Architecture-Based Verification of Software-Intensive Systems

Master thesis
Author: Andreas Johnsen
Supervisor: Jayakanth Srinivasan
Examiner: Kristina Lundqvist

Mälardalen University
School of Innovation, Design and Engineering
2010 Jan 20
Västerås Sweden
Abstract

Development of software-intensive systems such as embedded systems for telecommunications, avionics and automobiles occurs under severe quality, schedule and budget constraints. As the size and complexity of software-intensive systems increase dramatically, the problems originating from the design and specification of the system architecture becomes increasingly significant. Architecture-based development approaches promise to improve the efficiency of software-intensive system development processes by reducing costs and time, while increasing quality. This paradox is partially explained by the fact that the system architecture abstracts away unnecessary details, so that developers can concentrate both on the system as a whole, and on its individual pieces, whether it's the components, the components' interfaces, or connections among components. The use of architecture description languages (ADLs) provides an important basis for verification since it describes how the system should behave, in a high level view and in a form where automated tests can be generated. Analysis and testing based on architecture specifications allow detection of problems and faults early in the development process, even before the implementation phase, thereby reducing a significant amount of costs and time. Furthermore, tests derived from the architecture specification can later be applied to the implementation to see the conformance of the implementation with respect to the specification. This thesis extends the knowledge base in the area of architecture-based verification. In this thesis report, an airplane control system is specified using the Architecture Analysis and Description Language (AADL). This specification will serve as a starting point of a system development process where developed architecture-based verification algorithms are applied.
Contents

1 Introduction
  1.1 Background .................................................. 2
  1.2 Purpose .................................................... 4
  1.3 Problem Formulation ....................................... 4

2 Related Work .................................................. 6
  2.1 Deriving Tests From Software Architectures ............... 6
  2.2 Using Software Architecture for Code Testing ............ 7
  2.3 Specification-based Testing of Reactive Software: Tools and Experiments ......................................... 7
  2.4 Towards Software Architecture-based Regression Testing ................................................................. 8
  2.5 Specification-based Test Oracles for Reactive Systems ................................................................. 8
  2.6 Approaches to Specification-based Testing ................. 9
  2.7 UML-based Integration Testing for Component-based Software ......................................................... 9
  2.8 Testing: a Roadmap ........................................... 9
  2.9 An Overview of Integration Testing Techniques for Object-oriented Programs ........................................ 10
  2.10 Selecting and Using Data for Integration Testing .......... 11
  2.11 Design Complexity Measurement and Testing ............... 11
  2.12 Automatically Generating Test Data from a Boolean Specification ................................................... 12

3 Architecture-based Verification Algorithms .................. 13
  3.1 What Should Be Tested and How Should it Be Tested? .... 13
  3.2 Facilitate Regression Testing ................................ 17
  3.3 Overview of Uppaal ........................................... 17
  3.4 Overview of Junit ............................................ 20

4 The Architecture Analysis and Design Language ............... 22
  4.1 An Overview of the SAE AADL ................................. 22
    4.1.1 Component Type ........................................ 24
    4.1.2 Component Implementation ............................. 26

5 A Flight Control System AADL Specification .................. 31
  5.1 Basic Concepts of Flight Control Systems ................ 31
  5.2 The AADL Specification ..................................... 32
  5.3 Specification Verification .................................... 39
    5.3.1 Applying Architecture-based Testing Criteria ......... 40
    5.3.2 Verification Using Uppaal .............................. 42

6 Implementing the AADL Flight Control specification in Java 54
  6.1 Java Implementation ......................................... 54
  6.2 Unit Testing with Junit ..................................... 58
  6.3 Architecture-based Testing of the Implementation upon Junit .................................................... 61

7 The Refined Flight Control System .............................. 63
  7.1 Refined Implementation ...................................... 63
  7.2 Refining the AADL Specification ............................ 65
  7.3 Architecture-based Testing of the Refined Implementation .......................................................... 66
8 Summary and Conclusions
9 Future Work
A Architecture-based Testing Paths
B Architecture-based Uppaal Queries

List of Figures

1 Development Process [28] .................................................. 2
2 Waterfall Model .............................................................. 3
3 Architecture-based Development Model .............................. 4
4 The Uppaal concept .......................................................... 18
5 The Viking Problem .......................................................... 19
6 Junit Test Case Example .................................................... 21
7 Junit Test Result ............................................................... 21
8 Example of a Port Declaration ............................................ 24
9 Example of a Busaccess Declaration .................................... 25
10 Example of a Flow Declaration ........................................... 25
11 Example of Subcomponents Declaration ............................. 27
12 Example of Connection Declaration .................................... 28
13 Example of Modes Declaration .......................................... 29
14 Example of a Call Statement .............................................. 30
15 Airplane rudders: ailerons (blue), vertical fin rudder (red), elevator (yellow) 31
16 Specified Actuators .......................................................... 33
17 Specified Sensors ............................................................. 34
18 Specified Threads ............................................................ 35
19 Specified Process ............................................................ 37
20 Specified Application System ............................................ 38
21 Specified Execution Platform System ................................. 39
22 Specified System ............................................................. 39
23 AADL Graphical Representation ........................................ 41
24 Global Variables ............................................................. 43
25 Yoke Automaton .............................................................. 44
26 Pedals Automaton ............................................................ 45
27 Throttle Automaton .......................................................... 45
28 Autopilot Button Automaton .............................................. 46
29 Autopilot Setting Automaton ............................................. 46
30 Steering Control Thread Automaton .................................... 47
31 Engine Control Thread Automaton ...................................... 48
32 Autopilot Setting Control Thread Automaton ......................... 48
33 LeftAileron Automaton ..................................................... 49
34 Engine Automaton .......................................................... 49
35 Uppaal Verification .......................................................... 52
36 System Class Instance Variables ....................................... 55
37 addPort() method ........................................................... 55
addPortConnection() method .................................. 57
Flight control system creation ................................ 58
setUp() and tearDown() ......................................... 59
TestAddPort() .................................................. 59
TestAddPortConnection() ....................................... 60
Running the Test Suite ........................................ 60
Architecture-based Test Case ................................. 61
GUI of the Refined Flight Control System .................. 65
Signal Properties ............................................. 66
Thread Dispatch Properties ................................... 66
Code Coverage of Test Case, statements covered by green color were executed 68
AADL Graphical Representation ............................. 75

List of Tables

1 Architecture-based Testing Criteria ........................ 15
2 What Should Be Tested and How Should it Be Tested ...... 17
1 Introduction

1.1 Background

Software-intensive systems are systems where software interacts with sensors, actuators, devices, other systems and people [27]. Examples of such systems are embedded systems for avionics, automobiles and telecommunications. What these systems have in common is that they are rapidly growing in complexity and often operating in dynamic non-deterministic environments. Because of the growing complexity of these systems, development process elements such as cost, time and quality are increasingly important. Therefore, the major question is how to optimize development processes in order to reduce cost and time without losing quality or even better, increasing quality. An optimal development process would roughly speaking be, in a technical point of view, a process where system components are created and formed to an error free system. In today’s reality, this is not the case as illustrated in Figure 1 (although the figure is derived from an old investigation, the underlying message is still up-to-date). The figure illustrates a development process of a system where introduced errors (in %), detected errors (in %) and cost of correction per error (in Deutsche Mark) are represented by three graphs. As shown by the graphs, the majority of errors are introduced early in the process (note that errors are introduced before the programming phase) whereas the majority of errors are detected late in the process. Since time and consequently cost of correcting errors increase dramatically the later they are detected, prediction of possible errors is one of the main issues for developers.

![Figure 1: Development Process][28]

The solution to the problem of costly and time consuming error correction is to somehow move the detected errors graph as close as possible to the introduced errors graph since cost of correction is low where the majority of errors are introduced. Errors and problems are detected late in development processes where developers use strategies similar to the traditional waterfall model (shown in Figure 2) where the majority of testing activities takes place at the end of the process [29].

[28]
Architecture-based development approaches (shown in Figure 3), where a system architecture is modeled, analyzed and tested before implemented in order to predict if the system will satisfy the requirements, promise to improve the efficiency of software-intensive system development processes by reducing cost and time, while increasing quality since analysis and testing of architecture models (specifications) allow detection of errors early in the process. In order to preserve the usefulness of a well modeled and verified system architecture, an implementation of the system must be implemented in conformance with the model. Consequently, architecture-based development approaches do not only deal with testing of a system model, but also testing of an implementation, to verify its conformance with the model. Architecture-based testing of the implementation is feasible by generating test cases from the architecture model which can be mapped and applied to the implementation. Test cases must be mapped to the implementation since there is a traceability problem between a model and its implementation. A system architecture abstracts away unnecessary details, so that developers can concentrate both on the system as a whole, and on its individual pieces, whether it’s the components, the components’ interfaces, or connections among components. System architectures can be specified (modeled) by Architecture Description Languages (ADLs) which serve as a mutual communication blueprint and provide an important base for verification and early design decisions since they describe how the systems should behave at a high abstraction level.
1.2 Purpose

The purpose of this thesis is to extend the knowledge base in the area of architecture-based verification (testing from specifications) of software intensive systems by literature and case studies, in order to perform future research within the area. The importance is to get familiar with the state of the art research about testing from specifications, as well as its background. This includes gaining knowledge about modeling verification tools and architecture specification languages. The study should result in the ability to develop verification algorithms that effectively can be applied to a development process of a system, which has its origins in an architecture specification that later progresses to an implementation.

1.3 Problem Formulation

Tasks that have to be done in order to achieve the purpose of the thesis are:

- Detailed literature review on software testing, focused on model based testing and integration testing.
- Participate in a modeling and verification of embedded systems course at Mälardalens University (Västerås, Sweden) in parallel with the thesis work to further give ideas and techniques that can be used in the thesis.
- Study the Junit testing framework and develop testing skills using it in order to build architecture-based testing upon the unit testing framework.
- Study the Architecture Analysis and Design Language (AADL) and build a suitable specification of a system.
• Use knowledge described above to develop algorithms for architecture-based verification.

• Complete a case study where developed algorithms are applied to a system development process. The process will have its origins from the AADL system specification and later progresses into a Java implementation.
2 Related Work

The underlying goal of the thesis was to extend the knowledge base in architecture-based verification of software intensive systems. The first step to achieve this goal was to study the overall area of software testing and the state of the art research about testing from specifications as well as the background of testing from specifications. I studied the overall area of software testing by reading the book “Art of Software Testing” [4], which not only deals with testing techniques but also how psychology influence testing. To be able to perform a case study about architecture-based verification I was handed, by my supervisor, a list of papers focused on model based testing and integration testing. These papers gave me valuable ideas about different testing techniques that could be used during the thesis, which will be described in section 3. In this chapter, I summarize some papers to give the reader an insight of my literature studies and a better understanding of my work. Note that some of the summarized papers below present large and complex research projects and thus can the summaries be hard to thoroughly understand.

2.1 Deriving Tests From Software Architectures

Jin Z. and Offut J. presents in [24] an architecture-based testing technique to test software. Software architectures abstract away details from applications so the applications can be viewed as sets of components with connectors that describe the interactions among components. The architecture of a system generally contains of four elements: components, interfaces, connectors and configurations. Components are computational units in a system that interacts through their interfaces where interfaces are logical points of interactions between a component or connector and its environment. Connectors are defined as the interactions within the architecture whereas configurations define how each element is combined to form an architecture. Architecture description languages are used to model software architecture for analysis and development. The authors distinguish architecture testing from system testing because system testing test the overall system to see if it meets its requirements while architecture testing attempts to test the interactions/relations among components at the architecture level. Their approach defines general testing criteria that define the test requirements, which are used to generate test inputs. Their technique generates test criteria from traditional data-flow and control-flow criteria by using properties from data flow and control flow, at the architecture-level. The properties are data flow reachability, control flow reachability, connectivity and concurrency.

Data flow reachability: “A data element should be able to reach its designated target component from its source component through the connectors. The data element should reach the target component without having its value modified.”

Control flow reachability: “Every architecture element should be able to reach its designated next element on its control thread.”

Connectivity: “A component or connector interface with either no next element or no previous element is said to be “dangling”. Dangling components and connector interfaces indicate potential problems because they may not be connected to other components or connectors, which causes a discontinuity in the data or control flow.”

Concurrency: “Interactions that in isolation are deadlock free can interact in such a way as to cause a deadlock situation. The system should be deadlock free.”
From these properties they define five general architecture-based test criteria that they apply on a behavioral graph derived from the ADL Wright. When applying the criteria on the graph, test paths are derived. These tests can later be applied to the implementation, by mapping the architecture description to the implementation.

2.2 Using Software Architecture for Code Testing

In paper [16], Miccini H. et al. presents a fully developed approach, which uses a software architecture as a reference model for testing the conformance of the implementation with respect to the architecture specification (architecture reference model). Their goal is "to provide a test manager with a systematic method to extract suitable test classes for higher levels of testing and to refine them into concrete tests at the code level".

Their approach to software architecture conformance testing is based on four steps. The first step is to look at the software architecture dynamics, which is modeled by Labeled Transition Systems (LTSs), and define different "obs-functions" (observation-functions) over the software architecture model. Obs-functions abstract away actions (labels) that are uninteresting in a particular testing criterion. The second step is to apply the obs-functions to the LTS which derives a suitable LTS abstraction, called Abstract LTS (ALTS). ALTS is a reduced LTS, that by concentrating on relevant features and abstracting away those who are not, offers a specific view of the software architectural dynamics. The third step is to derive an adequate set of tests from the ALTS. This is done by deriving a set of paths that appropriately covers the ALTS according to the testing criterion. Although an ALTS has relative small dimensions, a tester has to carefully derive paths since an ALTS just specifies behavioral of the architecture at a higher abstraction level. One complete path in an ALTS corresponds to several possible paths in the LTS, which corresponds to several possible architectural tests. The fourth and last step is to run these high level tests on the implementation by converting the architectural tests into code-level tests, to see if the implementation conforms to its architecture model. If some interactions specified in the architectural level isn’t allowed in the implementation, the implementation doesn’t conform the architecture model.

2.3 Specification-based Testing of Reactive Software: Tools and Experiments

Jagadeesan L. J. et al. describes in paper [11] a specification-based testing tool they have built for automatic testing of reactive system software, which often is safety-critical software. Their tool tests the applications for violations of safety properties expressed in temporal logic. Specification-based testing is a typical approach for testing of telephone switching systems, which are reactive systems. So by applying their tool on several implementations of an "automatic protection switching system", they were able to assess the suitability and advantages of their approach. Their tool contains of finite state machine oracles that are produced by specifications of the safety properties (expressed in temporal logic). The state machines generates inputs to the reactive system to see if the output of the system violates the safety properties. The testing process between the system and the oracle state machine is generated and coordinated by a test harness that is produced automatically from the safety property and a description of the I/O signals. Their tool automatically alert and inform the user if a violation of the safety property has occurred. It is shown that their tool is highly effective at finding defects in the implementation but it still needs some work before
an industrial study can be made.

