom-up approach starts at the very low level of the system with a set of states that has no more substates. If the SLDV is applied to these states, they will be integrated into their superstate. The SLDV is then applied

to this superstate in conjunction with the other states at the specific level. That workflow iterates several times, until the top-level is reached.

Top-Down approach: The top-down approach starts at the top-level of the system, while all lower level states are excluded (consider figure B.2 again). In each iteration, the next lower level is included in the verification process, until the lowest level is reached.

5.1.2 Integration or Separation of Verification Components

As mentioned in section 4.3, the SLDV provides two strategies to include verification objectives for property proving in the model:

Direct integration: In small models, the direct integration of verification components is much easier, because they can directly be added by 'drag&drop'. However, if it comes to more complex models and properties that need to be proved, the direct integration provides a poor overview over the aim and the relation of verification objectives. Another disadvantage is the need for a change in the model.

Verification subsystem: Verification objectives can also be bundled in a separate subsystem called verification subsystem, a model block that references the design model and contains one or more subsystems that define the properties and any required constraints. If many or complicated properties need to be proved, the verification subsystem provides a better overview over the verification objectives. Also, the verification subsystem can be built in isolation of the design model. Section 5.2 contains a description of how to build up such a verification model. Inside that subsystem, it is possible to use any Simulink block to model a requirement, including Matlab function blocks. Although not used during this scalability analysis, Matlab functions offer the possibility to describe difficult requirements within the Matlab language, thus avoid complicated Simulink models.

5.1.3 Decision

I decided to use verification subsystems for the property proving mode. Apart from the fact that a key requirement to the analysis process was not to touch the actual design model, there are several other advantages. The verification objectives to each requirement can be grouped in a separate subsystem, which maintains a good level of traceability. Different requirements also have different assumptions and objectives on input and output signals. With the integration method, it would only be possible to put a single assumption or objective on a signal, which is not the case within a verification model.
In terms of the analysis approach, I decided for a top-down approach with certain modifications. Lower states often rely on variables that are assigned in their superstates. A bottom-up approach would thus just not be possible. Consider table D.1 on page 58, which is a changelog that tracks which parts of the model have been added in the specific steps, together with the nesting level, especially the state numbering, in figure B.2 on page 44 to follow this explanation. At first, I determined the statecharts that affected the properties to be proved. I kept these charts and their direct supercharts and deleted all other statecharts. In step 3, all statecharts that directly affected the properties in figure C.1 on page 48 and C.2 were added. In step 4, I build the top level of the system which can also be seen in figure 5.1, where all of the underlying statecharts were empty, except for the charts added in the steps before. In the next steps, I tried to evenly add new states and transitions by adding substates. Instead of adding a certain number of states and transitions, I added full functionalities, such as one ground operation per step in steps 6-11, to minimise

the risk of errors caused by interdependent statecharts. When all ground operations were added, I continued with adding the equipment monitoring and command charts. These contain the same control sequence for every pump and valve. I finished with the flight operations. The selection of the steps was arbitrary. When adding full functionalities, the model is complete after 18 steps. However, the ordering of these steps could also be done in any other way. Because many parts of the model are interdependent, this ordering might also affect the intermediate results of the verification, although this was not observed in this analysis.

5.2 Define Test Requirements as Proof Objectives

A crucial step in the application of the SLDV is the actual translation of textual requirements into verification objectives. Although a translation into temporal logic is not necessary, one still has to consider how to express a requirement with the Simulink blocks that are available and one must find a way to ensure that the verification objectives actually meet the given requirements. To understand how the subsystems which contain the

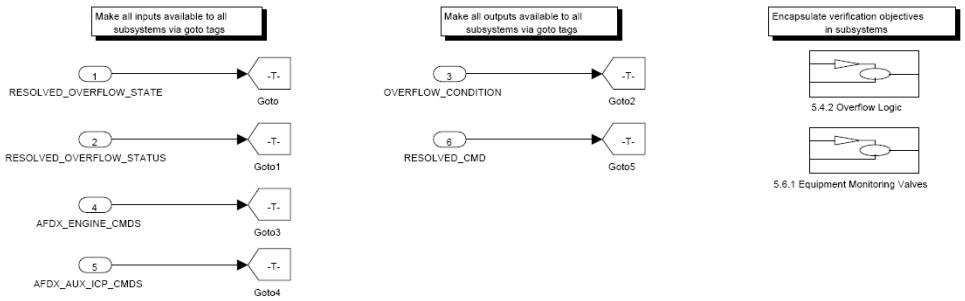

Figure 5.2: Top level of the Verification Subsystem

property proving blocks work, an understanding of the model architecture is necessary. An abstraction of the system's top level can be seen in figure B.1 on page 43. On the top level of the verification subsystem, the inputs and outputs of the design system are mapped to Simulink GoTo-blocks. A pair of a GoTo- and a From-block in Simulink has the same functionality as a regular line connection, but it avoids complicated wiring by making a signal of a system globally available with the GoTo-block, which can then be fetched with the From-block. The verification objectives to a specific group of requirements are encapsulated into several subsystems which are grouped and named by the respective section in the requirements document (see figure 5.2). This architecture achieves an adequate overview and a good level of maintainability.
The underlying requirements document is the *SSRD*. The requirements in the document are located between high- and low-level requirements as they define specifically how certain signals shall behave over time but do not describe a concrete implementation. A group of requirements I used for the analysis is shown below. I modified the original text of the requirement by annotations in brackets to facilitate the reference to signal names and values.

```
REQ:
The 'Overflow Condition' (OC) shall be set to true when any of the
following conditions is true:
-The Resolved Overflow status in the Left Wing Surge Tank (ROSL) is
NOT FAILED(0) and the Left Wing Surge Tank Resolved Overflow state
(LTOS) is WET(1).
-The Resolved Overflow status in the Right Wing Surge Tank (ROSR) is
NOT FAILED(0) and the Right Wing Surge Tank Resolved Overflow state
(RTOS) is WET(1).
REQ:
The 'Overflow Condition'(OC) shall be set to true when the Resolved
Overflow status in the Left (ROSL) or Right Wing Surge Tank (ROSR)
is FAILED(1).
REQ:
The 'Overflow Condition' (OC) shall be set to false when all of the
following conditions are true:
-The Resolved Overflow status in the Left Wing Surge Tank (ROSL) is
NOT FAILED(0) and the Left Wing Surge Tank Overflow state (LTOS) is
DRY(0).
-The Resolved Overflow status in the Right Wing Surge Tank (ROSR) is
NOT FAILED(0) and the Right Wing Surge Tank Resolved Overflow state
(RTOS) is DRY(0).
```

These requirements are representative for the rest of the document as they describe the behaviour of specific output signals in a certain environment. The signals that are used are mainly of boolean type, parameters such as *NOT FAILED* and *WET* are defined in the system and work as placeholders for the boolean values *TRUE* and *FALSE*. These characteristics of requirements offer a great opportunity for designing proof objectives and to assure that they meet the requirements: Truth tables. The truth table for the requirements above is shown in table 5.1. The computer science offers various techniques to

ROSL	ROSR	LTOS	RTOS	OC
0		1		1
	0		1	1
1				1
	1			1
0	0	0	0	0

Table 5.1: Truth table for the overflow requirement

derive a logical circuit from a truth table, e.g. Karnaugh maps or the Quine-McCluskey algorithm [28]. In most cases it was possible to find a suitable logical circuit at first sight. In the given example, a logical OR-connection of the four input signals would satisfy the desired behaviour. The verification task is to prove that the signal *Overflow Condition* meets the output of on OR-connection of the two input signals. In figure 5.3 on the next page, the respective Simulink model is shown. Three From-blocks catch the input signals *RESOLVED_OVERFLOW_STATE* and *RESOLVED_OVERFLOW_STATUS*, which each carry a signal for the left and for the right tank, and the output signal *OVERFLOW_CONDITION*. The relational operator proves if the OR-connection of the input

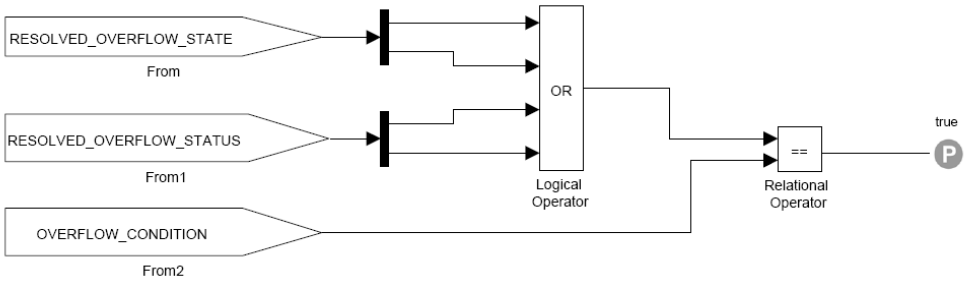

Figure 5.3: Simulink model for the Overflow requirement verification

signals and the output signal have the same value. In the new version, the SLDV also offers support for temporal properties. The temporal properties in the SSRD are limited to delays in signal changes. These can be implemented with the *Detector* block. The Detector-block detects the input duration of a signal while it outputs *FALSE*. After a given time, the output signal of the block becomes *TRUE* and stays like that as long as the input signal continues to be *TRUE*. The truth tables are not intuitively applicable anymore when it comes to properties that contain delays, unless a delay is treated as a signal itself that is true after the delay has elapsed, which I did in the particular cases. This is applicable without problems but might turn out as a lack of easy readability. An example implementation can be seen in figure C.10 on page 55. A glance in the whole requirement document and in every statechart revealed that more than 50% of all requirements deal with delays. Since the purpose of the model is to deal with actual hardware equipment, which means that response times and execution times have to be considered, this number is no surprise.
Note here that all requirements that have been used for the analysis could be proven valid at all time.

5.3 Limitations/Problems arisen

During the analysis, several problems arose which impact either the analysis in a way that does not allow to continue or the usefulness of the gained results. These problems and limitations have to be considered when the SLDV is applied to a system. In the following list, I will present these problems, together with a valuation of how severe a problem impacts the applicability of the SLDV on the given Fuel Quantity Management System and a possible solution to that problem.

Incomplete Model Coverage

Problem As mentioned earlier in section 2.3 on page 13, it is necessary to trigger a transition in a statechart as *TRUE* and as *FALSE* to reach full decision coverage. Transitions are executed in a specific, predicted order; unconditional transitions are executed last. The problem which occurs when applying the SLDV to the FQMS can be seen at the very top level of the statecharts. Figure 5.4 on the facing page shows a screenshot of the top level statechart after the test case generation with the SLDV. The results of the generation are highlighted by the tool. The respective log contains the following entry:

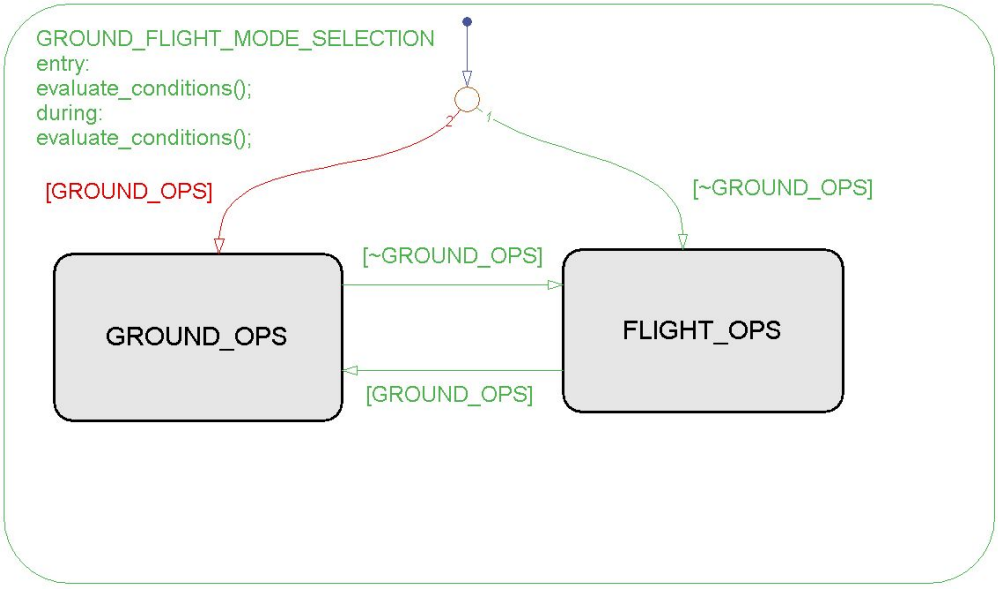

Figure 5.4: Top level statechart for the system behaviour of the FQMS

Objectives Proven Unsatisfiable		
Type	**Model Item**	**Description**
Decision	Chart.[...].[GROUND_OPS]	Transition: Transition trigger expression F

The transition that caused the unsatisfiable objective is highlighted red in figure 5.4. The reason for the incomplete coverage is that when the tool tries to trigger the transition as *FALSE*, it picks the signal $\sim GROUND\_OPS$. To reach the transition, the other transitions have to be executed in the given order, which can be read from the numbers on every transition. Obviously, when $\sim GROUND\_OPS$ is selected and the transition number 1 is executed, that transition will be taken. Therefore, transition number 2 can never be executed with $\sim GROUND\_OPS$ and therefore denotes an infeasible path.

