Liam O'Reilly, Markus Roggenbach (Swansea University, UK)

Lab: Verification of Ladder Logic Programs

Indicated duration 120 minutes
Work in teams of two
Assessed

Learning Outcomes

- Understanding of automated theorem proving
- Ability to verify Ladder Logic programs through Inductive Verification and Inductive Strengthening

Further Reading

- Roggenbach et al. *Formal Methods for Software Engineering*, Springer 2022, Section 4.3

Required Resources

- Access to the web-based interface of the tool Hets `http://rest.hets.eu`
- Python 3
- File: `ladderLogicSim.py`
- File: `deMorgan.casl`
- File: `ladderLogic.casl`
- File: `controller.casl`

Hints:

- It will be best to work with a source code editor such as Notepad++.
- When working with Hets, it is advisable to submit the CASL specifications using the text field: Open your specification in an editor, copy the specification, paste it into the Hets website.

  We advise this as when uploading the same file several times, Hets changes its behaviour according to results from previous sessions and might therefore behave differently from what is described on this lab sheet.
- When checking in Hets if a proof obligation holds or not, focus on texts such as "Results for Law-1 (Proved)" or "Results for WrongEquivalence-3 (Disproved)". In the context of this lab, you can ignore all further information that Hets displays.

Tasks

Task 1 – *Exploring Automated Theorem Proving with CASL*

Introduction to the Task: In propositional logic, De Morgan's laws are a pair of transformation rules that allow the expression of conjunctions and disjunctions purely in terms of each other via negation. Logically speaking, these transformations are tautologies, i.e., logical formulae that hold in any model.

Consider the following incorrect[1] CASL specification of De Morgan's Laws:

```
spec DeMorgan =
    preds A, B: ()
then %implies
     . not (A \/ B) <=> (not A) /\ (not B) %(Law-1)%
     . not (A /\ B) <=> (not A) \/ (not B) %(Law-2)%
then %implies
     . not (A \/ B) <=> (not A) \/ (not B) %(WrongEquivalence-3)%
     . not (A /\ B) <=> (not A) /\ (not B) %(WrongEquivalence-4)%
end
```

First, the above algebraic specification in CASL correctly formulates the two De Morgan laws named

- %(Law-1)% and
- %(Law-2)%.

The CASL specification also includes wrong formulations of the De Morgan laws named

- %(WrongEquivalence-3)% and
- %(WrongEquivalence-4)%.

The mistake in %(WrongEquivalence-3)% is that rather than changing from disjunction on the left-hand-side to conjunction on the right-hand side, the formula uses disjunction on the right-hand side; similarly for %(WrongEquivalence-4)%.

Explanation – *Annotations to CASL Specifications*

When a CASL specification is uploaded into the tool Hets, the annotation %implies following the keyword **then** triggers the tool to turn all formulae stated *after* the keyword **then** to be treated as proof goals, which a theorem prover will have to discharge using all the formulae stated *before* the keyword **then**.

In CASL, the notation %(__)% is used to give a name to a formula. This allows later on to trace in the output of a prover, which formula has been proven or disproven.

[1]Syntactically, the specification is correct, however, semantically not all %implies annotations hold.

- Enter the CASL specification DeMorgan from the file `deMorgan.casl` into the web-based interface of the tool Hets, available at `http://rest.hets.eu` and press the `submit` button.

  Effect: The specification is parsed by the tool Hets and represented as a so-called Development Graph: each node of the graph represents a specification fragment corresponding to part of the original CASL specification.

- Under `Commands` select `auto`.

  Effect: The Development Graph of the CASL specification DeMorgan is transformed in such a way that theorem provers can work on it. The red nodes indicate that there are open proof obligations.

- Click on the upper red node.

  Effect: You obtain a window that shows the theory over which you are working, where the open proof goals are marked as **%implied**.

- Click `Prove`.

  Effect: You obtain a Results window that lists shows whether the two formulae could be proven or not.

- Check that

  - Law-1 and
  - Law-2

  have been proved.

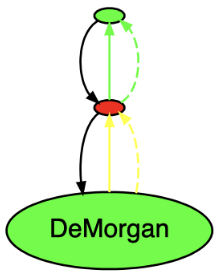
- Click on `ReturnToDGraph` to return to the development graph.

