Applied Verification Tasks

Markus Anders Sören Kwasigroch Matthias Lederer

June 20, 2019

1 NuSMV

1.1 What is the tool about?

NuSMV is a symbolic model checker. Its input language enables the symbolic description of finite state machines. The tool can then check whether the given FSM is a valid model for a LTL or CTL specification. For this procedure, the user can choose between BDD-based methods and SAT-based methods such as bounded model checking (BMC) and (rudimentary) induction.

1.2 What can it be used for?

There are many ways in which the tool can be applied:

1. For the description of reactive systems and proving properties on them.
2. Optimal synthesis by searching for counterexamples for myriad kinds of problems (e.g. synthesis of circuits or solving games and puzzles).
3. As a backend tool for verification in a model-based design flow.

1.2.1 How can you use the tool?

A good starting point is the NuSMV Tutorial [3] and a good reference is the NuSMV User Manual [4].

Description Language. The description language enables several ways to define initial states, transitions or generally the state space in the FSM. A complete description of the input language can be found in the user manual [4].

Generally, modules (`MODULE`) form the basis of the defined FSMs. They have local variables (`VAR`) that are in a certain state and can transition using direct assignments of new values based on the current state (`ASSIGN`) or by defining constraints on the transitions directly (`TRANS`), also enabling non-deterministic transitions. Modules can be made of submodules, enabling the piecewise construction of complex structures.

As an example, consider the following simple module from [3]:

```
1  MODULE main
2  VAR
3  request : boolean;
4  state : {ready, busy};
5  ASSIGN
6  init(state) := ready;
7  next(state) := case
8  state = ready & request = TRUE : busy;
9  TRUE : {ready, busy};
10 esac;
```

It has two variables, *request* and *state*. Both are basically boolean, but *state* uses symbolic names (refer to [4] for more specifics). The `ASSIGN` declaration then explicitly tells us the transitions of the state machine: the initial state is *ready*. If the current *state* is ready and there is a request, we switch to *busy*. Else, the state may be either *ready* or *busy* non-deterministically. Note that since *request* is never assigned, its value is also non-deterministic.

Specifications. LTL and CTL specifications are possible using `LTLSPEC` (for LTL) and `SPEC` (for CTL). NuSMV can then check whether the given FSM is a model for the specification.

Variable Orderings. Variable ordering and clusterings can be given manually in auxiliary order (*.ord*) files. These files simply list a (partial) order of variables to be used. They can be used by adding the *-i* command line option, specifying the desired file.

Proving Specifications. The tool can be called in different ways to enable BMC, induction or the BDD-based methods. Simply running

```
1 NuSMV input-file
```

will try to verify all `SPEC` and `LTLSPEC` using the BDD-based methods. Adding the *-bmc* command line option will switch to bounded model checking. Then, *-bmc_length* will determine the maximal length of path to which properties will be checked. BMC only works with LTL specifications. It should however be noted that in our testing, BMC methods ran considerably faster using the available incremental options. Details on these options can be found in Section 3.3 of [4].

Interactive Usage. The tool also enables interactive usage and exploration of the state space. By calling it with e.g.

```
1 NuSMV -int input-file
```

the interactive mode is entered. From there, special commands can guide NuSMV through states of the finite state machine. For more information on the interactive mode consult Section 3 of [4].

1.3 Exercises

In this section, we will present some exercises and their respective solutions for NuSMV.

1.3.1 Alice and Bob Share a Room

The following algorithms are taken from [2].

Problem A. Alice and Bob can both access a *critical* section. If Alice is in the critical section, Bob must not be (and vice versa). They can only communicate via auxiliary variables to coordinate their requests. Consider the following four algorithms:
Algorithm 1.

```
1 A0: Maybe go to A1.
2 A1: If l go to A1, else to A2.
3 A2: Set l = 1, go to A3.
4 A3: Critical, go to A4.
5 A4: Set l = 0, go to A0.
```

```
1 B0: Maybe go to B1.
2 B1: If l go to B1, else to B2.
3 B2: Set l = 1, go to B3.
4 B3: Critical, go to B4.
5 B4: Set l = 0, go to B0.
```

Algorithm 2.

```
1 A0: Maybe go to A1.
2 A1: Set a = 1 go to A2.
3 A2: If b go to A2, else to A3.
4 A3: Critical, go to A4.
5 A4: Set a = 0, go to A0.
```

```
1 B0: Maybe go to B1.
2 B1: Set b = 1 go to B2.
3 B2: If a go to B2, else to B3.
4 B3: Critical, go to B4.
5 B4: Set b = 0, go to B0.
```