2.4 Towards Software Architecture-based Regression Testing

In paper [15], Muccini H. et al. explores how regression testing, at the software-architecture level, can be systematically used in order to reduce cost of retesting modified systems. In general, the authors describe two goals, 1) how existing implementation-level test cases can be reused to test the conformance between modified code and the architectural specifications, and 2) how to reuse architecture-level test cases to test the conformance of the source code with respect to the evolved software architecture. Their approach to address both goals relies on integrating code-level regression testing with architecture-based regression testing. They use selective testing technique for code-level regression testing and use the same logical steps for architecture-based regression testing. The first goal is achieved by the following basic steps.

The first step is to represent the program P and the modified program P' in to graphs which is a common approach for regression testing.

The second step is to compare the graphs and look for changes in nodes and edges.

The third step is to build a test history report that records which nodes and edges in graph P has been covered by the original test cases during their execution over P.

The last step is to select test cases for P' by the information gathered in the second and third step. The idea here is if execution of P on a test case t ∈ T covers a node that is modified in P', then t needs to be retested on P'.

The second goal is achieved by following steps.

The first step is to create a new software architecture specification for the evolved architecture S'.

The second step involves earlier applied testing criterion on S now applied to S' which abstracts the software specification, in order to show only behaviors/components one want to test.

The third step is to compare the original architecture specification with the modified and look for topologic and behavioral changes.

The fourth step is to select architectural test cases (ATCs) for S'. Here the idea is that those ATCs from the original software architecture that are affected by the modification are retested in P' which is the implementation for S'.

The fifth step is to map the ATCs into code-level test cases to be able to create test cases T' for S'. The last step is to run T' over P' and evaluate the result of the software architecture-based regression testing.

2.5 Specification-based Test Oracles for Reactive Systems

Output from a test execution is typically examined by someone visually, to determine if the system behaved correctly. In paper [18], Richardson, D. J. et al. presents a specification-based testing approach that they are developing, to be applicable on several language paradigms and several specification languages. The approach is to derive test oracles from multi-specifications in conjunction with testing to represent an oracle for each test class specified by a testing criterion. Derivation of test oracles from specifications entails in mapping test data, which is in the implementation name space, to the oracle name space (the specification name space). Execution of the test data produces an execution profile which is produced by the test monitor. The test monitor requirements are determined by the requirements of the oracles derived from each specification language. These requirements are control points and data states to be
mapped, reflected in the mappings for each specification. The execution profile is then verified by the oracles related to test classes that are satisfied by the test data. Their approach is illustrated by testing of an elevator system implemented in Ada, specified in Real-time interval logic (describes temporal logic) and Z (describes scheduling of services).

2.6 Approaches to Specification-based Testing

Paper [17] by Richardson D. et al. propose four approaches to specification-based testing by extending fault- and error-based techniques, which are implementation-based testing techniques, to be applicable to formal specification languages. So the methodology of their approach is to combine implementation-based and specification-based to derive a "new" (paper written 1989) specification-based testing technique. The four approaches are named specification/error-based testing, specification/fault-based testing, oracle/error-based testing and oracle/fault-based testing. Specification/error-based testing and specification/fault-based testing are testing techniques, which goal is to detect errors/faults in specifications or errors/faults that derives from misunderstanding a specification, by applying fault- and error-based technique to the specification and implementation. Oracle/error-based testing and oracle/fault-based testing on the other hand treats the specification as an oracle to be violated by applying error- and fault-based technique to the implementation.

2.7 UML-based Integration Testing for Component-based Software

In paper [20] Wu Y. et al. presents a model for describing component-based software, when source code are not available, by using UML collaboration and sequence diagrams to represent interactions among different objects in a component. The model combines statechart diagrams with collaboration diagrams to give a further precise model of a component’s behavior. They propose a test model to test interfaces, events, context-dependence relationships and content-dependence relationships. Interfaces of components are access points that provide different operations of the component. Each interface of an integrated environment should be tested at least once. Events invoke interfaces in response to the incident and can be triggered by a different interface through an explicit user input or through an exception. Their test model presents that every event in the system has to be covered by some test. Context-dependence relationships are relationships where an event e1 triggers, directly or indirectly, an event e2. Content-dependence relationships are relationships between two interfaces that have a data-dependence relationship. Although interfaces and events specification can be obtained without source code, information for content- and context-dependence relationships is not available. These two elements are important for component integration testing and effective in finding faults. So by using the UML based diagrams to capture component relations, these two elements can be examined.

2.8 Testing: a Roadmap

In report [9] Harrold M.J. briefly presents the state of the art (paper written year 2000) of software testing and also gives some directions in what should be developed. They represent a roadmap for testing that leads to the destination: "providing practical methods, tools, and processes that can help software engineers develop high-quality software". The roadmap contains of seven different areas: testing component-based systems, testing based on precode artifacts, testing evolving software, demonstrating effectiveness of testing
techniques, establishing effective processes for testing, using testing artifacts and other testing techniques. The "testing component-based systems" section informs the problem of increasing use of component-based systems. The issue is the availability of the components source code. The component provider has access to the source code but the component user typically does not. Standards for representing, computing and storing testing information about a component should be developed. The "testing based on precode artifacts" section discusses the use of software's architecture for testing. Increased size and complexity of software systems has led to emerging of formal specifications for software architectures. Effective architecture testing techniques such as regression testing, assess testability, integration and unit testing are being developed and can facilitate dynamic analysis. The "testing evolving software" section describes how regression testing can be more efficient and effective. Regression testing stands for as much as one-third of the total cost of a system development. To make regression testing more efficient we need to develop techniques that can be applied on different abstractions of software such as its architecture or requirements. We also need to develop techniques that can reduce test suites to a desired coverage of the code. Techniques for assessing the testability of both software and test suites, to be able to evaluate alternative designs, also have to be developed. The "using testing artifacts" section describes how artifacts from testing such as execution traces of the software's execution with test cases can be stored for retesting modified software. The paper ends with some points to improve development of efficient methods and tools. Software technology takes as much time as 18 years on average to be transferred into practice. This time can be shortened by researchers working with the industry by develop methods and tools that implement the techniques and can be used to show their efficiency. Industrial systems are large and complex so an important criterion is that these tools and methods can be scalable to large systems.

2.9 An Overview of Integration Testing Techniques for Object-oriented Programs

In paper [6] Chan W.K. et al. gives an overview of the "current" (written 2002) work of integration testing for object-oriented programs. The paper covers state-based testing, event-based testing, fault-based testing, deterministic techniques, reachability techniques and formal techniques. Object-oriented programs can be tested at four different abstraction levels and this paper focus on testing of the cluster level since the implementation- and class-level have been widely studied. The cluster level concerns about the integration of classes and mostly focused on synchronization of components and interclass method invocation. The briefly explained testing techniques are:

- State-based testing models an integrating system by interacting finite state machines. The models can then be verified for selected properties such as "deadlock free".

- Event-based techniques views on the relationships between pairs of synchronization events. In this testing paradigm, relationships between events build a testing framework (harness) to check for violations.

- Mutation testing (fault-based testing) is a testing approach where the programs are modified (mutated) according specific mutation rules. Then the mutants (modified program) are tested to verify that different results are obtained from the same test cases. This approach is successfully applied to an imperative program paradigm but need some
further work to be able to apply it on an objective-oriented paradigm for integration testing.

- Deterministic testing is a technique that forces the synchronization to be executed in a certain order. A deterministic oracle checks the deterministic result.

- UML-based techniques contain useful diagrams, which express the relationships between method invocations, for integration testing.

- Data flow analysis is proposed to address the concern of whether variables are appropriately created and used. Data flow analysis can be dynamically approached by inserting probes to be able to analyze the actions on variables during program execution.

The techniques explained are mostly extended techniques used in the imperative programming paradigm.

**2.10 Selecting and Using Data for Integration Testing**

Harrold M.J and Soffa M.L describes in [10] a testing tool for integration testing based on data-flow testing. Integration testing is testing of a modular program that contains procedures that interact with other procedures. Not only do the programs procedures need to be unit tested, errors can still exist in the procedures interfaces. Their approach for integration testing is to extend dataflow testing (definition-use pairs) to interprocedural testing by developing both an analyze technique and a testing technique. The analyze technique computes the required information about the interprocedural dependencies of definition-use pairs, for both direct and indirect dependencies (one procedure "A" can have direct dependencies with a procedure "B" that has direct dependencies with another procedure "C", which means that procedure "A" has indirect dependencies with "C"). The method for computing interprocedural dependencies has four steps, namely 1) abstract control-flow and dataflow information, 2) represent interprocedural control flow and data flow, 3) get interprocedural information, 4) compute interprocedural definition-use information. The testing technique uses the information from the analyze technique to select and execute subpaths across procedure boundaries, to determine if there is a definition-clear subpath from the definition to the use. The technique of their tool supports recursive procedures.

**2.11 Design Complexity Measurement and Testing**

In article [14], McCabe T.J. and Butler C.W. proposes three design metrics, module design complexity, design complexity and integration complexity that are derived from the mathematical technique "cyclomatic complexity" to be applicable on architectural design of a system. The concept of these design metrics is to calculate the design complexity and drive a testing process. They also propose a structured integration testing methodology that utilizes the design metrics to produce a testing strategy from the design specification. Design metrics are important for software engineers to consider the design complexity and understand the implication of it. Cyclomatic complexity is a measure, derived from a flowgraph, of the number of basis paths through a program (number of decision statements plus one). Module design complexity is a design metric which is defined as the cyclomatic complexity of its reduced graph of a component. There are four reduction rules that are performed to eliminate complexity in the graph that doesn’t influence the interrelationship between design modules. Design complexity
metric can be calculated from a structured component chart when the module design complexity of each individual component has been determined. Design complexity is defined as the sum of its own module design complexity and its descendant’s module design complexity. The third and last design metric, integration complexity, is a metric that quantifies a basis set of tests that should be done to qualify the design. Integration complexity is in general defined as the design complexity minus the number of modules plus one. The testing methodology they propose, which uses the described metrics, can be applied at two levels, module integration testing and design integration testing. Module integration testing is testing of a module and its immediate subordinates. It describes the amount of tests required to qualify a module’s integration with subordinate modules. Design integration testing determines the amount of effort for testing a design.

2.12 Automatically Generating Test Data from a Boolean Specification

In paper [19], Weyuker E. et al. presents a tool they have developed that automatically generates test sets to test implementation faults that they call "variable negation faults". The test sets are automatically generated, independent of the format, from a Boolean specification. This is done by the tool converting the Boolean specification formulae to disjunctive normal form, also known as sum-of-products form. The resulting formulae is then used to generate test cases. Their strategy, which is a non-deterministic strategy, is to select test cases that demonstrate the meaningful impact of each literal occurrence on each possible value of a Boolean formula. Literals are each occurrence of a variable or its negation in a Boolean formula and a literal is said to have a meaningful impact on the value of a Boolean formula if a change of the literal’s value, and no change to any other literal, would result in a change of the formula’s value. Their strategy can be viewed in another way, if there are no test cases that demonstrate the meaningful impact on the formula’s value for a given literal occurrence, a literal can be changed without any noticeable effect and thus the implementation can be wrong and would go undetected. To assess the effectiveness of their tool they use mutation analysis which is a fault-based testing technique. The approach is to mutate the program by doing small syntactic changes that can represent typical faults, which should produce a different output for the same input. If the test data can detect the faults in the mutants it would indicate an effective testing strategy.
3 Architecture-based Verification Algorithms

Several ideas and testing techniques caught my interest while I was studying literature to get into the subject, where some of them seemed to be more applicable for the thesis than others. Additionally, I participated in a Ph.D. course "Modeling and Verification of Real-time Systems" while working on the thesis, which also gave me valuable techniques and ideas to use. This chapter describes developed verification algorithms used in the thesis, by answering two questions: what should be tested and how should it be tested?

3.1 What Should Be Tested and How Should it Be Tested?

Architecture-based testing, by Jin Z. and Offut J [24], deals with testing of interactions at the architecture level. Based on traditional data-flow and control-flow testing criteria, they define architectural properties that should be tested:

- Data flow reachability: A data element should be able to reach its designed component, from its source component, unmodified.
- Control flow reachability: An architecture element should reach its next element on its control thread.
- Connectivity: A component interface with no next or previous element may not be connected to other components, where "dangling" components can cause problems.
- Concurrency: deadlock free.

From these properties they define architecture relations which are used to define testing paths whereas the testing paths are used to define architecture-based testing criteria. These testing criteria with underlying test paths and relations are defined for general ADLs, which can be interpreted to conform a specific ADL. Since these criteria will be applied to an AADL specification, interpretation must be performed in conformance with the AADL standard. The defined criteria for general ADLs take into consideration that connectors have explicit interfaces, which is something that the AADL standard does not support (overview of AADL in section 4). Therefore, interfaces of connectors in following definitions below are seen as implicit interfaces. Defined relations and properties based on data transfer, control transfer and execution sequencing rules (interfaces that have execution rules such as parallel or sequence execution) are:

1. Component/Connector_Internal_Transfer_Relation: exists if there is a data or control transfer between two interfaces of a component/connector.
2. Component/Connector_Internal_Sequencing_Relation: exists if two or more interfaces of a component/connector have to execute by rules such as parallel or sequence execution.
3. Component/Connector_Internal_Relation: exists if relations 1 or 2 exist between interfaces of a component/connector.
4. Component_Connector_Relation: exists if an interface of a component is coupled with an interface of a connector.
5. **Direct Component Relation**: exists if there is a **Connector Internal Relation** between components A and B where A and B have Component Connector Relations to the connector.

6. **Indirect Component Relation**: exists between components A and C if there is a Direct Component Relation between A and B as well as between B and C where a Component Internal Relation in B connects the direct component relations.

From these relations and properties, six testing paths are derived:

1. **Component/Connector Internal Transfer Path**: If a component/connector has a Component/Connector Internal Transfer Relation, there exists a path between two interfaces of the component/connector.

2. **Component/Connector Internal Sequencing Path**: If a component/connector has a Component/Connector Internal Sequencing Relation, there exists a path between two interfaces of the component/connector.

3. **Component/Connector to Connector/Component Path**: If a component and connector have a Component Connector Relation, there exists a path from the component/connector to the connector/component.

4. **Direct Component to Component Path**: If two components have a Direct Component Relation, there exists a path between the components via a connector.

