Static Analysis of Physically Constrained Software Systems Using SMT

Maximilian Levi Heisinger

Institute for Formal Models and Verification

Supervisors: Martina Seidl, Armin Biere

Bachelor's degree thesis for the acquisition of an academic degree
Bachelor of Science (BSc)

Sunday 7th October, 2018
Johannes Kepler University
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at | fmv.jku.at

I hereby declare under oath that the submitted Bachelors Thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited.

Contents

1. Introduction

1.1. Motivation

Correctness of source code has always been a problem in software development. Because the cost of correcting errors made during software construction and found during system or in-field testing can increase by a factor of 10 or 15 [8, p. 29], it is desireable to detect defects and problematic corner cases as early as possible.

When narrowing down the problem domain, a multitude of failure classes can be eliminated using handcrafted tests, like serialising a specified type of data structure into a clearly defined output stream or single transitions in a finite state machine. Others, like control code for remote controlled and autonomous vehicles, have a more open problem definition. To still be able to verify such systems in a timely manner, one option is to develop automated tools to check for dangerous corner cases in unknown source code (black-box testing) and to sort out all possible inputs to a software system to find potential defects. Testing is, however, only suited to a limited extent for this task, as famously stated by Edsger W. Dijkstra:

> Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence [3].

The analysis of options and the development of a prototype for such a loosely specified system is the goal of this thesis, with special focus on the practical implementation on the Versatile Nature Exploration Rover (VERNER) platform.[1] This platform is a custom-built, six wheeled exploratory vehicle, designed to be extendable and capable of carrying varying sensors read by multiple operators at the same time. Exploring new ideas for controlling code should not endanger the entire device. Instead, the code should be proven to be safe before running full system tests on the device.

Software produced in this work is published under the MIT license on github.com/HARPTech, in the repositories RVerify and RTest.

[1] www.harptech.eu

1.2. Problem Statement

Ensuring high quality and correctness of any steering strategy while constructing new controlling software called Regulation Kernels (RKs) for the VERNER platform is challenging during development. Certain corner cases can be easily overseen or the software could crash during use because of a calculation error. The most critical failure class manifests in a technically working RK, which then fails with input parameters that produce out of range outputs. The cost of finding such failures during system testing is time consuming and potentially expensive, therefore such errors should be detected on the development machine before commencing the test on real hardware. Because the two main input variables every RK has to process are of type `int16`, the problem space stretches over $2^{16} \cdot 2^{16}$ inputs. This is too big to be searched exhaustively using regular testing methods over the whole input range.

The VERNER platform itself consists of six wheels, four of which can be individually controlled in speed and steering direction. The input parameters for the direct motor controls are in the range of $[-255, 255]$ (for speed v) and $[0, 255]$ (for steering s). The remote control abstracts the individual wheels by just sending two values for the steering direction S and the speed V, each in the range of $[-2^{15}, 2^{15}-1] = [-32768, 32767]$. These two values represent the two primary variables mentioned in the paragraph above. An RK can therefore be seen like a function, mapping the input vector (S, V) to the output vectors (s, v) for every wheel.

1.3. Aim of the Work

This work aims to create a software tool called *RVerify*, based on satisfiability modulo theories (SMT) solvers to analyse the given RK written in the programming language Python. SMT is a decision problem around finding valid assignments for variables in first-order logic formulas over theories like Int, Real, Bit-Vectors, etc. It has received much research attention in the computer science community and was implemented in a wide variety of solvers, competing against others and ranked by their performance in benchmarks like the SMT-LIB benchmark set [12]. The standard input format is called SMT-LIB2 [1].

The developed tool is able to automatically parse RKs, to generate SMT-LIB2-compatible code as a (printable) intermediate step and then use that representation to verify the intended behaviour of the RK against a given specification. The conversion between the imperative environment of Python into SMT-LIB2 supports common control structures, trigonometry and small utility functions used in such environments. The most critical failure cases to detect consist of:

- Motors being on while servos point against other servos, effectively jamming the wheels into each other and trying to move in opposite directions.

- Servos getting an invalid steer-setting s with $s > 255$ or $s < 0$, which would cause a crash of the RK. Such a crashed kernel would not send any more commands to the rest of the system, taking control from the operator and making the device unresponsive.
- Motors receiving an invalid speed-setting v with $v > 255$ or $v < -255$, which would also cause a crash of the RK.

In addition to finding these defects, RVerify is also able to identify internal logic inconsistencies in the source code, like multiple assignments to the same variable in the same scope. If it finds any, they are printed in a well-readable form. To display the mentioned programming defects, another tool is provided to present the collected information in a visual manner, called *RTest*.

1.4. Approaches for Formal Verification of Source Code

As has been stated previously, software tests often specify desired outputs (or ranges for them to fit into) for fixed inputs. This makes them a very well suited tool during software construction and to check correctness over the lifetime of an application. This approach has issues though. One of them is that testers can never be certain that all possible failure cases have been covered. Even very diligent testing cannot mitigate this problem because new corner cases might be introduced with every change.

What differentiates testing from verification is that tests can only find deficiencies in programs but never validate their correctness. To achieve that level of confidence, software needs to be checked to work for all possible inputs being run through all execution code paths [10]. The following section describes multiple approaches for such a task.