Algorithm 3.

```
1 A0: Maybe go to A1.
2 A1: Set a = 1 go to A2.
3 A2: If b go to A3, else to A4.
4 A3: Set a = 0, go to A1.
5 A4: Critical, go to A5.
6 A5: Set a = 0, go to A0.
```

```
1 B0: Maybe go to B1.
2 B1: Set b = 1 go to B2.
3 B2: If a go to B3, else to B4.
4 B3: Set b = 0, go to B1.
5 B4: Critical, go to B5.
6 B5: Set b = 0, go to B0.
```

Algorithm 4.

```
1 A0: Maybe go to A1.
2 A1: Set a = 1 go to A2.
3 A2: Set l = 0, go to A3.
4 A3: If b go to A4, else to A5.
5 A4: If l go to A5, else to A3.
6 A5: Critical, go to A6.
7 A6: Set a = 0, go to A0.
```

```
1 B0: Maybe go to B1.
2 B1: Set b = 1 go to B2.
3 B2: Set l = 1, go to B3.
4 B3: If a go to B4, else to B5.
5 B4: If l go to B3, else to B5.
6 B5: Critical, go to B6.
7 B6: Set b = 0, go to B0.
```

Note that in all of these algorithms, it is assumed that Alice and Bob can not linger in any state but $A0$ and $B0$. They are however allowed to operate at different speeds.

Model the specified mutual exclusion protocols sufficiently in NuSMV. Then, prove or disprove whether they actually implement the following properties:

1. Mutual exclusion: both parties can not enter the critical section simultaneously.
2. Deadlock-free: both Alice and Bob can always make progress.
3. Fairness: if a request is made to enter the critical section, it will be entered eventually.

Solution. *(Tool Input)* All of the algorithms can be formalized in the same manner: Alice and Bob can be modelled using a `MODULE`. Both clearly implement finite state machines. A state can be associated with each of the lines of the algorithms.

Specifying the properties is then quite simple. Mutual exclusion simply becomes checking whether both can not be in the according critical state, for Algorithm 1 that is $A3$ and $B3$:

```
1 LTLSPEC G (((a.state = 3) -> !(b.state = 3)) & ((b.state = 3) -> !(a.state = 3)));
```

Fairness is simply that if a request is made ($A1$ or $B1$ entered), eventually (F) state $A3$ or $B3$ must be reached:

```
1 LTLSPEC G (((a.state = 1) -> F (a.state = 3)) & ((b.state = 1) -> F (b.state = 3)));
```

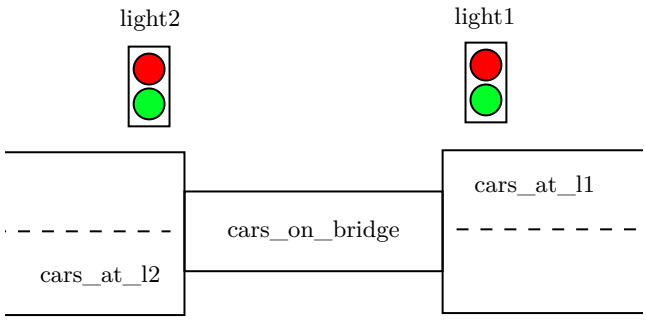
The specific input can be found in *mutex_alice_bob_nomutex.smv* for *Algorithm 1*, *mutex_alice_bob_deadlock.smv* for *Algorithm 2*, *mutex_alice_bob_disjkstra.smv* for *Algorithm 3* and *mutex_alice_bob_peterson.smv* for *Algorithm 4*.

(Tool Output) Using the BDD-based methods, NuSMV either proves the given properties correct or provides a counterexample.

(Conclusions) *Algorithm 1* does not implement mutual exclusion. *Algorithm 2* suffers from deadlocks. *Algorithm 3* is not fair. Finally, *Algorithm 4* is deadlock-free, fair and implements mutual exclusion.

1.3.2 One-lane Bridge Controller

Problem B. Consider a one-lane bridge over a river:



The bridge is only wide enough for a single car. So the directions have to take turns using the traffic light for coordination.

The task is now to design a traffic light controller for the bridge that has the following verified properties:

1. Safety: only one side at a time can have green light at a time. Only switch to a green light if the bridge is empty.
2. Fairness: if a car is waiting on one side, it must eventually get a green light.