- Repeat the process for the lower red node and check that

  - WrongEquivalence-3 and
  - WrongEquivalence-4

  have been disproved.

Assessment: The development graph with the upper node being green, the middle node being red, and the lower node being green, as shown below:



Task 2 – *Verifying a Ladder Logic Program via Manual Simulation*

Introduction to the Task: You are going to simulate a Ladder Logic control program of a Pelican Crossing. The Pelican Crossing helps pedestrians cross a road safely. It has red and green lights to prevent and allow pedestrians to cross the road; and red, amber, and green lights to prevent and allow vehicles to travel. Some lights can also flash to indicate that the lights are about to change.

- Use Python 3 to run the file `ladderLogicSim.py`. If your `python` command is setup for Python 3 then you would do this with the command:

```
python ladderLogicSim.py
```

- Upon running the simulator you will be presented with the following:

```
Initial State
-------------
Internal State
     sh        : 0
     sl        : 0
     request   : 0
     button    : 0
Lights
     vehicle   : tg
     pedestrian: pr


Cycle: 1
-----------
Is the button pushed (enter 0 for No, 1 for Yes, q to quit)?
```

Here you can see one of the initial states of the controller: what values the internal variables have (Internal State) and the state of the pedestrian red light and the vehicle green lights (both turned on). Note that only the active lights are displayed, unlit lights are false and suppressed in the output.

The lights are encoded as:

- tg: Vehicle green light
- ta: Vehicle amber light
- tr: Vehicle red light
- taf: Vehicle amber light flashing
- pg: Pedestrian green light
- pgf: Pedestrian green light flashing
- pr: Pedestrian red light

The Pelican Crossing changes its state regularly according to the Ladder Logic program. The new state depends on the previous state and the information if a pedestrian has pressed a button. At the beginning of the new cycle, either a button has been pressed or not. Entering this information as `0` (button not pushed) or `1` (button pushed) in the simulator triggers the simulator to continue the cycle and produce a new state.

For example, if we indicate that a button has been pressed by entering `1` then the simulator presents:

```
Cycle: 1
-----------
Is the button pushed (enter 0 for No, 1 for Yes, q to quit)? 1

Initial State
-------------
Internal State
     sh        : 0
     sl        : 1
     request   : 1
     button    : 1
Lights
     vehicle   : tg
     pedestrian: pr


Cycle: 2
-----------
Is the button pushed (enter 0 for No, 1 for Yes, q to quit)?
```

Here you can see that pressing the button has no visible effect yet (both the pedestrian red light and the vehicle green light are still on). However, the internal state has changed. You can then enter values by pressing the button over further rounds to simulate the system.

- Explore the simulation and write two proper cycles consisting of 5 or 6 states (i.e., traces that start and end with the same state, and proper means to not have any repeated states in between).

  Explanation – *Suggested Notation for Traces*

  To write a trace, it is advised to follow the following example scheme:

  ```
      0 0 0 0, tg, pr  --1-->   0 1 1 1, tg, pr   --0--> ...
  ```

  Each state is represented by the truth values of the variables sh, sl, request, button, followed by the information on which lights are on. Here we start in a state where sh is 0, sl is 0, request is 0, button is 0 and the lights tg and pr are the only lights which are on. The arrow shows we pressed the button and end up in a new state. Further steps can follow.

- Decide if the following "Usability-Condition" holds for each of your cycles: After pressing the button, the pedestrian light will become green (variable pg becomes true) after two or three further steps.
- Find as many cycles as you can within, say, 3 minutes. Did you find all cycles of the control program? Write down a reason for your response.

Assessment: Two proper cycles and your evaluation, if the Usability-Condition holds; argument why you found all / did not find all proper cycles within 3 minutes.

Aside – *Relation Between the Python Program and the CASL Specification*

The Python program `ladderLogicSim.py` is written in a generic style that allows to simulate *any* Ladder Logic specification using the functions:

- `pre_initial_state` provides the initialisation of the variables.
- `compute_next_state` captures the transition relation.
- `read_input` reads the new values for input variables.
- `print_output` prints the output.