For software development, some workflows around direct formal verification of source code revolve around manually translating imperative source code from programming languages like Python or C++ into some kind of descriptive representation of a theorem, which can then be automatically proved (like with KeYmaera X [5]). This representation could be completely independent (e.g. describing what a program does using a language like SMT or Coq [14]) and, additionally, could be generated in multiple steps with different intermediate representations [7].

Likewise, such complementary languages can also be given side-by-side to the source code of the actual implementation, providing meta-information about how the code should behave if it is correct (compare to the Java Modelling Language [2]). Because of the high variance in languages and grammars, only a few programming languages have yet received automatic translators, like LLVM IR [4, 6].

New automatic translators have to translate their source language into a target like SMT and can choose to utilise projects like Boogie [7], if the problem at hand needs

to have deeper language understanding. This thesis focuses on the development of a direct translation from Python into SMT formulas without such intermediary steps, to speed up the translation and to enhance the verification workflow for the RK developer.

1.5. Approach

To archieve the goals stated above, multiple levels of processing the given source code were implemented. The list below describes the main challenges of this work, each of which resulting in an independently usable tool that can be combined with other software.

1. Conversion of the Python source code into SMT-LIB2 representation.
2. Calculation of approximatory lookup-tables for tan and arctan to be used in the SMT-solver.
3. Checking the generated SMT formulas for inconsistencies using parallelized delta debugging [15] techniques.
4. Verifying the generated SMT formulas against the specified behaviour to check for defects.

1.6. Structure of the Work

This work is structured in a similar way as the milestones were completed to guide through the important pieces of the implementation and show how to use each component individually.

First, the concepts behind the VERNER platform are explained in Section 2.1, including a visualisation of the steering subsystem. A requirements analysis for RVerify (verification tool developed in this work) is given in Section 2.2 and the list of supported Python language features in Section 2.3.

In the next section, we discuss the generation of the SMT code from a RK together with the challenges encountered during this translation: The trigonometric lookup tables in Subsection 2.4.1, python parsing in Subsection 2.4.2, and finally the SMT generation itself in Subsection 2.4.3.

Having the parsing and translation phases complete, we check for defects in Section 2.5. Afterwards, failure states are analysed and the RK is compared against the specified behaviour in Section 2.6.

In the last chapter, we test the developed RVerify tool against errors encountered in the past during development of the current RK to check if these defects could have been prevented using this new tool. The first of these defects is a crash bug stemming from a wrong calculation in Subsection 3.1.1 and the second one in Subsection 3.1.2 was caused by an invalid negation. Finally yet importantly, it is discussed how RVerify could handle future RK defects and how it could be improved further.

2. Architecture

2.1. Concepts

The primary concept this work builds upon is the separation of a robotic platform into the following interchangeable parts:

- Core software, which passes commands between the other parts (*RMaster*).
- Supplementary tooling for controlling the whole system through a dedicated GUI (*RController*).
- Controlling actuators by issuing electronic signals (*RBreakout*).
- Code for interpreting the steering commands into direct motor controls (*Regulation Kernel*).

This thesis focuses on the *Regulation Kernel*, a visualisation of which is given in Figure 2.1 on page 11. The rest of the software stack has been designed in modules and specified to minimise the number of corner cases, which makes it well testable using automated tests. The RK on the other hand makes such testing harder to do, because it has no clear relation between its inputs and outputs. The problem is not as clearly definable as with the remaining stack, as there can be many ways of controlling a rover such as VERNER. Wheels can be parallel or on circular paths, speeds can be proportional to the individual angles of the wheels, etc. Even though such programs can be very short, corner cases could be missed, causing expensive damage to hardware.

2.2. Requirements Analysis

2.2.1. Goals of the Software

The software to be developed should be able to process generic Regulation Kernel code from the VERNER Rover Platform, which is typically written in Python. Through processing the source code of such a program, the following pieces of information have to be gathered:

- Will running the given RK have a vector of inputs (S, V) which produces a result contradicting the specification?

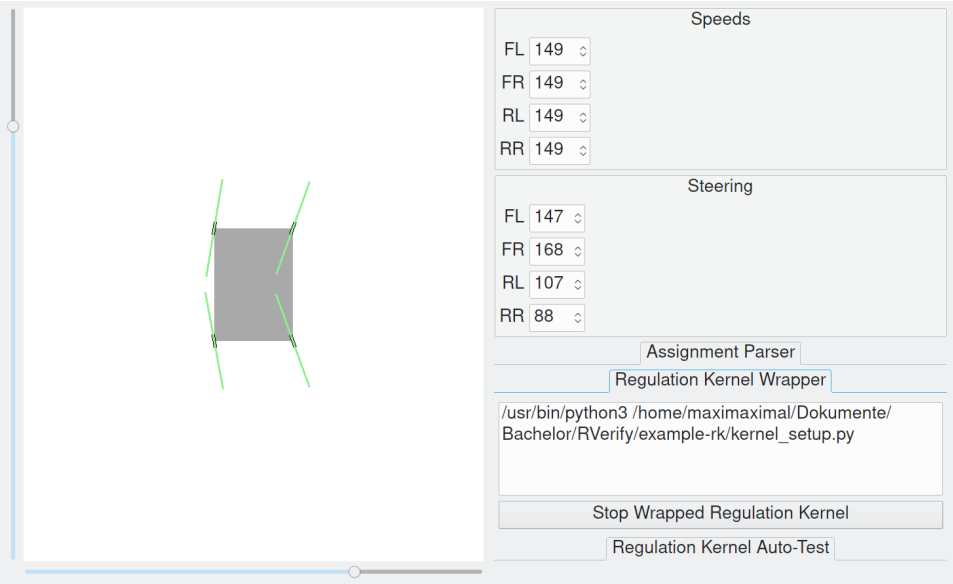


Figure 2.1.: Visualisation of the VERNER platform in RTest (visualisation tool for RK-code), running the Optimised Regulation Kernel as given in Appendix A.1

- If a possible error has been found, which failure case (which result) was reached and what input vector (S, V) caused that error?