The available sensors include `cars_on_bridge` which determines whether a car is currently driving on a bridge. `cars_at_l1` and `cars_at_l2` each indicate whether a car is waiting at the respective light. `light1` and `light2` have to be controlled by the controller. They can take values `red` and `green`, with their obvious meanings. The following properties can be assumed about the car behaviour:

```
1  -- cars leave the bridge
2        TRANS
3              ((cars_on_bridge & (light1 = red) & (light2 = red))-> !next(cars_on_bridge));
4  -- if a light is not green, cars do not leave the light
5        TRANS
6              (!(light1 = green) & cars_at_l1 -> next(cars_at_l1)) &
7              (!(light2 = green) & cars_at_l2 -> next(cars_at_l2));
8  -- if the light is green and there is a car at a light, it enters the bridge
9        TRANS
10             ((cars_at_l1 & (light1 = green)) -> next(cars_on_bridge)) &
11             ((cars_at_l2 & (light2 = green)) -> next(cars_on_bridge));
12 -- if no light is green, no car enters the bridge
13       TRANS
14             (light1 = red & light2 = red & !cars_on_bridge) -> !next(cars_on_bridge);
```

Design a suitable controller and properties utilizing the given sensors and actors. Then, verify the properties using NuSMV.

Solution. *(Tool Input)* Several different solutions are possible. We chose to go with a timed green light and switching priorities every time the light is green. The solution is contained in *one_lane_bridge.smv*.

First, some variables have to be defined. The controller only has two variables: a state variable and a timer:

```
-- variables of the controller itself
    state : {red_empty_last_1, red_full_last_1, red_empty_last_2, red_full_last_2, l1_green, l2_green};
    time : 0..5;
```

The exercise already defined the available sensors and actors:

```
-- sensors
    cars_at_l1 : boolean;
    cars_at_l2 : boolean;
    cars_on_bridge : boolean;

-- actors
  light1 : {red, green};
  light2 : {red, green};
```

Now, the state transition has to be modelled. We will start in a *red_full* state that will wait for the birdge to be empty.

```
init(state) := red_full_last_1;
```

A case distinction will now be made to determine the next state. In any of the *red_full* states we will want to wait for the bridge to be empty while the lights are red. Only when the bridge becomes empty a new green light should be activated.

```
next(state) := case
       state = red_full_last_1 & !cars_on_bridge: red_empty_last_1;
       state = red_full_last_2 & !cars_on_bridge: red_empty_last_2;
```

Once that is the case, we will activate the green light on demand. If the last green light was *light*1, we prioritize *light*2 (and vice versa):

```
       state = red_empty_last_1 & cars_at_l2: l2_green;
       state = red_empty_last_1 & cars_at_l1: l1_green;
       state = red_empty_last_2 & cars_at_l1: l1_green;
       state = red_empty_last_2 & cars_at_l2: l2_green;
```

Once the timer runs out, we switch the light to red:

```
       state = l1_green & time = 5: red_full_last_1;
       state = l2_green & time = 5: red_full_last_2;
         TRUE : state;
     esac;
```

The timer is simply implemented using the following assignment:

```
       init(time) := 0;
       next(time) := (time + 1) mod 6;
```

Finally, we have to define the light actors. However, these can be directly derived from the *state* variable:

```
       light1 := (state = l1_green)?green:red;
       light2 := (state = l2_green)?green:red;
```

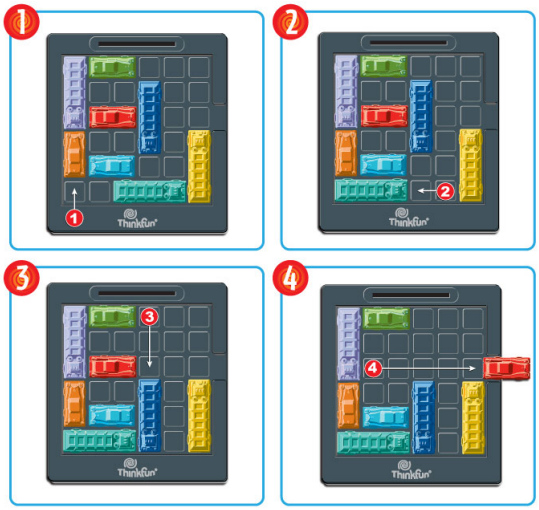


Figure 1.1: How to play Rush Hour.