5. **Indirect Component to Component Path**: If two components A and C have an Indirect Component Relation, there exist a path from A to C via a component B and two connectors (A-connector1-B-connector2-C).

6. **All Connected Component Path**: If four or more components are connected through relations, there exists a path through all components via connectors.

These paths, which identify architectural relations and properties, are testing requirements in order to assure interactions behavior of the architecture. Jin and Offut define five architecture-based testing criteria which when applied to ADL specifications derives testing paths where the criteria requires a test suite to fully exercise all identified paths (relations and properties). The architecture-based testing criteria are shown in Table 1.
Table 1: Architecture-based Testing Criteria

<table>
<thead>
<tr>
<th>Architecture-based testing criteria</th>
<th>Requires</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual_Component_Interface_Coverage</td>
<td>Tests to cover all Component_Internal_Transfer_Paths and Component_Internal_Sequencing_Paths</td>
</tr>
<tr>
<td>Individual_Connector_Interface_Coverage</td>
<td>Tests to cover all Connector_Internal_Transfer_Paths and Connector_Internal_Sequencing_Paths</td>
</tr>
<tr>
<td>All_direct_Component_to_Component_Coverage</td>
<td>Tests to cover all Component/Connector_to_Connector/Component_Paths and Direct_Component_to_Component_Paths</td>
</tr>
<tr>
<td>All INDIRECT_Component_to_Component_Coverage</td>
<td>Tests to cover all Indirect_Component_to_Component_Paths</td>
</tr>
<tr>
<td>AllConnected_Component_Coverage</td>
<td>Tests to cover all AllConnected_Component_Paths</td>
</tr>
</tbody>
</table>

As described above, paths for testing and simulation are derived when these testing criteria are applied to an ADL specification. The idea of derived paths is to test and simulate the paths to look for behaviors that oppose relations that generated the paths. For example, if a path is derived from a data transfer relation, testing of the path entails in looking for behaviors that oppose the data to reach its designated element from its source element unmodified. Furthermore, these tests can later be applied to the implementation-level where paths are exercised by generated inputs instead of simulations.

Miccini H. et al. [16] tests the conformance of an implementation with respect to a model (modeled by labeled transition systems) by testing if interactions specified in the model are allowed in the implementation. Since this is precisely what is done when test paths described above are applied to the implementation level, it can be used for conformance testing. Tests derived from the architecture level have to be converted to be applicable on the implementation level, since there is a traceability problem between a specification and its implementation. This problem is controlled in the thesis by using equal name space at both levels. Structural testing of Java implementations is feasible by the Junit framework described in chapter 4.3.

Ouimet M. describes in [21] the importance of a specification’s completeness and consistency properties. Ouimet M. describes criteria, for an embedded system specification to be complete and consistent, as:

"In the context of the specification of embedded systems, completeness of the specification is defined as the specification having a response for every possible class of inputs. In the same context, consistency is defined as the specification being free of contradictory behavior".
Ouimet applies these criteria to models specified by the Timed Abstract State Machine (TASM) language which is a language that extends the theory of abstract state machines with time. In the context of abstract state machines, completeness is defined as "a rule will be enabled for every possible combination of its monitored variables" whereas consistency is defined as "no state having more than one transition enabled at the same time"[21], i.e. no more than one rule is enabled at the same time. State machine transitions may have rules with guards where a transition is enabled when the rule’s guard evaluates to the Boolean value true.

As mentioned before, I participated in a course "Modeling and Verification of Real-time Systems" where we studied how to use the model checker Uppaal (overview of Uppaal described in section 3.3) for verification of real-time systems. In Uppaal are systems modeled by timed finite state automata and verified by branching temporal logics, or Computational Tree Logic (CTL) to be specific. A model is verified by giving CTL queries in form of path formulae and state formulae [22]. As the names states, state formulae are expressions that describe properties of individual states while path formulae are expressions that describe properties of paths. Path formulae can be divided into reachability, safety and liveness properties. Reachability property formulae test whether a formula can be satisfied by a reachable state along some path. Safety property formulae test whether "something bad will never happen" whereas liveness property formulae test whether "something will eventually happen" [22]. For example, a flight control system model could be tested by asking if it’s possible to be in a situation where the autopilot controls the plane incorrect and can’t be disengaged, or asking if the auto pilot engage button is pressed results in that the plane will eventually be automatically controlled, which are safety and liveness properties respectively. These formulae are more thoroughly explained in section 3.3.

In order to link completeness and consistency with architecture-based verification and the Uppaal toolbox, an architecture specification specified in AADL could be translated to automata/automaton (finite abstract state machines) in Uppaal to be verified for completeness and consistency using CTL. Furthermore, Uppaal provides easy deadlock checking which can be used to verify the translated specification for concurrency properties.

Properties that should be tested and how these properties should be tested are summarized in the following table:
Table 2: What Should Be Tested and How Should it Be Tested

<table>
<thead>
<tr>
<th>Property</th>
<th>Verification technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data flow reachability</td>
<td>Apply architecture-based testing criteria to the AADL specification</td>
</tr>
<tr>
<td>Control flow reachability</td>
<td>Apply architecture-based testing criteria to the AADL specification</td>
</tr>
<tr>
<td>Connectivity</td>
<td>Apply architecture-based testing criteria to the AADL specification</td>
</tr>
<tr>
<td>Concurrency</td>
<td>1) Apply architecture-based testing criteria to the AADL specification, 2) Translate the AADL specification to an Uppaal model and use CTL to verify the property</td>
</tr>
<tr>
<td>Completeness</td>
<td>Translate the AADL specification to an Uppaal model and use CTL to verify the property</td>
</tr>
<tr>
<td>Consistency</td>
<td>Translate the AADL specification to an Uppaal model and use CTL to verify the property</td>
</tr>
<tr>
<td>Conformance</td>
<td>Apply test paths, derived from the AADL specification (when the architecture-based criteria are applied to the specification), to the implementation</td>
</tr>
</tbody>
</table>

3.2 Facilitate Regression Testing

Muccini H. et al. describes in [15] the importance of regression testing at the architecture level in order to reduce cost of retesting modified systems. The underlying problem of their work is the phenomenon of "architectural drift". They explain that during system development it's common to change the implementation, due to tight deadlines, without updating the architecture according to the implementation. When the architecture drifts out of conformance with the implementation, many benefits of an architecture specification are lost. Therefore, modifications and refinements of an architecture specification/implementation should result in an update of the implementation/architecture specification to preserve the valuable use of an architecture specification.

3.3 Overview of Uppaal

Uppaal (freely downloaded at www.uppaal.com), developed by Uppsala University and Aalborg University, is a toolbox for modeling and verification of real-time systems. The basic concept of the Uppaal tool box, shown in Figure 4, is to model a system by timed automata (timed finite-state machines) and to verify model requirements by the Uppaal query language, which is a subset of CTL.
As mentioned, systems are modeled by timed automata where an automaton consists of nodes (states, locations in Uppaal), edges (transitions) and labels (alphabet). Time is modeled by clock variables where all clocks progress synchronously and evaluates to real numbers [22]. In Uppaal, timed automata are extended by discrete variables that can be read, assigned or used for arithmetic operations. Edges can be labeled with events in terms of synchronizations, timing constraints in terms of constraints on clock variables and data variables in terms of transition guards and variable assignments. Since a system can be modeled by several automata, transitions in different automata can be synchronized using Uppaal channels.

A state of a system is defined by "the locations of all automata, the clock constraints, and the values of the discrete variables" [22]. Locations (nodes) can be declared with invariants and stated as urgent or committed. Invariants are atomic properties (e.g. $x<10$) that invariantly holds within a location. In an urgent location, time isn’t allowed to progress which is semantically equivalent with having an invariant $cl<=0$ in a location where "$cl$" is a clock and is reset by all ingoing edges to the location. In a committed location, time isn’t allowed to progress and the next transition must involve one of its outgoing edges. As an example of an Uppaal model, we will look at the so called "Vikings problem". Four Vikings with one torch are supposed to traverse a damaged bridge in the middle of the night in order to get away from the unsafe side to the safe side of the bridge. Since it’s dark and the bridge is damaged, a maximum number of two Vikings can traverse the bridge simultaneously by using the torch. The Vikings traverse the bridge in different time units where Viking 1 needs 5, Viking 2 needs 10, Viking 3 needs 20 and Viking 4 needs 25 time units. Since there is only one torch, two Vikings traverse the bridge together and consequently the time needed is equal to the slowest Viking. Remember there is only one torch, so in order to get all Vikings at the safe side the torch must be carried back by a Viking so two more Vikings can traverse the bridge to the safe side. The underlying question is if all Vikings can be at the safe side within 60 time units. A Uppaal model of this system is partly shown in Figure 5.

- Automaton "A" represents one Viking (all Vikings are modeled by four automata similar to automaton "A" where they differ only by the variable delay) and automaton "B"
represents the torch.

- The automata synchronize by two channels "take!" and "release?" to model a Viking taking or releasing the torch. A channel name with an exclamation mark represents the "sender" where a question mark represents the "receiver", i.e. "take!" can synchronize with "take?".

- "L" is a global integer variable (reached by all automata) that can be assigned the value 0 or 1 to model which side the torch is at where the value 0 represents the unsafe side and 1 represents the safe side.

- "y" is a local clock (reached by one automaton/Viking) that is used with the local integer constant "delay" to model time needed to traverse the bridge. "delay" is assigned to the value 5, 10, 20 or 25 depending on which Viking the automaton represents.

- "unsafe", "safe", "free", "two" and "one" are names of locations where the "unsafe" and "safe" locations represent which side of the bridge the Viking is at. Locations "free", "two" and "one" represents the torch being held by zero, one or two Vikings.

Figure 5: The Viking Problem

The query language used to verify models is a subset of CTL which contains path- and state-formulae. A State formula can be evaluated to valid or invalid for a state without looking at the model’s behavior to or from the state. A state formula is an atomic proposition such as "x==4" or "x>0" where these formulae are valid in a state whenever "x equals four" or "x is greater than zero". Path formulae can be divided into reachability, safety and liveness properties where these formulae use atomic propositions in order to verify properties along paths of a model. A Reachability formula tests whether a state formula "q" can be valid in a reachable state. Reachability properties are verified using temporal operators "E" (pronounced "for some path" or "exist one path") and "<>" (also denoted as "F", pronounced "eventually"). So, in order to verify if state formula "q" is reachable in a model we simple check it by the formula E<> q (pronounced "for some path eventually q holds"). The Vikings problem described above can now be tested by using the reachability formula "E<> (Viking1.safe and Viking2.safe and Viking3.safe and Viking4.safe and time<=60)" where time is a global clock and never reset. Safety formula test whether "something bad will never happen", which can be positively formulated as "something good is always true". Safety properties are verified using
temporal operators "A" (pronounced "for all paths"), "E" and [] (also denoted as "G", pronounced "always" or "globally"). Formula A[] q (pronounced "for all paths globally q holds") is used if the property should hold in all states for all paths whereas formula E[] q (pronounced "for some path globally q holds") is used if the property should hold for all states in at least one path. Liveness formula tests whether "something will eventually happen", where these properties are tested by using temporal operators A, <> and -> (pronounced "leads to"). Formula A<> q (pronounced "for all paths eventually q holds") tests whether q will eventually hold in all paths whereas p -> q (pronounced "whenever p holds eventually q holds") tests whether q eventually holds whenever p holds. The Uppaal query language is extended with easy deadlock checking where a special deadlock state formula can be used with reachability and safety formulae.

3.4 Overview of Junit

JUnit is a unit-level testing framework for writing automated test cases. The framework contains tools for creation, execution and result of repeatable unit tests where related tests can be grouped together in order to give a manageable structure [23]. The JUnit framework contains the following concepts:

- Test case: a related set of tests written as methods
- Test fixture: objects that a test case need to run
- Test suite: a related set of test cases

Optimally, a test case should be written for each Java class. A test case is created by declaring a class as a subclass of the junit.framework.TestCase class. In order to create particular tests, a test fixture and test methods have to be declared. A test fixture is created by creating objects within a method named setUp() where objects that aren’t garbage collected have to be released within a method named tearDown(). Tests are created by declaring methods whose names start with "test" while the testing is carried out by using different assert methods provided by the Junit framework. Furthermore, Junit handles exceptions to pass or fail tests.

As an example, we will look at a simple test case of the java.util.Stack class shown in Figure 6. The TestStack class is a test case since it is a subclass of the junit.framework.TestCase class. The test fixture, which is set up by the test case setUp() method, is an instance of the class under test. Since the object will be garbage collected we don’t need to release it by the tearDown() method. The stack class is generic and in this example chosen to contain String objects. Finally, in the testPush() test method, we simply test the stack class’ push() method by checking if a string object can be pushed on the stack. This is tested by the assertEquals() method, which checks that the value of the test variable is equal to the peeked value. As can be seen, messages can be added to the assert method so faults easily can be distinguished if several assert methods are used in the same test.
Test cases such as the test case described above, can be executed alone or as a part of a test suite by the Junit GUI. An example of the result, after running the test case, is shown in Figure 7.

```java
import java.util.Stack;
import junit.framework.*;

public class TestStack extends TestCase { // test the java.util.Stack class
    private Stack<String> test_stack;

    public void setUp(){
        test_stack=new Stack<String>();
    }

    public void tearDown(){
    }

    public void testPush() { // test the push() method
        String test = "test";
        test_stack.push(test);
        assertEquals("Peeked string should be test", test,
            test_stack.peek());
    }
}
```

Figure 6: Junit Test Case Example

Figure 7: Junit Test Result
4 The Architecture Analysis and Design Language

AADL by Feiler et al. [1][2] was released and published as a Society of Automotive Engineers (SAE) Standard AS5506, in November 2004. It is a textual and graphical language used to model, specify and analyze software- and hardware-architectures of real-time, embedded and high dependability systems. The AADL language is based on a component-connector paradigm that describes components, component interfaces and the interaction between components. A system is modeled as a hierarchy of components where components that represent the application software are mapped onto the components that represent the hardware platform. Properties can be associated with different AADL elements (e.g. components, subcomponents, features, connections etc.) to provide descriptive information about the respective AADL element. Changes to the runtime architecture can be described by modes and transitions of modes. Timing and performance are modeled by explicit properties (such as deadlines, worst-case execution time, arrival rate, period etc.) associated with components and defined concurrency and interaction semantics. Detailed description about abstract paths of information through a system can be defined and analyzed as a flow path.

AADL models can be developed by the OSA TE environment (can be found through http://www.aadl.info) which is an Eclipse-based open-source tool.

