# Automated Numerical Testing of Models for a Helicopter Simulation Framework

## Bachelor Thesis

Lukas Schmierer

6th September 2017

| | |
|---|---|
| Bearbeitungszeitraum | 12 Wochen |
| Matrikelnummer, Kurs | 3529124, TINF14ITIN |
| Ausbildungsfirma | Deutsches Zentrum für Luft- und Raumfahrt, Köln |
| Betreuer der Ausbildungsfirma | M. Sc. Melven Röhrig-Zöllner |
| Gutachter der Dualen Hochschule | Prof. Dr. Harald Kornmayer |

# Eidesstattliche Erklärung

gemäß § 5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. September 2015.

Ich habe die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Köln, den 6. September 2017

## Abstract

The Versatile Aeromechanic Simulation Tool (VAST) performs aeromechanic simulations by interconnecting single isolated state-space models (first order ordinary differential equations) to form complex coupled systems. These systems are solved by applying numerical methods. To ensure the stability of these methods and achieve reasonable results, all models must meet certain conditions such as determinism. However, in coupled systems of multiple models, it is difficult to find the root cause of errors. The aim of this work is to design and implement an extensible test concept and validate its feasibility to solve this problem. The tests are applied to individual models at arbitrary times during a simulation, detached from the coupled system. Two different tests are implemented to evaluate this concept. The execution of the tests on VAST models showed faulty implementations of two models. This shows that the new test concept can help to detect typical errors in model implementations. Further tests can be developed on this foundation. A simple and clearly defined interface makes the implementation of new tests straightforward.

# Contents

# List of Abbreviations

| | |
|---|---|
| **DLR** | Deutsches Zentrum für Luft- und Raumfahrt e.V. (German Aerospace Center) |
| **VAST** | Versatile Aeromechanic Simulation Tool |
| **ODE** | Ordinary differential equation |
| **XML** | Extensible Markup Language |
| **XSD** | XML Schema Definition |
| **RK4** | Runge-Kutta 4 |
| **PID** | Proportional–integral–derivative |

# List of Figures

# Preface

This Bachelor thesis has been written in 2017 at the *German Aerospace Center* (DLR[1]) in Cologne at the facility of *Simulation and Software Technology*[2]. This work concludes my studies in *information technology* at the *Baden-Wuerttemberg Cooperative State University* (DHBW[3]) Mannheim.

I would like to thank Melven Röhrig-Zöllner, who supervised my work at DLR, and Prof. Harald Kornmayer as supervising professor of the DHBW.

DLR is the national aeronautics and space research center of Germany and it is commissioned by the Federal Government for administering the German space program. DLR engages about 8000 employees at 20 locations in Germany mainly for research in the areas of aeronautics, safety, energy and transport. Additional offices are maintained in Brusseles, Paris, Tokyo and Washington D.C. DLR is member of the *European Space Agency* (ESA) where it represents German interests and manages the German aeronautics dedicated budget. [1]

Founded in 2009, DHBW is the first German higher educational institution that integrates academic studies with workspace training. Spread over nine locations, it offers a broad range of study programs including primarily business and engineering. [2]

[1]German: Deutsches Zentrum für Luft- und Raumfahrt e.V.
[2]department of *High Performance Computing.*, led by Dr. Achim Basermann
[3]German: Duale Hochschule Baden-Württemberg

# 1. Introduction

The *Versatile Aeromechanic Simulation Tool (VAST)* is a DLR software project launched in 2016 by the facility of *Simulation and Software Technology* (department of *High Performance Computing* in Cologne) and the institute of *Flight Systems* in Brunswick. VAST is meant to support the DLR research in the areas of aeromachanics as a numerical simulation tool.

However, numerical calculations are always affected by numerical errors. In many cases these problems are hard to find.

## 1.1. Motivation

The *Versatile Aeromechanic Simulation Tool* is intended to support DLR research by enabling physical simulations. While the focus is on aeromechanic simulations, it is developed as a multi-purpose simulation tool. It oughts to integrate different physical subdomains like structure or flow simulation into one large system. Currently, the different domains are approached by individual strong-focused simulation tools.

VAST models physical systems with ordinary differential equations (ODEs) using the state-space representation (refer to section 2.1.1). Thereby, complex physical systems are assembled from separate state-space models. In complex systems of state space models, errors in single models can lead to numerical problems in solving the whole system.

For meaningful and numerical stable simulation results, VAST models need to fulfill some basic assumptions[1]. Some of these are that

- state-space models behave deterministic. The same input should result in the same output.
- the model outputs depend smoothly on their input. They are not stiff with respect to the desired time step size.
- the models are initialized with valid data and the models deliver meaningful data right from the beginning.

However, it is difficult to find out where errors originate as they can propagate through the whole calculation.

[1]See section 2.1 and section 2.2.3 for more details about the definition of state-space models in VAST.

## 1.2. Problem

A way to isolate the origins of numerical variances shall be found. Therefor, single models shall be analyzed for numerical stability and algorithmic correctness alone, detached from large coupled systems. Correct behavior of single models is a mandatory precondition for solving complex coupled systems.

An extensible test-interface shall be developed. The interface should make it easy to augment existing system tests with model tests and allow the tests to be run at arbitrary points in time during a simulation on the current simulation states.

The feasibility of this testing concept shall be examined.

## 1.3. Approach

As a first step, the current software architecture of VAST is studied. It is determined, how the new testing concept fits in the current architecture and new interfaces are designed.

The next step is to implement the actual test-interface and to finally integrate it in the VAST software. On top of that interface, specific model tests are implemented.

Finally, the test concept is evaluated by applying it to a set of VAST models. For that, currently existing system tests are adapted.

## 1.4. Structure of the work

The thesis begins with the chapter *Background and Fundamentals.* This chapter gives necessary background information on the VAST software. Mathematical fundamentals on ODEs and numerical stability are explained.

The chapter *Implementation* explains the design and the implementation of the newly developed test-interface. On top of the new interface, the implementation of concrete tests is explained.

In the chapter *Experiments*, the implemented tests are applied to VAST models. After running the tests on preexisting models, the obtained results are evaluated.

In the last chapter *Conclusion and Outlook*, the feasibility of the whole implemented test concept is finally discussed. Further, an overview of outstanding issues and possible future enhancements is given.

# 2. Background and Fundamentals

The following chapter outlines backgrounds and fundamentals needed for approaching the given problem.

This work has emerged in the context of the *VAST* software project. The concepts and the architecture of numerical simulations in VAST are introduced.

Subsequently, a brief introduction of the mathematical backgrounds is given. Besides stability analysis of numerical algorithms, *ordinary differential equations (ODEs)* and the concept of *state-space representation*, two concepts that are used in VAST, are introduced.

## 2.1. Versatile Aeromechanic Simulation Tool

Physical simulations are commonly accomplished by solving *ordinary differential equations (ODEs)* using numerical methods. VAST enables modularization by refining this concept. Complex simulation systems are assembled from single self-contained models in so-called *state-space* representation that take care of different concerns. These models have declared inputs and outputs. The inputs an outputs are used to form interconnections between the models. The concept of *state-space models* is elaborated in section 2.2.3.

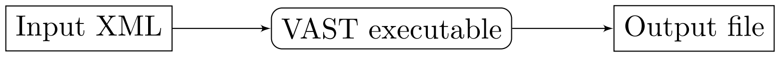


Figure 2.1.: VAST workflow

VAST is implemented in the programming language *Fortran.* The compiled application is usable as a command line executable. The simulation is specified by a configuration file using the *Extensible Markup Language (XML).* The configuration file defines the physical system that shall be simulated by specifying its models and connections between these models. Additionally, the XML configuration file can specify a solver and time steps at which the simulation should be evaluated. The path to the configuration file is committed to the VAST application as a command-line parameter. The application parses the file and instantiates necessary models and the desired solver. Afterwards, VAST performs the numeric calculations and writes the calculation results to an output file.

The quality of the VAST source code is assured by automatic execution of tests. A complete simulation is calculated and results are compared to expected output.

### 2.1.1. Architecture

On the top level, VAST parses a given input file and sets up the configured simulation. Every solver and model that can be specified in the configuration file corresponds to a specific Fortran type in the VAST source code. The corresponding type is then instantiated at runtime. VAST solvers extend the abstract type *AbstractStateSpaceSolver*. All models need to implement the *AbstractStateSpaceModel* type. Figure 2.2 shows an overview of the architecture.

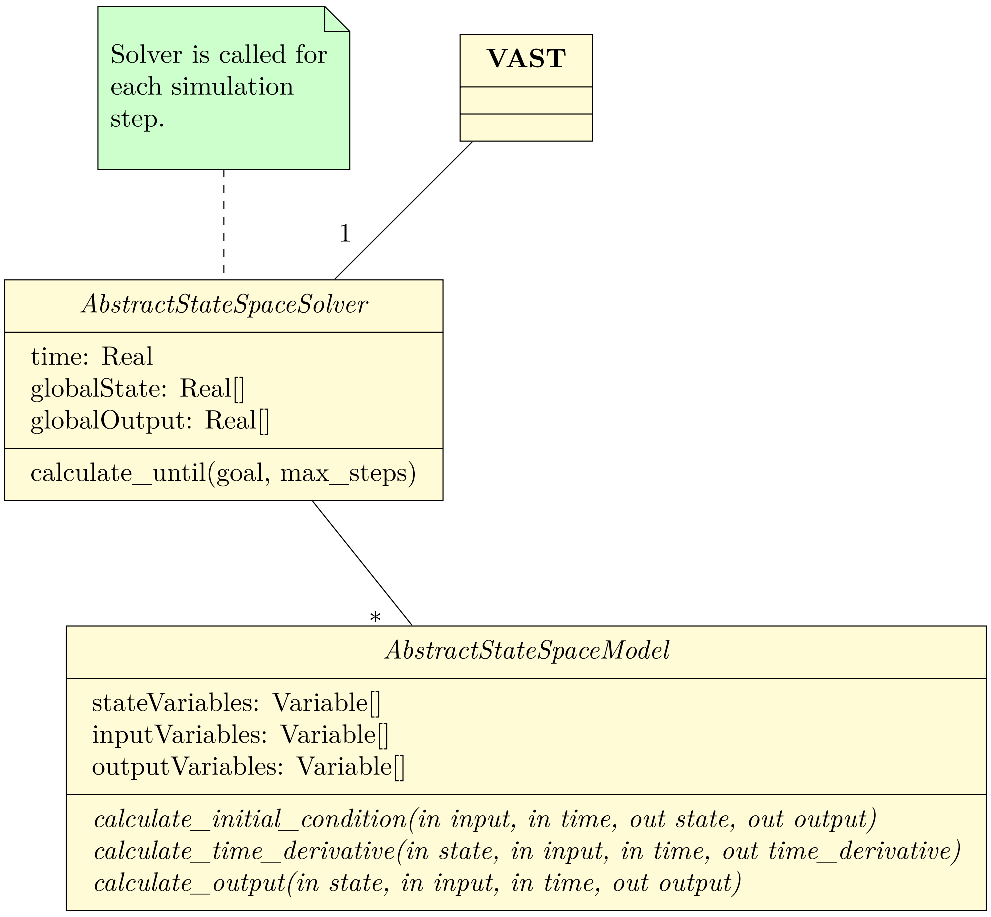

Solver is called for each simulation step.
VAST
1
AbstractStateSpaceSolver
time: Real
globalState: Real[]
globalOutput: Real[]
calculate_until(goal, max_steps)
*
AbstractStateSpaceModel
stateVariables: Variable[]
inputVariables: Variable[]
outputVariables: Variable[]
calculate_initial_condition(in input, in time, out state, out output)
calculate_time_derivative(in state, in input, in time, out time_derivative)
calculate_output(in state, in input, in time, out output)