The specifications are now straightforward. For safety, first we specify that only one light is green at a time:

```
LTLSPEC G (light1 = green) -> (light2 = red);
LTLSPEC G (light2 = green) -> (light1 = red);
```

Furthermore, if there are cars on the bridge, we do not switch to a green light in the next state:

```
LTLSPEC G ((cars_on_bridge & (light1 = red)) -> (X light1 = red));
LTLSPEC G ((cars_on_bridge & (light2 = red)) -> (X light2 = red));
```

For the fairness property, we just state that if a car is at *light*1, eventually *light*1 must turn green:

```
LTLSPEC G (cars_at_l1 -> (F light1 = green));
LTLSPEC G (cars_at_l2 -> (F light2 = green));
```

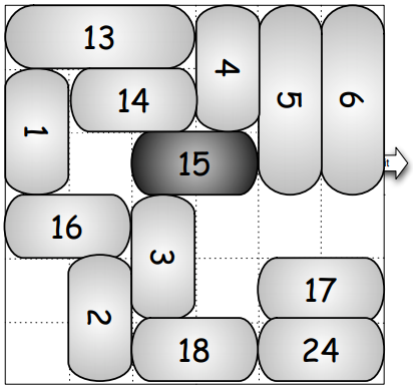
(Tool Output) The tool proofs all of the properties correct.

(Conclusions) The implemented traffic controller keeps drivers on the bridge safe.

1.3.3 Rush Hour

Rush Hour is a puzzle game where the goal is to remove the red car from the board. Figure 1.1 shows an example of how to play the game. You can move the cars along their given axis back and forth. The red car has to go all the way to the right to the opening of the board, which solves the puzzle.

Problem C. Get yourself familiar with the rules of Rush Hour. In [1], a hardest 6×6 Rush Hour board was proposed, which is the following:



The block labelled 15 is the red car that is supposed to leave to the right. Use NuSMV to solve the puzzle.

Solution. *(Tool Input)* The rules of the game can be encoded into a FSM. A state can transition to another if the movement of cars is possible within the rules. Towards this goal, several invariants can be defined.

An auxiliary script (see *rush_hour.rb*) was written in Ruby to produce the NuSMV input for a given board. It consists of the following, general parts: First, for every car its x and y position and length is defined. The car can only transition on its movement axis (either x or y) from e.g. x to $x+1$ or $x-1$. This is done using a `TRANS` declaration. Furthermore, an `INVAR` is added to prevent the car from leaving the field at all and another `INVAR` ensures that not too many cars are moved at once, preventing invalid moves.

Finally, collisions between cars have to be simulated. Since we only move cars 1 position at a time, a simple `INVAR` stating that cars should not overlap suffices. Cars that can never collide do not need collision invariants. Two different kinds of collisions are possible: either cars in the same row or column collide on their axis (x-x and y-y collisions). Or one car is on the x axis and the other on the y axis (x-y collision).

The LTL specification used as a proof goal simply states that the red car can never reach the x position related to the exit. If a counterexample is found, it will state all of the moves necessary such that the red car can reach the exit.

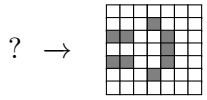
(Tool Output) Then (using BMC), the necessary moves can be read from the output.

(Conclusion) We were able to solve tough instances using BMC but not the BDD-based methods. Furthermore, we know that the solution found is the shortest possible solution: we verified that for all shoter paths, the specification holds, which means the puzzle can not be solved.

1.3.4 Reverse Game of Life

The following exercise is again inspired by the discussion in [2].

Problem D. Consider the following configuration of Conway's famous Game of Life:



Get yourself familiar with the rules of the Game of Life. Use NuSMV to find a generation that may precede it while not leaving the 7×7 field. Furthermore, proof the maximal amount of generations that can precede it within the given 7×7 field. The 7×7 field should act as if the bordering cells are dead, and should never find itself in a configuration that would bring them to life, i.e., your calculated generations should act the same when being placed on a larger field.