4.1 An Overview of the SAE AADL

The AADL language describes system component abstractions including software and hardware to, by Feiler et al. p. 4 [1]:

- "Specify and analyze real-time embedded systems, complex systems of systems, and specialized performance capability systems"
- "Map software onto computational hardware elements"

A component is modeled by a component type and a component implementation. The component type specifies the external interfaces of the component in which other components can interact through. Since the component type only specifies the functional external interfaces, and no internal interfaces or operations, it can be viewed as a "black-box". The component type consists of an identifier, features, extends, flows and properties. These elements and sub clauses will be described later. The component implementation specifies the internal structure of a component and must be coupled to a component type. The component implementation can be viewed as a "white-box" since it shows the internal structure of a component. A component implementation consists of an identifier, extends, refines, subcomponents, connections, call sequences, modes, flows and properties. These elements and sub clauses will be explained later.

The AADL language defines ten types of component abstractions which can be divided into three groups:

Application software

- Process: represents a protected address space containing of threads. Process components can have thread, thread group and data components as subcomponents. A process
must have at least one thread as subcomponent to represent an actively executing component.

- **Thread**: represents a schedulable concurrent unit of sequential executed source code. Execution properties can be associated with a thread such as dispatch protocol and timing. For instance a thread can have a scheduling related dispatch protocol property that is assigned periodic (other supported protocols are: aperiodic, sporadic and background), meaning that repeated dispatches occurs in specified time intervals. A thread can have data components as subcomponents.

- **Thread group**: represents an abstraction for organization of thread and data components. A thread group doesn’t represent a unit of execution, it represents a single reference to a group of threads that have common characteristics. Thread group can have thread, data and other thread groups as subcomponents.

- **Data**: represents data types and data abstractions in source text (target programming language). Data types are used to type ports which are interaction points to other components. Data can have data components as subcomponents.

- **Subprogram**: represents a callable piece of sequentially executed source code that operates on data or provides functions to the component that call it. Subprograms are not used as subcomponents, they are reached by call declarations with parameters in threads or subprograms. Subprogram can’t have any subcomponents.

**Hardware/Execution platform**

- **Processor**: represents hardware with associated software that schedules and executes threads. Processors can have memory subcomponents.

- **Memory**: represents a storage component for executable code and data. Memory components include abstractions of complex storage such as disks as well as randomly accessible storage such as RAM. Memory can have memory subcomponents.

- **Bus**: represents a component that can exchange control and data between processors, memories and devices. It’s a communication channel that interconnects execution platform components. Processor, memory, device and system components can be connected to a bus by require bus access declarations. Furthermore, other bus components can be directly connected to a bus by an access statement. Bus components can’t have any subcomponents.

- **Device**: represents abstraction of a component that interfaces the external environment such as sensors and actuators. Devices may be a simple unit or a complex system that may internally have processors, memory and software. The internal system of a device can be modeled by a system component associated with the device through an "Implemented_as" property. Devices are physically connected to processors via buses and logically connected to application software components. Device components can’t have any subcomponents.

**Generic/Composite**

- **System**: represents an assembly of software, hardware and system components and thus can have such subcomponents.
4.1.1 Component Type

A component type declaration is a component’s externally visible interface and is declared with a unique identifier. The interface contains partly of a "features" sub clause which contains of named interaction points used to exchange data and control to other components. The "features" sub clause contains of four different AADL elements, two of them explained thoroughly:

- **Ports**: represents a communication interface where components exchange directional data, events or both data and events. A port can either be a data port, an event port or an event data port. Data port represents an interface for communication of a data type where the connection between data ports can be immediate or delayed but not queued. Event port represents an interface for communication of events where communication between event ports may be queued. Event data port represents communication of data that associated with an event. Ports are directional and can either be an in port, an out port or an in out port. In port denotes a component’s input, out port denotes a component’s output and in out port denotes both a component’s input and output where the input and output can be connected to different components. An example of an out data port as a feature of a device component named "yoke" is shown in Figure 8. There are two different component type declarations in the Figure, one data component and one device component. Out data port, in yoke device features, is named "yoke_status" and receives "yoke_type" data type.

```
data yoke_type
end yoke_type;

device yoke
  features
    yoke_status: out data port yoke_type;
  end yoke;
```

Figure 8: Example of a Port Declaration

- **Data/Bus Access**: supports modeling of interfaces for access to static data and modeling of hardware components communicating through buses. Access declarations are named and can either be declared with a provides or requires statement. A provides statement denotes that a component provides access to a specific data or bus component that is internal to it. A requires statement denotes that a component requires access to a data or bus component that is external to it. An example of bus access declarations is shown in Figure 9. There are three component types in the Figure: bus component named "ARINC_629", memory component named "SDRAM" and processor component named "PENTIUM". The connectivity of the execution platform through the bus is declared by require bus access statements in the memory and processor component features. Both access statements require bus access to the "ARINC_629" bus component and are identified as "controller_cpu" and "controller_memory".
Figure 9: Example of a Busaccess Declaration

- **Subprogram**: represents an entry point into source text that operates on the component.

- **Parameter**: represents call and return data values that are transmitted into and out of a subprogram component.

A component type also contains of "flows", "extends" and "properties" sub clauses. A flow is logical flow of control and data through components, port connections and data access connections to support analysis such as timing, latency, reliability etc. Flows can be declared with relevant properties in a component by specifying whether the component is a flow source (i.e. the flow begins in the component), a flow sink (i.e. the flow ends in the component), or a flow path (i.e. there exist a flow through a component from an in port to an out port). End-to-end flow declarations are also supported to specify a complete flow path through a sequence of components, port connections and data access connections. Component type flow declarations specify externally visible flow through a component’s port and data access features. An example of a flow declaration is shown in Figure 10. The declaration specifies that a logical flow of data starts (flow source) in the device component named "throttle". The flow is named "flow1" and goes through out data port named "throttle_status" which transmits "throttle_type" data type. The flow is associated with a property which will be described later.

Figure 10: Example of a Flow Declaration

A component type can inherit characteristics from another component type (i.e. a subclass of the original component) by the "extends" sub clause. By declaring a component type as an extension of another component type allows modeling of interface variations of related
"Properties" sub clause provides property declarations, which is information about AADL elements and in this case, as a sub clause of a component type, information about the component type. Other elements that can be associated with property declarations are: component implementations, subcomponents, features, connections, flows, modes, mode transitions, sub-program calls and packages. A property has a name, a type and a value. The name of the property specifies the identifier to it along with information about to which AADL elements it applies. The property type specifies a set of values that is accepted for a given property and each property must be assigned a value or a list of values. There are built-in predeclared properties in the language but creation of new properties is supported. An example of a property statement is shown in Figure 10. The property name is "Latency", property type is time and the property value is "10 Ms". This property specifies that it takes ten milliseconds until an output is available at the output port when the throttle is adjusted.

4.1.2 Component Implementation

A component implementation declaration represents the internal structure in terms of subcomponents and their connections, flow sequences, properties, component modes and mode transitions. Component implementations are coupled with a component type by writing the associated component type identifier with the component implementation identifier separated by a dot. A component type can have zero, one or several component implementations. The component implementation "subcomponents" sub clause represents a component’s internal components. These internal components can themselves have subcomponents which results in a hierarchy that eventually describes a whole system. An example of a subcomponents declaration is shown in Figure 11. The Figure describes a component implementation, identified as "impl", of a component type identified as "ac_computer" (component type "ac_computer" is not shown in the figure) and three component types: "PENTIUM", "SDRAM" and "ARINC_629". The component implementation “impl” describes the internal structure of a system component. The "subcomponents" sub clause specifies three subcomponents: "SDRAM", "ARINC_629" and "PENTIUM", which are identified in the component implementation as "airplane_memory", "airplane_bus" and "airplane_processor" respectively.
Components interact through interface elements specified in the "features" sub clause (sub clause of a component type). The connections between interface elements are explicitly declared within a component implementation sub clause named "connections". Connections have an identifier, descriptor, source and destination. There are three types of connections: port connections, component access connections and parameter connections. The descriptor specifies whether the connection is a port (data, event or data event port), an access (bus or data access) or parameter connection. Port connections represent directional transfer of data and control between components with port relations. An example of data port connections is shown in Figure 12. The system implementation "ac_application.impl" consists of three subcomponents: "airplane_control.impl", "throttle" and "engine". The component types for these components are shown in the figure. The connection declarations states that, 1) there is a data port connection named "C1" from out data port "throttle_status" in device "THROTTLE" to in data port "throttle_status" in process "AC", 2) there is a data port connection named "C2" from out data port "engine_setting" in process "AC" to in data port "engine_setting" in device "ENGINE".

```
system implementation ac_computer.impl
subcomponents
  airplane_memory: memory_SDRAM;
  airplane_bus: bus ARINC_629;
  airplane_processor: processor PENTIUM;
connections
  C1: bus access airplane_bus -> airplane_memory.controller_memory;
  C2: bus access airplane_bus -> airplane_processor.controller_cpu;
end ac_computer.impl;

processor PENTIUM
features
  controller_cpu: requires bus access ARINC_629;
end PENTIUM;

memory SDRAM
features
  controller_memory: requires bus access ARINC_629;
end SDRAM;

bus ARINC_629
end ARINC_629;
```
A Component Access connection represents the path from the component providing access to the component requiring access. An example of a bus access connection is shown in Figure 11. Within the system component "ac_computer.impl" there are three subcomponents: a processor, a memory and a bus component where the processor and memory component requires bus access to the bus component. The connection paths are declared in the connection sub clause and states that, 1) there is a bus access path named "C1" from "airplane_bus" ("ARINC_629") to the requires bus access feature named "controller_memory" in "airplane_memory" ("SDRAM") component, 2) there is a bus access in path named "C2" from "airplane_bus" to the requires bus access feature named "controller_cpu" in "airplane_processor" ("PENTIUM") component.

Parameter connections represent flow of data into and out of subprograms and data flow through a sequence of subprogram calls. Parameter connections can be declared between subprogram parameters or between a data port and a subprogram parameter.

Components can have different states, within a state machine abstraction, with specific configuration of contained components, connections and property value associations. These alternative states are represented by modes in component implementations. A component’s state machine abstraction must contain at least two modes where one mode must be declared as the initial mode. Each mode must have an explicitly defined configuration and an event that cause transition to another mode. An example of a state machine abstraction is shown in Figure 13. The process component "control" has one in data port, one out data port and one in event port. In the "control" implementation, there are two declared thread subcomponents ("AUT" and "MAN") and two declared modes: "manualmode", which is specified to be the initial mode, and "automode". In "manualmode" the process implementation is configured

```
device throttle
features
  throttle_status: out data_port throttle_type;
end throttle;

device engine
features
  engine_setting: in data_port engine_type;
end engine;

process airplane_control

features
  throttle_status: in data_port throttle_type;
  engine_setting: out data_port engine_type;
end airplane_control;

subcomponents
  AC: process airplane_control.impl;
  THROTTLE: device throttle;
  ENGINE: device engine;

connections
  C1: data_port THROTTLE.throttle_status -> AC.throttle_status;
  C2: data_port AC.engine_setting -> ENGINE.engine_setting;
end ac_application.impl;
```

Figure 12: Example of Connection Declaration
to have connections "C1" and "C2" while in automode it is configured to have connections
"C3" and "C4". Transition from "manualmode" to "automode" and vice versa occurs by the
"trigger" in event port.

```
thread automatic
  features
    in_data : in data port;
    out_data : out data port;
  end automatic;

thread manual
  features
    in_data : in data port;
    out_data : out data port;
  end manual;

process control
  features
    input : in data port;
    output : out data port;
    trigger : in event port;
  end control;

process implementation control.impl
  subcomponents
    AUT: thread automatic;
    MAN: thread manual;
  connections
    C1: data port input -> MAN.in_data in modes (manualmode);
    C2: data port MAN.out_data -> output in modes (manualmode);
    C3: data port input -> AUT.in_data in modes (automode);
    C4: data port AUT.out_data -> output in modes (automode);
  modes
    manualmode: initial mode;
    automode: mode;
    manualmode ->{ trigger }-> automode;
    automode ->{ trigger }-> manualmode;
end control.impl;
```

Figure 13: Example of Modes Declaration

Subprogram components are not declared as subcomponents, they are accessed through
call statements. Subprograms are callable pieces of sequentially executed source code that
operates on data or provides functions to the component that call it. Calls are declared in
the component implementation "calls" sub clause. An example of a call statement is shown
in Figure 14. The figure consists of one thread "example.impl" that calls a subprogram
"tank.temp" where the identifier of the call is "get_temp".
Figure 14: Example of a Call Statement

```
thread example
end example;

thread implementation example.impl
calls
{
  get_temp: subprogram tank.temp;
};
end example.impl;

subprogram tank
end tank;

subprogram implementation tank.temp
end tank.temp;
```
5 A Flight Control System AADL Specification

One task of the thesis was to develop an AADL specification of a software-intensive system and to use the specification for architecture-based verification. Since AADL originates from the field of avionics, development of a flight control system was chosen. Instead of doing extensive research about how different systems control airplanes, basic concepts of how different rudders and engine are controlled were derived by surveying the book "Understanding Flight" [25], and adding some auto pilot features. From these basic concepts, described in section 5.1, an AADL fly-by-wire flight control system was specified, which is described in section 5.2.

5.1 Basic Concepts of Flight Control Systems

An airplane typically has four different rudders, as shown in Figure 15, that are controlled by the pilot: one on each wing called ailerons (blue), one on the horizontal stabilizer called elevator (yellow) and one on the vertical stabilizer called vertical fin rudder (red). The left and right aileron moves up and down in opposite directions and controls the rolling movement of the plane. For example, if the right aileron is moved down and consequently the left aileron is moved up, the lift power of the right wing increases while it decreases on the left wing and as a result the airplane rolls to the left. The elevator can be moved up or down and controls upward or downward movements of the plane. An upward or downward movement of the elevator forces the nose of the airplane to lift or lower respectively, which changes the lift power of the wings. A Lift of the nose results in increasing lift power of the wings and decreasing lift power when the nose is lowered. The vertical fin rudder can be moved left or right in order to control left or right yaw movement of the airplane. A left or right movement of the vertical fin rudder forces the nose to the left or right respectively and consequently the plane turns (yaws) to the left or right.

Figure 15: Airplane rudders: ailerons (blue), vertical fin rudder (red), elevator (yellow)
In order to perform efficient and safe maneuvering of a plane these rudders are often used simultaneously. Each rudder is controlled by the pilot from the cockpit typically by a yoke and two pedals. The yoke controls ailerons and elevator while the pedals control the vertical fin rudder. A yoke can be turned left or right and moved forward or backward. Left and right turning movements of the yoke control the ailerons where a left turn of the yoke rolls the plane to the left (left aileron up and right aileron down) while a right turn of the yoke rolls the plane to the right (left aileron down and right aileron up). Forward and backward movements of the yoke control the elevator where a forward movement lowers the nose (elevator down) whereas a backward movement lifts the nose (elevator up). The pedals control the vertical fin rudder in a contradictory manner, i.e. both pedals can’t be pressed at the same time. If left pedal is pressed the airplane yaws to the left (vertical fin left) whereas the airplane yaws to the right (vertical fin right) when right pedal is pressed.