Figure 2.2.: VAST architecture

### State-space solver

At any time, only one solver exists. The solver takes care of all models of the configured simulation and is responsible for the mapping of model inputs and outputs. Additionally, the solver maintains the current state of the simulation. That is the *globalState* and the *globalOutput* vector. The *globalOutput* is stored as it is mapped to the model input of further time steps. The solver has a method *calculate_until* which is called for each time step that should be calculated.

### State-space models

Models have a method for the initial setup, *calculate_initial_condition*, and the methods necessary for solving state-space models, *calculate_time_derivative* and *calculate_output* (regarding state-space representation refer to 2.2.3). Additionally, the class fields *stateVariables*, *inputVariables* and *outputVariables* define the inputs, state and outputs of a model.

### VAST variables

The *Variable* type is used for indexing into the global (state and output) vectors stored in the solver. It defines shape and rank of (multidimensional) variables and assigns a name to them.

```
inputVariables(1) = Variable_type('input1', rank=2, dim=(/2,2/),
                                            dimComment=(/'x','y'/))
```

The name is used by the *AbstractStateSpaceSolver* for coupling of models. The *stateVariables*, *inputVariables* and *outputVariables* fields need to be set by actual model implementations.

The *Variable* type has a method *getPointer*, which gives a pointer to the underlying data. In the following listing, *input* is a real array that is for example passed to the model. The subroutine call assigns the *var_ptr* to the correct memory location in the *input* array.

```
call this%inputVariables(1)%getPointer(input, var_ptr, ierr)
```

The variable *var_ptr* can then be used to access the value of the input variable.

### 2.1.2. Configuration and output files

VAST uses XML for configuration files. The XML structure is defined using a XML Schema Definition (XSD).

Calculation result outputs are plain text files.

#### Configuration files

On the top level, the *VAST_configuration* tag specifies that we are dealing with a configuration file for VAST.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<VAST_configuration
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="config.xsd">
  ...
</VAST_configuration>
```

*VAST_configuration* has three three children, *simulation_steps*, *solver* and *models*.

The *simulation_steps* child defines points in time where the simulation shall be simulated.

```xml
<simulation_steps>
  <time_evolution start_time="0.0" end_time="1.0" />
  <time_evolution end_time="2.0" />
</simulation_steps>
```

Using the *solver* child, the solver that is used for the simulation can be selected. Different solvers are available and can be configured.

```xml
<solver>
  <explicit_coupled_runge_kutta
      type="global explicit RK4" dt="0.001" />
</solver>
```

Finally, *models* defines the system of models that should be calculated. Different models are specified using different child tags. The models can have attributes for configuration. Models belong to specific categories. The *example_model* in the following example is part of the *dummy_models* category.

```xml
<models>
  <dummy_models>
    <example_model inverted="true"
        otherInput_dimension="23" stateVar2_dimension="37" />
    <example_model inverted="false"
        otherInput_dimension="23" stateVar2_dimension="37" />
  </dummy_models>
</models>
```

### Configuration file specification

An XML Schema Definition (XSD) is used to define the configuration XML structure. Using the XSD, the configuration file is parsed and validated on VAST execution.

Further, as part of the VAST software development process, the XSD is used for automatic code generation. The generated code gives easy access to supplied configuration.

The following source code listing corresponds to the configuration of the previous demonstrated *example_model*:

```xml
<xsd:complexType name="example_model">
  <xsd:annotation>
    <xsd:documentation>
      Simple example for a model with a single setting.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute
      name="inverted" type="xsd:boolean" default="false"/>
  <xsd:attribute
      name="stateVar2_dimension" type="xsd:int" default="3"/>
  <xsd:attribute
      name="otherInput_dimension" type="xsd:int" default="7"/>
</xsd:complexType>
```

With the generated Fortran code, the configuration options can be easily accessed. For the *example_model*, this is shown in the following source code listing:

```
call config%read(abstractConfig, ...)

! configure the model
newModel%inverted = config%inverted
newModel%m = config%stateVar2_dimension
newModel%n = config%otherInput_dimension
```

### Output files

The output of calculations is written to a text file. VAST writes the the current state of the solvers, states and outputs, for each time step specified in the configuration.

```
#### NEW SNAPSHOT ####
# Section: AbstractSolver_timeStepping
# Variable: name="time", unit="s", rank=0
    0.0000000000E+00

# Variable: name="dtSpec", unit="s", rank=1
#           dim="3", dimComment="min, last, and max dt"
    0.1000000000E+02    0.1000000000E+02    0.1000000000E+02