Solution. *(Tool Input)* First, the Game of Life itself has to be modelled. It is easy to start with a single cell, implementing the life and death rules.

```
1 MODULE cell(cSELF, cNW, cN, cNE, cW, cE, cSW, cS, cSE)
2 ASSIGN next(cSELF) := (4 < (2 * (cNW + cN + cNE + cW +
3                               cE + cSW + cS + cSE) + cSELF)) &
4                       (8 > (2 * (cNW + cN + cNE + cW +
5                               cE + cSW + cS + cSE) + cSELF))? 1 : 0;
```

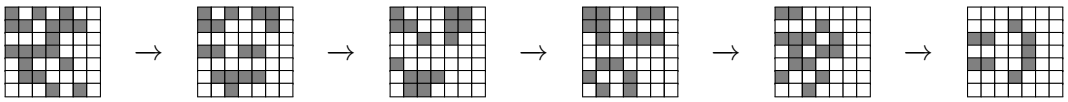
Cells are then connected in the main module, and the desired properties given. E.g., finally drawing a certain field but starting from another. Special care has to be taken that no living cells can propagate to the borders, thus reaching incorrect configurations. A simple LTL specification can be used to enforce that a certain amount of generations is actually produced, where α is the final desired configuration:

```
1 LTLSPEC G ! X ... X (α)
```

An auxiliary script *game_of_life.rb* was used to put together all of the constraints.

(Tool Output) Then (using BMC), it can be seen that the given configuration can be preceded by the following generations.

(Conclusion) Since for more generations the BMC could not find a counterexample within the appropriate length (the amounts of X's), it is not possible to precede the given generation in the 7×7 field once more. Therefore, the maximal amount of preceding generations is 5.



(Conclusion) Since for more generations the BMC could not find a counterexample within the appropriate length (the amounts of X's), it is not possible to precede the given generation in the 7×7 field once more. Therefore, the maximal amount of preceding generations is 5.

1.4 nuXmv

nuXmv extends NuSMV by the following features:

- The input language is extended with datatypes *real* and *integer* to enable the description of infinite state transition systems.
- New SMT-based algorithms to verify properties on infite state transition systems.
- More elaborate induction-based algorithms for the verification of finite state transition systems.

References

[1] Sébastien Collette, Jean-François Raskin, and Frédéric Servais. "On the Symbolic Computation of the Hardest Configurations of the RUSH HOUR Game". In: *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*. Ed. by H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers. Vol. 4630. Lecture Notes in Computer Science. Springer, 2006, pp. 220–233. ISBN: 978-3-540-75537-1. DOI: 10.1007/978-3-540-75538-8_20. URL: https://doi.org/10.1007/978-3-540-75538-8%5C_20.

[2] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. 1st. Addison-Wesley Professional, 2015.

[3] *NuSMV 2.6 Tutorial*. URL: http://nusmv.fbk.eu/NuSMV/tutorial/v26/tutorial.pdf.

[4] *NuSMV 2.6 User Manual*. URL: http://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf.

2 SPIN

SPIN [9] is a tool for verifying programs written in PROMELA (`.pml`). Since SPIN is mostly based on full state space exploration, the primary goal when modeling is to keep the state space as small as possible.

2.1 Promela

The semantics of PROMELA are mainly based on the concept of executablility of sequences and hence allow for the direct modelling of concurrent processes. Safety properties can be specified using assertions, liveness properties by using specially named labels. Full LTL-specifications can be encoded as Büchi-automata and implemented as separate processes.

For an introductory description of PROMELA, see [6]. For the language reference, see [7, 5].

2.2 Usage

By default, `spin` will run a random simulation of the model. Use the option `-a` to generate the C-code of a verifier (`pan.c`). With `-run`, `spin` will generate, compile and run the verifier. With no further options, the verifier will only check assertions and bad end-states. Use `-l` to generate a verifier that can check liveness. Error traces found by the verifier can be replayed using `-replay`.

See [8] for more options and compile-time parameters.

2.3 µP

From the Github-page [4] of P [3]:

> P is a language for asynchronous event-driven programming. P allows the programmer to specify the system as a collection of interacting state machines, which communicate with each other using events.

µP has been derived from P with the intention that more features of P can be added in the future. To that end, µP uses the same syntax as P does but some features are disabled. We implemented a translation of µP to C (for compilation/simulation) and to PROMELA (for verification). See `README.md` for how to call the compiler for the two targets.

2.3.1 Current Status

Features we implemented:

- basics
  - event specification
  - machine creation
  - event passing through FIFO-buffers
- assertions
- halt state

Features that could be added:

- deferred event handling
- internal events
- ghost machines
- printing

2.3.2 Notes on the Parser

Since we use the very same grammar as P does, we thought that the easiest way would be to use the same grammar specification as well. Because the original specification is written for the parser-generator ANTLR, our parser uses ANTLR, too. In general, the parser is kept to a minimum; at the moment, it generates an abstract syntax tree (AST) in one go using a visitor and no validity checks (types, whether variables are declared) are performed.

