DD2452 Formal Methods
Lab 1: Deductive Verification of an ABS Controller

Jonas Haglund

September 12, 2019

1 Introduction

In this lab, you will use the WP plugin of Frama-C to formally verify a C program controlling an anti-lock braking system (ABS) of a car. ABS is a safety system used in cars, motorcycles, trucks and airplanes to prevent the wheels from locking under braking, which would otherwise make the vehicle difficult to control. The purpose of ABS is to optimize braking distance and retain steerability and stability of the vehicle.

Sections 2 and 3 describe the algorithm implemented in the C program. Section 4 gives a brief overview of the structure of the C program. Section 5 describes your tasks, and Section 6 describes the grading. Section 7 gives some tips that are worthwhile to have in mind when verifying the C program.

2 Organization and Workings of an ABS system

The braking system in a car is organized roughly as a chain as follows:

1. The braking pedal is connected to the master cylinder.
2. The master cylinder is connected to the hydraulic modulator.
3. The hydraulic modulator is connected to the brake circuits (there are two brake circuits for safety reasons if one would fail: one for the front wheels and one for the rear wheels).
4. The brake circuits are connected to the brake calipers.

The master cylinder converts the mechanical force on the braking pedal to a proportional hydraulic force, by forcing brake fluid into the hydraulic modulator.

The hydraulic modulator increases, holds or decreases the brake pressure by controlling the amount of brake fluid in the brake circuits by opening and closing the inlet and outlet valves. When the inlet valve is open/closed, brake fluid is passed/not passed into the brake circuit. When the outlet valve is open/closed, brake fluid is pumped/not pumped out of the brake circuit into the master cylinder reservior. Hence, the hydraulic modulator controls the brake pressure as follows:

- Increasing brake pressure: The inlet vavle is open and the outlet vavle is closed. This results in the driver controlling the brake pressure.
- Holding brake pressure: Both valves are closed, resulting in constant brake pressure, irrespectively of how hard the driver pushes the braking pedal.
- Decreasing brake pressure: The inlet vavle is closed and the outlet vavle is open, resulting in the braking pedal having no influence on the braking of the car.

The brake fluid in the brake circuits causes a hydraulic force to press the brake pads in the calipers against the brake disk.

The ABS mechanism is implemented by the electronic control unit (ECU). In order for the ECU to control the brake pressure, the ECU is connected to the hydraulic modulator and continously sends a control signal to the hydraulic modulator of what the state of the valves shall be. To compute the control signal the ECU makes use of the following three sensors:

- Braking pedal sensor: Records the mechanical force on (or the displacement of) the braking pedal.
- Wheel angular velocity sensors: Records the rotational velocity of the wheels.
- Vehicle acceleration sensor: Records the change in velocity of the vehicle.

The ECU minimizes the braking distance by means of a concept called wheel slip ratio, defined as follows. A wheel is freely rolling when no braking nor driving torque is applied to the wheel. When a wheel is freely rolling its angular velocity is $\omega_0 = v/R$ where v is the velocity of the car and R is the radius of the wheels. When a braking torque is applied to a wheel, the wheel starts to slip and its angular velocity ω_t decreases: $\omega_t < \omega_0$. The wheel slip ratio $(\omega_0 - \omega_t)/\omega_0$ gives an indication of the amount of wheel slip. If the wheel is locked, $\omega_t = 0$, then the wheel slip ratio is at maximum one. If the wheel is slipping just a bit, $\omega_t \approx \omega_0$ and the wheel slip ratio is close to zero. The relationship between wheel slip ratio and the optimal braking distance depends on the velocity of the vehicle and surface conditions (wet or dry). Usually the optimal wheel slip ratio S_{ref} ranges between 0.10 and 0.60.

3 An Intelligent ABS Algorithm

The goal of the ABS controller in the ECU is, under braking, to keep the actual wheel slip $S = (\omega_0 - \omega_t)/\omega_0$ as close to S_{ref} as possible (where $S_{ref} \in [0.10, 0.60]$ depending on surface conditions and velocity). For simplification the ABS algorithm controls the braking of only one wheel. S_{ref} is defined to be 0.15: $S_{ref} = 0.15$.