# Section: AbstractSolver_stateVariables
# Variable: name="S_1", unit="", rank=0
    0.3600000000E+02

# Variable: name="T_1", unit="", rank=0
    0.1500000000E+02

# Section: AbstractSolver_outputVariables
# Variable: name="q", unit="", rank=0
    0.1290000000E-02
```

### 2.1.3. Quality assurance

The quality of developed source code is assured by implementing tests alongside functionality. It is our policy, that every line of source code should be covered by at least one test. We distinguish between *unit tests* and *system tests*:

- *Unit tests* are used to test single functionalities of the application. The Fortran unit test framework *pFUnit* is used. *pFUnit* is inspired by the popular JUnit Java test framework. Using a *Python*-based preprocessor, it enables a custom syntax for *assertion* statements.
- *System tests* assure that the software, especially models and systems of coupled models, behave as expected. Therefor, the VAST executable is called with a specific configuration file and the obtained output file of the simulation is simply compared to an expected reference output. This process is controlled and automatized using *Python* scripts.

## 2.2. Mathematical Backgrounds

In the following, the concepts of numerical stability are explained. The condition of a problem is discussed and floating-point arithmetic is outlined. This leads to stability analysis of algorithms.

VAST is built on the concept of *ordinary differential equations (ODEs)*, specifically *state-space models*. These concepts are explained and stability analysis of *ODEs* is discussed.

### 2.2.1. Numerical Stability

Performing algorithmic computations on digital computers always leads to numerical errors in calculation results.

Dahmen and Reusken [3, p. 12] differentiate between the *condition of a problem* and the *stability of an algorithm*. While the condition is set by the given problem, ways for avoiding or reducing the error of algorithms can be found.

#### Condition number

Dahmen and Reusken [3, pp. 18-23] define the *condition number* as the relation of the expected output error $\Delta y$ to a given input error $\Delta x$ for a function

$$f : X \to Y. \tag{2.1}$$

The input error $\Delta x$ is defined by

$$\Delta x = \tilde{x} - x, \quad x, \tilde{x} \in X \tag{2.2}$$

and the output $\Delta y$ is defined by

$$\Delta y = f(\tilde{x}) - f(x) = \tilde{y} - y, \quad y, \tilde{y} \in Y \tag{2.3}$$

with $\tilde{x}$ being the error affected input and $\tilde{y}$ being the error affected output.

The *absolute condition* $\kappa_{abs}$ is determined by division of the absolute input and output errors,

$$\kappa_{abs} = \frac{\|\Delta y\|_Y}{\|\Delta x\|_X} \tag{2.4}$$

where $\|\cdot\|_Y$ and $\|\cdot\|_X$ are some arbitrary norms on $X$ and $Y$. In many cases, these are simply the *absolute-value norm* $\|x\| = |x|$ or the *Euclidian norm* $\|x\| := \sqrt{x_1^2 + \cdots + x_n^2}$

The relative condition $\kappa_{rel}$ is given by the division of the relative input and output errors,

$$\kappa_{rel} = \frac{\delta_y}{\delta_x} \tag{2.5}$$

with

$$\delta_x = \frac{\|\Delta x\|_X}{\|x\|_X}, \quad \delta_y = \frac{\|\Delta y\|_Y}{\|y\|_Y} \tag{2.6}$$

being the relative error of the inputs and outputs.

### Floating-point arithmetic

Real numbers are usually stored using the *floating-point representation.* In general, a number is approximated by a fixed number called *mantissa* $f$ and scaled using an exponent $e$ in some fixed base[1] $b$ [3, p. 35]. This model is shown in equation 2.7.

$$x = f * b^e \tag{2.7}$$

Because the amount of numbers that can be represented by a computer is finite, real numbers can only be approximated. Different rounding strategies can be applied. Commonly the the nearest representable number is chosen. The number

$$eps := \frac{b^{1-m}}{2} \tag{2.8}$$

states the *relative precision* of represented numbers. It is the smallest number that can be added to 1.

Additionally to the error introduced by the composition of floating-point numbers, further error results from performing arithmetic operations on these.

Dahmen and Reusken show that that for multiplication and division, the relative error remains within the bounds of the representations precision [3, p. 41]:

$$\left| \frac{\tilde{x}\tilde{y} - xy}{xy} \right| \leq 2eps \tag{2.9}$$

It is further shown that in contrast to multiplication and division, addition and subtraction can have a a very large error amplification. This is severe especially in the case of addition of two numbers with different signs[2].

[1] On todays computers the base is 2 in the most cases.

[2] The addition of two numbers with different sign is analogues to a subtraction of two numbers with the same sign.

### Stability of an algorithm

In principle, the actual numerical algorithms can be understood as concatenation of elemental floating-point operations approximating a function $f$. Thereby, existing errors are propagated and new errors are introduced by each operation.

However, the algorithm can be implemented in different ways. When designing numerical algorithms, it is the task to find a sequence of operations that produces as few errors as possible.

Dahmen and Reusken [3, p. 42] denote a numerical algorithm to be *stable*, if the produced error stays within the magnitude caused by the non-avoidable *condition* of the problem.

## 2.2.2. Ordinary differential equations

*Ordinary differential equations (ODEs)* can be used to describe *dynamic systems* in a broad spectrum of applications. Especially in the modeling of physical systems, ODEs are of interest.

An ODE is a *differential equation* that is composite by one or more functions of one independent variable and its derivatives.

In equation 2.10, let $y$ be a dependent variable. $t$ denotes an independent variable[3]. $y = y(t)$ is an unknown function for which a solution is to be found. An equation of the form

$$y^n = f\left(t, y'(t), \dots, y^{(n-1)}\right) \tag{2.10}$$

is called *ordinary differential equation of order n.*

The task of finding a solution for a given initial condition

$$y(t_0) = y^0 \tag{2.11}$$

is called *initial value problem* (also called *Cauchy problem*). [4, p. 375]

A number of coupled differential equations can form a system of equations.

$$\begin{aligned} \boldsymbol{y}^{(n)} &= \boldsymbol{f}\left(t, \boldsymbol{y}(t), \boldsymbol{y}'(t), \boldsymbol{y}''(t), \dots, \boldsymbol{y}(t)^{(n-1)}\right) \\ &= \\ \begin{pmatrix} y_1^{(n)}(t) \\ y_2^{(n)}(t) \\ \vdots \\ y_m^{(n)}(t) \end{pmatrix} &= \begin{pmatrix} f_1\left(t, \boldsymbol{y}(t), \boldsymbol{y}'(t), \boldsymbol{y}''(t), \dots, \boldsymbol{y}(t)^{(n-1)}\right) \\ f_2\left(t, \boldsymbol{y}(t), \boldsymbol{y}'(t), \boldsymbol{y}''(t), \dots, \boldsymbol{y}(t)^{(n-1)}\right) \\ \vdots \\ f_m\left(t, \boldsymbol{y}(t), \boldsymbol{y}'(t), \boldsymbol{y}''(t), \dots, \boldsymbol{y}(t)^{(n-1)}\right) \end{pmatrix} \end{aligned} \tag{2.12}$$

[3]The dependent variable is primarily the *time* $t \in [t_0, T]$ in physics applications.

Dahmen and Reusken [4, p. 380] show that *ODEs of order $n$* can be reformulated into an equivalent *system of first order equations*. This characteristic can be exploited to simplify the development of ODE solution algorithms. Instead of implementing specific algorithms for many different cases, the problem can be transformed to fit the general case of a system of ODEs of first order.

### Stability of time-integration methods

In VAST, systems of ODEs are solved using explicit time-integration methods like Runge-Kutta 4 (RK4). To estimate the stability of integration methods, an ODE

$$\frac{dy}{dt} = \lambda * x, \quad y(0) = y_0 \tag{2.13}$$

with $\lambda \in \mathbb{C}$ is introduced (Anderson [5]). The *region of stability* is the region in the complex plane, so that the solution of the ODE is stable (regarding stability refer to section 2.2.1).

Figure 2.3 shows the stability regions of the Runge-Kutta 4 method. The algorithm is stable for all $\lambda\Delta t$ that lay within the red outlined area.

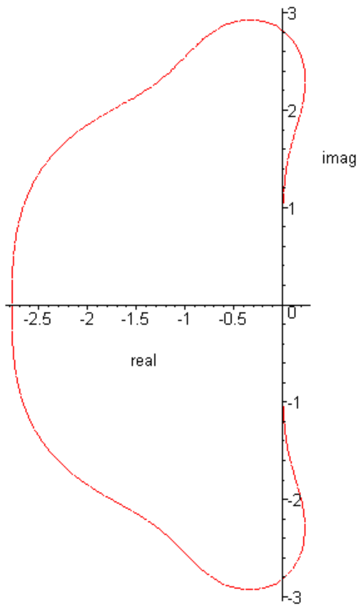


Figure 2.3.: Stability region of Runge-Kutta 4[4]

For systems of ODEs it is relevant to observe the behavior of $\frac{dx}{dt} = f(x)$ for small changes in $x$. $\lambda$ can be estimated and be used as indication for selecting a sensible magnitude for the time-step size to achieve numerical stable time-integration.

[4]Source: `http://u.cs.biu.ac.il/~schiff/Teaching/377/stab15.gif`

### 2.2.3. State-space representation

The *state-space representation* is a mathematical model for describing time-varying physical systems. It is based on the general concept of state. According to Hangos, Bokor and Szederkényi [6, p. 23], state-space models are the ”natural form” of system models in many engineering applications as the *state variables* have a clear physical meaning.

State-space models are characterized by a set of independent *input* variables and input-dependent *state* and *output* variables. State-space models are described by two sets of equations:

- *State equations* describe the change of state as a function of state and input (refer to 2.14).
- *Output equations* describe the output of the model as function of state and input (refer to 2.15).

The general form of *nonlinear continuous state-space model* equations is

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, t) \qquad (\textit{state equation}) \qquad (2.14)$$

$$\boldsymbol{y} = \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{u}, t) \qquad (\textit{output equation}) \qquad (2.15)$$

with $x$, $u$ and $y$ being state, input and output vectors [6, p. 32].

As the state often has a clear physically meaning, the most straight-forwarded approach on designing state-space models is by simply deriving them from the physical problem that should be described.

#### Solving simple state-space models

State-space models are solved in two steps. Since the output equation depends on the state, the state equation needs to be solved first.

The state equations are *ODEs of the first order.* Thus, the solution requires a numerical method.

On the other hand, the following evaluation of the output equations is purely algebraic.

### Coupling of state-space models

Beside easy physical-based description of models, state-space models have the advantage, that several of these can be *coupled* to complex systems. This is possible thanks to the clear definition of input and output variables. The inputs of models are defined by the outputs of other models. The inputs of a model are *coupled* to matching outputs of other models.

This modeling using interconnected state-space models is used in VAST (section 2.1). Thereby, it is required to solve nonlinear systems. This is, however, not further discussed at this point.

# 3. Implementation

This chapter describes the implementation of a common test-interface and the implementation of actual tests using it.

First of all, the test-interface is designed. The integration into the preexisting source code of VAST is inspected. After designing, the actual interface is built into VAST and actual model tests are implemented on top of the new interface.

Each step of implementing the interface and the tests is accompanied by the implementation of corresponding unit tests.

## 3.1. Design of the test-interface

The test interface is integrated into the VAST architecture described in section 2.1.1. Beside the *AbstractStateSpaceSolver*, a new abstract type *AbstractStateSpaceTest* that operates on *AbstractStateSpaceModel*s is introduced.

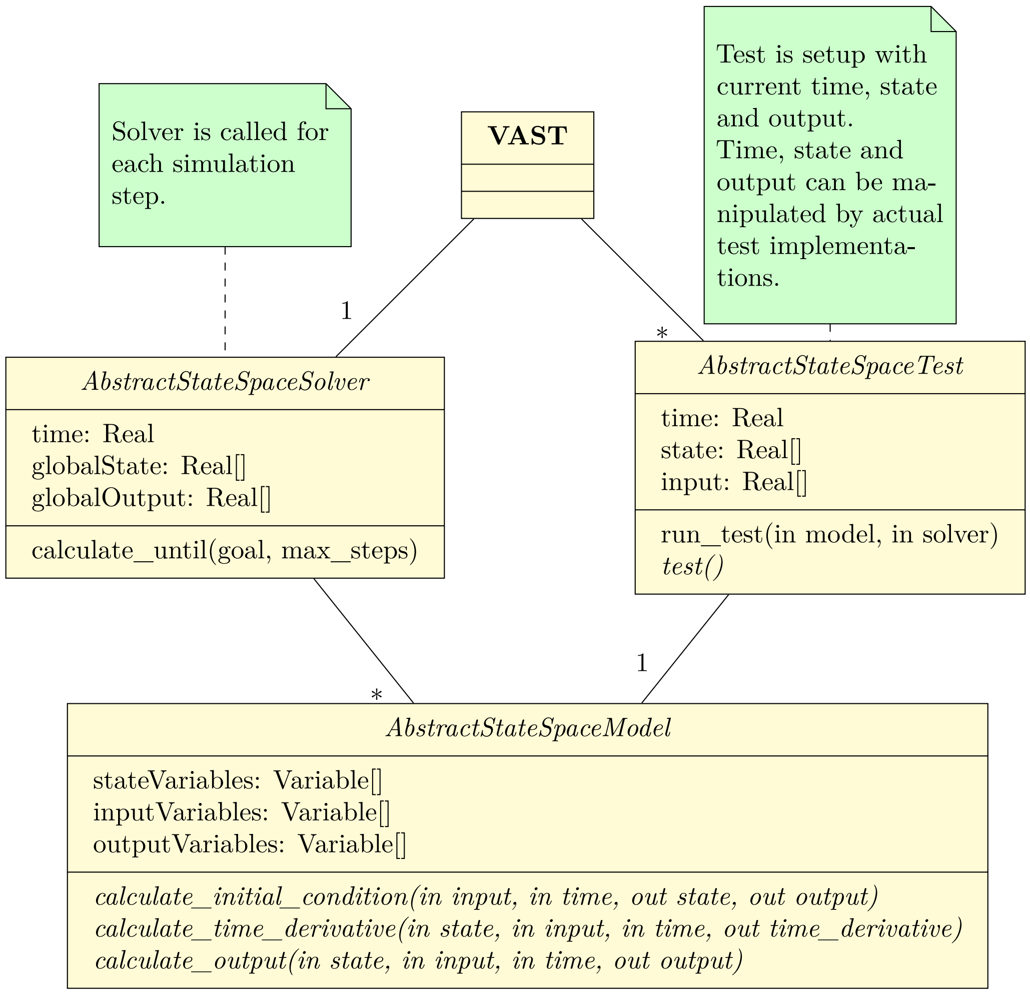
On the top level, *AbstractStateSpaceTest*s behave similar to *AbstractStateSpaceSolver*s. In contrast to the solvers, which take care of multiple models, an *AbstractStateSpaceTest* controls only one *AbstractStateSpaceModel*. Tests are setup according to a given configuration. Since tests shall be run on the current simulation state, they need access to the *models*, the current *time* and the *globalState* and the *globalOutput* vectors of the solver.

Alongside the model that should be tested, VAST passes the configured solver to the *run_test* method of the *AbstractStateSpaceTest*. The *run_test* method accesses the solver and stores copies of the current time and the state variables in class fields. Additionally, the input for the model is gathered from the solver. The gathered input is stored in a class field as well. Finally, *run_test* calls the abstract method *test*.

Actual tests, need to implement the *test* method. The implemented method can then access the previously stored class fields. Since these are copies of the actual state in the solver, it is save to manipulate these without interfering with the current simulation state still stored in the solver.

Tests are run on all models that are controlled by the solver, one by one. For each model, the test is newly instantiated and destroyed afterwards.

This integration in VAST is visualized in figure 3.1. The figure is an enhancement of the original architecture diagram in figure 2.2.



Solver is called for each simulation step.
VAST
Test is setup with current time, state and output.
Time, state and output can be manipulated by actual test implementations.
1
*
AbstractStateSpaceSolver
time: Real
globalState: Real[]
globalOutput: Real[]
calculate_until(goal, max_steps)
AbstractStateSpaceTest
time: Real
state: Real[]
input: Real[]
run_test(in model, in solver)
test()
*
1
AbstractStateSpaceModel
stateVariables: Variable[]
inputVariables: Variable[]
outputVariables: Variable[]
calculate_initial_condition(in input, in time, out state, out output)
calculate_time_derivative(in state, in input, in time, out time_derivative)
calculate_output(in state, in input, in time, out output)

Figure 3.1.: Test interface in VAST (extension of figure 2.2)

### 3.1.1. Integration of tests in VAST configuration files

Tests shall be run on determined points in time during a calculation. Therefore, test are defined in the *simulation_steps* section of the configuration file.

A *model_tests* tag can be added as child to *simulation_steps*. Tests are simply run using the current simulation state. That means that the test initialization is defined by previous *time_evolution* steps (especially its *end_time* attribute). The model-tests do not change the current simulation state, so they can be inserted without changing the simulation result (unless they fail, in that case the simulation is aborted).