One more vital airplane control element is the power produced by the airplane engine. Power produced by the engine is typically controlled by a throttle lever which can be moved forward to increase engine power or moved backward to decrease engine power.

Flight control systems are commonly built on mechanical or hydro-mechanical links between the control devices (e.g. yoke) and the controlled elements (e.g. ailerons) through cables and hydraulics. But since these systems aren’t software-intensive systems I had to take another approach. Instead, I considered a fly-by-wire system which controls the plane by electrical signals through sensors, actuators and computers, which indeed is a software-intensive system. The basic concept of a fly-by-wire system is that pilot commands (e.g. sensor data from yoke) are read by a computer which computes output signals, transmitted by wire, to different actuators [25]. Actuators convert electrical signals into mechanical actions such as moving a rudder.

In addition, flight control systems usually have auto pilot systems where it is possible to set course, height and speed. The auto pilot system is enabled by the autopilot engage button and controls the plane according to the settings without any interactions with the pilot.

From these basic concepts I developed a flight control system specification (model) without any detailed research about such systems.

5.2 The AADL Specification

As described in the previous chapter, fly-by-wire flight control systems contain of some basic elements, i.e. different sensors that are read by a computer which controls different actuators. Sensors and actuators are modeled by device components where the internal complexity of the device typically isn’t modeled. Since these elements are main parts of a flight control system it’s beneficial to model them first. Specified actuators are each rudder and the engine, shown in Figure 16. Each actuator has a specified input data port from where input signals are derived. Ports are typed with generic data types such as "aileron_type" where declarations of these data types are left out in the figure. For the purpose of testing, engine component is specified with a flow sink declaration (flow ends in the component). Since it’s possible to test time constraints on flow paths by the OSTATE environment (methods for flow testing by the OSTATE environment are described in [26]) an end to end flow path from the throttle lever to the engine is specified with time constraint properties (latency). In this particular case,
the latency property specifies that it takes 450 milliseconds from the point where input data is available to the point where the engine power is set according to the input. As described above, this flow latency property isn’t based on any reason other than to have a flow path to test, i.e. property value is set to a "random" value.

```plaintext
device left_aileron_actuator
  features
    left_aileron_setting: in data port aileron_type;
  end left_aileron_actuator;

device right_aileron_actuator
  features
    right_aileron_setting: in data port aileron_type;
  end right_aileron_actuator;

device vertical_fin_actuator
  features
    vertical_fin_setting: in data port vertical_fin_type;
  end vertical_fin_actuator;

device elevator_actuator
  features
    elevator_setting: in data port elevator_type;
  end elevator_actuator;

device engine
  features
    engine_setting: in data port;
  flows
    flow: flow sink engine_setting { 
      Latency => 450 Ms; 
    };
  end engine;
```

Figure 16: Specified Actuators

Specified main sensor parts are yoke, pedals and throttle as well as additional sensors such as course sensor, height sensor, speed sensor, autopilot button and autopilot setting panel, shown in Figure 17. Course, height and speed sensors are important navigation instruments for pilots but they are not included in the control system for that intention. In order to control the airplane by the autopilot, according to the autopilot settings, these sensors are vital. Each sensor has out data ports except for the "auto_pilot_button" (device "auto_pilot") which has an out event port. When the "auto_pilot_button" is pressed it should trigger a change of mode in the system, i.e. from manual mode to autopilot mode or vice versa.
Reading of sensors and calculation of output to actuators are controlled by software executed on a computer. The software is specified by three threads within one process. Thread declarations are shown in Figure 18, where thread identified as "steering_control" derives input from yoke and pedals (in manual mode) and produces output the each rudder. Thread "engine_control" derives input from the throttle lever (in manual mode) and produces output to the engine. Since there is an explicit end-to-end data flow from the throttle lever to the engine, which goes through the "engine_control" thread, a flow path is declared. The last thread, which is named "ap_setting_control" is the thread that indirectly controls the plane in autopilot mode. Each thread has an in event port where events from the autopilot button is received. These events control the threads weather they should operate in manual mode or autopilot mode (specification described below). In autopilot mode, "steering_control" and
"engine_control" will get input from "ap_setting_control", instead of different sensors (e.g. yoke). "Ap_setting_control" derives input from autopilot setting panel, course sensor, height sensor and speed sensor. "Ap_setting_control" produces output on four ports that are typed with the same data types as yoke-, pedals- and throttle-device ports (which feed data to "steering_control and engine_control"). This means that "steering_control" and "engine_control" can be connected with "ap_setting_control" (in autopilot mode) since they have ports with same data types. Specification of connections in different modes is described below.

```plaintext
thread steering_control
features
  yoke_status: in data port yoke_type;
  left_pedal_status: in data port pedal_type;
  right_pedal_status: in data port pedal_type;
  auto_pilot_button: in event port;
  left_aileron_setting: out data port aileron_type;
  right_aileron_setting: out data port aileron_type;
  vertical_fin_setting: out data port vertical_fin_type;
  elevator_setting: out data port elevator_type;
  ap_data: requires data access ap_setting.impl (Required_Access =>
    access read only));
properties
  Dispatch_Protocol => Periodic;
end steering_control;

thread engine_control
features
  throttle_status: in data port throttle_type;
  auto_pilot_button: in event port;
  engine_setting: out data port engine_type;
  ap_data: requires data access ap_setting.impl (Required_Access =>
    access read only));
flows
  throttle_flow: flow path throttle_status -> engine_setting {
    Latency => 50 Ms;
  };
properties
  Dispatch_Protocol => Periodic;
end engine_control;

thread ap_setting_control
features
  auto_pilot_button: in event port;
  ap_setting: in data port ap_type;
  course: in data port course_type;
  altitude: in data port altitude_type;
  speed: in data port speed_type;
  yoke_status: out data port yoke_type;
  left_pedal_status: out data port pedal_type;
  right_pedal_status: out data port pedal_type;
  throttle_status: out data port throttle_type;
  ap_data: requires data access ap_setting.impl (Required_Access =>
    access read write);
properties
  Dispatch_Protocol => Aperiodic;
end ap_setting_control;
```

Figure 18: Specified Threads

Declaration of the process named "airplane_control" containing the three threads is partly shown in Figure 19. Until now there have not been any component implementation decla-
rations. Since "airplane_control" has three threads as subcomponents, it’s declared in the component implementation "airplane_control.impl". The interfaces of the process contain all ports needed by the contained threads. For example, "steering_control" thread (Figure 18) has an in data port named "yoke_status" with data type "yoke_type". Since the port derives data produced by a component outside of the process component, there must be an interface of the process so the connection can be established, i.e. "airplane_control" must have an in data port with data type "yoke_type" (port identified as "yoke_status"). Furthermore, the process implementation contains declarations of connections, flows and modes. There are two modes declared, "manual_pilot" and "auto_pilot", where manual pilot is declared as the initial mode and change of modes are triggered by events derived on the "airplane_control" process port named "auto_pilot_button" (which described below is connected to the autopilot button device). Connection declarations connect process interface ports and subcomponent interface ports as intended where some connections depend on modes. For example, as mentioned before, "steering_control" thread should derive input from yoke and pedals in manual mode whereas in autopilot mode input should be derived by the "ap_setting_control" thread. This can be reflected in connections identified as "C1" and "C16". Connection "C1" specifies that in "manual_pilot" mode "steering_control" (identified as "S_C" in the process implementation) interface port identified as "yoke_status" is connected with process interface port identified as "yoke_status" (which can be seen below is connected with the yoke). On the other hand, connection "C16" specifies that in "auto_pilot" mode "steering_control" port "yoke_status" is connected to "ap_setting_control" interface port "yoke_status". The flow path through the process is declared in both the component type and component implementation. Flow path declaration in the component type specifies that there is a flow path between "throttle_status" and "engine_setting" interface ports, whereas flow path declaration in the component implementation specifies how this path is connected in the internal structure (via subcomponents).
Software and device components described above can now be formed into a subsystem of the flight control system as shown in Figure 20. The system component identified as "ac_application" connects the process and all devices as intended. The component type contains of a require bus access interface so devices can be bound to a bus (execution platform described below). Since the explicit data flow from the throttle to the engine starts and ends within the component, there is an end-to-end flow declaration in the component implementation. The end-to-end data flow latency property specifies a latency requirement where the total latency of all components in the specified data flow shouldn’t exceed 900 milliseconds.
In order to complete the flight control system, software and device components must be mapped to an execution platform, which is specified as another subsystem of the complete system. The execution platform system identified as "ac_computer" and its subcomponents are shown in Figure 21. Subcomponents of the execution platform are: one processor component identified as "PENTIUM", one memory component identified as "SDRAM" and one bus component identified as "ARINC_629". The system implementation consists of two bus access connections ("C1" and "C2"), which connects memory and processor to the bus. Furthermore, "ac_computer" component provides bus access so devices, which are outside of the system component, can be connected to the bus.
5.3 Specification Verification

In this section, we will look at derived test paths for simulation when architecture-based testing criteria (shown in Table 1) are applied to the specification. We will also look at a translation of the specification into timed automata in Uppaal (presented in section 5.3.2) in order to apply verification algorithms for completeness, consistency and concurrency.
5.3.1 Applying Architecture-based Testing Criteria

Test paths are easiest derived by looking at the textual representation in conjunction with a graphical representation of the AADL specification. A graphical representation of the specified software and devices is shown in Figure 23. As an example of how to derive test paths, consider Direct_Component_to_Component_Path (described in section 3.1) which is defined to be a path if two components "A" and "B" have a Direct_Component_Relation. A Direct_Component_Relation exists if there is a Connector_Internal_Relation between components A and B where A and B have Component_Connector_Relations to the connector. A Component_Connector_Relation exists if an interface of a component is coupled with an interface of a connector (note that connectors don’t have explicit interfaces in AADL so this can be interpreted as coupled with a connector) while a Connector_Internal_Relation exists if there is a data-transfer, a control transfer or a sequencing-rule relation between interfaces of a connector. Since the yoke device and steering_control thread are connected by connection 1 (Figure 23) and there is a data transfer (of yoke_type data type) from yoke to steering_control through connection 1, there is a Direct_Component_to_Component_Path. At the specification-level, these paths are for simulation and testing where the architecture-based criteria (shown in Table 1) require all test paths to be fully exercised. All derived paths can be seen in appendix A.
Instead of simulating every single path (there are more than fifty of them) it is feasible to derive a set of suitable paths that can be exercised so all paths are covered. Deriving suitable paths entails in extracting as much properties and relations a tester can handle simultaneously without slacking assurance that underlying paths will be fully exercised. For example, consider the All_Connected_Component_Path containing components connected by connections 1-4-10-11-13. The path contains following test cases:

**Test case 1:** If autopilot button isn’t pressed (In manual mode), steering control gets data input from yoke and produces output to left/right aileron and elevator. Since there are no specified execution rules in this path, there is nothing that contradicts data and control flow reachability properties which also results in that there are no concurrency property problems.

**Test case 2:** If autopilot button is pressed (In autopilot mode), steering control doesn’t get data input from yoke. This path can be concluded as the path in test case 1 except that this...
path has a security problem since there should be some relation that forces steering control to get data input from the yoke if it is moved in autopilot mode. The pilot should always have the highest control priority for safety reasons.

These test cases cover several others paths such as for example the Component/Connector/Internal_Transfer_Path from yoke_status input interface to rudders output interfaces of steering control. All simulated paths can be found in appendix A.

The results from simulations of test paths showed few problems in the flight control system specification. The majority of problems in the system were mainly because of missing safety relations between the autopilot mode and the manualpilot mode (control flow properties). Since the flight control system consists of many devices where the internal behaviors of these devices were not modeled, there are little execution rule properties, apart from AADL thread execution semantics, that could contradict the tested properties (shown in Table 2). The flight control system specification could be much more refined with behavioral annexes, specified runtime execution by subprograms and specified internal behavior of devices, and thus would be a more appropriate model for verification by the architecture-based testing criteria.

5.3.2 Verification Using Uppaal

One milestone of the architecture-based verification algorithms is to use the Uppaal toolbox to verify the specification for completeness, consistency and having the property of being deadlock free (concurrency). Although architecture-based criteria applied to the specification in previous section verify concurrency properties, modeling and verification of a system in Uppaal without deadlock checking would be inadequate because of the fact that the property is easily verified.

The AADL flight control specification is translated to the Uppaal toolbox by finite state automata where an AADL component abstraction is represented by an automaton. To start with, we will look at global variables (reached by all automata), shown in Figure 24, to get a better understanding of the model. The first five variables (yokeL, yokeR, pedals, throttle, apbutton) represent connections between steering control thread, engine control thread and its input devices (sensors) where arrays are used for appending size to the integer variables. For example, yokeL represents a connection of yoke left and right movement signals and is an integer that can be assigned the value -1, 0 or 1 which represent left, neutral and right movements of the yoke respectively. The following four variables represent connection between the autopilot control thread and the autopilot setting device. These variables are similar to the ones explained but there is one crucial difference. These variables represent a connection of course, height and speed signals and have nothing to do with output signals of other devices. The boolean variable "autopilot" is used for thread synchronization. The last seven statements are declaration of channels where leftaileron, rightaileron, verticalfin, elevator and engine channels represent connections between steering control thread, engine control thread and its output devices (actuators). These channels are arrays of channels to represent different states of the devices. For example, leftaileron channel number 1 (which is indexed zero) represent a signal which should result in that the leftaileron is positioned to decrease the lifting power ("upper state") whereas the other two represent a neutral and a lower state (increasing lifting power) of the left aileron. Note that any value described
by the first four variables corresponds to the right behavior according to the channels if the value is added by one. For example, if yokelr has the value -1, which means that the yoke is turned to the left, left aileron should be moved upward whereas right aileron should be moved downward. If we now add the value -1 with 1 and looks at the corresponding channels (leftaileron[0], rightailer[0] etc.), we get the correct behavior. All other variables that will appear in this section are local variables and can only be reached by a single automaton.