See `src/main/java/PAst/` for the definition of the AST and `src/main/java/PComp/PVisitor.java` for the visitor.

2.3.3 Notes on the Translation to Promela

Basically, every machine type is translated to one proctype and its states become labels inside the proctype. Switching states is thus a mere goto.

The main problem is the translation of machine queues in the presence of dynamic creation and halting of machines. The specified behavior of a halted machine in µP is that its queue will drop all further messages.

Channels are the natural counterpart of queues but there are some limitations due to the finite state nature of Spin: each process must have fixed resources, i.e. it is not possible to declare channels in a dynamic context, e.g. before a new process is run. Furthermore, Spin apparently lacks a garbage collector; once created and communicated to other processes, a channel will usually never be freed, even when the owning process has stopped. This leaves us with two options: either each process declares its own channel and communicates it to its parent or there is a fixed amount of pre-allocated channels (to be concrete: 255 since Spin allows as many processes).

In principle, µP does not forbid creating infinitely many machines; at any given moment only finitely many machines will be running. However, the above facts make it hard (if not impossible) to use Spin for checking programs that create and dispose processes in such a dynamic manner: As channels do not get freed, one would have to reuse channels, i.e. use a pre-allocated array of channels, indexed by, for

instance, the process id. Unfortunately, this is not sufficient as a halted machine's channel might still be in use when it is assigned to a new machine. As a workaround, one could pair each message with an instance counter. With small counter ranges this might still fail regardless; large ranges could blow up the state space. Generally speaking, one would have to retrofit a garbage collection of some kind to allow for infinitely many machine creations.

In the end, we chose to not allow for unrestricted creation and halting of machines and instead settled with the following rather natural translation: each machine declares its own channel and communicates it to its parent through a single rendezvouz-channels `p_publish`. When the machine halts, it transitions into a special halt state (a valid end state) and drops all incoming messages.

New machines are created atomically as follows exploiting the rendezvouz-channel for transferring control: The creating machine checks whether a new process can be started (`_nr_pr < 255`), starts the new process and transfers execution to the created process by sending a dummy value through `p_publish`. The newly created machine reads the dummy value and sends its channel back, thus transferring execution back to the parent. The parent reads the channel and the atomic sequence ends.

Another note-worthy detail of the translation is the definition of the machines' channels: In Spin, a single channel can only hold messages with the same fields. Though one could possibly convert every basic datatype into an integer, we chose to declare channels with the minimum number of fields for each type, so that every event type can be fit in without requiring conversions. With small messages, this should not waste too much space.

2.3.4 Notes on the Translation to C

The C backend uses the POSIX API for threads (*pthreads*) [1] to achieve concurrency among the machines; the semaphores [2] API is required for waiting for incoming events. Commonly used functionality (implementation of the buffers, creation of machines) is externalized in `runtime/Eventhandler.c`.

Each machine is simulated by a separate thread. Machines are translated into single functions were every state is a structured block of code marked by a label; switching states becomes a simple goto. Despite the use of goto, the resulting code is still somewhat structured in that the goto is always executed at the end of an event handling case. Overall, the translation is easy to trace.

The payload of an events is stored in a tuple/vector of void pointers, i.e. types are erased when the event message is composed, so casting to the right type is necessary when unpacking. See `runtime/Eventhandler.h` for the definition of the event structure.

The FIFO buffers of the machines are implemented as ring buffers. This was the most challenging part of the translation due to the concurrent accesses to the buffers. Waiting on a buffer for events is implemented by means of a semaphore; the semaphore's value indicates the number of messages in the buffer that have not been processed yet. Furthermore, every buffer has a mutex ensuring synchronized accesses.

References

[1] "<pthread.h>". In: *POSIX.1-2017*. The Open Group, 2018. URL: `pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html`.

[2] "<semaphore.h>". In: *POSIX.1-2017*. The Open Group, 2018. URL: `pubs.opengroup.org/onlinepubs/9699919799/basedefs/semaphore.h.html`.

[3] Ankush Desai et al. "P: safe asynchronous event-driven programming". In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 321–332. URL: `people.eecs.berkeley.edu/~ankush/assets/papers/p.pdf` (visited on 05/08/2019).