2.2.2. Available Time and Computing Resources

Because this tool will be executed often on a development machine, it needs to give results in $< 1\,\mathrm{s}$ for typical inputs. An input can be regarded as typical if it controls all major aspects of the hardware platform similarly to the default implementation called *Optimised-Regulation-Kernel* (see Appendix A.1). In lines of code, this would be equivalent to 250 lines total.

The amount of resources needed to stay inside that time frame is limited by the processing power of a 2018 high level laptop. In this context, this means a machine with 4 cores at 3 GHz each. To still be able to configure the precision and therefore the runtime of the tool, a precision parameter is introduced to scale the precision of the trigonometric lookup tables.

2.2.3. Definition of Success

The project is successful, when statements about the working state of a given piece of code can be proven, i.e. deviations from standard inputs as given in the reference implementation of a RK (see Appendix A.1) are detected successfully. Full verification

of an RK is not required however, as the focus of this work is on giving results during the development process, not on complete system verification of an entire rover. Completely different inputs also have to be analysed (with reported errors during processing) and if nothing went wrong in the processing stage, the result must still be reliable.

The supported language constructs from the inputs do not have to include all constructs the language supports. Instead, a minimal subset capable of implementing behaviours comparable to the default implementation is enough for this tool to be useful. More advanced language constructs can be added at a later stage though, so the general architecture should either be small enough to be replaced easily or allow these expansions.

2.3. Supported Language Features

Through having a problem statement with an exact definition of what kind of input the analysis software is required to handle, the input code could be adapted. In order to lessen the burden of parsing a grammatically complex programming language like Python, changes were made to the old steering code to accomplish the goal of making it automatically analysable.

Although the resulting program has been tailored to be more easily parsable, it is easier to understand for the programmer and, even more importantly, uses less grammar features. This decreases the amount of syntax the parser must handle. The chosen subset of the language consists of:

- Certain objects, methods and attributes given from the environment (such as registry methods from the rest of the codebase or methods like `numpy.interp` or `math.tan`).
- If-Statements with optional else branches.
- An all-encompassing while-loop as main control structure.
- Variable declarations.
- Regular calculations with variables and numbers (`+`, `-`, `/`, `*`) with arbitrary nesting.

2.4. Convert Python into SMT

Before explaining the translation of Python source code to SMT, the challenges are discussed. The main challenge is the translation to *declarative* representation. Because

Python is an *imperative* programming language, assignments have a different meaning depending on *when* they are made. The program therefore has to be seen more like a directed graph of variable assignments than a sequence of instructions for a machine. The following paragraph illustrates how the Python source of one of the test cases of RVerify represented:

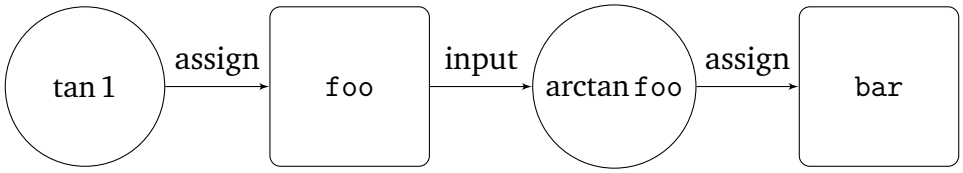
```
foo = math.tan(1)
bar = math.atan(foo)
```

The snippet above is automatically translated into the following SMT-LIB2 code, given in prefix notation:

```
(assert (= foo (tan 1 )))
(assert (= bar (atan foo )))
```

The `tan` and `atan` functions are hereby provided by a custom lookup table generated with a user-specified precision for the tan and arctan functions. More detail about this lookup table follows in the next subsection.

The directed graph for the snippet above looks like this:



This graph representation is already very close to the representation in SMT. It can be considered as a sequence of relations between variables, values and functions. To make such a translation automatically, every assignment in the python code has to be traced back to the top-most scope, where it is transformed into an `(assert ...)`, which is then interpreted by SMT-solvers similarly to the graph shown in the figure above. The solver primarily used in this thesis is Z3 [9] in the most recent version from its repository.[2] This decision was made because of its support for recursive functions and its good performance with real numbers and integers.

2.4.1. Generating a Trigonometric Lookup Table

Because the Optimised Regulation Kernel uses the tan and arctan functions, they also have to be interpreted correctly by the SMT solver. As can be seen in Figure 2.2 (p. 14), the tangent is especially difficult to calculate when a value at its extremes is looked for. To account for this highly varying function, it first gets reduced into only one instance (the middle one in the graph), which then gets split up into three parts to approximate the left and right extreme points together with the middle section.