Severity According to certain styleguides, almost every statechart in the FQMS contains that architecture where a junction decides between a variable and its negation. Therefore this issue is very severe.

Proposal for Solution If Stateflow would allow real concurrent execution, a dedicated execution order for transitions would not be necessary anymore and therefore the current design of the FQMS could be tested with full model coverage. The only solutions that would cope with the given issue with the current version of the SLDV would require a change in the model. Since the FQMS has already started to go through the certification process, every change in the model would require a significant amount of document changing and re-certification. Changing the model would just not be feasible, because almost every statechart is affected. Nevertheless, if a change in the model would be allowed, one could replace every transition that has a condition which is simply a negotiation of another transition by an unconditional transition. That would not affect the behaviour of the model because unconditional transitions are executed last and need to be executed just once to reach full MCDC coverage. Using the SLDV, one could then prove that the new

model behaves exactly the same as the old model, and apply the SLDV to the new model. Though, unconditional transitions are considered bad design.

Corrupt MCDC Coverage

Problem The difference between MCDC coverage and decision coverage is the way that combinations of inputs in conditions are handled (see section 2.3 on page 13). In the FQMS, input conditions are grouped in functions and variables. Consider the statechart in figure 5.4 on the previous page. In fact, *GROUND_OPS* is not a signal but a variable, which is evaluated in the *evaluate_conditions* block. The evaluation is similar to the following:

$$\text{GROUND\_OPS} = \text{RES\_OVERFLOW} \ \& \ \text{FAILURE}$$

An MCDC coverage would require to test all conditions as true and then every condition separately as false while the others are held true. Thus, an MCDC coverage analysis would require three test cases or input combinations within a test case. However, the SLDV does not recognize that *GROUND_OPS* is only a variable and tests only *GROUND_OPS* as *TRUE* and *FALSE* for an MCDC coverage analysis.

Severity Almost every condition in the FQMS is a variable that is evaluated in a separate function. It provides better reading and makes information available for lower states without the need of recomputing signal combinations several times. Therefore nearly all the information about MCDC coverage is not right. Testing with MCDC coverage is a non-mandatory requirement in the ABD0200. The generated test cases do not reach MCDC coverage, therefore it is not sufficient to derive the test cases from model testing for later system tests. That issue affects also the confidentiality about the right functionality of the model.

Proposal for Solution Functions in Stateflow are not covered by model coverage with the SLDV. However, functions that are evaluated in Matlab functions embedded into Stateflow are covered by model coverage. By replacing all Stateflow functions by embedded Matlab functions, one could work around that problem. This could be done with an automated script, and it would not mean changing, but instrumenting the model for the verification. Thus, this approach is promising. It would also be possible to work around that problem by consequently avoiding using variables. However, that would decrease the readability of the model significantly.

No Support of Unbounded Loops

Problem The SLDV is unable to cope with unbounded loops in property proving mode. Remembering the strategy of model checking (see section 4.1 on page 17), it is obvious that it is impossible to derive a deterministic finite state machine from an unbounded loop, because the number of possible states would be infinite. However, strategies such as *busy waiting* and other systems behaviour have to be excluded from the model. A lack of usability is the fact that the SLDV stops the analysis when detecting an unbounded loop but gives no information about the location of the loop.

Severity Again, a redesign of the model would fix the problem, but it is not said that an unbounded loop is not essential for the systems functionality, so it is not said that the redesign equals the original model.

Proposal for Solution However, there are only 2 unbounded loops in the FQMS, and the respective statecharts are working in isolation of other parts of the system, so I just excluded them from the model and accepted that the results are not 100% accurate.

Annulling Test Case Generation

Problem During the analysis, I encountered a situation whereby adding certain parts of the model led to an analysis time which was about 5% of the former analysis time, while suddenly about 95% of the objectives were classified as *undecided*. In the following, I describe this significant loss in the success rate as *annulling*. A manual debugging led to the conclusion that a certain architecture annuls the test case generation. Specifically, it is using an equality check between two boolean values that is used as a condition within Stateflow functions. An example can be constructed given the statechart in figure 5.1 on page 24. In the function at the bottom of the statechart, the expression in the square

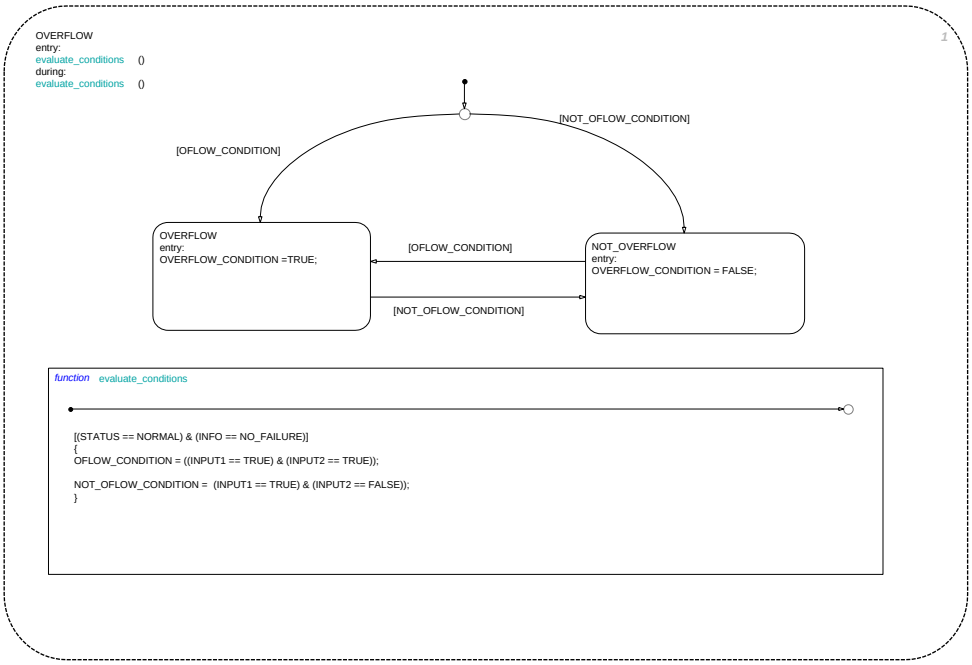

Figure 5.5: Statechart with function that annuls test case generation

brackets is the boolean equality check that leads to the given problem. If the check returns *TRUE*, the underlying equations are executed.

Severity The architecture in figure 5.1 is used several times throughout the model. An elimination of the respective statecharts would change the behaviour of the model and is therefore not applicable. This issue is very severe, because the SLDV can not produce useful results in test generation mode on the FQMS. In fact, it led to the fact that I

stopped the scalability analysis for the test generation mode after several iterations (see section 6 on page 35.)

Proposal for solution A query at Mathworks revealed that the equality check is allowed and commonly used in systems design, and that the problem is a bug that is fixed in the latest version 2012a of Matlab, which was not available for the scalability analysis. A workaround was suggested that consisted of changing the data type of one of the boolean values in the equality check to another data type such as integer. However, that is in contradiction to the solution of the problem in section 5.3 and was therefore not followed.

Reading the Status of Output Signals

Problem During the second part of the scalability analysis (see section 6), I encountered that also in property proving mode, adding certain verification objectives led to an analysis that could not produce any results and left all properties undecided. Specifically, the failure occurred when I added requirement ID 4 (figure C.4). I assumed that the origin of that issue is when the status of an output signal is a condition for another output signal. This assumption was confirmed when removing requirement ID 4 and adding requirement ID 5 (figure C.5), which does nothing else than proving that two output signals behave the same, led to the same empty set of results. However, this applies only for signals like *AFDX_MGMT_CMDS* in the given case, which are used and assigned in many different charts throughout the whole model. Note that *output signal* means an output signal of the FQMS, so of the own system, so the SLDV should know the status of these signals. This was confirmed when trying to read from *OVERFLOW_CONDITION*, another output signal, was successful, so in general the structure is allowed and the problem only occurs when the signal is subject to more complicated computations.

Severity About 80% of the equipment monitoring requirements contain the *AFDX* output signal as a condition to another output signal. However, the issue only occurs when trying to verify these requirements. Their implementation apparently does not affect the analysis. Thus, these requirements can not be directly verified, but a verification of other requirements is still possible.

Proposal for solution On the users side, a change in the requirements or in the implementation might solve the problem. Although it was not tested, I suggest that breaking a signal flow that underlies a requirement into smaller steps and verifying these steps leads to more appropriate results. For example, instead of using the status of *AFDX_MGMT_CMDS* as a condition for the verification objective, one could use the input signals that drive *AFDX_MGMT_CMDS*. If it does not affect the system's behaviour, the logic could also be changed in a way that it uses the value of the AFDX signal from the previous timestep rather than the current signal. On the Mathworks side, I can just guess where the error might occur. It is possible that not all possible states are derived from the model when creating the state space model.

Using Doubles Instead of Booleans

Problem Apart from the signals that carry the flight data and the fuel level in the tanks, all signals are booleans. However, around 20% of them are modelled as doubles because

this is the default data type in Simulink. That causes trouble in two ways. On the one hand, it leads to a larger state space, on the other, the SLDV finds counterexamples that are not achievable in practice.

Severity The workaround for this problem is not difficult. However, the problem has to be considered for further development of the model and for further work with the SLDV.

Proposal for solution Changing the data types from double to boolean is a five-minute-task. A change will not have any effect on the behaviour of the model because the respective signals are handled as booleans. Even if a change in the model is not feasible or not desired, the respective signals can be constrained with assumption blocks inside the verification objectives. Figure C.6 on page 53 shows such a workaround.

Runtime Error in Test Case Generation Mode

Problem When I tried to apply the test case generation mode to the complete model, a runtime error occurred five minutes later after the compatibility check, so before the actual analysis started. The stacktrace was displayed, together with the information about an unknown exception that was thrown. This failure was reproducible in 9 out of 10 times.

Severity For the analysis, this failure was not severe because I decided to stop the test case generation before. However, if that runtime error is not related to the problems that have been mentioned before, it makes the application of the test case generation mode on the whole model impossible.

Proposal for solution A support query at Mathworks revealed that the runtime error is caused by a bug that is fixed in the latest version 2012a.