Input to the algorithm is:

- Braking pedal pushing indicator: Instead of the braking pedal sensor mentioned above that records mechanical force on or displacement of the braking pedal, a "braking pedal pushing indicator" is used which returns 1 if the braking pedal is pushed and 0 otherwise.

- Wheel angular velocity sensor: Records the rotational speed ω_t of the wheel in radians/s.
- Vehicle acceleration sensor: Records the acceleration av of the car in m/s^2.

Output of the algorithm is a control signal u_c to the hydraulic modulator. The hydraulic modulator reacts to the control signal u_c as follows:

- $u_c > 0.15$: The hydraulic modulator increases the brake pressure by opening the inlet valve and closing the outlet valve.
- $u_c < -0.15$: The hydraulic modulator decreases the brake pressure by closing the inlet valve and opening the outlet valve.
- Otherwise: The states of the inlet and outlet valves are unchanged.

It can be assumed that the ABS algorithm is invoked once every 20 ms (a timer can be configured to raise one interrupt every 20 ms, and at each interrupt the ABS algorithm is invoked). The ABS algorithm considered in this lab is a combination of PID (Proportional, Integral, Derivative) and intelligent control.

PID is a common control method based on calculating the error of previous control signals (in this context, the difference between the actual wheel slip ratio S and the optimal wheel slip ratio S_{ref}). The proportional, integral and derivative components are used, respectively, to proportionally correct the control signal, correct accumulated errors over time, and correct the current error.

The intelligent control part is based on fuzzy logic, a successful control method suitable for handling non-linear behaviors, braking in this context. Fuzzy logic works basically as follows. Given inputs $i = (i_1, ..., i_n)$, a set of rules is applied on i. Each rule R_m has the form: **if** $P_m(i)$ **then** $\mu_m(i)$. The meaning of such a rule is: if $P_m(i)$ is true, then compute the membership function μ_m applied on i: $\mu_m(i)$. $0 \leq \mu_m(i) \leq 1$ gives an indication of the degree to which i is in a certain set. $\mu_m(i) = 0$ means that i is not in the set, $\mu_m(i) = 1$ means that i is definitively in the set, and $0 < \mu_m(i) < 0$ means that i is in the set to a certain degree. In this context $i = (e, e')$, where $e = S - S_{ref}$ is the error and e' is rate of change of the error, and one of the sets considered is whether e and e' are zero. Those $\mu_m(i)$ that have been computed, depending on whether the corresponding $P_m(i)$ are true, are then used to compute the final output (the control signal u_c in this context).

The PID part of the algorithm computes $e = S - S_{ref}$ and $e' = (S - S_{previous})/\Delta t$, where S is the most recent computation of the actual wheel slip, $S_{previous}$ is the wheel slip computed by the previous invocation of the algorithm, and $\Delta t = 0.020$ s. Recall $\omega_0 = v/R$ (where v is the speed

of the vehicle and R is the radius of the wheels) and $S = (\omega_0 - \omega_t)/\omega_0$. These two equations give $S = (v/R - \omega_t)/(v/R)$. Multiplying the right side by $R/R = 1$ gives $S = (v - \omega_t R)/v$. The velocity of the vehicle can be computed by integrating over the acceleration a over time and adding the velocity $v_0 = \omega_t R$ of the vehicle when the braking started (at which time ω_t is the angular velocity of a freely rolling wheel): $v = \int a \; dt + v_0$.