[2] github.com/Z3Prover/z3

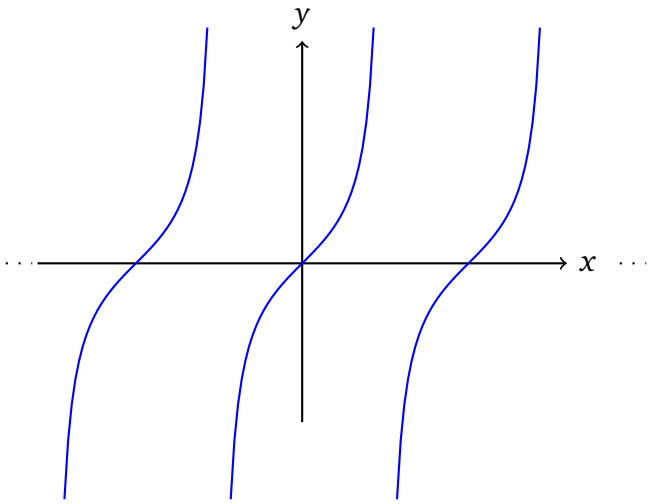


Figure 2.2.: Graph of the tan function.

To reduce the function to one instance, all queries into the lookup table with $x > \frac{\pi}{2}$ or $x < -\frac{\pi}{2}$ are given recursively to itself with an adjusted parameter x', which is set to either $x - \pi$ or $x + \pi$. After this first reduction, the main lookup table (a sequence of `(ite ... ... ...)` statements with a length defined by the used precision) comes into effect. The values to generate this lookup table are generated using `numpy` (a collection of performance-optimised algorithms and primitives for scientific computing in Python), the exact algorithm is provided in the Python source file `RVerify.smt_gen.trigonometric`, which can also be used independently. The approximation generated in a typical run (no precision adjustments) is illustrated in Figure 2.3 (p. 15) and shows an even distribution over the tan function.

Approximating arctan is done in a similar manner by switching inputs and outputs (x and y), so the function maps a x to given y values.

2.4.2. Parsing Python to an AST

On the way to process Python source code like in the example above, the code first has to be parsed into an Abstract Syntax Tree (AST). This is done using a Python package called `typed_ast` [13], a fork of the standard `ast` package. This fork allows the processing of PEP 484 Type Hints [11], but most importantly, its parsers are independent from the host Python version.

Using the `typed_ast` package, an unprocessed Abstract Syntax Tree (AST) is produced, as can be seen in Appendix A.1.4. This tree is then simplified to suit itself better to generating SMT-assertions. To simplify the tree, a Preorder traversal is done to identify all contained nodes. During that traversal, another tree is built, the *statement tree*. This tree contains the nodes gathered from the AST together with the line numbers of the

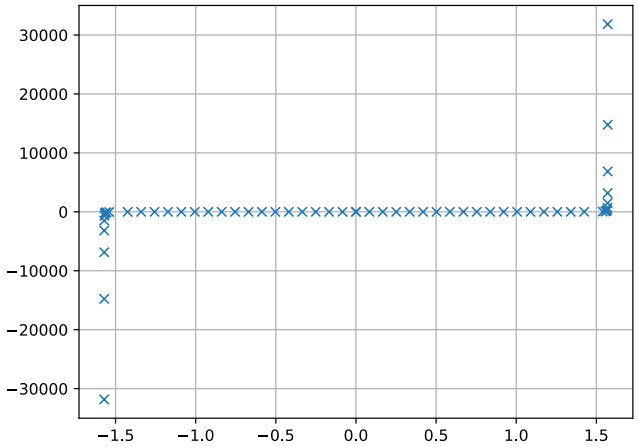


Figure 2.3.: Generated approximation of the tan function. Each mark is a value in the trigonometric lookup table.

original python script. When the statement tree is built completely, the 'information gathering phase' is complete and the SMT formulas can be generated.

2.4.3. Generating SMT

After the statement tree is build, a mixture of Inorder, Postorder, and Preorder traversals is done to gather all required information for each line of the resulting SMT formulas. All variables are collected in a separate data structure, the *variable store*, at the same time the statement tree is traversed. After the traversal is finished, all data points can be combined for the complete output in the following way:

1. The SMT Logic to be used
   (defined in `RVerify.parser.predefined.logic`, default logic is QF_UFNIRA)
2. Internal (predefined & global) variables and functions
   (defined in `RVerify.parser.predefined.internals`)
3. Computed lookup-tables for tan and arctan as seen in Subsection 2.4.1
4. Variable declarations
   (using `(declare-fun)`) from the *variable store*
5. The generated SMT-assertions from the *statement tree*
6. The predefined checks to be performed, as can be seen in Appendix A.2
   (defined in `RVerify.parser.predefined.checks`)

7. A call to `(check-sat)`
   (defined in `RVerify.parser.predefined.check_sat`)

The resulting SMT-code can be displayed using the `--print-smt` switch on RVerify. This output is then passed onwards to the checker.

2.5. Consistency Checking

The first phase of the checker analyses for internal consistency of the code by trying to find a satisfiable assignment, i.e. finding a solution consistent with the previously generated formulas for the generated SMT code. Inconsistent code contains multiple assignments to the same variable in the same scope (not separated by `if-else` statements), which results in invalid SMT formulas. Even though such statements are legal in Python, they are not supported by the parser to reduce its complexity.

If such errors are found somewhere in the code, a delta debugging process [15] is started. To debug the SMT formulas, they are split up into a list of strings, each entry having one line less than its respective predecessor in the list. This list is then passed to the solver again, which checks every entry for a possible satisfiable assignment. This process is parallelised over all available cores, so the result can be found quicker. The one with the highest number of contained lines of SMT formulas is selected, its lines are counted, and finally matched against the saved Python line-numbers in the statement tree. Once the fitting line number has been found, the offending Python lines can be printed to the user of RVerify.