// Place global declarations here.

int[-1,1] yokelr; //position of the yoke left/right 0=neutral, -1=left, 1=right
int[-1,1] yokelb; //position of the yoke forward/backward 0=neutral, -1=forward, 1=backward
int[-1,1] pedals; //position of the pedals 0=both neutral, -1=left pressed, 1=right pressed
int[0,1] throttle; //position of throttle 0=neutral, 1=pushed
int[0,1] spbutton; //position of spbutton 0=not pressed, 1=pressed
int[-1,1] ap_yokelr; //rotate 0=neutral, -1=left, 1=right
int[-1,1] ap_yokelb; //down/up 0=neutral, -1=down, 1=up
int[-1,1] ap_pedals; //left/right 0=both neutral, -1=left, 1=right
int[0,1] ap_throttle; //power 0=no power, 1=poworz

bool autopilot = false;

chan leftaileron[3]; // 0=to upper state, 1=to neutral state, 2=to lower state
chan rightaileron[3]; // 0=to lower state, 1=to neutral state, 2=to upper state
chan verticalail[3]; // 0=to left, 1=to neutral state, 2=to right state
chan elevator[3]; // 0=to lower state, 1=to neutral state, 2=to upper state
chan engine[2]; // 0=power off, 1=power on
chan threadsyncsteering, threadsynccengine;

Figure 24: Global Variables

The yoke is modeled by an automaton with nine locations, shown in Figure 25, which represents all possible positions (locations) and movements (edges) of the yoke. To easily understand the model one can think of a "x and y" coordinate system where the abscissa (x-axis) and ordinate (y-axis) have its origo (x = 0, y = 0) in the center state. The center state represents that the yoke left/right turning movements and forward/backward movement are both in neutral state. Therefore, every edge into to center location have an update on the yokelr and yokebr variables where these are assigned the value 0. With the coordinate system and its origo in mind, following rules can be applied:

- Every location that has an intersection with zero value location on the abscissa represent a neutral turning movement of the yoke
- Every location on the positive side of the abscissa represent a right turning movement of the yoke
- Every location on the negative side of the abscissa represent a left turning movement of the yoke
- Every location that has an intersection with zero value location on the ordinate represent a neutral forward/backward movement of the yoke
- Every location on the positive side of the ordinate represent a forward movement of the yoke
- Every location on the negative side of the ordinate represent a backward movement of the yoke

All edges update the \( yokefb \) and \( yokefr \) variables according to the locations.

![Yoke Automaton](image)

Figure 25: Yoke Automaton

The pedals (shown in Figure 26), throttle (shown in Figure 27) and autopilot button (shown in Figure 28) components are modeled in similar fashion (assigning global variables corresponding to locations and not anything to do with the coordinate system) as the yoke except that the throttle is slightly different since it was specified with a signal latency (10 milliseconds) property in the AADL specification. Consequently, edges to other locations are added with an extra location and edge in order to model the latency. For example, if the automaton is in location "Neutral" and a transition to the "Pushed" location is taken, the automaton needs to visit the "Delay1" location along the path. The edge to the delay location resets a local clock whereas the edge away from the location has a guard which states that the clock must be bigger or equal to the latency. In order to force the automaton to the "Pushed"
location when the clock is equal to latency, an invariant which states that the clock must be smaller or equal to the latency is declared in the location.

Figure 26: Pedals Automaton

Figure 27: Throttle Automaton
The autopilot setting component is modeled by a single location as shown in Figure 29. The idea here is, because of the simplicity and correctness, to model every possible combination of outputs from the device by model an edge for each possible value that can be assigned to each variable.

The steering control thread is modeled by an automaton where the initially location is "Waiting for Dispatch" and contains of two different paths, as shown in Figure 30. The paths represent manual mode (rightmost side) and autopilot mode (leftmost side) as specified in the specification. The initial location is added with an invariant using a local clock to force the automaton into action. If the autopilot button isn’t pressed (button can be pressed at the exact time when the thread dispatches which is synchronized with the other threads through the boolean variable "autopilot" and "apbutton" in order to prevent steering control.
and engine control to dispatch in different modes) when the thread dispatches, inputs from devices (sensors) are collected to produce output to actuators in form of synchronization with channels. Note that locations are declared as committed in order to represent data being obtained simultaneously. On the other hand, if autopilot mode is engaged by the autopilot button and autopilot setting control thread has set the "autopilot" variable to true (as seen below), the other path will be taken. As can be seen, this path uses the same variables for input as the manual mode path, but since threads are synchronized (shown below), autopilot control thread temporarily uses these variables for communication. Engine control thread is modeled in similar fashion as shown in Figure 31.

Figure 30: Steering Control Thread Automaton
Autopilot setting control thread, shown in Figure 32, is modeled as the other threads with two paths where one simply resets the local clock and the other is used for providing data to other threads when the system is in autopilot mode. Whenever the automaton goes into autopilot mode, it waits for synchronization with the other threads. When threads are synchronized, setting control thread collects inputs and produces output by temporarily using variables which are restored with default values before the thread is finished. Some locations of the automaton are provided with invariants for guaranteed progress.

Rudders and engine components are modeled in similar fashion where each device is represented by an automaton such as the left aileron automaton shown in Figure 33. Locations
represent the state of devices where each location can be reached by all other locations by using correct channels. Since the engine (shown in Figure 34), just as the throttle, was specified with a latency property, there is a slight difference since it is modeled with a delay property.

Figure 33: LeftAileron Automaton

Figure 34: Engine Automaton

The model which is a translation from the AADL specification can now be verified for
completeness, consistency and concurrency (deadlock free) as described in the developed algorithms explained in section 3.1. Completeness was described as the specification having response for all possible inputs whereas consistency was described as the specification being free of contradictory behavioral. And in the context of abstract state machines, completeness was defined as

"a rule will be enabled for every possible combination of its monitored variables"

whereas consistency was defined as

"no state having more than one transition enabled at the same time"[21].

In the context of Uppaal, completeness is then defined as a guard will be enabled for every possible combination of its monitored variables which can be stronger interpreted as the right outcome will be produced for every possible input. In the case of the flight control model, monitored variables are those who are assigned by yoke, pedals, throttle, autopilot button and autopilot setting automata. Therefore, to verify the completeness of the model, queries that completely covers all possible inputs with the correct behavioral have to be asked to the Uppaal tool box. As an example of how these queries are created, we will look at inputs produced by the pedals device.

Possible inputs (via the global "pedals" variable) produced by the pedals are:

- pedals = 0 (i.e. no pedals are pressed) which is equal as the pedals automaton being in location "Neutral"
- pedals = -1 (i.e. left pedal is pressed) which is equal as the pedals automaton being in location "LeftPressed"
- pedals = 1 (i.e. right pedal is pressed) which is equal as the pedals automaton being in location "RightPressed"

Correct outcome of these inputs are (outcome ordered in relation to input):

- Vertical fin automaton being in location "NeutralState"
- Vertical fin automaton being in location "LeftState"
- Vertical fin automaton being in location "RightState"

Since completeness properties are on the form "something will eventually happen" or to be more specific "an input leads to correct outcome", it is a liveness property which can be verified by using the temporal operator "->". Therefore, a query as "whenever pedals are in the "Neutral" location then eventually the vertical fin will be in the "NeutralState" location" or by using the Uppaal query language "ThePedals.Neutral --> TheVerticalFin.NeutralState", seems like a proper query in order to verify the first input with the first outcome shown in the pointlists above. But a verification of this property will get a "property not satisfied" result by the tool box. The property is not satisfied because of the fact that input devices aren’t modeled with time constraints except for the throttle which was specified with a latency property in the AADL specification. This means that, according to the model, pedals output can be changed "infinitely" many times before the steering control thread dispatches since time doesn’t elapse during changes of the outputs. Consequently, the query must be extended with the fact that
the steering control thread must dispatch with the pedals being in the "Neutral" location. The proper query is "ThePedals.Neutral and TheSteeringControlThread.GetInputsAndSync -> TheVerticalFin.NeutralState" which result is "property satisfied" when asked to the Uppal toolbox.

Verification of input from autopilot setting device with correct outcome is far more complex since synchronization between threads depends on several clock constraints, variables and channels. But instead of simply verifying that any guard will be enabled by an input for autopilot setting, completeness properties can be checked by dividing queries to easily checked parts which can be synthesized to a conclusion. For example, an input from the autopilot setting can be verified to lead to the correct outcome by checking if: the location where inputs are gained by the autopilot control thread (which involves the autopilot button) and the location where steering control thread and engine control thread gains input from the autopilot control thread where synchronization with actuators takes place, can be reached ("E<> (TheAPSettingControlThread.WaitForSyncWithSteeringAndReadSensorValues and TheEngineControlThread.WaitingForSync and TheSteeringControlThread.WaitingForSync)”). This doesn’t ensure that values of variables have been correctly communicated because of the nature of the query. To ensure correct values, a trace of the path can be analyzed.

During the process of gaining knowledge about the Uppaal toolbox, it came to my mind that consistency involves a model to be deterministic in the way that only one guard can be enabled at the same time whereas in Uppaal this isn’t feasible because of the fact that the automata theory is extended with synchronization (by channels) possibilities where several guards can be enabled at the same time. Therefore, consistency verification of the Uppaal model is discarded since the subject needs a more thorough investigation.

Concurrency properties are simply checked by querying in the form "for all paths globally not deadlock" which in Uppaal syntax is "A[ ] not deadlock".

All architecture-based queries with results can be found in appendix B.
Figure 35: Uppaal Verification

The results of the verification were that the model was complete in the context of a guard being enabled in response of every possible input. The stronger completeness verification, which was in the context of the right outcome of every possible input, showed the model to not be complete because of the latency property in the engine automata. Whenever the engine automaton was synchronized by the engine control thread in order to change the state of the engine, it was possible to reach a deadlocked state. Since the latency was specified to 450 milliseconds and the period of threads was 100 milliseconds, it is possible for the engine control thread to reach a location where the next edge involves synchronization with the engine that may still be in a delay location. The deadlock occur since the particular location of the thread is committed which means that the next transition must involve an edge of the thread
(and time isn’t allowed to progress) which involve synchronization (through channel and not synchronization of threads) with the engine whereas the engine can’t support this because it’s stuck in the delay location. This also answers results of the concurrency property (deadlock free) verification.

The architecture-based testing criteria applied to the AADL flight control system specification (described in section 5.3.1) could be applied to the translated Uppaal model instead of the AADL model in order to do make the verification approach more formal and partly automated. This approach was discarded because of schedule constraints but will be a topic for future research.
6 Implementing the AADL Flight Control specification in Java

Architecture-based verification involves testing of the implementation that is implemented to satisfy the specification. In this chapter we will look at a Java implementation of the flight control specification. In order to implement the AADL specification in Java, which contains specification of both hardware and software, the abstraction level of the implementation is analogous to the specification. The underlying idea is to translate the AADL specification to executable Java code and thereby making it possible to execute test cases derived from the architecture.

6.1 Java Implementation

To simplify understanding of the Java program, AADL component abstractions (e.g. device), component type subclause elements (e.g. data port) and component implementation subclause elements (e.g. connections) corresponds to a java class. Component abstraction classes have instance variables referenced to tables that correspond to each component type subclause and component implementation subclause. To illustrate an example, we can create a device component object and a data port object which we can added to the device component features data port table and thus representing a device component with one interface data port.

To start with, we will look at the java class that represents an AADL interface port. The Port class contains of "get methods" and instance variables representing AADL port attributes which are set by the constructor. The instance variables are able to contain information about a port’s identifier, direction (e.g. out port), connection descriptor (data or event port) and data type where the data type is a reference variable while others are simple variables (e.g. String variable). Since AADL ports are typed by data components, data type variable is a reference variable which can reference to a data component object. Methods of the port class are "get methods" from where information about a port’s attributes can be obtained.

AADL data access and bus access component features are implemented in similar way as ports. Data access and bus access classes have instance variables able to contain information about identifier, direction (provides or requires) and bus or data object to access.

Connections between ports, data accesses and port accesses are implemented as three different classes. The port connection class has three instance variables representing port connection attributes and thus they are able to contain information about a port connection’s identifier, source port and destination port. The port connection constructor executes a checking algorithm before instance variables are assigned since a connection between ports is constrained to be between ports with the same connection descriptor and data type.

Data access connection and bus access connection classes are implemented analogous to port connections except for the checking algorithms which checks if the source data/bus access object is a "provides access" and that the destination data/bus access is a "require access".

In order to assure semantics of AADL in the implementation, further checking of connections has to be carried out which will be described below.
As described above, each component abstraction corresponds to a Java class where these classes are implemented in similar fashion but differs according to the AADL standard. We will look at the system component class which context is applicable to each other component abstraction class.

The system class has instance variables that correspond to AADL component type and component implementation subclauses except for one containing the system identifier. Each instance variable (a few of them illustrated in Figure 36) is referenced to a hashmap where keys are subclause element identifiers (a String) and values are subclause element objects (e.g. ports).

![Figure 36: System Class Instance Variables](image)

Initially, a system object has an identifier and no other elements when it’s created. Other elements (e.g. subcomponents) are added to the system component by using its "add" methods. For example, we will look at the addPort() and addPortConnection () methods. The addPort() method, shown in Figure 37, acquires a port object as parameter which is checked before added to the "port_tab" hashmap. The method simple checks if there already exists a port in the system component feature with same identifier as the parameter port object. If there is no such port, the parameter port is added to the hashmap.

```java
public void addPort(Port po) {
    if (port_tab.containsKey(po.getName())) { // the port already exist
        JOptionPane.showMessageDialog(null, "ERROR: Port " + po.getName() + " already exist as a port in system component "+name);
    } else { // add it to the port feature hashmap
        port_tab.put(po.getName(), po);
    }
}
```

![Figure 37: addPort() method](image)

According to the addPort() method, it’s possible to add the same port object to the port
hashmap that may already been added to another component since it’s not checked. Instead of preventing this kind of behavior, the program allows use of the same port object in different components but the program doesn’t allow connection among port objects that are used more than once, as will be described below. The addPortConnection() method, partially shown in Figure 38, acquires a port connection object which is checked before added to the port connection hashmap. The checking algorithm checks whether each of the two ports in the connection exists only once in the system or its subcomponents, if directions of ports are accurate and if there doesn’t already exist a port connection with the same identifier. These constraints can be checked by going through the system’s and its subcomponents port hashmap, as partially shown in Figure 38. While going through the hashmaps (i.e. going through the entries of the hashmaps), each hashmap value is compared to the source and destination port of the connection object. If one of them is referenced to the same object as the hashmap value, an integer variable, "found1" or "found2", is increased by one and another integer variable, "component_level1" or "component_level2", is set to indicate whether the found port is a feature of the system component or its subcomponents (value 0 indicates component level while value 1 indicates subcomponent level). When the algorithm terminates, variables "found1" and "found2" should have the value one. If ports are found in different component levels, they must have the same direction whereas connection between ports in the same component level must be from an out port to an in port.
Figure 38: `addPortConnection()` method

Java classes described above can now be used to develop the specific flight control system. Creation of the system simply involves creation of objects according to the specification, which is partially shown in Figure 39.
The implementation described above can easily be verified by hand not to conform the specification since all AADL elements used in the specification aren’t implemented. For example, properties can’t be associated with different AADL elements. This can partially be explained with the decision of implementing another refined implementation, explained in section 7, instead of refining the first one. We will now look at automated testing of the program in order to assure correct behavior and conformance of the implementation with respect to the specification. A part of the test cases involves unit testing by the JUnit framework where the other part involves architecture-based testing built upon the JUnit framework. We will now look at the former part whereas the latter is described in section 6.3.