The code within `pre_initial_state` and `compute_next_state` is 'identical' with our propositional formulae as written in CASL – up to syntax representation due to using a different language.
By changing the above-listed four functions, it is possible to simulate any ladder logic program.

Task 3 – *Formally Verifying a Ladder Logic Program*

Introduction to the Task: Establishing the Usability-Condition through simulation in Task 1.2 turned out to be cumbersome at least, if not impossible. In this task, you will prove that two safety properties hold. To this end, you will apply two techniques: Inductive Verification and Inductive Strengthening.

Inductive Verification involves two proof steps: first, we show that the initial states are safe; in a second step we show that,U should we start in a safe state, all successors of this safe state are safe again. We will see that with this technique it is possible to show the safety property concerning pedestrian lights.

Explanation – *Inductive Verification*

Inductive Verification can be expressed through the following schematic specifications in CASL:

```
spec InitialStatesAreSafe =
    Init
then %implies
    . safety %(initial states are safe)%
end

spec TransitionsPreservesSafety =
    TransitionRelation
then %implies
    . safety => safety' %(safety is preserved under transitions)%
end
```

Here,

- `Init` is a specification of the system's initial state,
- `safety` is the property to be shown.
- `Transition` is a specification of the system's transition relation, and
- `safety'` is as `safety` but will all variables primed.

Unfortunately, this technique fails to prove the property concerning traffic lights. The reason for this is that Inductive Verification over-approximates the state set. In its second step, Inductive Verification considers all safe states rather than all safe and *reachable* states.

For situations like this, one can apply the proof technique of Inductive Strengthening. This technique utilizes a so-called invariant. An invariant is a formula that holds for all reachable states. We can use Inductive Verification to establish that a given formula is an invariant.

Given an invariant, like Inductive Verification, the technique of Inductive Strengthening also consists of two proof steps: first, we show that the initial states are safe; in a second step, we show that should we start in a safe state *for which the invariant holds*, all successors of this safe state are safe again. Using this technique, it is possible to show the safety property concerning traffic lights.

Explanation – *Inductive Strengthening*

Inductive Strengthening can be expressed through the following schematic specifications in CASL:

1. Establishing that the formula `inv` is actually an invariant:[a]

```
spec InvariantHoldsInInitialStates =
    Init
then %implies
    . inv %(invariant holds initially)%
end

spec InvariantHoldsForTransitions =
    TransitionRelation
then %implies
    . inv => inv’ %(invariant is preserved under transitions)%
end
```

2. Proving safety with the invariant:

```
spec InitialStatesAreSafe =
    Init
then %implies
    . safety  %(initial states are safe)%
end

spec TransitionsAreSafeUnderInvariant =
    TransitionRelation
then %implies
    . safety /\ inv => safety’
      %(safety is preserved under invariant)%
end
```

[a]Here we are actually proving that `inv` is inductive; note that an invariant is not necessarily inductive.

- Enter the CASL specification pelicanCrossing.casl into the web-based interface of the tool Hets, available at `http://rest.hets.eu` and press the `submit` button.
- Under `Commands`, select `auto`.
- Click on the small red node directly above `InitialStatesAreSafe` and prove that both safety properties hold in the initial state.
- Click on the red node `TransitionRelation` and prove that safety is preserved for pedestrians and establish that it is not the case that safety is preserved for vehicles.

  This means: Inductive Verification allows us to show safety for the pedestrian lights, however, it is not successful for the traffic lights.

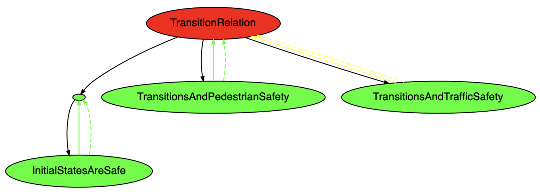
Explanation – *False Positives*

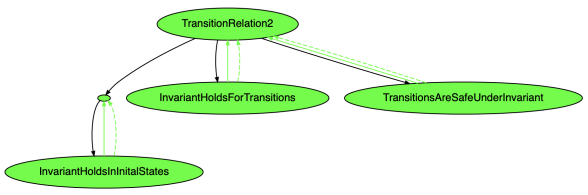
> With regards to the safety of traffic lights, at this point, we do not know if the program is wrong or if this result is a 'false positive', i.e., the proof tool reports an error where there is none. Therefore, we try the proof technique of Inductive Strengthening.