If no errors were found, analysis for possible failure states of the RK can begin.

2.6. Analyse for Possible Failure States

For the second analysis phase, the predefined set of possible failure conditions (as explained in Section 1.3) is checked against the generated SMT code. These failure states are not specified as concrete values, but instead as the negation of the desired properties. The specification used can be seen in Appendix A.2.

As soon as the solver finds a model, an assignment has been found that violates the specified behaviour. It is then printed to the user in the form of a list of all variables and their assignments to be pasted into the input form of RTest, which then displays the defect in its visualisation. The following chapter gives examples of such detected defects.

3. Results & Discussion

3.1. Checking for Known Defect-Classes

To verify the practicability of RVerify, it is used to check defect-classes encountered during the development of the current RK as seen in Appendix A.1. The two main failures with the current kernel concerned the speed setting underflowing ($v < -255$) because of a faulty calculation and invalid wheel positions because of missing negations in their final assignments at the end of the current RK. An additional find after using RVerify for checking the RK running on the rover, was the invalid interpretation of several steering positions caused by upside-down positioned servo motors, leading to a software fix to correct their behaviour.

3.1.1. Speed Setting Underflow

To provoke this defect again, the assignment for the forward velocity v of all motors is changed to be $v = \frac{V}{128}$, with V being the forward velocity in the range of $[-32768, 32767]$. If the velocity drops to -32768, the maximum negative velocity, the kernel crashed, because it let v become -256, which is out of its bounds.

After running RVerify against the changed RK, it produces the following output, finding the defect. The offending lines have been highlighted and `grep` has been used to limit output to the most relevant bits.

```
python3 -m RVerify -f RVerify/example-rk/kernel_code.py -check | grep -E
 ↪  "_.*_|[A-Z]{7,}"

CODE SOUNDNESS PASSED, CHECK FAILURE STATES
FAILURE STATES DETECTED!
 _servo_rr_ = 34,
 _motor_fr_ = -256,
 _servo_fl_ = 160,
 _motor_rl_ = -256,
 _steer_direction_ = 25976,
 _motor_fl_ = -256,
 _forward_velocity_ = -32768,
 _motor_rr_ = -256,
 _servo_fr_ = 221,
 _servo_rl_ = 95,
```

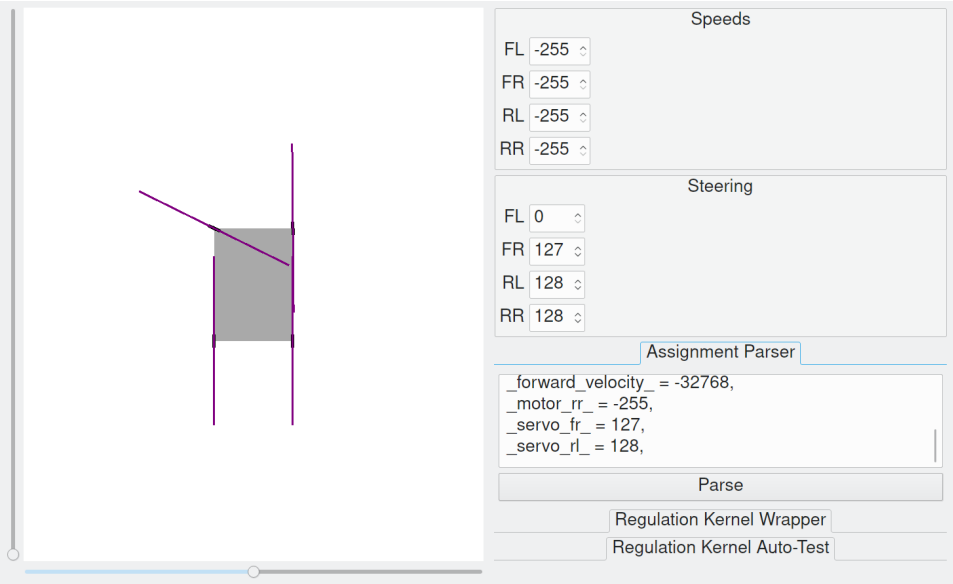


Figure 3.1.: Visualisation of the defect discussed in Subsection 3.1.2

3.1.2. Invalid Negations in Final Steering Assignments

The following defect occurred during small tweaks of the kernel. By mistake, one single minus was switched to a plus, which led to a wrong calculation for the final angle of a wheel (front left, in this case). By reproducing that issue and then running RVerify the now invalid RK, the following output is produced (again, using `grep` for brevity):

```
python3 -m RVerify -f RVerify/example-rk/kernel_code.py -check | grep -E
 ↪  "_.*_|[A-Z]{7,}"
CODE SOUNDNESS PASSED, CHECK FAILURE STATES
FAILURE STATES DETECTED!
 _servo_rr_ = 128,
 _motor_fr_ = -255,
 _servo_fl_ = -128,
 _motor_rl_ = -255,
 _steer_direction_ = -14,
 _motor_fl_ = -255,
 _forward_velocity_ = -32768,
 _motor_rr_ = -255,
 _servo_fr_ = 127,
 _servo_rl_ = 128,
```

This defect can again be read as individual assignments for angles and velocities, or the assignment parser of RTest can be used to visualise the behaviour, as seen in Figure 3.1.