[4] *P on Github*. URL: `github.com/p-org/P` (visited on 05/08/2019).

[5] *Promela grammar*. URL: `spinroot.com/spin/Man/grammar.html` (visited on 05/08/2019).

[6] *Promela introduction*. URL: `spinroot.com/spin/Man/Manual.html` (visited on 05/08/2019).

[7] *Promela reference*. URL: `spinroot.com/spin/Man/promela.html` (visited on 05/08/2019).

[8] *Spin reference*. URL: `spinroot.com/spin/Man/` (visited on 05/08/2019).

[9] *SPIN web page*. URL: `spinroot.com` (visited on 05/08/2019).

3 Boogie

3.1 Introduction

3.1.1 What is the tool about?

Boogie is both an intermediate verification language (IVL) for contract based methods and a verifier for this language developed by Microsoft Research. Usually, a program written in some language would be translated to the Boogie IVL using a front-end tool, e.g. SMACK for C/C++. That resulting `.bpl`-file can then be extended by assertions and loop invariants for verification purposes. Using an SMT-solver (by default Z3), the tool Boogie can now be used to check the `.bpl`-file.

3.1.2 What can it be used for?

There are many possible application areas:

Program verification check given pre- and post-conditions of program blocks. Furthermore, loop invariant can be checked.

Invariant Synthesis There is work on implementing automated invariant synthesis (e.g. see section 3.3).

The following table shows which different programming languages can be translated to Boogie.

Front-End Tool	Source language	Restriction
SMACK	C/C++	single threaded
Spec#	C#	
VCC	C	concurrent with function specification
Dafny	language independent	
AutoProof	Eiffel(OOP)	
Joogie	Java	detection of unreachable code
Viper	Java,Python,Rust,OpenCL,...	
GPUVerify	OpenCL, CUDA(GPU kernels)	Race Conditions, Invalid use of memory batteries

3.1.3 How can you use the tool?

For a description of the syntax and semantic of the Boogie IVL, see [3]. For some use cases, see section 3.2.

Execution

As mentioned before, BOOGIE is written in the Dot-Net framework and has the suffix .exe (see */Binaries/Boogie.exe*). For the start of Boogie in Linux you need to call:

$$mono\ PathToExe/Boogie.exe\ file1.bpl\ file2.bpl\ ...\ fileN.bpl\ .$$

This call expects at least one *file.bpl*. If multiple files are given, it concatenates all files and tries to find a prove for all methods. Each procedure needs to have a unique name.
To search for some special parameter type:
mono PathToExe/Boogie.exe /help.
For some useful flags let us go over a simple example (*test.bpl*). The semantic of these statement we will explain in the section section 3.2.

```
1  procedure main(){
2
3    var x: bool;
4
5    assert x || !x;
6
7  }
```

Running it with the standard command
mono PathToExe/Boogie.exe test.bpl
yields the output: *Boogie program verifier finished with 1 verified, 0 errors.*

Tracing

The flag */trace* outputs details about the execution. For example,when the trace flag is present Boogie produces the following output for the example test.bpl:

```
1  Parsing test.bpl
2  Coalescing blocks...
3  Inlining...
4
5  Running abstract interpretation...
6    [0,035907 s]
7
8  Verifying main ...
9    [0,224 s, 1 proof obligation] verified
10
11 Boogie program verifier finished with 1 verified, 0 errors
```

This shows further information abouut how Boogie verifies the program

Additional command line options

Also interesting might be the following options:

- /traceTimes: Show the used time after the different steps of execution
- /tracePOs: output information about the number of proof obligations
- /log[:method]: Print debug output during translation
- /print *pathToFile/parseFile* print Boogie program after parsing it in *parseFile*

- Different prover options
  - /errorLimit <num> : Limit the number (num) of errors produced for each procedure. (Default 5)
  - /timeLimit <num>: Limit the number of seconds spent trying to verify each procedure
  - /prover <tp>: use theorem prover <tp>, where <tp> is either the name of a DLL containing the prover interface located in the Boogie directory, or a full path to a DLL containing such an interface. The standard interfaces contains: *SMTLib,Z3,Simplify,ContractInference*
  - /proverlog *pathToFile/file*: Log input for the theorem prover to *file*.
  - /proverWarnings <num>: 0 (default) - don't print, 1 - print to stdout, 2 - print to stderr

3.1.4 Tool Chain for C and C++ verification

In this project we worked on .bpl files to keep it simple. If you want to apply Boogie on C and C++ programs you should take a look on the Front-End tool SMACK[1]. As the solver under Boogie the Z3 solver is a good choices.