The fuzzy logic part of the algorithm computes the final control signal u_c to the hydraulic modulator by means of e, e', and the following set of rules:

$$\mu_{NB}(x) = \begin{cases} 1 & \text{if } x \leq -1 \\ -2x - 1 & \text{if } -1 < x < -0.5 \\ 0 & \text{if } -0.5 \leq x \end{cases}$$

$$\mu_{NM}(x) = \begin{cases} 0 & \text{if } x \leq -1 \\ 2x + 2 & \text{if } -1 < x \leq -0.5 \\ -4x - 1 & \text{if } -0.5 < x < -0.25 \\ 0 & \text{if } -0.25 \leq x \end{cases}$$

$$\mu_{NS}(x) = \begin{cases} 0 & \text{if } x \leq -0.5 \\ 4x + 2 & \text{if } -0.5 < x \leq -0.25 \\ -4x & \text{if } -0.25 < x < 0 \\ 0 & \text{if } 0 \leq x \end{cases}$$

$$\mu_{ZE}(x) = \begin{cases} 0 & \text{if } x \leq -0.25 \\ 4x + 1 & \text{if } -0.25 < x \leq 0 \\ -4x + 1 & \text{if } 0 < x < 0.25 \\ 0 & \text{if } 0.25 \leq x \end{cases}$$

$$\mu_{PS}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 4x & \text{if } 0 < x \leq 0.25 \\ -4x + 2 & \text{if } 0.25 < x < 0.5 \\ 0 & \text{if } 0.5 \leq x \end{cases}$$

$$\mu_{PM}(x) = \begin{cases} 0 & \text{if } x \leq 0.25 \\ 4x - 1 & \text{if } 0.25 < x \leq 0.5 \\ -2x + 2 & \text{if } 0.5 < x < 1 \\ 0 & \text{if } 1 \leq x \end{cases}$$

$$\mu_{PB}(x) = \begin{cases} 0 & \text{if } x \leq 0.5 \\ 2x - 1 & \text{if } 0.5 < x < 1 \\ 1 & \text{if } 1 \leq x \end{cases}$$

where x is either e or e'. The membership functions have a triangular shape and state the degree to which e and e' are considered to be very negative (negative-big, NB), moderately negative (negative-medium, NM), slightly negative (negative-smallm, NS), zero (ZE), slightly positive (positive-small, PS), moderately positive (positive-medium, PM), and very positive (positive-big, PB).

The membership functions give a weight of what value the control signal u_c should have for the sets NB, NM, NS, ZE, PS, PM and PB. The control signal u_c to the hydraulic modulator is then calculated by means of the product-sum inference method:

$$u_c = \frac{\sum_{i,j} \mu_i(e) \cdot \mu_j(e') \cdot u_{i,j}}{\sum_{i,j} \mu_i(e) \cdot \mu_j(e')},$$

where i and j ranges over the set {NB, NM, NS, ZE, PS, PM, PB}, and $u_{i,j}$ is the control signal that is appropriate when for i and j. $u_{i,j}$ is derived as follows (recall $S = (v - \omega_t R)/v$, $e = S - S_{ref}$, and $e' = (S - S_{previous})/\Delta t$). Consider the desired behavior of the brakes for the possible ranges that e and e' can be in:

- NB:
  - e is very negative: This means that the wheel slip is far below optimal and that the wheel velocity is far too high. The wheel slip is increased by greatly increasing braking.
  - e' is very negative: This means that the wheel slip is decreasing fast, either towards or away from the optimum S_{ref}.
- NM:
  - e is moderately negative: The wheel slip is significantly below optimal, and the wheel velocity is too high. The wheel slip is increased by increasing the braking.
  - e' is moderately negative: The wheel slip is decreasing at a moderate rate, either towards or away from the optimum.
- NS:
  - e is slightly negative: The wheel slip is slightly below optimal, meaning that the wheel velocity is a bit to high. The wheel slip is increased by lightly increasing the braking.
  - e' is slightly negative: The wheel slip is decreasing slowly towards or away from the optimum.
- ZE:

  - e is zero: The wheel slip is optimal. No change in braking is needed.
  - e' is zero: The wheel slip is constant.

- PS:
  - e is slightly positive: The wheel slip is slightly above optimal, meaning that the wheel velocity is a bit too low. The wheel slip is decreased by lightly decreasing the braking.
  - e' is slightly positive: The wheel slip is increasing slowly towards or away from optimum.