3.2. Detecting Unknown Defects in the Future

The checks RVerify is based on, specifying allowed behaviour and detecting wrong results with previously unknown inputs over a search space with the size of 2^{32}, will be of great help during further development and in letting less experienced developers try themselves on writing RKs. If the need arises, its parsing capabilities can be expended and made more sophisticated. After the demonstrated detection of actual defects encountered during software construction, it is however clear, that the capabilities of SMT solvers can also be applied in environments such as this one, even without having more specialised software or specific expertise.

If new failure classes emerge out of added complexity, RVerify is able to either signal that it cannot parse the source code or that successfully checked the RK. Using that information, the verification capabilities can be further expanded during later development and used against new controlling software written for VERNER.

3.3. Lessons Learnt

The most important lesson learnt was to apply formal methods before building a physically constrained system. Through specifying dangerous corner cases, the RK code became much easier to reason about and to debug. Additionally, reducing the supported language features became the only way to fit the translator for Python into this thesis, without it being overrepresented.

Furthermore, the STDIN, STDERR and STDOUT (Posix) streams were very useful during development and testing of RTest, particularly during the first executions of old RK versions. The possibility to use dynamic linking and a stable ABI to expand the features of old kernels made creating such a generic visualisation possible. This area can also be expanded in the future, by building streaming capabilities into the remaining stack of VERNER, so that combining different components becomes easier.

3.4. Future Work

To further expand RVerify and its theoretical base, it would be advised to first improve the Python parser and the SMT translation. There has been much research in the field of writing such translators, for example Boogie [7] could be a big improvement once a frontend for Python is introduced.

The specification on how the rover should behave already works very good and could be expanded further, once more inputs and outputs are used in regulation kernels. One possible scenario would be the addition of motor utilisation to the steering code,

adding four additional `int8` inputs to the RK. Even though more complex kernels using that information do not exist yet, this could improve manoeuvrability in more difficult terrain drastically. The gyroscope is also a sensor, which provides possibly useful information for a RK. It is yet to be seen which data points will be used in RKs of the future, but RVerify can certainly be expanded to handle the additional inputs.

Bibliography

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2017.

[2] Lilian Burdy et al. 'An Overview of JML Tools and Applications'. In: *International Journal on Software Tools for Technology Transfer* 7.3 (June 2005), pp. 212–232. issn: 1433-2779, 1433-2787. doi: 10.1007/s10009-004-0167-4. url: http://link.springer.com/10.1007/s10009-004-0167-4 (visited on 07/13/2018).

[3] Edsger Wybe Dijkstra. 'The Humble Programmer, ACM Turing Lecture'. 1972.

[4] Michael Emmi et al. *SMACK: Software Verifier & Verification Toolchain*. url: https://smackers.github.io/ (visited on 07/24/2018).

[5] *KeYmaera X: An aXiomatic Tactical Theorem Prover for Hybrid Systems*. url: http://www.ls.cs.cmu.edu/KeYmaeraX/ (visited on 07/25/2018).

[6] Daniel Kroening. *CBMC*. url: https://www.cprover.org/cbmc/.

[7] Akash Lal et al. *Boogie: An Intermediate Verification Language*. url: https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/ (visited on 07/24/2018).

[8] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. 2nd edition. Redmond, Wash: Microsoft Press, June 19, 2004. 960 pp. isbn: 978-0-7356-1967-8.

[9] Leonardo de Moura and Nikolaj Bjørner. 'Z3: An Efficient SMT Solver'. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Mar. 29, 2008, pp. 337–340. isbn: 978-3-540-78799-0 978-3-540-78800-3. url: https://link.springer.com/chapter/10.1007/978-3-540-78800-3_24 (visited on 07/25/2018).

[10] Glenford Myers, Tom Badgett, and Corey Sandler. *The Art of Software Testing*. 3. ed. Hoboken, NJ: Wiley, 2012. isbn: 978-1-118-03196-4 1-118-03196-2.

[11] *PEP 484 – Type Hints*. url: https://www.python.org/dev/peps/pep-0484/ (visited on 07/24/2018).

[12] *SMT-LIB Benchmarks*. url: http://smtlib.cs.uiowa.edu/benchmarks.shtml (visited on 07/24/2018).

[13] *Typed_ast: Modified Fork of CPython's Ast Module That Parses '# Type:' Comments*. July 12, 2018. url: https://github.com/python/typed_ast (visited on 07/24/2018).

[14] *What Is Coq? | The Coq Proof Assistant*. url: https://coq.inria.fr/about-coq (visited on 07/13/2018).

[15] Andreas Zeller and Ralf Hildebrandt. 'Simplifying and Isolating Failure-Inducing Input'. In: *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 183–200.

Appendices

A. Source-Code and Examples

A.1. Optimised-Regulation-Kernel

This RK is the main regulation kernel used as a basis for developing new regulation kernels. It computes the correct wheel positions to be on imagined circular lines on the ground, so that no wheel is positioned against another one. To execute this regulation kernel directly, RBase (shared software libraries of VERNER) needs to be installed on the system (either locally in `/usr/local` or by the package manager in `/usr`). To directly test a regulation kernel, RTest embeds the required Python modules and can be directly installed on any Debian-based distribution. The latest release of the RTest software can be downloaded from the *Releases* page[3] on its GitHub repository.