6 Results of the Verification Concerning Scalability

The data that has been collected during the scalability analysis is shown in section D on page 58.
The parameters and the reason why they were recorded are explained in section 3.2. The data on memory consumption and CPU usage needs some kind of interpretation and also some background knowledge of how the data was collected. Matlab offers .NET support, so information on processes running on the machine can be directly gained inside Matlab. I used the *System Information Class for Windows* [25] from the Matlab File Exchange. The class offers functions to record the memory allocation by the Matlab process and the CPU-usage of the Matlab process. It also generates a plot where both parameters are plotted over time. The Matlab process apparently allocates system memory and does not release it after its usage. This was observed when the same iteration was executed on different days during different times and the maximum system memory used changed between 400 MB and 800 MB, dependent on how long and how intensely Matlab was used before. Therefore, in terms of memory consumption, the property of interest is the difference between low-peak and high-peak memory usage, because this value actually shows the impact of starting an SLDV analysis.
In terms of CPU usage, one needs to keep in mind that a CPU with 4 cores was used. Comparing one of the generated plots (figure B.4 on page 45) with the Windows Task Manager Performance Window (figure B.3 on page 45), it can be seen that the Matlab process obviously does multithreading within certain boundaries, because it sometimes uses more than 25% of the CPU and must therefore use more than one core. In more detail, the graphs allow the conclusion that the Matlab process almost constantly uses a single core until it is completely in use and distributes remaining threads over the spare cores to a total sum of maximum 50% CPU usage. During program execution, Matlab including its GUI is blocked. A query at Mathworks revealed that the SLDV itself is not multithreaded, so the tasks that are distributed over the other CPU cores belong to background operations such as data management.
For the graphs, I chose the median of the three iterations whenever applicable.
As described in section 5, the scalability analysis can be divided into the following two main parts: Property proving and test case generation on the growing model and property proving on the full model. These parts are in chronological order in terms of their processing. Due to the fact that the results of the test case generation were not useful, which is discussed below, and that the SLDV started to throw an *unknown exception of unknown type* when trying to start the test case generation, I decided to stop the analysis for that mode in step 10. As mentioned in section 5.1.3, model interdependencies might influence the analysis when certain statecharts are not yet added in the analysis. In step 14, all equipment monitoring and equipment commands statecharts, which affect all pumps and valves and might therefore influence other statecharts that deal with pumps and valves status, were added. I started another test case generation in this step to assure that the results were still not satisfying, which was right. This last attempt also took over two days

of trying to avoid the runtime error, so my decision to stop the test case generation turned out to be right.
The data collected during property proving on the growing model (section D.3 on page 61) allows the conclusion that the SLDV scales well with a growing number of states and transitions. All parameters show at least linear growth with the number of states and transitions. The same applies for property proving with an increasing number of verification objectives (section D.4 on page 65). Here, apart from the analysis time, the parameters even decreased with a growing number of objectives, so it can be assumed that the main challenge for the SLDV is the size of the model rather than the number of objectives. It has to be considered that the underlying requirements were easy requirements that could be captured by truth tables. When it comes to proofs that require timing or latency, the SLDV might still show poor scaling.
In terms of test case generation, the data in table D.3 on page 60 is missing any structure or observable trend. Adding more states and transitions directly leads to an increasing number of objectives, because e.g. a conditional transition needs to be triggered as *TRUE* and *FALSE* and therefore adds two objectives. After the 5th step, the number of satisfied objectives stagnated around 300 for the next 5 steps, while the number of undecided objectives increased significantly. In the 14th step, around 70% of the objectives were declared undecided. More than that, the number of unsatisfiable objectives decreased after the 8th step down to 1 in the 14th step, although the model contains almost one unreachable path per statechart as described in section 5.3. Looking at the memory usage and CPU usage, it seemed that the SLDV got stuck in computation in some parts of the model. Altogether, this unstructured behaviour is due to problems that could mainly traced back to an incompatible model design that constrained the interface between the model and the SLDV, which are presented in section 5.3. I did not plot any resulting graphs for the scalability because they could also be misleading. However, because the test case generation mode is using the proof engine to search for counterexamples, as it is also done in the property proving mode which scales well, it can be assumed that also test case generation shows good scalability on an appropriate model.

6.1 Possible Improvements

All possible improvements have the purpose to get more accurate results in the test case generation mode because the property proving mode scaled well and performed promisingly. Some improvements have already been mentioned in section 5.3. A summary of them which should be applied as *Best Practice* in model design, not only to facilitate the work with the SLDV but also to meet existing design habits, is given below:

- Use the latest version Matlab 2012a when working with the SLDV. All support queries to Mathworks concerning problems with the SLDV could be tracked down to bugs that had been fixed in the latest version.
- Use Matlab functions to compute transition conditions instead of Simulink functions.
- Always use the smallest design that is possible. Especially never use doubles instead of booleans for signal types.
- When using calculated outputs in equations, use the previous step's output.

7 Conclusion

As I discussed in section 6 on page 35, the results from the scalability analysis that were useful lead to the promising conclusion that at least parts of the SLDV scale very well on the Fuel Quantity Management System. In the introduction work for this thesis, I reviewed the work with the SLDV and its embedding into the model as logical and structured. Also, the modelling of requirements is a workflow that might be considered easy by test engineers, due to the fact that although the strategy of model checking and testing are completely different, building a test-case or a verification objective both requires exact understanding of the requirement.
Although many problems and limitations for the application of the SLDV on the FQMS were discovered, I would still come the the conclusion that it is feasible to use the tool in the current development process at Airbus. The test case generation mode is not yet applicable and it would require a considerable amount of redesign of the model to use it. Nevertheless, the fact that property proving works well makes the application of the SLDV interesting. It might be too much work and also not helpful to subsequently implement every requirement in the SSRD as a verification objective to verify it on the the model. But in case of slight modifications of a single statechart, the SLDV could be used to ensure that the corresponding requirements are still satisfied. The results of that verification can serve as additional evidence for the re-certification of the model after such a slight change.

7.1 Future Work

Future work on the topic of this thesis should at first focus on the possible improvements I gave in section 6.1 on the facing page. If all the suggestions and ideas are implemented, there should be no major problem with the application of the SLDV on the FQMS. It also became quite obvious in the context of the thesis that the SLDV and its applicability have improved with every new Matlab version. I therefore recommend to repeat the analysis which was done for this thesis with every new release of Matlab.
Another interesting approach would be to integrate the SLDV in a completely new model design process. The requirements could then directly be built as verification objectives before the system design has started. It would then be possible to check at any time of the design phase if the model already meets certain requirements and also avoid that the model architecture constrains the later application of the SLDV on the whole model.

Appendices

A List of Abbreviations

SLDV	Simulink Design Verifier
EASA	European Aviation Safety Agency
RTCA	Radio Technical Commission for Aeronautics
EUROCAE	European Organization for Civil Aviation Equipment
AFSME	Airbus Fuel System Modelling Environment
FQMS	Fuel Quantity Management System
A/C	Aircraft
API	application programming interface
SSRD	Sub-System Requirements Document
BDD	Binary Decision Diagram

B General Appendix

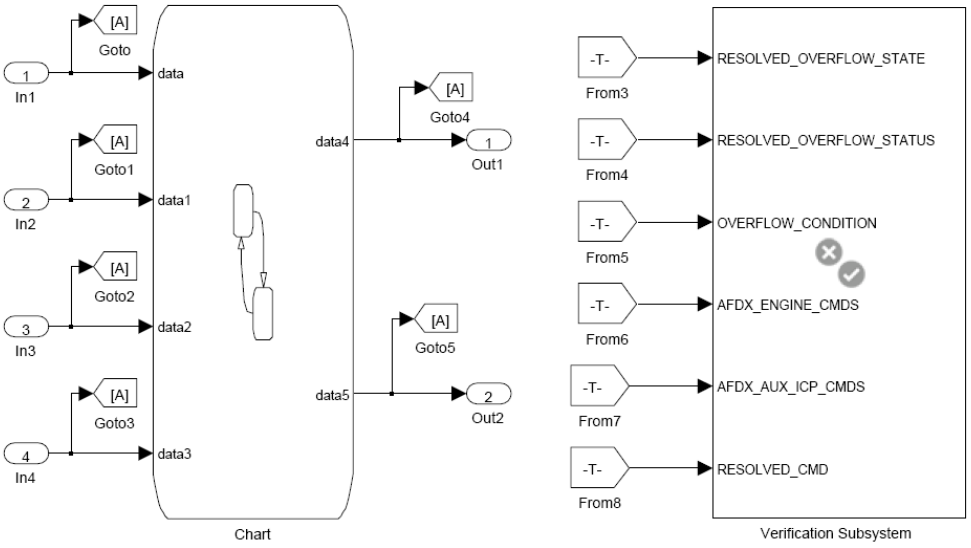

Figure B.1: Abstraction of the Top Level of the FQMS Testing Environment

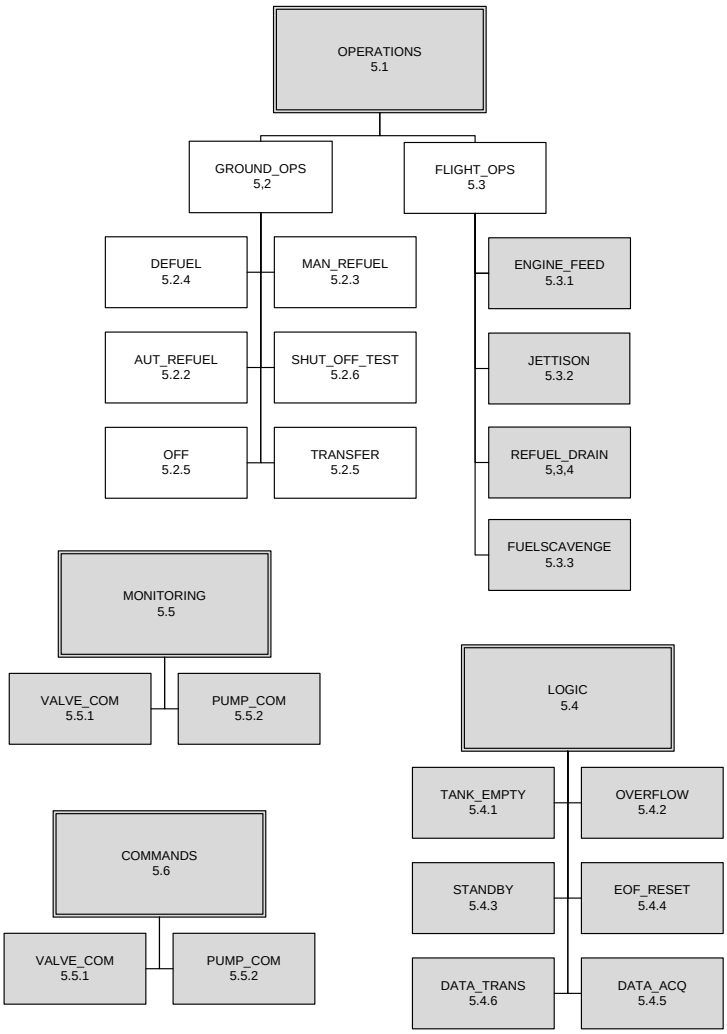

Figure B.2: System Hierarchy of the First Three Levels of the FQMS - Parallel states are marked grey. The numbering denotes the level of the specific chart, starting with chart 5 for the very top level