- PM:
  - e is moderately positive: The wheel slip is significantly above the optimum with the wheel velocity being too low. The wheel slip is decreased by decreasing the braking.
  - e' is moderately positive: The wheel slip is increasing at a moderate rate towards or away from optimum.

- PB:
  - e is very positive: The wheel slip is far above optimum with wheel velocity being far too low. The wheel slip is greatly reduced by no braking.
  - e' is very positive: The wheel slip is increasing fast towards or away from optimum.

The values of the control signal u_c are in the interval [-1, 1] and can be considered to have the following meanings when interpreted by the hydraulic modulator:

- $u_c = 1$: Increase brake pressure as much as possible.
- $u_c = 2/3$: Increase brake pressure moderately.
- $u_c = 1/3$: Increase brake pressure by a small amount.
- $u_c = 0$: No change.
- $u_c = -1/3$: Decrease brake pressure by a small amount.
- $u_c = -2/3$: Decrease brake pressure moderately.
- $u_c = -1$: Decrease brake pressure as much as possible.

With the desired behavior of the brakes depending on the values of e and e' and the interpretation of u_c, $u_{i,j}$ is defined as follows:

$u_{e,e'}$		e'						
		NB	NM	NS	ZE	PS	PM	PB
e	NB	1	1	1	1	2/3	1/3	0
	NM	1	1	1	2/3	2/3	0	-1/3
	NS	1	2/3	2/3	1/3	0	-1/3	-2/3
	ZE	1	2/3	1/3	0	-1/3	-2/3	-1
	PS	2/3	1/3	0	-1/3	-2/3	-2/3	-1
	PM	1/3	0	-2/3	-2/3	-1	-1	-1
	PB	0	-1/3	-2/3	-1	-1	-1	-1

For instance, when $e \in$ NM and $e' \in$ PS, then the appropriate control signal is 2/3. That is, the brake pressure should increase by a moderate amount. Also, for simplification, the verification does not need to deal with arithmetic overflows.

4 Structure of the C Program to Verify

You are given a C program, called tabs.c (thousand ABS since the program is working with multiples of 1000; see below), implementing the algorithm explained in sections 2 and 3. The most relevant variables and functions of tabs.c are:

- signal_to_hydraulic_modulator: Dummy variable representing the address of the register that causes the ECU to send the written value to the hydraulic modulator.
- wt_sensor: Dummy variable representing the address of the register used to read the angular wheel velocity sensor.
- bp_sensor: Dummy variable representing the address of the register used to read whether the brake pedal is pushed.
- at_sensor: Dummy variable representing the address of the register used to read the acceleration sensor.
- R: Global constant storing the radius of the wheels.
- delta_t: Stores the number of milliseconds between interrupts and invocation of the ABS software.
- acceleration_sum: Global variable accumulating the acceleration samples. Used to compute the velocity of the vehicle by integration.
- velocity_before_braking: Global variable storing the velocity of the vehicle just before braking.

- md(index, x): Function implementing the membership functions used to compute the degree to which x is in the set represented by index (index = 0 = NB, index = 1 = NM, index = 2 = NS, index = 3 = ZE, index = 4 = PS, index = 5 = PM, index = 6 = PB).
- compute_velocity_of_vehicle(): Function computing the velocity of the vehicle.
- compute_wheel_slip(v, wt): Function computing the wheel slip given the current velocity v and angular wheel velocity wt.
- compute_control_signal(): Function computing the control signal that shall be sent to the hydraulic modulator.
- hydraulic_modulator_driver(): The function the timer interrupt routine calls to update the brake pressure.

5 Tasks

Verifying programs with floating point computations with Frama-C requires interactive theorem proving, which we will not involve. Therefore all units are multiplied by 1000 and only integer computations are performed.