```xml
<simulation_steps>
  ...
  <time_evolution end_time="10."/>
  <model_tests>
    <dummy_state_space_test/>
    ...
  </model_tests>
  ...
</simulation_steps>
```

The *model_tests* section can have multiple children. The children specify which tests shall be run. The specified tests are run on all configured models.

In the above source code listing, a dummy test is specified as an example.

## 3.2. Implementation of the test-interface

An abstract type *AbstractStateSpaceTest* is defined. Actual tests are later instantiated using the factory pattern. Therefore an abstract type *AbstractStateSpaceTestFactory* is introduced. The *AbstractStateSpaceTest* and the *AbstractStateSpaceTestFactory* are finally integrated into the preexisting sourcecode of VAST.

### 3.2.1. Implementation of the AbstractStateSpaceTest

The abstract type *AbstractStateSpaceTest* is defined in a Fortran module *vast_state_space_test*. The type has four fields. A *model* pointer shall point to the model that should be tested. The *time* field stores the time of the current simulation. The vectors *state* and *input* store the input and state relevant for the model that shall be tested. These are subsets of the global vectors (*globalState* and *globalOutput*) stored in the solver.

```fortran
type, abstract :: AbstractStateSpaceTest_type
  class(AbstractStateSpaceModel_type), pointer :: model
  real                                         :: time
  real, allocatable                            :: state(:)
  real, allocatable                            :: input(:)

contains
  procedure :: run_test => AbstractStateSpaceTest_run_test
  procedure(AbstractStateSpaceTest_test), deferred :: test

end type AbstractStateSpaceTest_type
```

The *AbstractStateSpaceTest* has a procedure *run_test* and an deferred procedure *test*. The *test* procedure needs to be implemented by actual tests and is later called by the *run_test* procedure. This is visualized in figure 3.2.

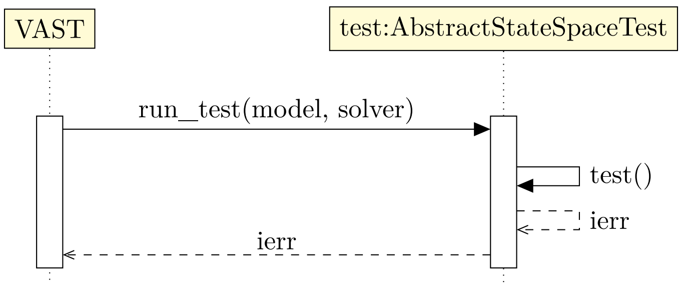


Figure 3.2.: Sequence diagram of the test process

The *run_test* procedure takes an *AbstractStateSpaceModelWrapper* and an *AbstractStateSpaceSolver* as input parameters. An integer output parameter is used as an error flag.

The *AbstractStateSpaceModelWrapper* contains an *AbstractStateSpaceModel* and additional model related information set and used by the solver. Among others, *AbstractStateSpaceModelWrapper* has information for indexing into the global state vectors of the solver.

```
subroutine AbstractStateSpaceTest_run_test(this, model_wrapper,
                                           solver, ierr)
  class(AbstractStateSpaceTest_type),    intent(inout) :: this
  class(AbstractStateSpaceModelWrapper_type), target,
                                         intent(in)    :: model_wrapper
  class(AbstractStateSpaceSolver_type),  intent(in)    :: solver
  integer,                               intent(out)   :: ierr
```

At first, the class field *model* is pointed to the actual model that should be tested. The model can be accessed from the field *p* of the *AbstractStateSpaceModelWrapper*. Subsequently, the memory for the allocatable model-local state fields *state* and *input* is allocated. The needed dimensions for the vectors are obtained from the model.

```
  this%model => model_wrapper%p

  allocate(this%state(model_wrapper%p%stateSize))
  allocate(this%input(model_wrapper%p%inputSize))
```

The *time* and *state* fields can be directly assigned. For assignment of the model-local *state*, the fields *x_begin* and *x_end* of the *AbstractStateSpaceModelWrapper* are used. *x_begin* and *x_end* define the indexes of model-local state date in the global state vector of the solver (*solver%x*).

The model-local input can not be directly obtained form the solver. Instead, a procedure of the solver, *gather_global_input*, is called. The procedure maps the last calculated global output of the solver (*solver%y*) to a passed vector (in this case called *global_input*). Thereby, it takes the *outputVariables* and *inputVariables* of all models stored in the solver into account. Using the *u_begin* and *u_end* fields of the of the *AbstractStateSpaceModelWrapper*, we can then obtain the model-local input from the *global_input* vector.

```
  real :: global_input(solver%globalInputSize)

  this%time = solver%time
  this%state = solver%x(model_wrapper%x_begin:model_wrapper%x_end)
  call solver%gather_global_input(solver%y, global_input, ierr)
  this%input = global_input(model_wrapper%u_begin:model_wrapper%u_end)
```

After the previously described setup, the deferred procedure *test* is called. At this point, the actual test itself is performed.

```
  call this%test(ierr)
```

After the test was executed, allocated memory for the local *state* and *input* vectors is cleaned up.

```
  deallocate(this%state)
  deallocate(this%input)
```

The call of the deferred *test* method has an integer error flag as output parameter. A zero value means, that no errors arose. In case of errors (e.g. a model did not pass the test), a non-zero integer is returned. The error code is checked and a respective message is logged.

```
  if (ierr == 0) then
    call log_info(TAG, "Test successfully completed!")
  else
    call log_info(TAG, "Test failed!")
  end if
end subroutine AbstractStateSpaceTest_run_test
```

The signature of the deferred *test* procedure is defined by an abstract interface. The procedure has only one input parameter. That is *this*, the test type itself. Implementations can access the model that should be tested and the current simulation state previously obtained from the solver. *ierr* is the previously mentioned error flag that is used to indicate the success of the calculation.

```fortran
abstract interface
  subroutine AbstractStateSpaceTest_test(this, ierr)
    class(AbstractStateSpaceTest_type), intent(in)  :: this
    integer,                            intent(out) :: ierr
  end subroutine AbstractStateSpaceTest_test
end interface
```

### Testing the AbstractStateSpaceTest

Unit tests for the *AbstractStateSpaceTest* are implemented in a pFUnit test case *test_state_space_test*. For testing the abstract type, an actual test *TestStateSpaceTest* is implemented. The test extends the *AbstractStateSpaceTest* and implementents the deferred *test* procedure.

```fortran
type, extends(AbstractStateSpaceTest_type) :: TestStateSpaceTest_type
contains
  procedure :: test => TestStateSpaceTest_test
end type
```

Primarily, it shall be tested if the passed data is correctly associated to the test class fields. Therefore, variables on the module level are introduced.

```fortran
real, allocatable :: test_time, test_state(:), test_input(:)
```

The implementation of the *test* procedure of the *TestStateSpaceTest* assigns the tests class fields to the previously desribed module-level variables.

```fortran
test_time =  this%time
test_state = this%state
test_input = this%input
```

The *test_TestStateSpaceTest_run_test* procedure performs the actual test. The *@test* annotation tells pFUnit that the procedure is a unit test and should be executed.

For performing the test, some dummy input is needed. The *run_test* method requires an *AbstractStateSpaceModelWrapper* and an *AbstractStateSpaceSolver*. Variables for these are declared and necessary memory is allocated. The actual used solver implementation does not matter, since no actual simulation is performed. In the following test, an *ExplicitRungeKuttaSolver* is used.

```fortran
@test
subroutine test_TestStateSpaceTest_run_test
  use pfunit_mod
  type(TestStateSpaceTest_type)             :: test
  type(AbstractStateSpaceModelWrapper_type) :: model_wrapper
  type(ExplicitRungeKuttaSolver_type)       :: solver
  real, allocatable                         :: tmp_input(:)
  integer                                   :: ierr

  ! memory allocations...
```

After all necessary allocations, the model wrapper and the solver need to be initialized with some valid data. The simulation states in the solver (global state $x$ and global output $y$) are initialized. The indexing information in the *model_wrapper* is set accordingly to the solver states.

```fortran
  ! setup model_wrapper
  model_wrapper%x_begin = 1
  ! ...