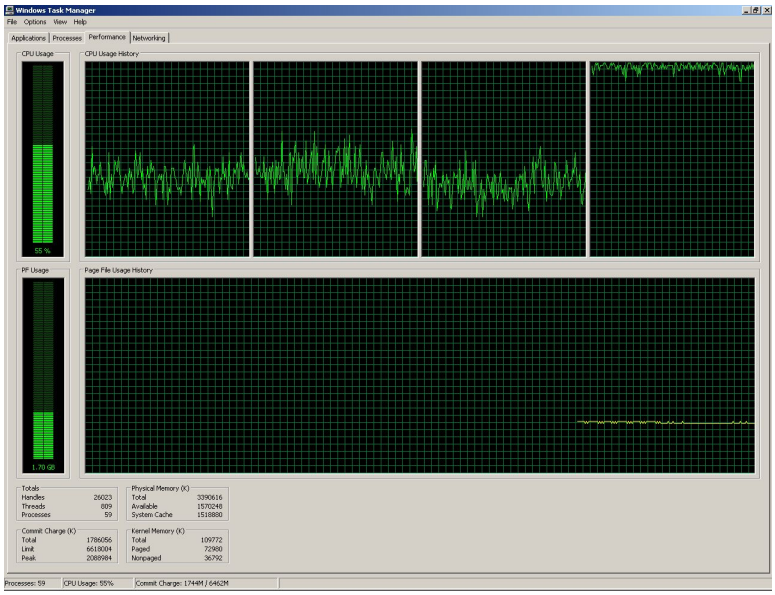

Figure B.3: Screenshot of the Windows Task Manager during test case generation with the SLDV

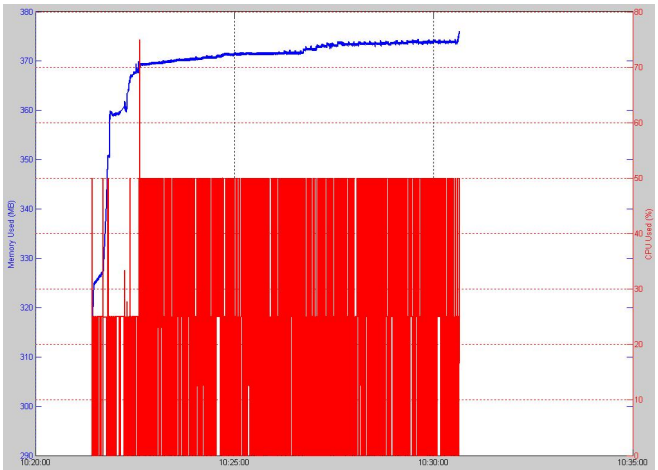

Figure B.4: Performance data gained with Matlab during test case generation with the SLDV

C Verification Subsystems

The following requirements are representatives of the relevant FQMS requirements and should not be used to imply about the functionality of the Fuel Quantity Management System used within Airbus.

C.1 Overflow Condition

Requirement ID 1

REQ:
The 'Overflow Condition' (OC) shall be set to true when any of the following conditions is true:
-The Resolved Overflow status in the Left Wing Surge Tank (ROSL) is NOT FAILED(0) and the Left Wing Surge Tank Resolved Overflow state (LTOS) is WET(1).
-The Resolved Overflow status in the Right Wing Surge Tank (ROSR) is NOT FAILED(0) and the Right Wing Surge Tank Resolved Overflow state (RTOS) is WET(1).
REQ:
The 'Overflow Condition'(OC) shall be set to true when the Resolved Overflow status in the Left (ROSL) or Right Wing Surge Tank (ROSR) is FAILED(1).
REQ:
The 'Overflow Condition' (OC) shall be set to false when all of the following conditions are true:
-The Resolved Overflow status in the Left Wing Surge Tank (ROSL) is NOT FAILED(0) and the Left Wing Surge Tank Overflow state (LTOS) is DRY(0).
-The Resolved Overflow status in the Right Wing Surge Tank (ROSR) is NOT FAILED(0) and the Right Wing Surge Tank Resolved Overflow state (RTOS) is DRY(0).

Truth Table

ROSL	ROSR	LTOS	RTOS	OC
0		1		1
	0		1	1
1				1
	1			1
0	0	0	0	0

Table C.1: Truth table for requirement ID 1

Architecture

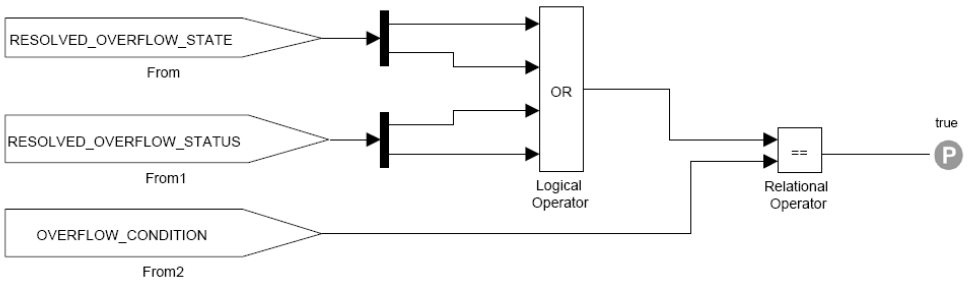

Figure C.1: Verification model for requirement ID 1

C.2 Equipment Monitoring Valves

Requirement ID 2

REQ:
The resolved command for an Engine LP Valve (RES_CMD_LP1) (RES_CMD_LP2) shall be OPEN(1) when all of the following conditions are true:
-The respective Engine Master Switch pushbutton (E1) (E2) is selected ON(1).
-The respective Engine Fire Handle on the cockpit ICP (ICP1) (ICP2) is selected OFF(0).
Otherwise the resolved command for the respective Engine LP Valve (RES_CMD_LP1) (RES_CMD_LP2) shall be SHUT(0).

Truth Table

E1	¬ICP1	RES_CMD_LP1
1	1	1
all others		0

(a) Engine LP1 Valve

E2	¬ICP2	RES_CMD2_LP2
1	1	1
all others		0

(b) Engine LP2 Valve

Table C.2: Truth table for requirement ID 2

Architecture

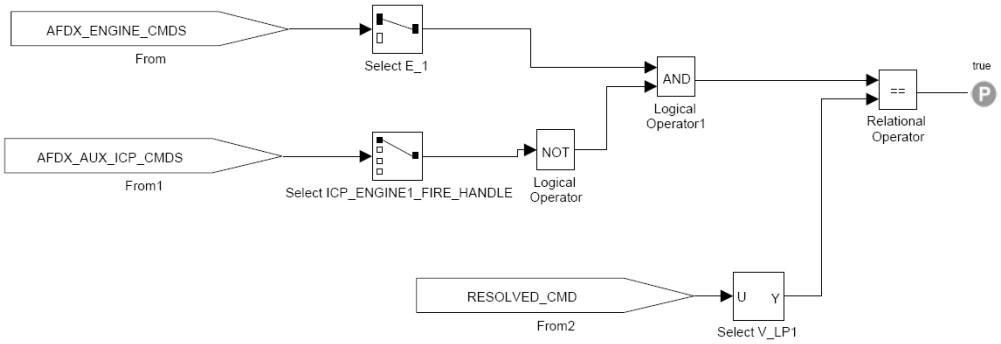

Figure C.2: Verification model for requirement ID 2 (LP2 analogous)

Requirement ID 3

REQ:
The resolved command for a Crossfeed Valve (RES_CMD_XA) (RES_CMD_XB) shall be OPEN(1) when any of the following conditions are true:
-The respective Crossfeed Valve (ICPXA) (ICPXB) pushbutton on the cockpit ICP is selected ON(1).
-The Crossfeed Open Override Relay signals that the respective Crossfeed Valve (XA_OPEN) (XB_OPEN) is TRUE(1) and there is an Electrical System Emergency Condition(EC == 1).
Otherwise the resolved command for the respective Crossfeed Valve (RES_CMD_XA) (RES_CMD_XB) shall be SHUT(0).

Truth Table

ICPXA	XA_OPEN	EC	RES_CMD_XA
1			1
	1	1	1
all others			0

Table C.3: Truth table for requirement ID 3 (XB analogous)

Architecture

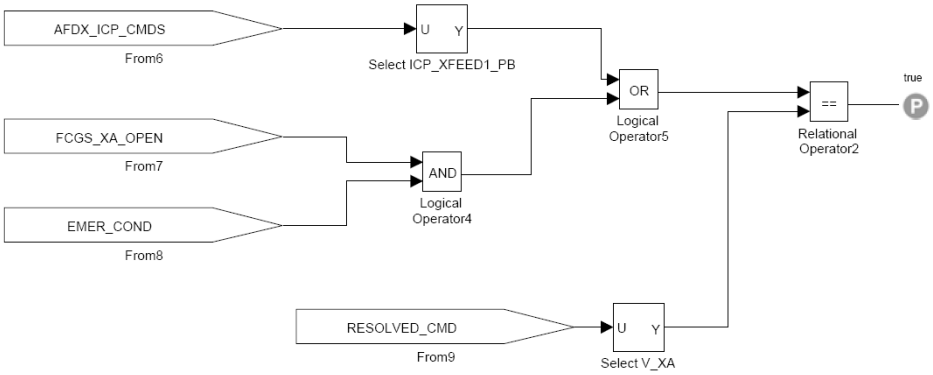

Figure C.3: Verification model for requirement ID 3 (XB analogous)

Requirement ID 4

REQ:
The resolved command for a Jettison Valve (RES_CMD_J1) (RES_CMD_J2) shall be OPEN(1) when all of the following conditions are true:
- Jettison is enabled via hardware pin programming (HPP).
- The FQMS commands the respective Jettison Valve (J1_OPEN) (J2_OPEN) OPEN(1).
- The Jettison ARM and ACTIVE pushbuttons on the cockpit ICP are selected ON.
- The resolved state of the Left Wing Tank Low Level sensors (LSENS) is WET(1).
- The resolved state of the Right Wing Tank Low Level sensors (RSENS) is WET(1).
Otherwise the resolved command for the respective Jettison Valve (RES_CMD_J1) (RES_CMD_J2) shall be SHUT(0).

Truth Table

HPP	J1_OPEN	ARM	ACTIVE	LSENS	RSENS	RES_CMD_J1
1	1	1	1	1	1	1
all others						0

Table C.4: Truth table for requirement ID 4 (J2 analogous)

Architecture

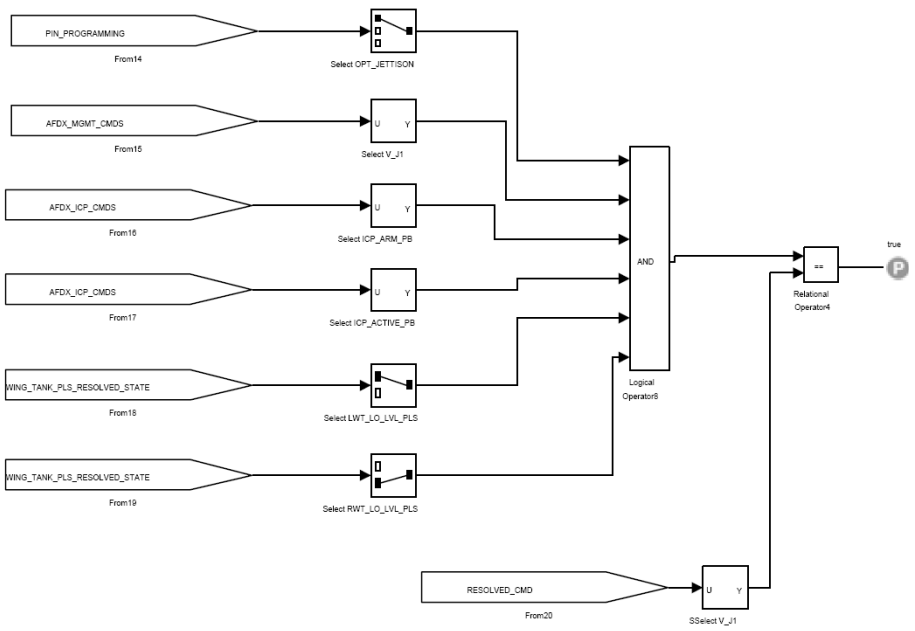

Figure C.4: Verification model for requirement ID 4 (J2 analogous)

Requirement ID 5

REQ:
The resolved command for a Scavenge Valve (RES_CMD_TL) (RES_CMD_TR) shall be OPEN(1) when the FQMS commands the respective Fuel Scavenge Valve (TL) (TR) OPEN(1).
Otherwise the resolved command for the respective Scavenge Valve (RES_CMD_TL) (RES_CMD_TR) shall be SHUT(0).