6.2 Unit Testing with JUnit

A JUnit test suite has been developed to assure correct behavior of the implementation. The test suite contains of test cases where a test case is a related set of tests of a particular Java class. Since most classes are built on the same context, i.e. they have the same behavior, I didn’t create a test case for all sixteen classes. As an example of how to assure some of the behaviors described above, we will look at the test case for the system component class. "TestSystem" class (shown in Figure 40), which is a Junit test case, contains tests to assure the behavior of the system class. The test case has one fixture object, an instance of the class under test, which is required by the tests to run. The fixture object is set up by the setUp() method shown in Figure 40. The resource is released by the teardown() method, which isn’t required in this case since it will be automatically garbage-collected anyway.
The test case consists partially of two tests, "testAddPort()" and "testAddPortConnection()", which test the behavioral of the addPort() and addPortConnection() methods in the system class. The testAddPort(), shown in Figure 41, tests whether it’s possible to add a port to the system and that it isn’t possible to add two ports with the same identifier. For example, to test if it’s possible to add a port we start with creating a port which then is added to the system fixture object. The assertSame() method is then used to assert that the created port exists in the system.

```java
import junit.framework.*;

public class TestSystem extends TestCase {

    private System testSystem;

    public void setUp(){
        testSystem=new System("testsystem");
    }

    public void tearDown(){
        testSystem=null;
    }

    // additional code
}
```

Figure 40: setUp() and tearDown()

The testAddPortConnection(), partly shown in Figure 42, tests whether, 1) a connection between two ports can be added even though the ports are not added in the system or its subcomponents, 2) a connection can be added when ports are added to the component, 3) a connection can be added even though one of the ports in the connection is added as feature in two components, 4) a connection can be added between two ports in the same component level even though the source port isn’t an out port and destination port isn’t an in port, 5) a connection between two ports in different component levels even though they don’t have the same direction.

```java
public void testAddPortConnection() {
    // check if port was added
    Port port1 = new Port("testport",0,0,new Data("testdata"));
    testSystem.addPort(port1); // add a port
    testSystem.addPortToSystem
    assertSame("error: add port to system failed",port, testSystem.getPort("testport");
    // check that different port with same name as a added one isn’t added
    Port port2 = new Port("testport",1,0, new Data("testdata"));
    testSystem.addPort(port2); // add another port with same name as the last one
    assertNotSame("error: another port with same name added in system", port2, testSystem.getPort("testport");

    // check that the last port wasn’t added
}
```

Figure 41: TestAddPort()
These automated tests revealed some faults in the implementation that had been overlooked. The tests were passed, shown in Figure 43, without any errors or failures after some small changes in the implementation.
6.3 Architecture-based Testing of the Implementation upon Junit

As Jin Z. and Offut J. describes in [24], derived testing paths from the specification (described in section 5.3.1) should be fully exercised in the implementation by a set of inputs. This is not feasible in this particular case because the implementation is a straight translation from AADL to Java, i.e. the implementation have the same abstraction level as the specification. Instead, relations and properties expressed in the specification can be tested for existence in terms of connections. This is done by using the Junit framework where a test case (partly shown in Figure 44) is written for the HWSW class which describes the flight control system. The test case simply consists of one fixture object, which is an instance of the HWSW class, and one test method. The test method consists of an algorithm which starts with asserting existence of connections (by using identifiers since the same namespace is used in the implementation as in the specification) in the top component and later through its subcomponents until the bottom components are reached. The testHWSW() method starts with asserting existence of the busaccess connection "C1" in the top system component HWSW. This connection should exist in the implementation since there is such a connection in the top system component of the AADL flight control system specification, as shown in Figure 22. Since there are no more connections in the top component, the algorithm proceeds by deriving the system subcomponents ("ac_application" and "ac_computer") of the top system component where connections in these components will be asserted for existence. The algorithm proceeds in this manner until all the components’ connections have been asserted for existence according to the AADL specification.

```java
public class TestHWSW extends TestCase {//test the Process class
    private HWSW testhsw;

    public void setUp(){
        testhsw = new HWSW();
    }

    public void tearDown(){
        testhsw = null;
    }

    public void testHWSW(){
        System hwsu = testhsw.getSystem();
        //check busw connections
        assertNotNull("error: busaccessconnection C1 doesn't exist in hwsu system", hwsu.getBusAccessConnection("C1");
    }
}
```

Figure 44: Architecture-based Test Case

The test case revealed some faults in the HWSW class where a few architecture elements were incorrectly created or mistakenly not created according to the specification and therefore
the implementation didn’t conform the specification. For example, the connection from the engine setting out data port in engine control thread to the engine setting out data port in the airplane control process was mistakenly not created in the system, which results in that an architectural data flow reachability property failed.
7 The Refined Flight Control System

Since the Java implementation of the AADL flight control specification was implemented at a high abstraction level, creating static objects according to the specification, there were no functionalities or meaningful execution in the program that could react on any kind of input for the purpose of testing. What I mean with no functionalities or meaningful execution is that the implementation didn’t capture the meaning of how the system should behave according to the specification. For example, it’s not possible to produce any input to the system that will start execution and communication of components that eventually will control the airplane according to the input. To be able to perform more sophisticated testing, the implementation had to be refined to a lower abstraction level by creating functionalities and meaningful execution of objects that responds and relies on interactions between objects.

In addition, a second thought had to be considered, about how much refinement needed to be carried out to produce an enough useful application. Because of the complexity of a real flight control system a refinement to a low abstraction level would acquire too much time and would be out of scope of this thesis. So the aim of the refinement is to lower the abstraction level just enough to provide useful functionalities and in the same time be simple to implement. Instead of refining the implementation by extensions and development I chose to develop a new application independently of the old application since the program was entirely built on semantics of the AADL standard which would be complex to refine to a lower abstraction level. Because of the fact that I have implemented an AADL specification (which describes both hardware and software) in Java, there would be inadequate to refine the execution platform abstraction of the implementation. Therefore I discarded the execution platform (processor, bus and memory) in the refinement and defend myself by saying with a smile "the refined application is actually executed on my computer, which has a Pentium processor, bus and RAM memory, as in the specification". So by looking at the device-application part of the specification which contains of devices, process and threads I developed a new refined application explained in section 7.1. Under the development of the refined application new architectural elements, properties and configurations were extracted from the application which weren’t specified in the AADL specification. Consequently, the specification was also refined (explained in chapter 8.2) to conform the refined implementation.

7.1 Refined Implementation

The device-application system of the AADL specification, which contains of a process with three concurrently executed threads and several devices (abstractions of sensors, actuators and hardware), can be reflected in the refined application as Java threads [3] and simulations of devices. There are three Java threads in the refined application namely: SteeringControl, EngineControl, APSettingControl and thirteen simulated devices namely: Yoke, Pedals, Throttle, APButton, APSetting, CourseSensor, HeightSensor, SpeedSensor, LeftAileron, RightAileron, VerticalFin, Elevator and Engine.

The concurrent threads are synchronous communicated through a BlockingQueue object [3] which is a Java standard class for communication between Java threads. When the auto pilot button device is pressed (system enters auto pilot mode), SteeringControl and EngineControl use this communication queue to get inputs produced by the APSettingControl thread instead of different devices as when auto pilot button isn’t pressed (system is in manual mode).
SteeringControl is a thread that dispatches periodically, and in manual mode, interacts with simulated devices such as Yoke and Pedals to get inputs that will be analyzed to produce outputs to devices such as LeftAileron, RightAileron, VerticalFin and Elevator. EngineControl acts in the same manner as SteeringControl except that, in manual mode, gets input from the simulated Throttle and produces output to the simulated Engine. APSettingControl thread dispatches aperiodically which means that it dispatches by an event-trigger. Whenever an auto pilot setting (instance variables in APSetting) is changed, an event is produced to trigger the APSettingControl thread and consequently updates the BlockingQueue with new steering and engine settings.

Devices in the AADL specification are implemented in the refined application as dynamic graphical objects in a JFrame window, which also is the application’s G graphical User Interface (GUI), represented in Figure 45. Devices that produce inputs to the threads are: Yoke, Pedals, Throttle, APButton, APSetting, CourseSensor, HeightSensor and SpeedSensor. Yoke controls LeftAileron, RightAileron and Elevator and is simulated by two JSlider objects that represent forward and backward as well as left and right turning movements of the yoke. Each JSlider can be moved, by the application user, to different integer values in the range of -10 to +10, which represents the voltage V of the output signals of the Yoke. The extreme values -10 and +10 represent maximum movements in opposite directions. Pedals and Throttle simulations, which controls VerticalFin and Engine respectively, work in the same logic. Pedals contains of two JS intersectors (one for each pedal) where each can have signal values between 0 and +10. Throttle contains of one JSlider that can have signal values between 0 and +10. CourseSensor, HeightSensor and SpeedSensor are "simulated" by JLabels which shows the current course (degrees), height (meters) and speed (km/h) of the airplane. These sensors were specified in the specification to produce input to the APSettingControl thread for calculation of how to regulate steering elements according to auto pilot settings (wanted course, height and speed). Doable simulations of these sensors would be extremely complicated since they depend on several complex factors. For this reason, I chose to simplify the auto pilot mode by not implement the phase from when the autopilot button is pressed to the point where the airplane is regulated according to the auto pilot settings. So when the auto pilot button is pressed you magically appear in the right course, height and speed (sensors are set according to the auto pilot settings). In manual mode are the sensors affected by simple mathematical algorithms that depend on settings of steering elements. APbutton and APSetting devices are simulated by JToggleButton and JSpinners [3] respectively. JToggleButton is simple a button that can be pressed to go into auto pilot mode (from manual mode) or pressed to go into manual mode (from auto pilot mode). JSpinners allows the user to set course, height and speed to a particular value setting.

Devices (actuators) that are direct affected by controlling threads and indirect affected by devices (sensors) explained above are: LeftAileron, RightAileron, VerticalFin, Elevator and Engine. All these devices are simulated by JSliders which represent the "position" of each steering element except for the Engine which represent amount of power produced by the engine. For example, if the yoke is moved forward to the value 10 (tip the nose of the airplane) the Elevator’s position will decrease to the value -10 (fully tilted down).
As explained in section 3.2, modifications and refinements of an implementation should result in an update of the architecture-specification to preserve the valuable use of an architecture specification. And since the refined implementation extracted new properties, which were not specified in the specification, the AADL flight control specification was consequently refined which is explained in following section.

### 7.2 Refining the AADL Specification

During refinements of the implementation new architecture properties emerged that were not specified in the AADL specification and consequently the architecture specification should be updated with the new properties in order to preserve regression testing. Devices (yoke, pedals, throttle lever, rudders and engine) were implemented with signal properties that simulated a signal voltage with the range from -10V to +10V or 0V to +10V. Sensor signals for auto pilot control were implemented with km/h, degrees and meter units for speed, course and height respectively. These properties can be specified by specifying signal properties and add them to the current data ports. Specified signal properties and an example of updated data ports are shown in Figure 46.

Figure 45: GUI of the Refined Flight Control System
Steering control and engine control threads were implemented with periodical dispatch properties with a period of 100 ms whereas autopilot control thread was implemented with an aperiodical dispatch (triggered by an event) property. Since these properties weren’t specified in the specification, they too was updated as shown by steering control thread in Figure 47.

### 7.3 Architecture-based Testing of the Refined Implementation

Paths derived from the specification (explained in section 5.3.1) can now be exercised by a set of inputs to the flight control program. Path executions are analyzed by a code coverage tool Eclemma (www.eclemma.org), which is used to determine results of test case inputs. As an example of applying paths to the implementation, we will look at test case 1 of the All_Connected_Component_Path that was simulated in section 5.3.1.
**Test case 1**: If autopilot button isn’t pressed (in manual mode), steering control thread gets data input (voltage signal) from yoke and produces output to left/right aileron and elevator. A positive/negative voltage input from yoke "left-right turn" to steering control results in equal voltage output to ailerons where left aileron gets a positive value whereas right aileron gets a negative value. A positive/negative voltage input from yoke "forward-backward" to steering control results in equal negative/positive output voltage value to the elevator.

The test case can be applied to the implementation by following input signals:

- Autopilot button set to manual mode (not pressed)
- Yoke "left-right turn" set randomly between -10 and +10
- Yoke "forward-backward" set randomly between -10 and +10

Note that the path should be more extensively tested than setting yoke output randomly. For example, the test case can be extended with boundary value analysis but because of the simplicity of the program I discarded such techniques as being confident that if one combination of input signals works, they all work.

The expected result of the inputs is that the path described in test case 1 should be fully exercised (which can be argued impossible since input signals are randomly set), i.e. all found relations in the path, according to the specification, should be executed in the implementation.