  ! setup solver
  solver%x = (/ 1.0, 2.0, 3.0 /)
  solver%y = (/ 4.0, 5.0, 6.0 /)
  solver%gather_u_offsets = (/ 0, 3 /)
  solver%gather_u_yidx = (/ 0 /)
  ! ...
```

The *gather_u_offsets* and *gather_u_yidx* fields of the solver are used by the solver for mapping the last global output $x$ to the input for the next time step. In an ordinary simulation, these would be determined by the solver on setup using the *outputVariables* and *inputVariables* definitions of all models used in the calculation.

Finally, the *run_test* method of the test is called.

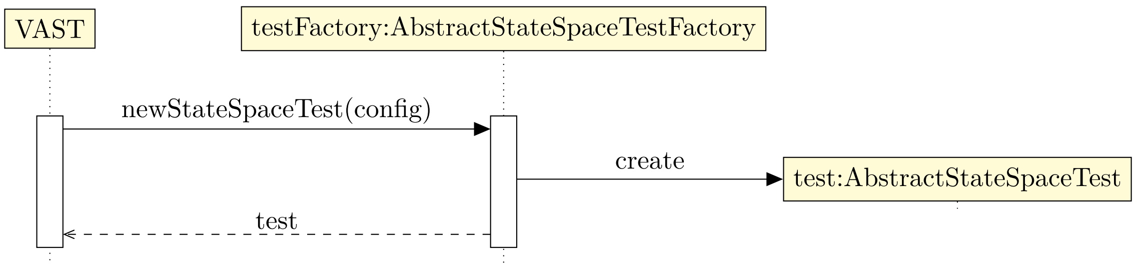
It is checked if the error return code is zero and the temporary module-level variables *test_time*, *test_state* and *test_input* are equal to the data previously passed to the *run_test* method. This is accomplished using the custom assertion syntax pFUnit provides (*@assert...*).

```
  @assertEqual(0, ierr)
  @assertEqual(0.0, test_time)
  @assertEqual(solver%x, test_state)

  call solver%gather_global_input(solver%y, tmp_input, ierr)
  @assertEqual(0, ierr)

  @assertEqual(tmp_input, test_input)
```

For checking the input for equality, the *gather_global_input* method of the solver performs the necessary mapping of the last output to the new input.

### 3.2.2. Implementation of the AbstractStateSpaceTestFactory

Tests are instantiated using the factory pattern. To provide a common interface for all test factories, an abstract type *AbstractStateSpaceTestFactory* is introduced. Figure 3.3 shows a sequence diagram of the creation of new tests.



VAST
testFactory:AbstractStateSpaceTestFactory
newStateSpaceTest(config)
create
test:AbstractStateSpaceTest
test

Figure 3.3.: Sequence diagram of test creation

The *AbstractStateSpaceTestFactory* exposes a procedure *newStateSpaceTest* that instantiates new tests.

```
type, extends(AbstractDictionaryEntry_type), abstract
                              :: AbstractStateSpaceTestFactory_type
contains
  procedure(CreateAbstractStateSpaceTest), deferred
                              :: newStateSpaceTest
end type AbstractStateSpaceTestFactory_type
```

An abstract interface defines the signature of the *newStateSpaceTest* procedure. An *AbstractConfigNode* is passed as input parameter. It is loaded from a VAST configuration file and it is used to configure the test. The output parameter *test* is the test itself that should be instantiated. Further, the *ierr* output parameter is again used as an error flag.

```fortran
abstract interface
  subroutine CreateAbstractStateSpaceTest(this, abstractConfig,
                                           test, ierr)
  class(AbstractStateSpaceTestFactory_type),
                                intent(in)  :: this
  class(AbstractConfigNode_type), intent(in)  :: abstractConfig
  class(AbstractStateSpaceTest_type), allocatable,
                                intent(out) :: test
  integer,                      intent(out) :: ierr

  end subroutine CreateAbstractStateSpaceTest
end interface
```

The *AbstractStateSpaceTestFactory* type extends an *AbstractDictionaryEntry* type. This enables *AbstractStateSpaceTestFactory* to be used as entry for a *GeneralDictionary*. *GeneralDictionary* is a utility type provided by VAST. It provides a mapping of character strings to types that extend *AbstractDictionaryEntry*.

```fortran
call factoryDict%add(factory, ierr)
factory => factoryDict%get(factoryName)
```

The dictionary is used in the next section, where the integration of the new *AbstractStateSpaceTest* type into the current VAST architecture is explained.

### Testing the AbstractStateSpaceTestFactory

A *unit* test is used for testing the *AbstractStateSpaceTestFactory* as well.

A type *EmptyStateSpaceTest* that is an *AbstractStateSpaceTest* is implemented. The implementation of the *test* method does basically nothing. The *EmptyStateSpaceTestFactory* implements the *AbstractStateSpaceTestFactory* that should be tested.

The *newStateSpaceTest* procedure of the *EmptyStateSpaceTestFactory* simply allocates an *EmptyStateSpaceTest* and returns it.

In the test itself an allocatable variable *factory* of the abstract *AbstractStateSpaceTestFactory* type is defined. Using *allocate*, the variable is initialized with the concrete *EmptyStateSpaceTestFactory* implementation.

```
@test
subroutine test_empty_state_space_test_factory
  ! some variables omitted...
  class(AbstractStateSpaceTestFactory_type), allocatable :: factory
  class(AbstractStateSpaceTest_type), allocatable        :: model_test

  allocate(EmptyStateSpaceTestFactory_type::factory)
  @assertTrue(allocated(myFactory))
```

The abstract factory is then used to create an actual *EmptyStateSpaceTest*.

```
  call factory%newStateSpaceTest(configNode, model_test, ierr)
  @assertEqual(0, ierr)
```

It is checked if the the newly created test is correctly allocated. It is asserted that it is allocated at all and that the test is of the correct type. Therefor a *select-type* statement is used.

```
  @assertTrue(allocated(model_test))
  select type(model_test)
  type is (EmptyStateSpaceTest_type)
    @assertTrue(.true.)
  class default
    @assertTrue(.false.)
  end select
end subroutine test_empty_state_space_test_factory
```

### 3.2.3. Integration of the test-interface in VAST

The new test-interface needs to be integrated into the current VAST system. The code that is responsible for reading the *simulation_steps* section of the configuration needs to be adapted.

A loop is used to iterate over all *simulation_steps* children. The name of each child is fetched and a *select case*-statement is used to distinguish between different possible simulation steps.

```
nSteps = configNode%getNumberOfChildren(ierr)
do iStep = 1, nSteps, 1
  stepNode = configNode%getChildNode(iStep, ierr)

  stepType = stepNode%getName(ierr)
  select case(stepType)
    case("time_evolution")
      ...

    case("model_tests")
      call model_tests(stepNode, solver, ierr)

  end select
end do
```

At this point, a new *case* is added. If a config node is called "model_tests", a subroutine *model_tests* is called. The config node itself and the solver of the simulation is passed.

The *model_tests* subroutine performs tests on all models of the passed solver.

```
subroutine model_tests(config, solver, ierr)
  type(AbstractConfigNode_type),         intent(in)    :: config
  class(AbstractStateSpaceSolver_type), intent(inout) :: solver
  integer,                               intent(out)   :: ierr
```

For the instantiation of the tests, the factory pattern is used. Therefore, the necessary factories need to be created. The creation of a factory for a simple *DummyTest* (later explained in section 3.3.1) for example is delegated to a subroutine *generate_DummyTestFactories*. The factories are stored in a dictionary *factoryDict* which is of type *GeneralDicitonary*.

```
  call generate_DummyTestFactories(factoryDict, ierr)
```

For all possible tests, factories need to be created at this point.

After creating the factories, a loop is used to iterate over all configuration sub nodes of the passed configuration. The sub nodes define the actual tests that should be executed.

```
  nTests = config%getNumberOfChildren(ierr)
  do i = 1, nTests, 1
    testNode = config%getChildNode(i, ierr)
    factoryName = testNode%getName(ierr)
    factory => factoryDict%get(trim(factoryName))
```

The factories for each test are fetched from the *factoryDict* using the name of the desired test. The name is obtained from the configuration nodes.

Models are obtained from the solver. The models are iterated and a new test is instantiated for each model. Test are finally run by passing the model and the solver to the *run_test* procedure of the test.

```fortran
    nModels = size(solver%models)
    do j = 1, nModels, 1
      call factory%newStateSpaceTest(testNode, test, ierr)

      call test%run_test(solver%models(j), solver, ierr)

      deallocate(test)
    end do

    ! check ierr for error...
  end do
end subroutine model_tests
```

The call of the *newStateSpaceTest* procedure allocates memory for the test. Thus, it needs to be deallocated after the test is executed.

### Testing the test-interface integration

A *unit* test for testing the integration of the interface into VAST is setup in a similar way to the *AbstractStateSpaceTest* unit test. It shall be tested if a passed configuration is parsed correctly and defined tests are executed.

Again, a solver with valid models and states is set up and configured.

Additionally an in-memory representation of a valid configuration is created. The shown configuration sets up a "dummy_state_space_test" (refer to section 3.3.1).

```fortran
tixiCreateElement(configNode%tixiHandle,
                  trim(configNode%xml_path),
                  'model_tests')
tixiCreateElement(configNode%tixiHandle,
                  trim(configNode%xml_path) // '/model_tests',
                  'dummy_state_space_test')
tixiAddTextAttribute(configNode%tixiHandle,
                     trim(configNode%xml_path) // '/model_tests
                                        /dummy_state_space_test',
                     'fail', 'false')
```

The earlier in section 3.2.3 described *model_tests* subroutine is then called with the setup configuration.

```
call model_tests(configNode, solver, outputWriter, ierr)
@assertEqual(0, ierr)
```

This validates that parsing the configuration and running the test does not cause any errors. However, it needs to be assured that the tests did actually run.

Therefore, the *DummyTest* is configured to fail and the *model_tests* subroutine is called again.

```
tixiAddTextAttribute(configNode%tixiHandle,
                     trim(configNode%xml_path) // '/model_tests
                                                   /dummy_state_space_test',
                     'fail', 'true')

call model_tests(configNode, solver, outputWriter, ierr)
@assertFalse(ierr == 0)
```

In this case, the error flag *ierr* should not equal zero. That indicates that an error has occurred as expected.

## 3.3. Implementation of actual tests

In the following, actual model tests are implemented on top of the developed interface. Three different tests are designed. A dummy test is primarily intended for testing the test interface. Further, it serves as reference for further test implementations. A determinism test checks that state-space models produce the same output for a given input when invoked multiple times. In addition, it checks how much the output changes for slightly different input. Finally, an initial condition test validates that the output of the *calculate_initial_condition* procedure is consistent with the output of *calculate_output*.

### 3.3.1. Dummy Test

The *DummyStateSpaceTest* is the first test that was implemented. It was developed alongside the implementation of the interface. In this way, experiences from the implementation of the *DummyStateSpaceTest* could be fed back into the design of the interface.

The *DummyStateSpaceTest* is no real test that invokes any procedures of the model. Instead, it can be configures if the model should fail or succeed. Models that are passed to the test are completely ignored.

If the test should fail or succeed can be configured in the VAST XML configuration file.