Your task is to specify and verify in Frama-C with the WP plugin the following properties:

1. If the brake pedal is not pushed, then the brakes are not applied ($u_c = -1000 = -1 \cdot 1000$, meaning that the outlet valve is open). That is, -1000 is written to the variable signal_to_hydraulic_modulator:

$$\text{signal\_to\_hydraulic\_modulator} = -1000.$$

2. If the brake pedal is not pushed, then the current velocity is correctly stored in the variable velocity_before_braking:

$$\text{velocity\_before\_braking} = \text{wt\_sensor} \cdot \text{R}.$$

3. If the brake pedal is not pushed, then the current acceleration is correctly stored in the variable acceleration_sum:

$$\text{acceleration\_sum} = \text{at\_sensor}.$$

4. If the brake pedal is not pushed, then S_previous is assigned zero:

$$\text{S\_previous} = 0.$$

5. The velocity is computed correctly (the first term is divided by 1000 since both factors of that term have already been multiplied by 1000):

$$\text{compute\_velocity\_of\_vehicle}() = \text{acceleration\_sum} \cdot \text{delta\_t}/1000 + \text{velocity\_before\_braking}.$$

6. The wheel slip is computed correctly:

$$\text{compute\_wheel\_slip}(\text{v}, \text{wt}) = (\text{v} - \text{wt} \cdot \text{R}/1000)/\text{v}.$$

7. If:

   - the brake pedal is pushed,
   - the wheel slip is below optimal by at least 500, and
   - the wheel slip is decreasing by at least 500 units per time interval,

   then the maximum brake pressure is applied.

The challenge in this lab is to verify property 7. It is recommended to verify property 7 according to the following steps:

1. Phrase the four subproperties in terms of equations and inequalities and convince yourself that the equations indeed represent the four subproperties.

2. Phrase property 7 as a logical formula.

3. Since the brake pressure is computed by compute_control_signal(), property 7 is a useful contract for this function. Write a pen-and-paper proof (can be digital) convincing yourself that compute_control_signal() satisfies property 7. This proof can be viewed as a proof plan that can guide the verification of that compute_control_signal() satisfies property 7. Without a proof plan, it difficult to know how to annotate a non-trivial C program in order for it to be verified.

4. Use the proof plan from the previous step to guide your annotations of compute_control_signal(). It might be helpful to write a statement annotation (//@ assert ...;) after each C statement (e.g. an assignment x = e) reflecting what is known at that control point. For instance:

   ```
   int wt = read_wheel_angular_velocity();
   //@ assert wt == wt_sensor;
   ```

   By running Frama-C immediately after the new annotation has been added, immediate feedback is received of whether Frama-C succeded with verifying the new annotation. If the verification succeded, then a new statement annotation can be added after the next C statement. If

the verification failed and the C statement invokes another C function the latter C function might need to be verified. If the verification failed (e.g. $0 \leq x \leq 9 \implies y = 0$ could not be verified) then it might be necessary to split up the failing annotation into several "subannotations". Each subannotation shall require less computational effort to verify compared to the failing annotation (e.g $0 \leq x \leq 3 \implies y = 0$, $4 \leq x \leq 7 \implies y = 0$ and $8 \leq x \leq 9 \implies y = 0$). If Frama-C can verify the "subannotations", and infer the failing annotation from the "subannotations", then Frama-C can verify the failing annotation, which can then be verified by inserting it after the "subannotations". For instance, if the verification of

```
//@ assert 0 <= x <= 9  ==>  y == 0;
```

fails, try to give Frama-C smaller verification tasks, such as:

```
//@ assert 0 <= x <= 3  ==>  y == 0;
//@ assert 4 <= x <= 7  ==>  y == 0;
//@ assert 8 <= x <= 9  ==>  y == 0;
//@ assert 0 <= x <= 9  ==>  y == 0;
```

where Frama-C derives the last (previously failing) annotation from the three "subannotations" preceding it.

This incremental approach of adding annotations after each C statement and checking them immediately helps you to immediately spot errors (failed verification attempts or incorrectly formulated formulas).