Truth Table

$$\text{RES\_CMD\_TL} == \text{TL}$$
$$\text{RES\_CMD\_TR} == \text{TR}$$

Architecture

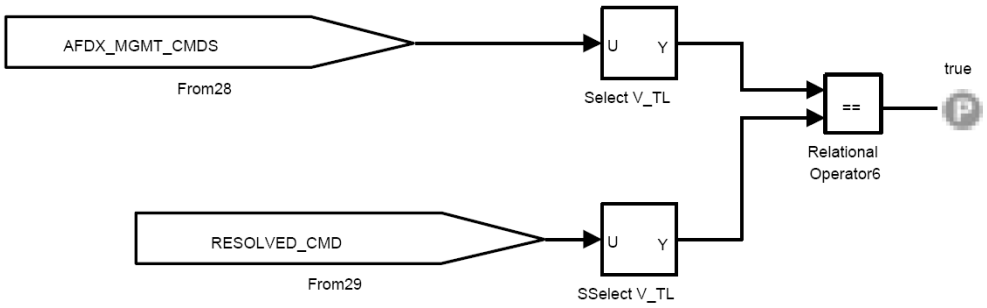

Figure C.5: Verification model for requirement ID 5 (TR analogous)

Requirement ID 6

REQ:
The resolved command for the APU Isolation Valve (RES_CMD_SA) shall be OPEN(1) when all of the following conditions are true:
- The APU Fuel Demand State (FDS) is TRUE(1).
- The APU Fuel Line Damage Signal (FDL) is FALSE(0).
Otherwise the resolved command for the APU Isolation Valve (RES_CMD_SA) shall be SHUT(0).

REQ:
The resolved command for the APU LP Valve (RES_CMD_LPA) shall be OPEN when all of the following conditions are true:
- The APU Fuel Demand State (FDS) is TRUE(1).
- The APU Fuel Line Damage Signal (FDL) is FALSE(0).
Otherwise the resolved command for the APU LP Valve (RES_CMD_LPA) shall be SHUT(0).

Truth Table

FDS	¬FDL	RES_CMD_LPA	RES_CMD_SA
1	1	1	1
all others		0	0

Table C.5: Truth table for requirement ID 6

Architecture

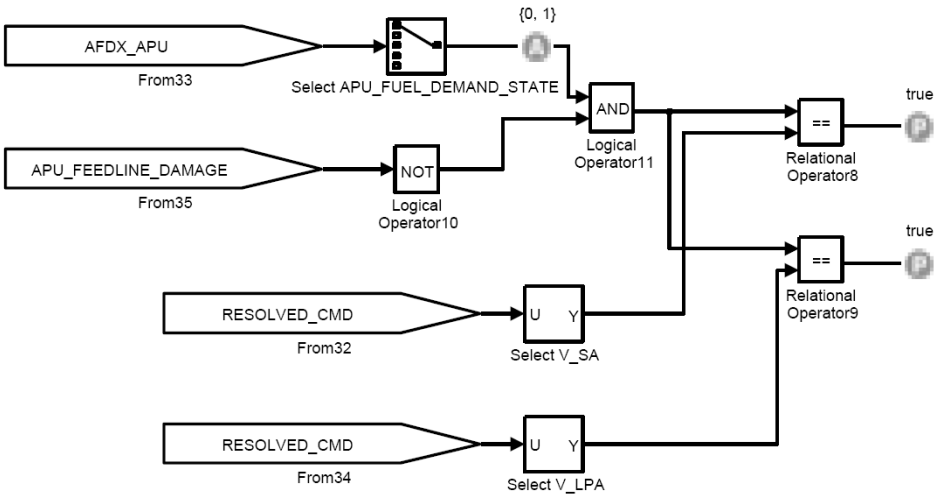

Figure C.6: Verification model for requirement ID 6

Requirement ID 7

REQ:
The resolved manual command for the Main Pumps (RES_MAN_CMD_PM1) (RES_MAN_CMD_PM2) shall be ON(1) when the load shedding conditions that apply for the Main Pumps
(PUMP_SHED_PM1) (PUMP_SHED_PM2) are FALSE(0).
Otherwise the resolved manual command for the Main Pumps (RES_MAN_CMD_PM1) (RES_MAN_CMD_PM2) shall be OFF(0).

Truth Table

$$\text{RES\_MAN\_CMD\_PM1} == \neg\text{PUMP\_SHED\_PM1}$$
$$\text{RES\_MAN\_CMD\_PM2} == \neg\text{PUMP\_SHED\_PM2}$$

Architecture

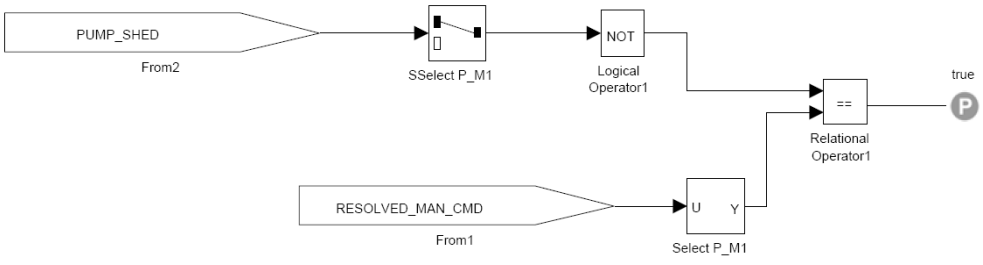

Figure C.7: Verification model for requirement ID 7 (PM2 analogous)

Requirement ID 8

REQ:
The resolved manual command for a Standby Pump (RES_MAN_CMD_PS1) (RES_MAN_CMD_PS2) shall be ON(1) when the respective Main Pump (PUMP_LP_PM1) (PUMP_LP_PM2) pressure switch state indicates LOW PRESSURE(1).
Otherwise the resolved manual command for the respective Standby Pump (RES_MAN_CMD_PS1) (RES_MAN_CMD_PS2) shall be OFF(0).

Truth Table

$$\text{RES\_MAN\_CMD\_PS1} == \text{PUMP\_LP\_PM1}$$
$$\text{RES\_MAN\_CMD\_PS2} == \text{PUMP\_LP\_PM2}$$

Architecture

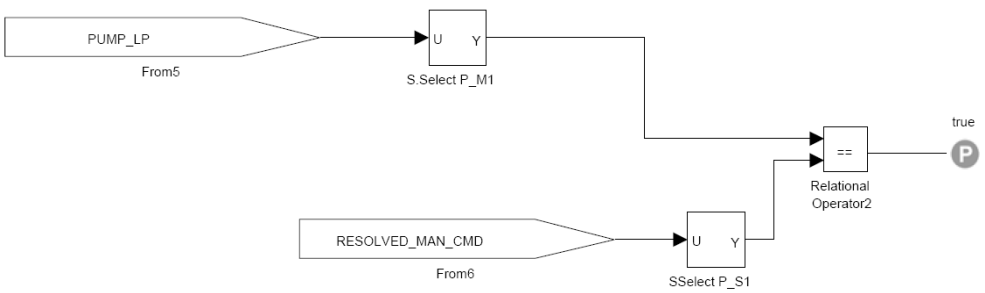

Figure C.8: Verification model for requirement ID 7 (PS2 analogous)

Requirement ID 9

REQ:
The resolved manual command for both Centre Tank Pumps (RES_MAN_CMD_PC1) (RES_MAN_CMD_PC2) shall be ON(1) when the Centre Tank Manual Feed pushbutton (ICP_CTR_TK_FEED_PB) is selected ON(1) on the ICP.
Otherwise the resolved manual command for both Centre Tank Pumps (RES_MAN_CMD_PC1) (RES_MAN_CMD_PC2) shall be OFF(0).

Truth Table

$$\text{RES\_MAN\_CMD\_PC1} == \text{ICP\_CTR\_TK\_FEED\_PB}$$
$$\text{RES\_MAN\_CMD\_PC2} == \text{ICP\_CTR\_TK\_FEED\_PB}$$

Architecture

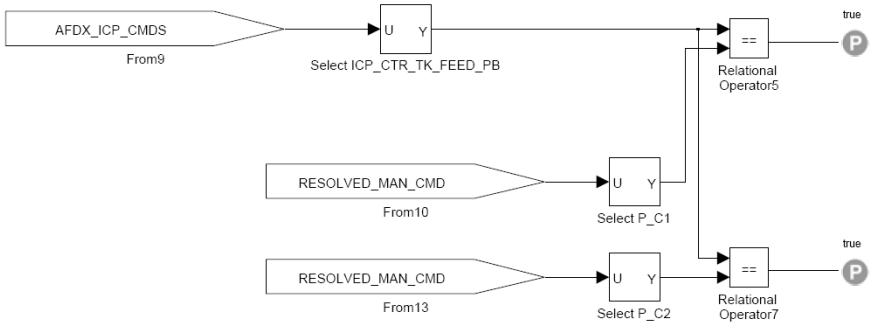

Figure C.9: Verification model for requirement ID 9

Requirement ID 10

REQ:
The FQMS shall command the Left Jettison Valve (J1) SHUT(0) when the Jettison mode enters the 'Complete' sub-mode (CMPL) and a confirm time, 'DELAY_JETT_VALVES' (DELAY) has elapsed.

REQ:
The FQMS shall command the Right Jettison Valve (J2) SHUT when the Jettison mode enters the 'Complete' sub-mode (CMPL) and a confirm time, 'DELAY_JETT_VALVES' (DELAY) has elapsed.