```xml
<model_tests>
  <dummy_state_space_test fail="true"/>
</model_tests>
```

The following XSD defines available configuration options. The only attribute *fail* has a default value of *false*. That means that the test will run successfully if nothing else is explicitly set.

```xml
<xsd:complexType name="dummy_state_space_test">
  <xsd:attribute name="fail" type="xsd:boolean" default="false"/>
</xsd:complexType>
```

### Implementation of the DummyStateSpaceTest

The *DummyStateSpaceTest* type extends the *AbstractStateSpaceTest*. A field *fail* determines if the test should succeed or not.

```
type, extends(AbstractStateSpaceTest_type) :: DummyStateSpaceTest_type
  logical :: fail
contains
  procedure :: setup => DummyStateSpaceTest_setup
  procedure :: test => DummyStateSpaceTest_test
end type
```

The *setup* procedure is used for initializing the test. In this simple test, it just assigns the *fail* class field.

```
subroutine DummyStateSpaceTest_setup(this, fail)
  class(DummyStateSpaceTest_type), intent(out) :: this
  logical,                         intent(in)  :: fail

  this%fail = fail
end subroutine DummyStateSpaceTest_setup
```

The implemented *setup* procedure simply sets the *ierr* error flag depending on the *fail* class field.

```
if (this%fail) then
  call log_debug(TAG, "fail as expected")
  ierr = 1
else
  ierr = 0
end if
```

### Testing the the DummyStateSpaceTest

The *DummyStateSpaceTest* is tested using a unit test. The test is configured in both possible ways (*fail* = *false* and *fail* = *true*) and the returned error flag *ierr* is checked.

```
call dummy_test%setup(.false.)
call dummy_test%run_test(model_wrapper, solver, ierr)

@assertEqual(0, ierr)

call dummy_test%setup(.true.)
call dummy_test%run_test(model_wrapper, solver, ierr)

@assertFalse(ierr == 0)
```

Although the implemented *DummyStateSpaceTest* does not access the passed model, the passed *model_wrapper* and the passed *solver* need to be configured correctly. That is because the extended *AbstractStateSpaceSolver* still obtains the model-local states and inputs from the passed solver, when *run_test* is invoked.

### Implementation of the DummyStateSpaceTestFactory

The *DummyStateSpaceTestFactory* extends the *AbstractStateSpaceTestFactroy* type.

```
type, extends(AbstractStateSpaceTestFactory_type)
                                               :: DummyStateSpaceFactory_type
contains
  procedure :: newStateSpaceTest
end type DummyStateSpaceFactory_type
```

The implementation of the *newStateSpace* procedure reads the passed configuration and sets up the model accordingly.

```
call config%read(abstractConfig, ierr)
call newTest%setup(config%fail)
```

#### Testing the DummyStateSpaceTestFactory

A unit test validates that the factory creates the correct model and the model is configured properly. Therefore, an in-memory *configNode* is passed to the *newStateSpace* procedure.

```fortran
tixiAddTextAttribute(configNode%tixiHandle,
                     trim(configNode%xml_path),
                     'fail', 'true')
call factory%newStateSpaceTest(configNode, model_test, ierr)
@assertEqual(0, ierr)
@assertTrue(allocated(model_test))
select type(model_test)
type is(DummyStateSpaceTest_type)
  @assertTrue(model_test%fail)
class default
  @assertTrue(.false.)
end select
```

After the model was created, it is checked if it is of the correct type and if it is configured correctly according to the previously passed configuration.

### 3.3.2. Determinism Test

The *DeterminismStateSpaceTest* validates that a model behaves in a deterministic way.

The procedures of the VAST models are numerical algorithms. The input data consists of floating-point numbers. Thus, already the input is affected by numerical error. The algorithm itself is a sequence of floating-point operations that results in even more error. While computers are essentially deterministic, this characteristic might get lost when computer programs are for example parallelized.

Goal of the *DeterminismStateSpaceTest* is to assure that the output variance for same (or similar) input is as small as necessary. Two primary sources of error shall be eliminated:

- The algorithm has no implementation failure like uninitialized or random output and
- the algorithm is numerically stable. Though we do not know the real condition, the user can configure an allowed magnitude for the error

An allowed relative variance in the model outputs can be configured in the XML configuration. Further, it can be specified that the model inputs are varied by a given relative variance.

```
<determinism_state_space_test use_current_state="true"
                              rel_input_variance="0"
                              rel_output_variance="0"/>
```

It can be configured what model input the test should use. When *use_current_state* is set to *true*, the test uses the current state obtained from the solver. Alternatively, *const_input* and *const_state* can be used to set constant model inputs that are used instead.

```
<xsd:complexType name="determinism_state_space_test">
  <xsd:attribute name="use_current_state" type="xsd:boolean"
                                          default="false"/>
  <xsd:attribute name="const_input" type="xsd:double" default="0"/>
  <xsd:attribute name="const_state" type="xsd:double" default="0"/>
  <xsd:attribute name="rel_input_variance" type="xsd:double" />
  <xsd:attribute name="rel_output_variance" type="xsd:double" />
</xsd:complexType>
```

The configuration defaults to use the constant set input and state. These in turn default to a value of zero.

### Implementation of the DeterminismStateSpaceTest

The *DeterminismStateSpaceTest* extends the *AbstractStateSpaceTest*. The previously explained configuration options are exposed as class fields.

```
type, extends(AbstractStateSpaceTest_type)
                                    :: DeterminismStateSpaceTest_type
  logical :: use_current_state
  real :: const_input
  real :: const_state
  real :: rel_input_variance
  real :: rel_output_variance
contains
  procedure :: setup => DeterminismStateSpaceTest_setup
  procedure :: test => DeterminismStateSpaceTest_test
end type
```

The *setup* procedure takes the configuration options as parameters and sets the class fields accordingly. This is analogous to the setup procedure of the *DummyStateSpaceTest*.

In the *test* procedure, the actual determinism test on the model is performed. Initially, the model input that should be used is assigned to the variables *base_input* and *base_state*.

```
if (this%use_current_state) then
  base_input = this%input
  base_state = this%state
else
  base_input = this%const_input
  base_state = this%const_state
end if
```

Thereby, it is differentiated if the current simulation calculation state or the configured constant inputs should be used. The current simulation state can be obtained from the the class fields *input* and *state*. These are set by the extended *AbstractStateSpaceTest* which in turn obtains it from the solver of the current simulation. The configured constant input and state can be obtained from the class fields *const_input* and *const_state*.

The testing of the model itself is performed by calling the specific model procedures multiple times and comparing the outputs. A constant module-level parameter *TEST_ITERATIONS* specifies how often each single procedure is invoked. By the definition of *TEST_ITERATIONS*, the needed number of iterations needs to be weighed up. While two iterations can already prove determinism, random derivation of the input data and more iterations are needed to assure stability.

For each iteration, a utility subroutine *nearby_rel* is called. *nearby_rel* assigns given reference data (in the following listing *base_input*) to a target vector (*input*), thereby varying the values by a random value in the range of the passed variance. Thus, the inputs for the model procedures can be slightly different for each iteration (depending on configuration).

```
do i = 1, TEST_ITERATIONS
  call nearby_rel(this%model%inputVariables, input, base_input,
                  this%rel_input_variance, ierr)

  call this%model%calculate_initial_condition(input, this%time,
                                                    states(:,i),
                                                    outputs(:,i), ierr)
end do
```

The returned outputs of the model (*states* and *outputs* for the *calculate_initial_condition* procedure in the listing above) are stored in a multi-dimensional vector.

After all iterations completed, the stored outputs are compared. Thereby, an utility function *compare_rel* is used. The function compares two passed arrays and returns an array of the same size and dimensions. The returned array contains the comparison results for all elements passed to the function. Using the function *all*, it is assured that all elements are *true*. *compare_rel* further allows a relative variance.

```
do i = 2, TEST_ITERATIONS
  VAST_ASSERT(all(compare_rel(this%model%stateVariables, states(:,i),
                              states(:,1), this%rel_output_variance,
                              .false., ierr)), ierr)
  VAST_ASSERT(all(compare_rel(this%model%outputVariables, outputs(:,i),
                              outputs(:,1), this%rel_output_variance,
                              .false., ierr)), ierr)
end do
```

*VAST_ASSERT* is a macro that is used at this point. It asserts that its argument evaluates to *true* and sets the passed error flag *ierr* to a non-zero value if it is not. Further, it immediately returns the currently executes routine or function in the error case. If the difference between the compared outputs is not within the expected variance, the *compare_rel* function further prints an error message to the console. This error message contains information on the obtained values and the difference between these. This difference can be though of as the estimated $\lambda$ explained in section 2.2.2 (equation 2.13).

Beside the *calculate_initial_condition* procedure, the procedures *calculate_time_derivative*, *calculate_output* and *calculate_output_gradient* are tested analogous. The complete determinism test completes successfully if the determinism requirement is fulfilled by all tested model procedures.

### Testing the DeterminismStateSpaceTest

For testing the *DeterminismStateSpaceTest*, a unit test is written. It needs to be assured, that the test detects inconsistencies in the model outputs. This is achieved by implementing a model *TestStateSpaceModel* that purposefully varies the returned data.

```
type, extends(AbstractStateSpaceModel_type) :: TestStateSpaceModel_type
  real       :: state_variance, time_derivative_variance,
                output_variance
contains
  procedure :: calculate_initial_condition => test_initial_condition
  ! ...
end type
```

The *TestStateSpaceModel* exposes the class fields *state_variance*, *time_derivative_variance* and *output_variance*. These are used to set the variation the model should produce.

The implemented model procedures pass through their inputs. In the following listing the implementation of the *calculate_initial_condition* procedure is shown.

```fortran
call random_number(state)
state = input + (2 * state - 1) * this%state_variance
call random_number(output)
output = input + (2 * output - 1) * this%output_variance
```

At this point, the variance defined in the class fields is applied. The *random_number* subroutine creates random values in the range of zero to one. The random values are multiplied by the desired variance and added to the the input. The *calculate_time_derivative*, *calculate_output* and *calculate_output_gradient* procedures are implemented in a similar way.

In the actual test code, the just implemented test model is used. It is configured in different ways and the determinism test is set respectively.

As a first test, the variance of the model is set to zero.