Finally, as the return value of compute_control_signal() depends on the value of the array u, the values in u must be known to compute_control_signal(). The values of u can be specified by means of a predicate, e.g. u_init, which is defined immediately after the C declaration of u:

```
/*@ predicate u_init =
  @ u[NB][NB] == 1000 && u[NB][NM] == 1000 && ...
```

u_init can then be listed in a requires clause of the function contract of compute_control_signal().

Your solution shall be tabs.c with ACSL annotations such that Frama-C verifies the properties listed above (ACSL is an annotation language a subset of which is implemented in Frama-C). **The C code in tabs.c must not be modified, and the annotated C code shall be submitted in Canvas.**

In addition, write a succinct report containing at least the following:

1. Did you follow the steps listed above?
2. Your thoughts about the difficulty of the lab. What was most difficult?

3. Annotation overhead (number of lines and number of goals to be proved).
4. Your reasoning behind the annotations (e.g. the pen-and-paper proof of compute_control_signal()).
5. Verification time (in seconds).
6. Is there any code in tabs.c that you would rewrite in order to ease the verification?
7. Will you write code differently in the future? Why (not)?
8. What properties do you think are relevant to verify that were not verified?
9. Do you see any flaws in this verification approach where only source code is analyzed?
10. Do you see any practical limitations of what programs/properties that can be verified with Frama-C and the WP plugin?
11. Do you see any shortcomings in the WP-plugin with respect to the expressiveness of annotations?
12. Were there any annotations that you expected to be proved/not to be proved but was not proved/proved in Frama-C? If so, what was the annotation?
13. Have you experienced any strange behavior of (or bugs in) Frama-C? For instance, were you forced to add a statement (assert) annotation that was included in a requires clause?

6 Grading

For the grade E, do all tasks mentioned in Section 5. For D, the report shall be well-structured with toughtful answers.

For a C, the requirements for E and D must be fulfilled, but in addition the following annotations must be verified for tabs_loop.c:

- After the first for-loop at lines 320-321:

  //@ assert ep <= -500 ==> ep_sum == 1000;

- After the second for-loop at lines 324-325:

  //@ assert -1000 <e <= -500 && ep <= -500 ==> numerator == -2000*e - 1000000;

The loop invariants do not need to "look good", and anything that works is accepted. tabs_loop.c with annotations shall be submitted in Canvas.

7 General Tips

- If you use logic functions and predicates, check that Frama-C interprets them as you expect by testing them in statement annotations (Frama-C seems to not always handle logic functions and predicates as specified, perhaps due to bugs).
- Each time you add an annotation, test that Frama-C interprets them as you expect. Such checks can be made by modyfing the preceding C code.
- It can be useful to double check that a function contract is actually correct, since the contract annotation before a function can be complicated and easily misunderstood. A way of double checking is by adding a statement annotation immediately before the return statement stating the desired properties of the return value.
- Implications are useful to check conditional properties, but be careful to not introduce contradictions with a false antecedent as $\bot \implies P$ is true for any property P. Have this also in mind when writing a precondition of a function contract.
- Check whether a variable in an annotation refers to the value of the variable at the time of function invocation, at the current control point, or its value immediately after the function returns. For instance, what is the difference between \at(variable, Pre) and \at(variable, Post)? How is a global variable interpreted when mentioned in a requires clause and in an ensures clause?
- If a formula is not verified but is true, help the automated theorem prover by splitting the formula into several simpler formulas.
- Question critically whether each annotation you introduce really expresses the intended property.
- Useful ACSL constructs (make sure you really understand every detail of the meaning of the constructs you use):
  - requires
  - ensures
  - assigns

  - behavior
  - assumes
  - let
  - assert
  - \result
  - \nothing
  - \at

- Make annotations readable with the let construct.
- If a function f1 satisfies its contract, but a function f2 calling f1 does not satisfy the preconditions of f1 at the invocation point, then f2 is not verified.
- Each statement annotation is independent of all other statement annotations, meaning that the variables in one annotation is not affected by other annotations (even if the statement annotations are on consecutive lines).
- Work incrementally by iteratively adding one annotation, checking it, and fixing it if necessary.