Truth Table

CMPL	DELAY	J1
1	1	0
all others		not covered

Table C.6: Truth table for requirement ID 10 (J2 analogous)

Architecture

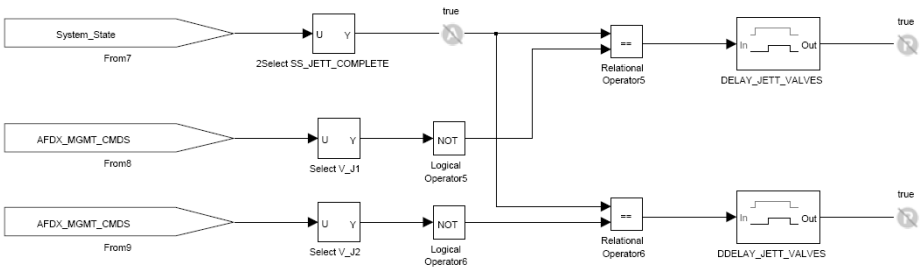

Figure C.10: Verification model for requirement ID 10

D Data from the Scalability Analysis

D.1 Changelog: Parts Added in Each Step in the First Part

Step	Annotation
1	Common_Logic_Overflow only charts added: 5.0 5.4, 5.4.2
2	Valve Equipment Monitoring LP_1 added charts added: 5.6, 5.6.1, 5.6.1.17 1 Objective added
3	Valve Equipment Monitoring LP_2 added charts added: 5.6.1.18 1 Objective added
4	Operations added charts added: 5.1, 5.2(empty), 5.3(empty)
5	all ground_ops functions added charts added: 5.2(2), 5.2(3), 5.2.X(empty)
6	Manual Refuel functionality fully added charts added: 5.2.3 all, 5.2.3.1, 5.2.3.1.X
7	automatic refuel functionality first level added charts added: 5.2.2(1), 5.2.2(2), 5.2.2.2, 5.2.2.1(empty)
8	automatic refuel functionality fully added charts added: 5.2.2.1, 5.2.2.1.X 5.2.2.1.3(2) and 5.2.2.1.3(3) excluded due to bad performance
9	SHUT_OFF_TEST fully added charts added: 5.2.6, 5.2.6(X)
10	added OFF functionality and GROUND_TRANSFER fucntionality charts added: 5.2.7, 5.2.5.ALLSUBSTATES
11	added DEFUEL functionality completely charts added: 5.2.2.1.3(2) and 5.2.2.1.3(3), 5.2.4.ALLSUBSTATES(*) *DELETED 5.2.4.ALLSUBSTATES because of unbounded loops
12	added ALL Equipement commands charts added: 5.5.ALLSUBSTATES
13	added ALL Valve-Monitoring commands charts added: 5.6.1.ALLSUBSTATES
14	added ALL Pump-Monitoring commands charts added: 5.6.2.ALLSUBSTATES test cases with decision coverage tried
15	added TANK_EMPTY, STANDBY_PUMP_LOGIC, EOF_RESET from COMMON_LOGIC charts added: 5.4.1ALLSUBSTATES, 5.4.3.ALLSUBSTATES, 5.4.4
16	added DATA_ACQUISITION and DATA_TRANSMISSION from COMMON_LOGIC charts added: 5.4.5.ALLSUBSTATES, 5.4.6.ALLSUBSTATES(*) *DELETED 5.4.6.1.2(4) because of unbounded loops
17	added ENGINE_FEED, FUEL_SCAVENGE, REFUEL_GALLERY_DRAIN from FLIGHT_OPS charts added: 5.3.1, 5.3.3, 5.3.4
18	added JETTISON from FLIGHT_OPS. Model completed charts added: 5.3.2.ALLSUBSTATES

Table D.1: Changelog

D.2 Data Collected in the First Part

Property proving with 2 requirements (ID1 and ID2) that result in 3 objectives and test case generation on the growing model.

Step	General Information		time [s]		memory usage [MB]			CPU [%]	objectives
	states	transitions	overall	sldv	low peak	high peak	difference	av. usage	proven
1	5	4	64.60	1.00	549.40	562.92	13.52	21.74	1
			64.21	0.00	562.89	568.31	5.41	22.72	
			62.73	0.00	565.58	569.00	3.43	23.10	
2	19	34	64.69	1.00	603.94	607.29	3.35	23.07	2
			64.36	0.00	605.11	608.73	3.62	23.24	
			64.94	0.00	605.83	609.07	3.24	23.20	
3	31	61	62.90	0.00	500.08	526.43	26.34	22.98	3
			63.08	0.00	524.26	529.59	5.32	23.05	
			63.31	1.00	527.40	532.51	5.11	23.26	
4	34	66	64.74	0.00	598.52	610.89	12.37	22.83	3
			64.95	0.00	610.71	619.44	8.72	22.86	
			65.32	0.00	616.71	625.05	8.34	22.99	
5	46	97	66.19	0.00	479.94	569.05	89.11	22.58	3
			66.85	0.00	571.45	588.27	16.83	22.98	
			68.35	0.00	584.39	591.93	7.54	22.81	
6	61	125	69.71	1.00	774.63	778.91	4.28	22.67	3
			70.38	0.00	775.02	780.39	5.38	22.82	
			70.16	1.00	776.46	781.05	4.59	22.60	
7	72	151	68.71	0.00	852.39	861.72	9.34	22.84	3
			68.75	0.00	857.39	863.03	5.64	22.63	
			69.19	0.00	858.71	864.80	6.09	22.67	
8	85	169	73.54	1.00	529.24	534.94	5.70	21.16	3
			74.04	0.00	530.64	536.40	5.76	21.11	
			76.05	0.00	531.26	536.86	5.60	21.16	
9	89	181	71.32	1.00	799.79	805.44	5.65	22.51	3
			72.24	0.00	799.45	804.89	5.45	22.41	
			72.79	0.00	799.64	805.64	6.00	22.53	
10	109	219	72.75	1.00	822.87	829.39	6.52	22.17	3
			72.33	0.00	824.34	829.91	5.57	22.28	
			72.47	0.00	824.67	830.14	5.47	22.39	
11	DEFUEL MODE causes unbounded loops in top level. Deleted again.								
12	163	321	80.83	1.00	822.87	829.42	6.55	22.05	3
			80.73	1.00	824.12	831.97	7.85	22.14	
			80.56	0.00	826.61	833.47	6.86	22.12	
13	381	803	110.55	3.00	716.93	743.96	27.04	19.85	3
			113.16	3.00	726.41	756.61	30.20	19.72	
			119.34	3.00	739.43	760.88	21.45	19.50	
14	540	1162	141.89	3.00	795.37	829.03	33.66	18.10	3
			144.99	3.00	800.51	831.75	31.23	17.86	
			141.89	3.00	805.70	834.28	28.59	18.18	
15	594	1307	159.19	3.00	979.14	1066.74	87.60	19.57	3
			145.31	3.00	1025.99	1072.13	46.14	19.95	
			144.85	3.00	1033.36	1075.22	41.87	20.01	
16	697	1505	167.31	4.00	745.51	802.27	56.77	18.67	3
			168.09	3.00	751.47	810.91	59.44	19.28	
			167.57	4.00	757.04	814.60	57.56	18.83	
17	726	1562	193.96	3.00	834.05	893.27	59.23	18.42	3
			182.95	3.00	839.14	897.45	58.31	17.39	
			176.45	3.00	845.04	901.05	56.01	18.83	
18	787	1680	205.31	4.00	925.36	999.71	74.36	18.64	3
			183.50	4.00	928.30	1003.99	75.69	18.67	
			186.35	5.00	936.39	1008.88	72.49	19.07	

Table D.2: Data Collected in First Part for Property Proving

Table D.3: Data Collected in First Part for Test Case Generation

Step	General Information		time [s]		Coverage Data				memory usage [MB]			CPU [%]
	states	transitions	overall	sldv	objectives	satisfied	undecided	unsatisfiable	low peak	high peak	difference	av. usage
1	5	4	59.87	1.00	10	9	0	1	566.34	571.78	5.44	21.99
2	19	34	141.53	77.00	99	92	0	7	606.64	611.68	5.04	24.22
3	31	61	415.74	350.00	188	139	36	13	641.94	652.50	10.56	25.74
4	34	66	412.87	348.00	198	141	43	14	663.32	669.75	6.43	26.57
5	46	97	553.12	480.00	502	297	149	56	298.89	376.02	77.13	24.56
6	61	125	723.11	646.00	562	300	205	57	452.67	463.30	10.63	25.35
7	72	151	728.45	645.00	636	310	195	131	464.13	478.01	13.88	25.30
8	85	169	772.12	684.00	676	300	205	171	505.01	533.17	28.16	24.49
9	89	181	1162.99	1076.00	698	304	367	27	798.25	805.72	7.48	24.35
10	109	219	1388.85	1296.00	776	305	446	25	825.42	836.69	11.27	24.91
11	TEST CASE GENERATION PAUSED HERE											
12	163	321										
13	381	803										
14	540	1162	5315.36	5041.00	2493	734	1758	1	757.33	883.79	126.46	25.07
15	594	1307	TEST CASE GENERATION STOPPED HERE									

D.3 Resulting Graphs from the First Part: Property Proving

The data for the test case generation was too sparse and without any structure, therefore graphs were just plotted for the property proving data.