```fortran
model%state_variance = 0.
model%time_derivative_variance = 0.
model%output_variance = 0.
call determinism_test%setup(.false., 2., 2., 0., 0.)
call determinism_test%run_test(model_wrapper, solver, ierr)
@assertEqual(0, ierr)
```

In this case, the model just returns the input. The determinism test is configured to not accept any variation in the model outputs.

In a subsequent test, the output variances are set to a value of one. The test is setup with a constant input and state of two. With an absolute variance of one and input value of two, the expected output lies in the range of one to three. This result in an absolute output variance of two. When set in relation to the lower and upper bounds of the output range, this is a relative tolerance between two and two thirds ($2/1 = 2$ and $2/3 = \frac{2}{3}$). Thus, a relative tolerance of two is accepted.

```fortran
model%state_variance = 1.
model%time_derivative_variance = 1.
model%output_variance = 1.
call determinism_test%setup(.false., 2., 2., 0., 2.)
call determinism_test%run_test(model_wrapper, solver, ierr)
@assertEqual(0, ierr)
```

Following, the some variance is applied, but the allowed output variance is reduced. A value of zero in the following listing assures that the test fails. This expected behavior is asserted by checking the *ierr* error flag.

```
call determinism_test%setup(.false., 2., 2., 0., 0.)
call determinism_test%run_test(model_wrapper, solver, ierr)
@assertFalse(ierr == 0)
```

#### Implementation of the DeterminismStateSpaceTestFactory

A factory for the *DeterminismStateSpaceTest* is implemented in a similar way to the *DummyStateSpaceTestFactory* in section 3.3.1. The configuration is read and the model is set alike. Merely the actual configuration parameters differ.

### 3.3.3. Initial Condition Test

The state-space models in VAST expose a procedure *calculate_inital_condition*. The procedure is called at the beginning of a simulation by the solver. Given an initial *input* and a *time*, it initially sets *state* and *output*.

The *InitialConditionStateSpaceTest* shall ensure that the output of *calculate_inital_condition* is consistent with the output of the *calculate_output* procedure. Thereby, *calculate_output* is called with the *state* obtained from *calculate_initial_condition*.

In analogy to the determinism test, the initial condition test can either use the current input state from the solver or a configured constant as input for the model procedures.

```
<initial_condition_state_space_test const_input="0"
                                    rel_output_variance="0"/>
```

In contrast to the determinism test, only input can be set. The state is obtained from the calling the *calculate_inital_condition* procedure.

```
<xsd:complexType name="initial_condition_state_space_test">
  <xsd:attribute name="use_current_input" type="xsd:boolean"
                                          default="false"/>
  <xsd:attribute name="const_input" type="xsd:double" default="0"/>
  <xsd:attribute name="rel_output_variance" type="xsd:double" />
</xsd:complexType>
```

### Implementation of the InitialConditionStateSpaceTest

Like the other model tests, the *InitialConditionStateSpaceTest* extends *AbstractStateSpaceTest*. Again, the configuration options are stored in class fields.

In the implementation of the *test* procedure, it is distinguished if the input obtained from the solver or the configured constant input should be used. Dependently, a variable *input* is initialized.

```
if (.not. this%use_current_input) then
  input = this%const_input
else
  input = this%input
end if
```

The *input* is then passed to the *calculate_initial_condition* model procedure. *calculate_initial_condition* sets the passed *state* and *outputs* vectors.

```
call this%model%calculate_initial_condition(input, this%time, state,
                                             outputs(:,1), ierr)
```

The *outputs* vector has to dimensions. The output of *calculate_initial_condition* is stored to the index one.

Subsequently, the *calculate_output* procedure is invoked. The same *input* and the *state* obtained from *calculate_initial_condition* is passed.

```
call this%model%calculate_output(state, input, this%time,
                                 outputs(:,2), ierr)
```

The output is saved to the index two of the *outputs* vector.

Finally, the outputs of both procedures are compared using the *comapre_rel* utility function. A configures relative variance is allowed.

```
VAST_ASSERT(all(compare_rel(this%model%outputVariables,
                            outputs(:,1), outputs(:,2),
                            this%rel_output_variance, ierr)), ierr)
```

### Testing the InitialConditionStateSpaceTest

For the *InitialConditionStateSpaceTest*, also a test model *TestStateSpaceModel* is implemented. The model is much simpler than the test model of the determinism test. Only a boolean class field *fail* describes the model behavior.

```
type, extends(AbstractStateSpaceModel_type) :: TestStateSpaceModel_type
  logical   :: fail
contains
  procedure :: calculate_initial_condition => test_initial_condition
  ! ...
end type
```

The *calculate_initial_condition* procedure just passes the given input through to the *state* and *output* output parameters.

```
state = input
output = input
```

The *calculate_output* procedure behaves depending on the *fail* class field. If the test should fail, the procedure adds the value one to the input and returns it as output. If the test should not fail, the given *input* is simply returned as output. This way, *calculate_initial_condition* and *calculate_output* are surely returning the same outputs if they receive the same input.

```
if (this%fail) then
  output = input + 1.
else
  output = input
end if
```

In the implementation of the test code, the model is firstly configured to not fail. After that the model is setup to fail. After the initial condition test is run, it is asserted that the *ierr* error flag is as expected for each configuration.

```
model%fail = .false.
call initial_condition_test%setup(.false., 2., 0)
call initial_condition_test%run_test(model_wrapper, solver, ierr)
@assertEqual(0, ierr)

model%fail = .true.
call initial_condition_test%setup(.false., 2., 0)
call initial_condition_test%run_test(model_wrapper, solver, ierr)
@assertFalse(ierr == 0)
```

#### Implementation of the InitialConditionStateSpaceTestFactory

Like the *DeterminismStateSpaceTestFactory*, the *InitialConditionStateSpaceTestFactory* is implemented in a similar fashion to the *DummyStateSpaceTestFactory*. Reading the configuration and setting the model accordingly is explained in section 3.3.1.

# 4. Experiments

This chapter describes the application of the previously implemented tests. To evaluate the implemented test-concept, the developed tests are run on existing VAST models. Afterwards, the test results are evaluated.

## 4.1. Running tests on models

VAST currently has a collection of about 60 model tests, bundled in the so-called "generic_testsuite". These are *system tests* that validate that implemented state-space models behave as expected. The system tests are run by passing configuration files to the VAST executable. Resulting output is compared to expected reference output. Thus, each test consists of a configuration input file and an expected reference output file. Each Test is located in its own directory. Figure 4.1 outlines this file hierarchy for a "single_body_const_vel" test. The file "vast_config.xml" is the configuration input file and "vast_output.txt" is the expected reference output.

```
generic_testsuite
│
└── single_body_const_vel
    │
    ├── ref
    │   │
    │   └── vast_output.txt
    │
    └── vast_config.xml
│
└── ...
```

Figure 4.1.: File structure of a system test

The implemented determinism and initial condition tests shall be applied to the system tests contained in the *gegenric_testsuite*. Therefore, a *model_tests* section needs to be added to the *simulation_steps* section of the configuration files. The following listing shows the test configuration that is added to the tests.

```
<model_tests>
  <initial_condition_state_space_test const_input="0"
                                      rel_output_variance="0"/>
  <determinism_state_space_test use_current_state="true"
                                rel_input_variance="0"
                                rel_output_variance="0"/>
  <determinism_state_space_test use_current_state="true"
                                rel_input_variance="1.e-8"
                                rel_output_variance="1.e-4"/>\
</model_tests>\
```

To circumvent the adaption of every single test, a utility script “modeltest.sh” is written. The script iterates all tests in the *gegenric_testsuite* folder and adapts their configuration. The new configuration is saved to a new location and finally the *VAST* simulation is run. The adapted configurations and calculation outputs are collected in a new folder “modeltest”.

```
for tc in ${test_cases}; do
  mkdir ${tc}
  cd ${tc}
  cat "../../${test_dir}/${tc}/vast_config.xml"
    | sed "s#</simulation_steps>#${test_config}</simulation_steps>#"
    > vast_config.xml
  ../../${vast} vast_config.xml &> vast.log || echo "failed"
done
```

The augmentation of the configuration files is achieved using the unix command *sed*. The string “</simulation_steps>” is searched and replaced by the “${test_config}</simulation_steps>”. Thereby, “${test_config}” is a variable containing the *model_tests* section that should be added. As a result, the *model_test* configuration is added as the last child to the *simulation_steps* section. This assures that the simulation already ran and models and solver are initialized. For finding problems in failing system tests, however, tests can also be inserted as first step, as failing tests will abort before reaching the final simulation steps.

The newly created configuration is passed to the VAST executable. Output is piped to a file "vast.log". In the case of an error, the message "failed" is printed to the console. For further information on why a test may have failed, the "vast.log" file can be examined.

Figure 4.2 shows the resulting file structure that the script creates.

```
modeltest
├── single_body_const_vel
│   ├── vast.log
│   ├── vast_config.xml
│   └── vast_output.txt
└── ...
```

Figure 4.2.: File structure of model test results

## 4.2. Evaluation of test-results

The test run successfully on the most part of the *generic_testsuite* tests. However, the console output of the “modeltest.sh” test script shows that performing the tests on a “pid controller” model and on a “simple rotor” model (the *quasisteady_helicopter_velocity_sweep_noeulerangles* test) yielded errors.