- Click on the small red node directly above `InvariantHoldsInInitalStates` and prove that the invariant holds in initial states.
- Click on `TransitionRelation2` and prove that the invariant is preserved when making a transition and safety is preserved for vehicles under an invariant.

  This establishes the safety property with regards to vehicle lights (together with the proof of "Initial states are safe for vehicles", which we already established above).

Assessment: Development graphs as shown below (where we omit the middle part of the picture concerning the specification `TransitionRelationAlternative` as it is not affected by our proof activities):




TransitionRelation2
InvariantHoldsForTransitions
TransitionsAreSafeUnderInvariant
InvariantHoldsInInitalStates

Task 4 – *Verifying a Room Control System*

Introduction to the Task:

In this task you shall verify a small control program on your own, using the technique of Inductive Strengthening. The 'challenge' will be to find a suitable invariant.

Narrative - A Room Controller

As part of a state-of-the-art lecture theatre, a new room controller is proposed that automatically controls the blinds (which we all know, are too complicated to operate for lecturing staff). When the lights are off, the blinds should be up to allow natural light in. When the lights are on, the controller shall lower the blinds. This allows for optimal viewing of the projector. Of course, safety is important. Thus, there is a safety light. The controller has to guarantee that there is light at all times, i.e., the lights are on or the safety light is on.

This fictitious room controller has been realised in a ladder logic program as follows:

```
spec Init =
    preds s,                %% safety light
          b : ()            %% blinds
    . not b /\ s
end

spec TransitionRelation =
    preds s,s',            %% safety light
          b,b',            %% blinds
          i:    ()         %% input
    . b' <=> i \/ (not s /\ b)
    . s' <=> s
end
```

This ladder logic program uses state variables "b" and "s", as well as an input variable "i". The variable encoding is as follows:

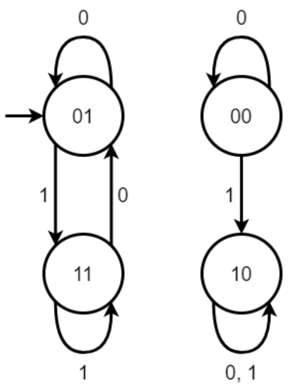
- b = 1 means the blinds are down.
- s = 1 means the security light is on
- i = 1 means the user switches the room light on

Initially, the blinds are up and the safety light is on.

Using the above encoding, the desired safety property can be expressed as

$$\neg \mathtt{b} \vee \mathtt{s}.$$

The following picture depicts the transition system resulting from this program:



The states are written as pairs `b s`. 01 is the initial state. The label on the transitions is the value of the input i.

- Inspect the given, finite transition system and figure out if all its reachable states are safe.
- (Optional step) Modify the Python program `ladderLogicSim.py` to simulate the room controller. To this end, study the above "Aside" box.
- Establish with Hets that safety can't be proven with Inductive Verification.
- Invent a logical formula that can serve as an invariant for inductive strengthening and does not characterise reachability.

  Explanation – *Reachability Versus Invariants*

  > Usually, a controller has many states and hence several invariants are possible. In verification practice, we try to find invariants that are 'easier' to identify than formulae that characterise reachability. As invariants are more abstract than reachability (i.e., hold in more states), one can take advantage of this and find a simple and intuitive formula.

  Hint: Analyse the above transition system and figure out which transitions give rise to false positives in Inductive Verification. Your invariant ought to fulfil two properties:

  - it needs to exclude all unreachable safe states which have a transition to an unsafe state, and
  - it needs to hold in all reachable states.

  Note that this technique of inspecting a transition system and reading off an invariant works out only for examples with a small state space. For large examples, one needs special algorithms to automatically construct invariants such as the IC3 algorithm (see, e.g., Aron R. Bradley: *Understanding IC3*, LNCS 7317, Springer, 2012).

- Demonstrate that safety can be proven by Inductive Strengthening.

Assessment: The invariant, the development graph of Inductive Strengthening in Hets with all proof obligations discharged.