Overall Time

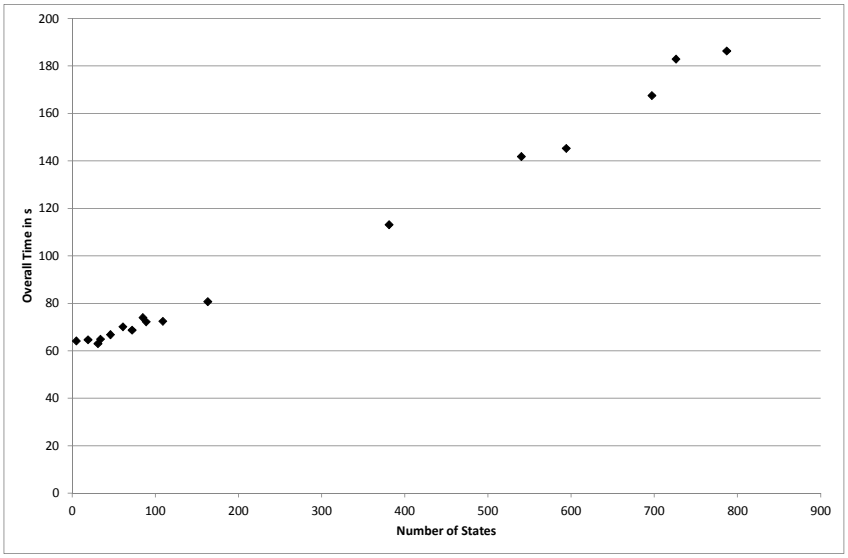

Figure D.1: Overall Time against Number of States in Property Proving Mode

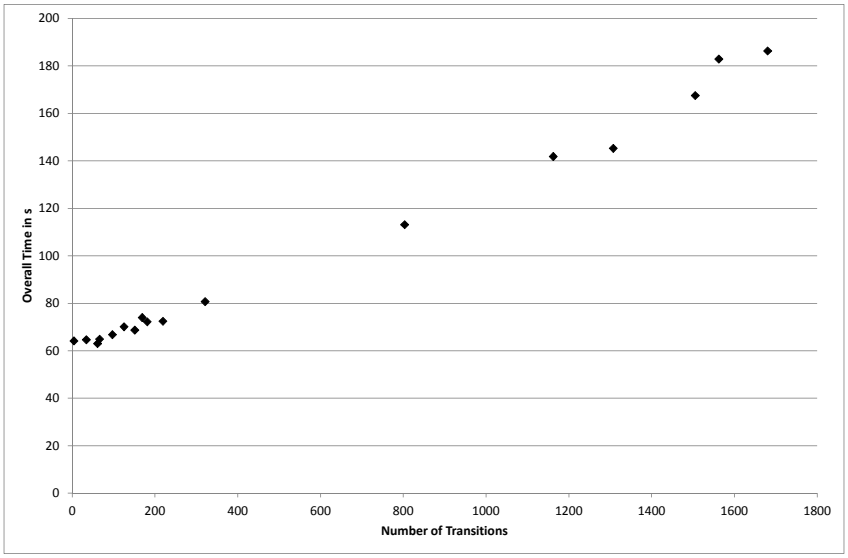

Figure D.2: Overall Time against Number of Transitions in Property Proving Mode

SLDV Analysis Time

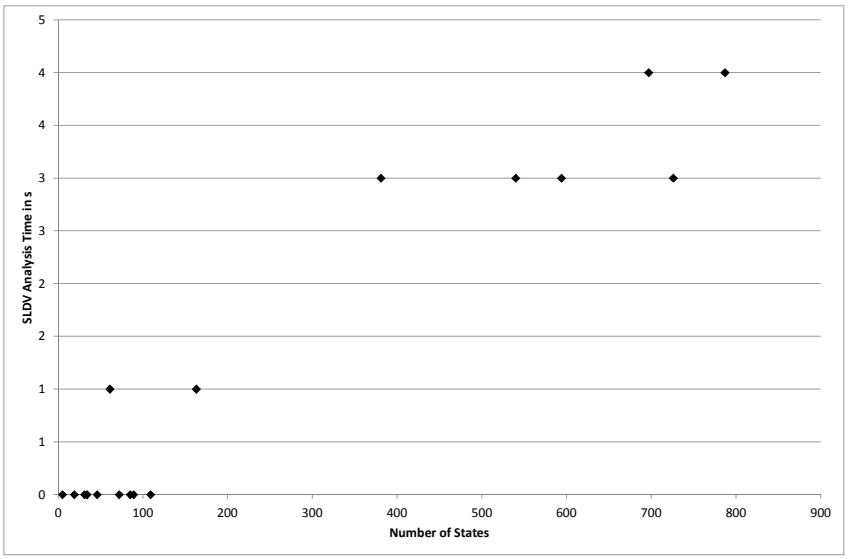

Figure D.3: SLDV Analysis Time against Number of States in Property Proving Mode

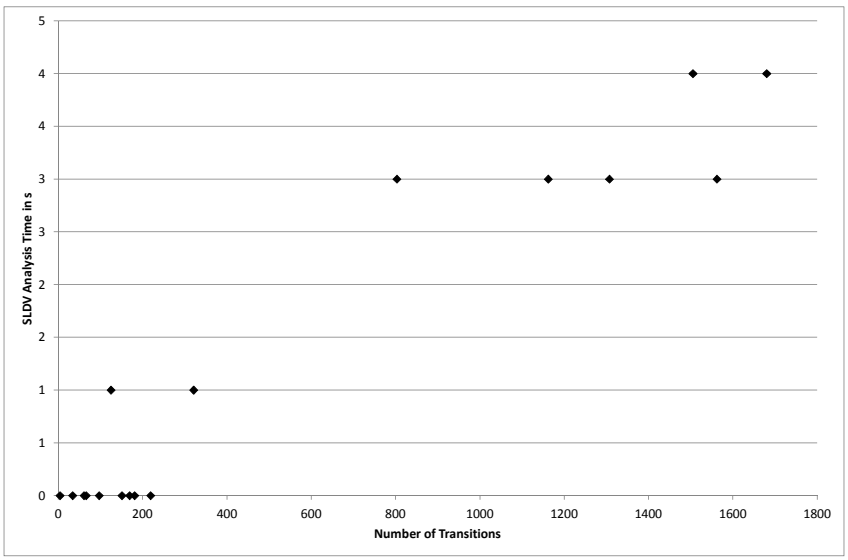

Figure D.4: SLDV Analysis Time against Number of Transitions in Property Proving Mode

System Memory Usage

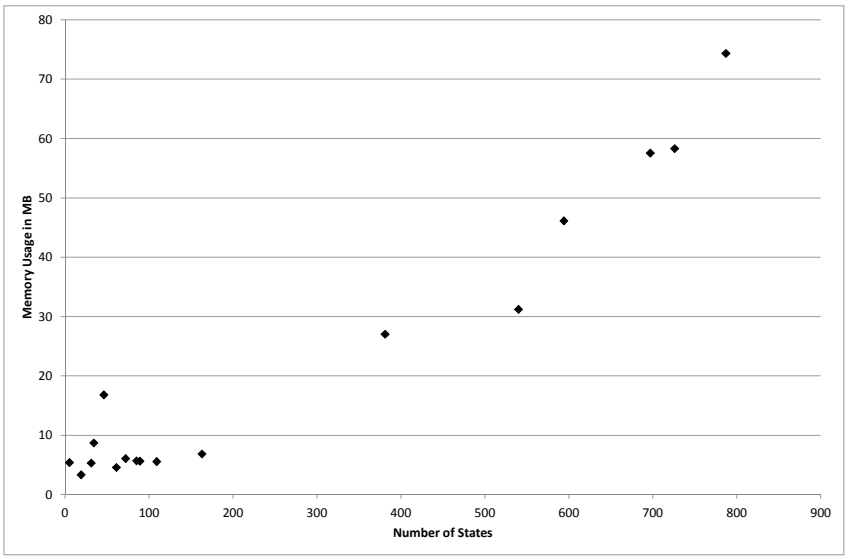

Figure D.5: Memory Usage against Number of States in Property Proving Mode

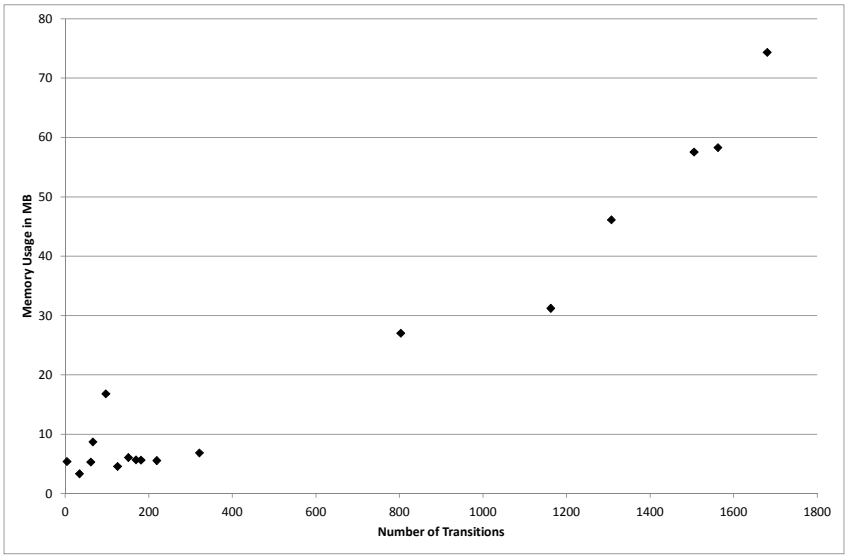

Figure D.6: Memory Usage against Number of Transitions in Property Proving Mode

Average CPU Usage

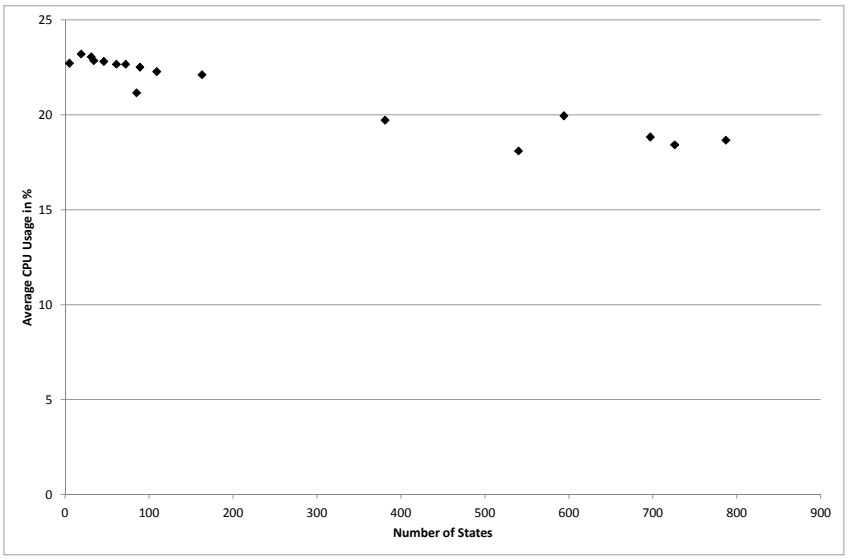

Figure D.7: Average CPU Usage against Number of States in Property Proving Mode

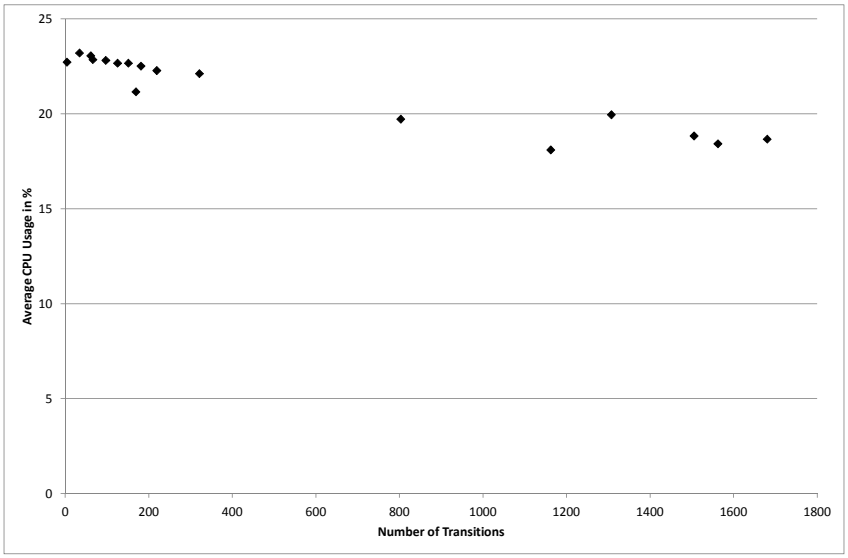

Figure D.8: Average CPU Usage against Number of Transitions in Property Proving Mode

D.4 Data Collected in the Second Part

Increasing the number of objectives to prove on the full model, therefore:

- number of states fixed to 787
- number of transitions fixed to 1680

Note that adding one requirement can result in adding more than one objective, because the requirement might cover multiple equipment parts.

Table D.4: Data Collected in Second Part for Property Proving

Step	time [s]		memory usage [MB]			CPU [%]	objectives
	overall	sldv	low peak	high peak	difference	av. cpu usage	proven
A1	190.78	2.00	550.60	635.52	84.92	18.41	1
	193.15	1.00	580.98	641.15	60.16	18.80	
	193.08	2.00	588.93	649.39	60.46	18.69	
A2	135.47	5.00	543.72	641.63	97.90	18.01	3
	137.70	4.00	561.77	656.07	94.30	17.90	
	138.18	5.00	575.64	661.79	86.14	17.71	
A3	142.81	4.00	584.00	677.27	93.27	17.55	5
	138.26	5.00	612.38	683.30	70.92	18.17	
	140.24	4.00	622.52	706.16	83.65	17.76	
A4	165.58	8.00	270.63	503.77	233.14	17.65	7
	143.63	8.00	459.29	522.57	63.28	17.96	
	143.58	8.00	472.18	532.05	59.88	18.07	
A5	148.79	7.00	480.00	538.88	58.88	17.23	9
	144.41	7.00	493.75	545.80	52.05	18.07	
	145.75	7.00	502.13	551.38	49.25	18.10	
A6	151.39	7.00	505.77	555.59	49.82	17.45	11
	143.70	7.00	516.73	562.71	45.99	18.06	
	145.79	7.00	524.84	570.30	45.46	18.04	
A7	149.37	7.00	529.44	574.29	44.85	17.84	13
	146.38	7.00	540.38	580.80	40.42	18.01	
	147.36	8.00	547.57	587.29	39.72	18.19	

Table D.5: Requirements Added in Each Step in Second Part

Step	Annotation
	Requirements Proven
A1	ID1
A2	ID1, ID2
A3	ID1, ID2, ID3
A4	ID1, ID2, ID3, ID6
A5	ID1, ID2, ID3, ID6, ID7
A6	ID1, ID2, ID3, ID6, ID7, ID8
A7	ID1, ID2, ID3, ID6, ID7, ID8, ID9