```
single_body_const_acc
single_body_const_angularAcc
...
pid_controller_model
failed
quasisteady_helicopter_velocity_sweep_noeulerangles
failed
```

The “pid controller” model simulates a proportional–integral–derivative (PID) controller. Basically, a PID controller is a mechanism that applies responsive correction to a control function trying to optimize a measured process variable. This is achieved by calculating proportional, integral, and derivative responses. [7]

For further investigation on why the test failed, the output log of the failed test is examined. The output of the “pid_controller_model” test can be found at the file location “modeltest/pid_controller_model/vast.log”. An excerpt from the output containing information of the exact error is attached in appendix A.1. Examination of the log reveals, that the initial condition test failed. That means that the output of the *calculate_initial_condition* procedure is not consistent with the output of the *calculate_output* procedure.

The “simple rotor” model, a model that simulates rotor aeromechanics, fails the determinism test. An excerpt from the log output can be found in appendix A.2. The log shows that the determinism test runs successfully on the current state without any input variation. When the input is varied, the test fails. This shows that the model behaves deterministic, but is not numerically stable. The issue was found in a failing system test that consists of eight coupled models. Applying the model-tests revealed the exact model that causes the error.

Further investigation towards the exact reason for the failing tests is not done at this point as it goes beyond the scope of this thesis. Insights about the implementation details of the the PID model and the physics behind the aeromechanic rotor simulation would be required.

Regarding all other system tests, no problems were noticed.

# 5. Conclusion and Outlook

In this last chapter, the present thesis is summarized and final results are discussed.

A complementary test concept was developed for the coupled simulation tool VAST. The test concept can be used to test individual models in isolation to determine whether they meet certain requirements.

The feasibility of the implemented test concept is evaluated and an outlook on possible future work is given.

## 5.1. Conclusion

In order to evaluate the test concept, an extensible test-interface was developed. On top of this interface, two specific tests, a fuzzy determinism and an initial condition test, were implemented. The fuzzy determinism test ensures that models behave numerically stable, e. g. that they provide similar output for similar input. The initial condition test assures the consistency of model implementations.

Running the implemented tests on the *generic_testsuite* revealed that the tests can indeed be used to detect misbehavior of models. Some issues with the calculation of the initial condition of a “pid controller” model and stability issues with a “simple rotor” model were found. Especially regarding the “simple rotor” model, the new test concept has proven to be valuable. The model test exposed a faulty model that caused a coupled simulation to fail.

However, most of the VAST models seem to be in a good shape. In these cases, the tests give more confidence that the underlying code behaves as expected.

The new test concept is not meant to replace the already existing test concepts. Instead, it is a good addition to the current approach of plain unit tests and system tests that replicate simulation results. Together they can make up a solid testing foundation that results in high quality source code without unexpected side-effects.

## 5.2. Outlook

In the short-time future, the reasons for the failure of the “pid controller” model and the “simple rotor” model are to be investigated and errors in their implementations need to be fixed. To find more potential errors, the implemented determinism and initial condition test can be configured with different parameters (other *const_input* and *const_state* values) and run on the models again.

Further it can be investigated in what extent the new testing approach can help and streamline the process of developing new models in VAST. In particular, the model tests can be easily automated for all (new) system tests using a continuous integration service such as Jenkins CI.

An important point is also to analyse what assistance the new test system can provide when problems in large coupled systems shall be revealed. Further tests in larger systems can be carried out.

In the long term, the test framework can be extended by implementing additional tests. This is possible thanks to the extensibility of the developed interface.

# Bibliography

[1] *DLR at a glance.* URL: `http://www.dlr.de/dlr/en/desktopdefault.aspx/tabid-10443/637_read-251` (visited on 04/09/2017).

[2] *About Us.* URL: `http://www.dhbw.de/english/dhbw/about-us.html` (visited on 04/09/2017).

[3] Wolfgang Dahmen and Arnold Reusken. 'Fehleranalyse: Kondition, Rundungsfehler, Stabilität'. In: *Numerik für Ingenieure und Naturwissenschaftler.* 2nd ed. Aachen: Springer-Verlag Berlin Heidelberg, 2008. Chap. 2, pp. 11–50. ISBN: 978-3-540-76492-2. DOI: `10.1007/978-3-540-76493-9_2`.

[4] Wolfgang Dahmen and Arnold Reusken. 'Gewöhnliche Differentialgleichugnen'. In: *Numerik für Ingenieure und Naturwissenschaftler.* 2nd ed. Aachen: Springer-Verlag Berlin Heidelberg, 2008. Chap. 11, pp. 375–454. ISBN: 978-3-540-76492-2. DOI: `10.1007/978-3-540-76493-9_11`.

[5] Chris Anderson. *Regions of Absolute Stability.* 2008. URL: `http://www.math.ucla.edu/~anderson/270e.1.08f/summaries/RegionsOfAbsoluteStability.html` (visited on 04/09/2017).

[6] Katalin M. Hangos, József Bokor and Gábor Szederkényi. 'State-space Models'. In: *Analysis and Control of Nonlinear Process Systems.* Budapest: Springer-Verlag London, 2004. Chap. 3, pp. 23–37. ISBN: 978-1-85233-600-4. DOI: `10.1007/1-85233-861-X_3`.

[7] *PID Theory Explained.* URL: `http://www.ni.com/white-paper/3782/en/` (visited on 04/09/2017).

# A. Appendix

## A.1. Log of pid_controller_model test

```
2017-08-28 14:09:46,450 [INFO] VAST: Trying to create
                               "initial_condition_state_space_test"
                               test...
2017-08-28 14:09:46,450 [INFO] AbstractStateSpaceTest: Setup test
                               InitialConditionStateSpaceTest for model
                               pid_controller_model
2017-08-28 14:09:46,450 [INFO] AbstractStateSpaceTest: Copying time...
2017-08-28 14:09:46,450 [INFO] AbstractStateSpaceTest: Copying state...
2017-08-28 14:09:46,450 [INFO] AbstractStateSpaceTest: Gathering
                               input...
2017-08-28 14:09:46,450 [INFO] AbstractStateSpaceTest: Running test...
2017-08-28 14:09:46,450 [DEBUG] InitialConditionStateSpaceTest:
                                const_input: 0.000
2017-08-28 14:09:46,450 [DEBUG] InitialConditionStateSpaceTest:
                                rel_output_variance: 0.000
2017-08-28 14:09:46,450 [DEBUG] VARIABLE COMPARE: variables
                                'secondVar', 'secondVar' mismatch
2017-08-28 14:09:46,450 [DEBUG] VARIABLE COMPARE:
                                var=0.000, var_ref=11.05 (scalar)
2017-08-28 14:09:46,450 [DEBUG] VARIABLE COMPARE: expected relative
                                difference abs(-11.05) to be
                                <= tolerance 0.000 * abs_norm(var_ref)
                                = 0.000
2017-08-28 14:09:46,450 [INFO] AbstractStateSpaceTest: Test failed!
```

## A.2. Log of quasisteady_helicopter_velocity_sweep_noeulerangles test

```
                                ...
2017-09-04 11:12:26,852 [INFO] AbstractStateSpaceTest: Setup test
                               DeterminismStateSpaceTest for model
                               quasisteady_rotor_aerodynamics_model
2017-09-04 11:12:26,852 [INFO] AbstractStateSpaceTest: Copying time...
2017-09-04 11:12:26,852 [INFO] AbstractStateSpaceTest: Copying state...
2017-09-04 11:12:26,852 [INFO] AbstractStateSpaceTest: Gathering
                               input...
2017-09-04 11:12:26,852 [INFO] AbstractStateSpaceTest: Running test...
2017-09-04 11:12:26,852 [DEBUG] DeterminismStateSpaceTest: on current
                                state
2017-09-04 11:12:26,852 [DEBUG] DeterminismStateSpaceTest:
                                rel_input_variance: 0.000
2017-09-04 11:12:26,852 [DEBUG] DeterminismStateSpaceTest:
                                rel_output_variance: 0.000
2017-09-04 11:12:26,852 [DEBUG] DeterminismStateSpaceTest: running
                                determinism checks on
                                calculate_initial_condition
2017-09-04 11:12:26,852 [DEBUG] DeterminismStateSpaceTest: setting
                                input
2017-09-04 11:12:26,852 [DEBUG] DeterminismStateSpaceTest: calculating
                                initial condition
2017-09-04 11:12:26,853 [DEBUG] DeterminismStateSpaceTest: comparing
                                results...
2017-09-04 11:12:26,853 [DEBUG] DeterminismStateSpaceTest:
                                calculate_initial_condition passed
                                determinism test
                                ...
2017-09-04 11:12:26,856 [DEBUG] DeterminismStateSpaceTest: determinism
                                tests passed
2017-09-04 11:12:26,856 [INFO] AbstractStateSpaceTest: Test
                               successfully completed!
                               ...
2017-09-04 11:12:26,873 [INFO] AbstractStateSpaceTest: Setup test
                               DeterminismStateSpaceTest for model
                               quasisteady_rotor_model
2017-09-04 11:12:26,873 [INFO] AbstractStateSpaceTest: Copying time...
2017-09-04 11:12:26,873 [INFO] AbstractStateSpaceTest: Copying state...
```

```
2017-09-04 11:12:26,873 [INFO] AbstractStateSpaceTest: Gathering
                               input...
2017-09-04 11:12:26,873 [INFO] AbstractStateSpaceTest: Running test...
2017-09-04 11:12:26,873 [DEBUG] DeterminismStateSpaceTest: on current
                                state
2017-09-04 11:12:26,873 [DEBUG] DeterminismStateSpaceTest:
                                rel_input_variance: 0.1000E-07
2017-09-04 11:12:26,873 [DEBUG] DeterminismStateSpaceTest:
                                rel_output_variance: 0.1000E-03
2017-09-04 11:12:26,873 [DEBUG] DeterminismStateSpaceTest: running
                                determinism checks on
                                calculate_initial_condition
2017-09-04 11:12:26,868 [DEBUG] DeterminismStateSpaceTest: calculating
                                initial condition
2017-09-04 11:12:26,873 [DEBUG] DeterminismStateSpaceTest: setting
                                input
                                ...
2017-09-04 11:12:26,873 [DEBUG] DeterminismStateSpaceTest: comparing
                                results...
2017-09-04 11:12:26,873 [DEBUG] VARIABLE COMPARE: variables
                                'Mainrotor Moments',
                                'Mainrotor Moments' mismatch
2017-09-04 11:12:26,873 [DEBUG] VARIABLE COMPARE:
                                var=-0.8720E-08, var_ref=0.5697E-08
                                at index (1)
2017-09-04 11:12:26,874 [DEBUG] VARIABLE COMPARE: expected relative
                                difference abs(0.3055E+15) to be
                                <= tolerance 0.1000E-03
                                  * abs_norm(var_ref)
                                = 0.3055E+11
2017-09-04 11:12:26,874 [DEBUG] VARIABLE COMPARE:
                                var=0.7458E-08, var_ref=0.8762E-08
                                at index (2)
2017-09-04 11:12:26,874 [DEBUG] VARIABLE COMPARE: expected relative
                                difference abs(0.9219E+14) to be
                                <= tolerance 0.1000E-03
                                  * abs_norm(var_ref)
                                = 0.3055E+11
2017-09-04 11:12:26,874 [DEBUG] VARIABLE COMPARE: variables
                                'Mainrotor flapping motion 0+1/rev',
                                'Mainrotor flapping motion 0+1/rev'
                                mismatch
```

```
2017-09-04 11:12:26,874 [DEBUG] VARIABLE COMPARE:
                                var=-0.8720E-08, var_ref=0.5697E-08
                                at index (1)
2017-09-04 11:12:26,874 [DEBUG] VARIABLE COMPARE: expected relative
                                difference abs(-0.9989E+08) to be
                                <= tolerance 0.1000E-03
                                  * abs_norm(var_ref)
                                = 0.1348E+06
                                ...
```