A.1.1. Main Code

The following code (after the `# RVERIFY_BEGIN` comment) is checked by RVerify.

```
1  import RRegistry as RR
2  import RSupport as RS
3  import kernel_globals as g
4  from numpy import interp
5  import math
6  rsupport = g.rsupport
7  registry = g.registry
8  # RVERIFY_BEGIN
9
10 d_fl = 33
11 d_fr = 33
12 d_rl = 31
13 d_rr = 31
14 G = 30
15
16 while(True):
17     rsupport.service()
18
19     steer_direction = registry.getInt16(RR.Int16_MVMT_STEER_DIRECTION)
20     vel = registry.getInt16(RR.Int16_MVMT_FORWARD_VELOCITY)
21
22     forward_velocity = interp(vel, [-32768, 32767], [-255, 255])
```

[3] github.com/HARPTech/RTest/releases

```
23
24      motor_fl = int(forward_velocity)
25      motor_fr = int(forward_velocity)
26      motor_rl = int(forward_velocity)
27      motor_rr = int(forward_velocity)
28
29      beta_sub = (interp(steer_direction, [-32768, 32767], [-72, 72])) / 100
30      beta = (math.pi / 2) - beta_sub
31
32      A = math.tan(beta) * G
33
34      beta_fl = math.atan((A + d_fl) / G)
35      beta_fr = math.atan((A - d_fr) / G)
36
37      beta_rl = -math.atan((A + d_rl) / G)
38      beta_rr = -math.atan((A - d_rr) / G)
39
40      multiplicators_1 = (beta_fl * 2) / math.pi
41      multiplicators_2 = (beta_fr * 2) / math.pi
42      multiplicators_3 = (beta_rl * 2) / math.pi
43      multiplicators_4 = (beta_rr * 2) / math.pi
44
45      if steer_direction >= 0:
46          values_1 = 128 + (1 - multiplicators_1) * 128
47          values_2 = 128 + (1 - multiplicators_2) * 128
48          values_3 = 128 - (1 + multiplicators_3) * 128
49          values_4 = 128 - (1 + multiplicators_4) * 128
50      else:
51          values_1 = 128 - (1 + multiplicators_1) * 128
52          values_2 = 128 - (1 + multiplicators_2) * 128
53          values_3 = 128 + (1 - multiplicators_3) * 128
54          values_4 = 128 + (1 - multiplicators_4) * 128
55
56      # Assign the calculated variables into the registry.
57      registry.setInt16(RR.Int16_MVMT_MOTOR_PWM_FL, int(motor_fl))
58      registry.setInt16(RR.Int16_MVMT_MOTOR_PWM_FR, int(motor_fr))
59      registry.setInt16(RR.Int16_MVMT_MOTOR_PWM_RL, int(motor_rl))
60      registry.setInt16(RR.Int16_MVMT_MOTOR_PWM_RR, int(motor_rr))
61
62      registry.setUint8(RR.Uint8_MVMT_SERVO_FL_POSITION, int(values_1))
63      registry.setUint8(RR.Uint8_MVMT_SERVO_FR_POSITION, int(values_2))
64      registry.setUint8(RR.Uint8_MVMT_SERVO_RL_POSITION, int(values_3))
65      registry.setUint8(RR.Uint8_MVMT_SERVO_RR_POSITION, int(values_4))
```

A.1.2. Setup Code

The source code in the following listing initialises the connection between RMaster (central VERNER software component) and RVerify and sets up the local RRegistry (central list of named properties of the rover, like velocity and steering direction) mirror.

```
import sys

sys.path.append("/usr/local/share/python3/")

import RRegistry as RR
import RSupport as RS

import optimized-regulation-kernel-globals as g

print("Created handle! Trying to connect")

# Connect to default path.
status = g.rsupport.connect("/tmp/lrt_pipe_path.pipe")
if status != RS.RSupportStatus_Ok:
    print("Error while connecting: " + RS.rsupport_status_msg(status))

# After connecting, options can be set.
g.rsupport.service()

# The frequency should be regulated automatically.
g.rsupport.setOption(RS.RSupportOption_AutoFrequency, True)
# After each loop, the movement state should be
# forwarded to the hardware and the Arduino.
g.rsupport.setOption(RS.RSupportOption_AutoMovementBurst, True)

# Receive the registry instance.
registry = g.rsupport.registry()
g.registry = registry

# Subscribe to inputs.
g.rsupport.subscribe(RR.Type_Int16, RR.Int16_MVMT_STEER_DIRECTION)
g.rsupport.subscribe(RR.Type_Int16, RR.Int16_MVMT_FORWARD_VELOCITY)

# Run the kernel code.
import optimized-regulation-kernel-code
```

A.1.3. Global Variables

This file serves as a bridge between the setup and the looping code, containing global variables.

```
import RRegistry as RR
import RSupport as RS

rsupport = RS.RSupport()
registry = None
```