D.5 Resulting Graphs from the Second Part

Overall Time

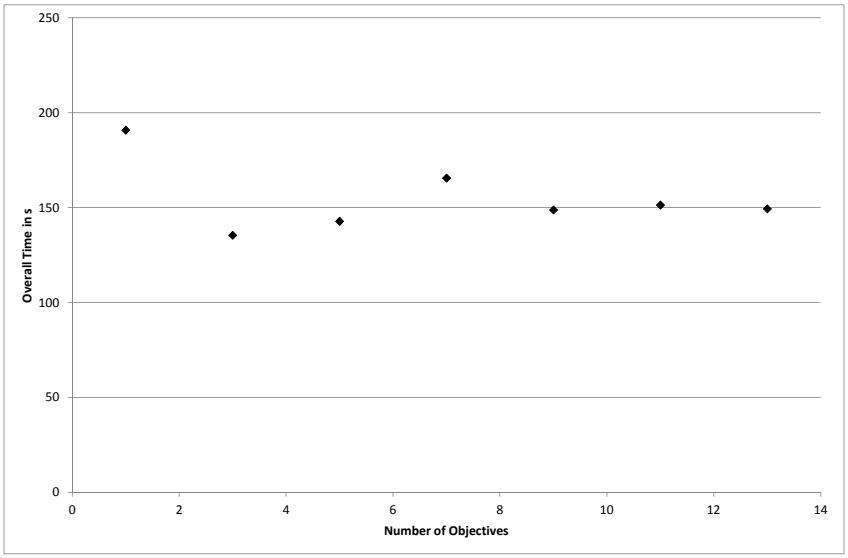

Figure D.9: Overall Time against Number of Objectives in Property Proving Mode

SLDV Analysis Time

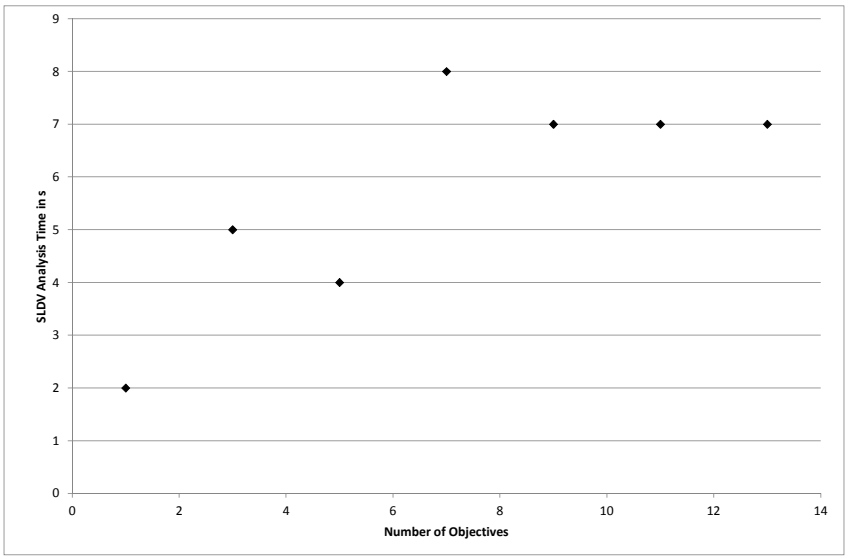

Figure D.10: Analysis Time against Number of Objectives in Property Proving Mode

System Memory Consumption

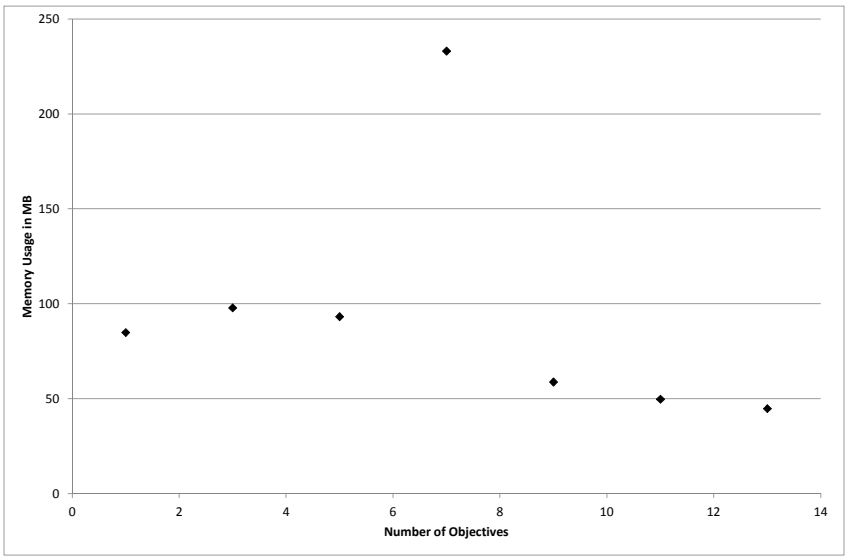

Figure D.11: Memory Usage against Number of Objectives in Property Proving Mode

Average CPU Usage

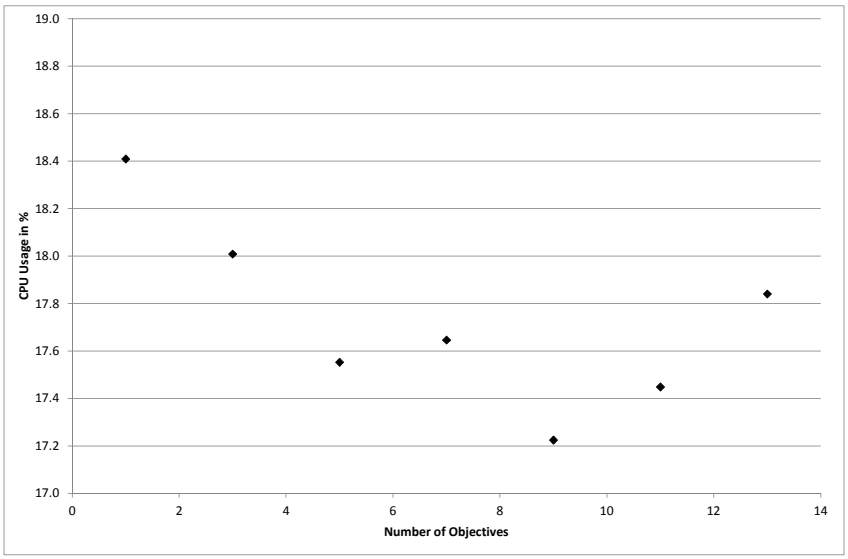

Figure D.12: CPU Usage against Number of Objectives in Property Proving Mode

Bibliography

[1] William Aldrich. 'Using Model Coverage Analysis to Improve the Controls Development Process'. In: *AIAA Modeling and Simulation Technologies Conference, Monterey (US)*. The Mathworks Inc. 2002.

[2] Anne Angermann et al. *MATLAB - Simulink - Stateflow*. 6th ed. München: Oldenbourg Wissenschaftsverlag GmbH, 2009. ISBN: 978-3-486-58985-6.

[3] Goran Begic. *Webinar: Introduction to Simulink Design Verifier*. The Mathworks Inc. URL: http://www.mathworks.co.uk/webex/recordings/link_desver_061207/index.html.

[4] R. Black and J.L. Mitchell. *Advanced Software Testing - Vol. 3: Guide to the ISTQB Advanced Certification as an Advanced Technical Test Analyst*. Rocky Nook, 2011. ISBN: 9781457112201. URL: http://books.google.co.uk/books?id=QF2ZtucOLikC.

[5] André B. Bondi. 'Characteristics of scalability and their impact on performance'. In: *Proceedings of the 2nd international workshop on Software and performance*. WOSP '00. ACM, 2000, pp. 195–203. ISBN: 1-58113-195-X. DOI: 10.1145/350391.350432. URL: http://doi.acm.org/10.1145/350391.350432.

[6] J.N. Buxton and B. Randell, eds. *Software Engineering Techniques*. Apr. 1970.

[7] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999. ISBN: 0-262-03270-8.

[8] J. A. Dabney. *Return on Investment of Independent Verification and Validation Study Preliminary Phase 2B Report*. Tech. rep. NASA IV and V Facility, 2003.

[9] Leticia Duboc, David Rosenblum, and Tony Wicks. 'A framework for characterization and analysis of software system scalability'. In: *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ESEC-FSE '07. ACM, 2007, pp. 375–384. ISBN: 978-1-59593-811-4. DOI: 10.1145/1287624.1287679. URL: http://doi.acm.org/10.1145/1287624.1287679.

[10] E. Allen Emerson. '25 Years of Model Checking'. In: ed. by Orna Grumberg and Helmut Veith. Berlin, Heidelberg: Springer-Verlag, 2008. Chap. The Beginning of Model Checking: A Personal Perspective, pp. 27–45. ISBN: 978-3-540-69849-4. DOI: 10.1007/978-3-540-69850-0_2. URL: http://dx.doi.org/10.1007/978-3-540-69850-0_2.

[11] David Harel. 'Statecharts: A Visual Formalism for Complex Systems'. In: *Science of Computer Programming 8*. 1987, pp. 231–274.

[12] Christopher Hellwig. *TRW Automotive Develops and Tests Electric Parking Brake Using Simulink and Simulink Design Verifier*. TRW Automotive Inc. 2009. URL: http://www.mathworks.de/company/user_stories/TRW-Automotive-Develops-and-Tests-Electric-Parking-Brake-Using-Simulink-and-Simulink-Design-Verifier.html.

[13] The Mathworks Inc. *User's Guide: How Simulink Works*. URL: http://www.mathworks.de/help/toolbox/simulink/ug/f7-882.html.

[14] Florian Leitner. *Evaluation of the Matlab Simulink Design Verifier versus the model checker SPIN*. Tech. rep. University of Konstanz, Department of Computer and Information Science, 2008.

[15] *Mathworks Inc. Product Overview*. June 13, 2012. URL: http://www.mathworks.de/products/.

[16] *NuSMV Model Checker*. June 21, 2012. URL: http://nusmv.fbk.eu/.

[17] *Online Newspaper article on new Software Standard DO-178C*. June 13, 2012. URL: http://www.avionics-intelligence.com/articles/2009/10/upgrade-to-do-178b-certification-do-178c-to-address-modern-avionics-software-trends.html.

[18] *Prover Technoloy AB Product Page*. June 21, 2012. URL: http://www.prover.com/products/prover_plugin/.

[19] *SCADE Design Verifier*. June 21, 2012. URL: http://www.esterel-technologies.com/products/scade-suite/add-on-modules/design-verifier.

[20] Sibylle Schupp. 'Lecture Script on Software Engineering SoSe 11'. Institute for Software Systems Hamburg University of Technology (TUHH). 2011.

[21] Mary Sheeran and Gunnar Stålmarck. 'A Tutorial on Stålmarck's Proof Procedure for PropositionalLogic'. In: *Form. Methods Syst. Des.* 16.1 (Jan. 2000), pp. 23–58. ISSN: 0925-9856. DOI: 10.1023/A:1008725524946. URL: http://dx.doi.org/10.1023/A:1008725524946.

[22] *Simulink Design Verifier User's Guide*. The Mathworks Inc. URL: http://www.mathworks.de/help/releases/R2011b/pdf_doc/sldv/sldv_ug.pdf.

[23] Christopher Slack. 'Model Based Design for Fuel System Development'. internal presentation. May 2010.

[24] *SPIN Model Checker*. June 21, 2012. URL: http://spinroot.com/spin/whatispin.html.

[25] *System Information Class for Windows on Matlab File Exchange*. July 4, 2012. URL: http://www.mathworks.com/matlabcentral/fileexchange/26662-system-information-class-for-windows.

[26] Marvin Tunnat. 'Integration modellbasierter Methoden in den Entwicklungsprozess hybrider Flugzeugregelungssysteme am Beispiel des Ventilation-Control-System'. MA thesis. Technische Universität Hamburg-Harburg, 2011.

[27] Volker Turau. 'Lecture Script on Operating Systems SoSe 11'. Institute of Telematics Hamburg University of Technology (TUHH). 2011.

[28] John F. Wakerly. *Digital Design: Principles and Practices*. 4th ed. Pearson Prentice Hall, 2010. ISBN: 978-0-13-613987-4.

Declaration

I, Max Schürenberg, solemnly declare that I have written this bachelor thesis independently, and that I have not made use of any aid other than those acknowledged in this bachelor thesis. Neither this bachelor thesis, nor any other similar work, has been previously submitted to any examination board.

Bristol, August 8, 2012