The result of the test case, based on inputs and outputs of the program as well as code coverage analysis (partly shown in Figure 48), was that all relations were found in the implementation.
Figure 48: Code Coverage of Test Case, statements covered by green color were executed

Testing of all paths can be found in appendix A. Almost all paths were fully exercised in the flight control program except for a couple of test cases where failures of these test cases were known beforehand but anyway result in that the refined implementation doesn’t conform the specification. For example, autopilot setting control thread didn’t get input (data flow reachability property) from course, height and speed sensor as expected in a test path. Since regulation of the plane from the point where auto pilot button is engaged to the point where the plane is regulated according to the auto pilot settings was not implemented in the application, the relations doesn’t exist in the implementation.
8 Summary and Conclusions

Development processes of software-intensive systems acts under severe budget, schedule and quality constraints where architecture-based approaches promise to improve the process by making it more predictable with early error detection and design decision. The purpose of this thesis was to extend the knowledge base within the area of architecture-based verification of software intensive systems by literature and case studies. The underlying problem was to develop verification algorithms, by studying literature and tools such as AADL, Uppaal and Junit, in order to be applicable on a case study enclosing a development process originating from a system specification and later progressed to an implementation. In this work, an AADL flight control system specification was built to serve as base for the development process. Before the process progress to an implementation, developed verification algorithms are applied to the specification. The architecture-based verification algorithms were derived from data-flow reachability, control-flow reachability, connectivity, concurrency, completeness and conformance properties. Furthermore, the algorithms considered issues concerning architecture-based regression testing. Data-flow reachability, control-flow reachability, connectivity and concurrency properties were assessed by derived architecture-level criteria which could be applied to both the specification-level and the implementation-level. In order to apply verification for concurrency (deadlock free) and completeness properties, algorithms were supported by using the Uppaal tool box where the specification was modeled by finite state automata and verified by temporal logic. The process was progressed by a Java implementation, as a translation from AADL semantics to a Java program, of the flight control system specification where a Junit test suite was developed to verify correct behavior and to apply architecture-level criteria. Since the implementation was implemented at a high abstraction level, creating static objects with no meaningful functionalities that could respond to inputs, a refinement to a lower abstraction level was required to perform more sophisticated testing. Consequently (in the purpose of developing knowledge and not to produce an error free implementation), a refined implementation was developed that could be verified by test case inputs and the code coverage tool Eclemma. Developed architecture-level testing criteria were shown to effectively finding problems and faults in both the specification and its implementation since when applied, they command all architectural relations and properties to be fully simulated and tested. The testing criteria were also indirectly used as a base for conformance testing, where the results of all testing paths showed that both implementations didn’t conform with the specification. These criteria should have been applied to the translated specification (Up-paal model), instead of directly to the original, in order to make the verification algorithms more formal and partly automated. Algorithms supported by the Uppaal toolbox, which initially were intended to verify the specification for completeness, concurrency and consistency, were feasible by a translation from AADL to finite-state automata. Because of the fact that the Uppaal modeling language extends finite-state automata theory with synchronization between automata, knowledge about specification concurrency properties wasn’t sufficient to be applicable on the translated model (which uses synchronization properties) and consequently was discarded whereas the topic will be more carefully analyzed in a future research. The translated specification was verified to be complete whereas concurrency verification showed the specification to not be free from deadlock because of specified timing constraint properties. However, one can question the conclusions that the AADL specification was complete but not free from deadlock because a timed automata model of the specification was complete but not free from deadlock. The conclusions certainly rely on how the specification is modeled since
an informal translation is easily done with extended or lost properties. Since the translation in this work was made by an informal approach, there are no underlying proofs that the AADL specification actually is complete and not free from deadlock. As far as known, there are no formal methods to do this translation which is an interesting topic for future research. As to summarize this section, it can be stated that derived verification algorithms resulted in that important test cases resulted in a successful test run, in the context of a problem or an error was found, which is a positive outcome since testing is about exposing the presence of faults and not the absence.
9 Future Work

The purpose of this thesis was to extend the knowledge base within the area of architecture-based verification in order to perform future research within the area. Consequently, one implicit task of the thesis was to find interesting topics for future work. The area of architecture-based verification is quite large with several interesting topics for future research but the topics that are closest related to the thesis work are as follow:

- Refine the AADL flight control system specification with behavioral annexes, specified runtime execution by subprogram components and specified internal behavior of devices
- How to perform consistency verification of Uppaal models
- Develop a tool for formal and automated translation from AADL to Uppaal
- How to perform automated simulations (in Uppaal) of test paths when architecture-based testing criteria are applied to Uppaal models
- Develop algorithms for architecture-based regression testing
References


A Architecture-based Testing Paths

Figure 49: AADL Graphical Representation

All paths:

Component (Connector) internal transfer paths:

1. Connections 1 to 17
2. steering control (yoke status 'left/right aileron and elevator setting
3. steering control(left/right pedals status 'vertical fin setting)
4. steering control (ap_setting data type 'left/right aileron, vertical fin and elevator setting)
5. engine control (throttle status 'engine setting)
6. engine control (ap setting data type ‘engine setting)
7. ap setting data type (input from ap setting control ‘output to steering and engine control)
8. ap setting control (ap setting, course, height and speed ‘output to ap setting data type)

Component (Connector) internal sequencing paths:

9. All interfaces has sequencing rules since they all have data or control transfer

Direct component to component paths:

10. components associated with connections 1 to 17

Indirect component to component path:

11. 1-10
12. 1-11
13. 1-13
14. 2-12
15. 1-4
16. 2-4
17. 16-4
18. 3-14
19. 3-5
20. 17-5
21. 15-16
22. 15-17
23. 17-14
24. 16-10
25. 16-11
26. 16-12
27. 16-13
28. 6-15
29. 7-15
30. 8-15
31. 9-15

All connected components path (These can be divided to several sub paths!):

32. 1-2-4-10-11-12-13
33. 6-7-8-9-4-15-16-10-11-12-12
34. 6-7-8-9-4-15-17-14
35. 3-5-14

Path All connected components paths: 1-4-10-11-13

Test case 1: If autopilot button isn’t pressed (In manual mode), steering control gets data input from yoke and produces output to left/right aileron and elevator.
Test case 2: If autopilot button is pressed (in autopilot mode), steering control doesn’t get data input from yoke. This path has a security problem since there should be some relation that forces steering control to get data input from the yoke if it is moved in autopilot mode. The pilot should always have the highest control priority for safety reasons. Test cases cover paths: 1 (connections 1, 4, 10, 11, 13), 2, 10 (components associated with connections 1, 4, 10, 11, 13), 11, 12, 13, 15

Implementation testing

Test case 1: Input signals: autopilot button not pressed, yoke "left-right" set randomly between -10 and +10, yoke "forward-backward" set randomly between -10 and +10.
Expected result: Path described in test case 1 should be fully exercised in the implementation i.e. all found relations in the path, according to the specification, should be executed in the implementation.
Actual result: All identified relations in the path were fully exercised.

Test case 2: Input signals: autopilot button pressed, yoke "left-right" set randomly between -10 and +10, yoke "forward-backward" set somewhere between -10 and +10.
Expected result: Path described in test case 2 should be fully exercised in the implementation.
Actual result: All identified relations were fully exercised.

Path Indirect component to component paths: 2-4-12

Test case 1: If autopilot button isn’t pressed (in manual mode), steering control gets input from pedals (left and right pedal) and produces output to vertical fin.
Test case 2: If autopilot button is pressed (in autopilot mode), steering control doesn’t get input from pedals. This path has a security problem since there should be some relation that forces steering control to get data input from pedals if they are moved in autopilot mode.
Test cases cover paths: 1 (connections 2, 4, 12), 3, 10 (components associated with connections 2, 4, 12), 14, 16

Implementation testing

Test case 1: Input signals: autopilot button not pressed, pedals set randomly between 0 and +10.
Expected result: Path described in test case 1 should be fully exercised in the implementation.
Actual result: All identified relations in the path were not fully exercised. Since the pedals have a type of contradictory behavior, i.e. both pedals cannot be pressed at the same time, steering control first get data input from the left pedal and if the data indicates that it is pressed steering control doesn’t need to get data input from right pedal. So to fully exercise the path I needed to either set both pedals to 0 or set the right pedal between 1 and +10.

Test case 2: Input signals: autopilot button pressed, pedals set randomly between 0 and +10.
Expected result: Path described in test case 2 should be fully exercised in the implementation.
Actual result: All identified relations were fully exercised.
**Path** Indirect component to component paths: 3-5-14

Test case 1: If autopilot button isn’t pressed (in manual mode), engine gets input from throttle and produces output to engine.

Test case 2: If autopilot button is pressed (in autopilot mode), engine control doesn’t get input from throttle. This path has a security problem since there should be some relation that forces engine control to get data input from the throttle if it is moved in autopilot mode.

Test cases cover paths: 1(connections 3, 5, 14), 5, 10(components associated with connections 3, 5, 14), 18, 19

**Implementation testing**

Test case 1: Input signals: autopilot button not pressed, throttle set randomly between 0 and +10.

Expected result: Path described in test case 1 should be fully exercised in the implementation.

Actual result: All identified relations in the path were fully exercised.

Test case 2: Input signals: autopilot button pressed, throttle set randomly between 0 and +10.

Expected result: Path described in test case 2 should be fully exercised in the implementation.

Actual result: All identified relations were fully exercised.

**Path** All connected components paths: 6-7-8-9-15

Test case 1: autopilot setting control dispatches by an event (autopilot button) and gets input from autopilot setting device, course sensor, height sensor and speed sensor and produces yoke, pedals and throttle output to ap_setting data type which is an abstract data type in source text.

Test case 2: autopilot setting control don’t dispatch when autopilot button isn’t pressed.

Test cases cover paths: 1(connections 6, 7, 8, 9, 15), 8, 10(components associated with connections 6, 7, 8, 9, 15), 28, 29, 30, 31

**Implementation testing**

Test case 1: Input signals: auto pilot settings are randomly set, auto pilot is pressed to dispatch the autopilot setting control.

Expected result: Path described in test case 1 should be fully exercised in the implementation.

Actual result: All identified relations in the path were not fully exercised. Autopilot setting control didn’t get input from course, height and speed sensor. Since regulation of the plane from the point when auto pilot button is set to the point where the plane is regulated according to the auto pilot settings was not implemented in the application, the relations doesn’t exist in the implementation.

Test case 2: Input signals: auto pilot settings are randomly set, auto pilot is not pressed.

Expected result: Path described in test case 2 should be fully exercised in the implementation.

Actual result: All identified relations in the path were fully exercised.
**Path** All connected components paths: 4-16-10-11-12-13

Test case 1: If autopilot button is pressed (In autopilot mode), steering control gets input from ap_setting data type and produces output to left/right aileron, vertical fin and elevator.
Test case 2: If autopilot button isn’t pressed (In manual mode), steering control doesn’t get input from ap_setting data type.
Test cases cover paths: 1(connections 4, 16, 10, 11, 12, 13), 4, 10(components associated with connections 4, 16, 10, 11, 12, 13), 21, 24, 25, 26, 27

**Implementation testing**

Test case 1: Input signals: course and height auto pilot settings are randomly set, auto pilot button is pressed.
Expected result: Path described in test case 1 should be fully exercised in the implementation.
Actual result: All identified relations in the path were fully exercised.
Test case 2: Input signals: course and height auto pilot settings are randomly set, auto pilot is not pressed.
Expected result: Path described in test case 2 should be fully exercised in the implementation.
Actual result: All identified relations in the path were fully exercised.

**Path** All connected components paths: 5-17-14

Test case 1: If autopilot button is pressed (In autopilot mode), engine control gets input from ap_setting data type and produces output to engine.
Test case 2: If autopilot button isn’t pressed (In manual mode), engine control doesn’t get input from ap_setting data type.
Test cases cover paths: 1(connections 5, 17, 14), 6, 10(components associated with connections 5, 17, 14), 20, 23, 22

**Implementation testing**

Test case 1: Input signals: speed auto pilot settings are randomly set, auto pilot button is pressed.
Expected result: Path described in test case 1 should be fully exercised in the implementation.
Actual result: All identified relations in the path were fully exercised.
Test case 2: Input signals: speed auto pilot settings are randomly set, auto pilot is not pressed.
Expected result: Path described in test case 2 should be fully exercised in the implementation.
Actual result: All identified relations in the path were fully exercised.

Total test cases covers paths: 7, 9, 17, 32, 33, 34, 35, 36
B Architecture-based Uppaal Queries

Architecture-based queries

Property:
TheYoke.NeutralNeutral and TheSteeringControlThread.GetInputsAndSync →
TheLeftAileron.NeutralState and TheRightAileron.NeutralState and
TheElevator.NeutralState
Satisfied

Property:
TheYoke.NeutralLeft and TheSteeringControlThread.GetInputsAndSync →
TheLeftAileron.UpperState and TheRightAileron.LowerState and
TheElevator.NeutralState
Satisfied

Property:
TheYoke.ForwardLeft and TheSteeringControlThread.GetInputsAndSync →
TheLeftAileron.UpperState and TheRightAileron.LowerState and
TheElevator.LowerState
Satisfied

Property:
TheYoke.ForwardNeutral and TheSteeringControlThread.GetInputsAndSync →
TheLeftAileron.NeutralState and TheRightAileron.NeutralState and
TheElevator.LowerState
Satisfied

Property:
TheYoke.ForwardRight and TheSteeringControlThread.GetInputsAndSync →
TheLeftAileron.LowerState and TheRightAileron.UpperState and
TheElevator.LowerState
Satisfied

Property:
TheYoke.NeutralRight and TheSteeringControlThread.GetInputsAndSync →
TheLeftAileron.LowerState and TheRightAileron.UpperState and
TheElevator.NeutralState
Satisfied
Property:
TheYoke.BackwardRight and TheSteeringControlThread.GetInputsAndSync ->
TheLeftAileron.LowerState and TheRightAileron.UpperState and
TheElevator.UpperState
Satisfied

Property:
TheYoke.BackwardNeutral and TheSteeringControlThread.GetInputsAndSync ->
TheLeftAileron.NeutralState and TheRightAileron.NeutralState and
TheElevator.UpperState
Satisfied

Property:
TheYoke.BackwardLeft and TheSteeringControlThread.GetInputsAndSync ->
TheLeftAileron.UpperState and TheRightAileron.LowerState and
TheElevator.UpperState
Satisfied

Property:
ThePedals.Neutral and TheSteeringControlThread.GetInputsAndSync ->
TheVerticalFin.NeutralState
Satisfied

Property:
ThePedals.LeftPressed and TheSteeringControlThread.GetInputsAndSync ->
TheVerticalFin.LeftState
Satisfied

Property:
ThePedals.RightPressed and TheSteeringControlThread.GetInputsAndSync ->
TheVerticalFin.RightState
Satisfied

Property:
TheThrottle.Neutral and TheEngineControlThread.GetInputsAndSync ->
TheEngine.NoPower
Satisfied

Property:
TheThrottle.Neutral and TheEngineControlThread.GetInputsAndSync ->
TheEngine.NoPower or TheEngine.Delay1
Satisfied

Property:
TheThrottle.Pushed and TheEngineControlThread.GetInputsAndSync →
TheEngine.Power
Satisfied

Property:
TheThrottle.Pushed and TheEngineControlThread.GetInputsAndSync →
TheEngine.Power or TheEngine.Delay2
Not Satisfied

Property:
E<> TheSteeringControlThread.WaitingForSync
Satisfied

Property:
E<> TheEngineControlThread.WaitingForSync
Satisfied

Property:
E<> (TheAPSettingControlThread.WaitForSyncWithSteeringAndReadSensorValues and
TheEngineControlThread.WaitingForSync and TheSteeringControlThread.WaitingForSync)
Satisfied

Property:
E<> TheAPSettingControlThread.WaitForSyncWithEngine
Satisfied

Property:
A[] not deadlock
Not Satisfied