A.1.4. AST-Dump of `kernel_code.py`

This dump has been created using the `--dump-ast` switch to RVerify. The contents are minified to the most relevant bits, namely the dump of a function call, assignments and an `if`, `else` combination. The most relevant lines are highlighted.

```
Module(
  body=[
    Assign(
      targets=[Name(id='d_fl', ctx=Store())],
      value=Num(n=33),
      type_comment=None,
    ),
    ...
    While(
      test=NameConstant(value=True),
      body=[
        Expr(
          value=Call(
            func=Attribute(
              value=Name(id='rsupport', ctx=Load()),
              attr='service',
              ctx=Load(),
            ),
            args=[],
            keywords=[],
          ),
        ),
        Assign(
          targets=[Name(id='steer_direction', ctx=Store())],
          value=Call(
            func=Attribute(
              value=Name(id='registry', ctx=Load()),
              attr='getInt16',
              ctx=Load(),
            ),
            args=[
              Attribute(
                value=Name(id='RR', ctx=Load()),
                attr='Int16_MVMT_STEER_DIRECTION',
                ctx=Load(),
              ),
            ],
            keywords=[],
          ),
          type_comment=None,
        ),
        ...
        If(
          test=Compare(
            left=Name(id='steer_direction', ctx=Load()),
            ops=[GtE()],
```

```
            comparators=[Num(n=0)],
          ),
          body=[
            Assign(
              targets=[Name(id='values_1', ctx=Store())],
              value=BinOp(
                left=Num(n=128),
                op=Add(),
                right=BinOp(
                  left=BinOp(
                    left=Num(n=1),
                    op=Sub(),
                    right=Name(id='multiplicators_1', ctx=Load()),
                  ),
                  op=Mult(),
                  right=Num(n=128),
                ),
              ),
              type_comment=None,
          ...
          orelse=[
            Assign(
              targets=[Name(id='values_1', ctx=Store())],
              value=BinOp(
                left=Num(n=128),
                op=Sub(),
                right=BinOp(
                  left=BinOp(
                    left=Num(n=1),
                    op=Add(),
                    right=Name(id='multiplicators_1', ctx=Load()),
                  ),
                  op=Mult(),
                  right=Num(n=128),
                ),
              ),
              type_comment=None,
              ...
            ],
            keywords=[],
          ),
        ),
      ],
      orelse=[],
    ),
  ],
  type_ignores=[],
)
```

A.2. Specification used in RVerify

This specification defines allowed behaviour for the rover and its RK. If the regulation kernel would produce any output that is outside this spectrum, it would be faulty and should be reworked.

```
(assert (not (and

(>= _motor_fl_ (- 255))
(<= _motor_fl_ 255)
(>= _motor_fr_ (- 255))
(<= _motor_fr_ 255)
(>= _motor_rl_ (- 255))
(<= _motor_rl_ 255)
(>= _motor_rr_ (- 255))
(<= _motor_rr_ 255)

(=> (or (> _steer_direction_ 1) (< _steer_direction_ (- 1)))
    (and
     (>= _servo_fl_ 0)
     (<= _servo_fl_ 255)
     (>= _servo_fr_ 0)
     (<= _servo_fr_ 255)
     (>= _servo_rl_ 0)
     (<= _servo_rl_ 255)
     (>= _servo_rr_ 0)
     (<= _servo_rr_ 255)
     ))

;; Combined Properties
;; ----------

(=> (and (> _servo_fl_ 128) (< _servo_fr_ 128)) (or (and (>= _motor_fl_ 0) (<=
 ↪  _motor_fr_ 0)) (and (<= _motor_fl_ 0) (>= _motor_fr_ 0))))
(=> (and (> _servo_fr_ 128) (< _servo_fl_ 128)) (or (and (>= _motor_fl_ 0) (<=
 ↪  _motor_fr_ 0)) (and (<= _motor_fl_ 0) (>= _motor_fr_ 0))))
(=> (and (> _servo_rl_ 128) (< _servo_rr_ 128)) (or (and (>= _motor_rl_ 0) (<=
 ↪  _motor_rr_ 0)) (and (<= _motor_rl_ 0) (>= _motor_rr_ 0))))
(=> (and (> _servo_rr_ 128) (< _servo_rl_ 128)) (or (and (>= _motor_rl_ 0) (<=
 ↪  _motor_rr_ 0)) (and (<= _motor_rl_ 0) (>= _motor_rr_ 0))))
(=> (and (> _servo_fl_ 128) (> _servo_fr_ 128)) (or (and (>= _motor_fl_ 0) (>=
 ↪  _motor_fr_ 0)) (and (<= _motor_fl_ 0) (<= _motor_fr_ 0))))
(=> (and (> _servo_rl_ 128) (> _servo_rr_ 128)) (or (and (>= _motor_rl_ 0) (>=
 ↪  _motor_rr_ 0)) (and (<= _motor_rl_ 0) (<= _motor_rr_ 0))))

)))
```

B. Acronyms

AST Abstract Syntax Tree 2, 13

C++ A compiled, multi-paradigm programming language originally based on C 6

RK Regulation Kernel 4, 5, 7–9, 15–18, 20, 25, 27

SMT Satisfiability modulo theories 2, 5–7, 11–15, 18

VERNER Versatile Nature Exploration Rover 4, 5, 7, 9, 10, 18, 20, 21, 27

C. Glossary

Python An interpreted programming language with very concise syntax. It is known to be easy to learn for beginners and to have a very rich grammar 2, 5–7, 10, 11, 13, 15, 18, 20, 27

RBase Shared software libraries of VERNER. Includes the code of RRegistry and Python modules for RKs 20

RMaster Central orchestrating component of the VERNER software stack, running on the rover 21

RRegistry Central list of named properties of the rover, categorised by their data type and purpose, like velocity and steering direction 21, 27

RTest Visualisation tool for RK-code. Accessible via github.com/HARPTech/RTest 9, 15, 17, 20

RVerify The tool developed in this work. Accessible via github.com/HARPTech/RVerify 3, 7, 8, 11, 12, 15–18, 20, 21, 23, 25