Z3

Z3 is a back-end SMT solvers for Boogie. It tries to find a satisfying model for the assertions of the Boogie file.
As reminder, the translated file for Z3 can be obtained by the parameter *–proverLog:log.txt*. This prints a log in log.txt in the current folder of the Boogie script.
For more details go to this Z3 tutorial.

Additional information

- Online Boogie interpreter
- official documentation site

3.2 Basic Features

In this section, we will demonstrate some basic features of the Boogie language by example. This examples can be found in the folder *common/boogie/IntroductionExamples/Basic_feature/*. First, simple sequential code is shown. Then, loops and the definition of manual loop invariants are presented. And finally, pre- and postconditions of procedures are discussed. This section is in no way a comprehensive showcase of all features of Boogie. For that, please refer to [3].

3.2.1 Sequential Code

BOOGIE is able to check assertions of straight-line code consisting of expressions, variable assignments or `if`-statements. An example would be the following simple code:

```
1 procedure main()
2 {
3   var x: int;
4   assume x == 1;
5   x := x + 2;
6   assert x == 3;
7 }
```

First, variable x is declared. Allowed types include `bool`, `int` and `real`. Then, the assumption statement expresses that it is assumed that x equals 1. Assumptions are not checked by Boogie and are simply believed to be true. Thus, after the assignment of $x + 1$ the property that $x = 3$ obviously holds. In order to check this using Boogie, we can run it using the following command (assuming the program is saved as `ex1.bpl`):

```
1 mono Boogie.exe -trace ex1.bpl
```

Running this command should yield the following output, telling us that our assertion was successfully verified:

```
1 Boogie program verifier finished with 1 verified, 0 errors
```

Under the hood, Boogie translates the given program into a SMT formula. In this example, the translated formula is

$$x_1 = 1 \rightarrow (x_2 = x_1 + 2 \rightarrow (x_2 = 3)).$$

Notice how each statement of the program implies the next one, while finally the assertion is implied by the result of the program execution under the defined assumptions. The formula is then dispatched to a SMT solver to prove its correctness. To illustrate the difference between assumptions and assertions, consider now changing the first assumption to an assertion:

```
1 procedure main()
2 {
3   var x: int;
4   assert x == 1;
5   x := x + 2;
6   assert x == 3;
7 }
```

Running BOOGIE on this example yields the following output:

```
1   [0.139 s, 2 proof obligations] error
2 ex2.bpl(5,3): Error BP5001: This assertion might not hold.
3 Execution trace:
4     ex2.bpl(5,3): anon0
5
6 Boogie program verifier finished with 0 verified, 1 error
```

BOOGIE tells us that the assertion on Line 4 might not always hold, which is clearly true: the assertion $x = 1$ does not hold for every integer x, e.g. $x = 0 \neq 1$. Notice also that BOOGIE now reports 2 proof obligations, one for each assertion in the program: so while in the first example BOOGIE just assumed $x = 1$ to hold, the condition was now actually checked. Next, let us consider a simple `if`-statement:

```
procedure main()
{
  var x,y: int;
  if(x < 0)
  {
    y := -x;
  } else
  {
    y := x;
  }
  assert y >= 0;
}
```

The program calculates the absolute value of x and puts it into y. Therefore, we expect the assertion $y \geq 0$ to always hold. Notice that in this example the condition of the `if`-statement $x < 0$ is necessary to show correctness of the property $y \geq 0$. Only if x is smaller 0, the sign of x must be flipped. Here, the translated SMT formula looks as follows:

$$((x_1 < 0) \rightarrow y_1 = -x_1 \rightarrow y_1 \geq 0) \wedge$$
$$((x_1 \geq 0) \rightarrow y_1 = x_1 \rightarrow y_1 \geq 0)$$

When a certain branch is executed, its corresponding condition can be assumed to be true. Running Boogie on this example confirms that the condition holds.

Furthermore, Boogie offers a special statement called `havoc`. The `havoc` statement sets the given variables non-deterministically to an arbitrary value of its corresponding type. Consider again our first example, with the following statements added:

```
procedure main()
{
  var x: int;
  assume x == 1;
  havoc x;
  x := x + 2;
  assert x == 3;
}
```

Without the `havoc`, we know that the assertion holds. Running Boogie shows that this is however not the case now:

```
  [0.141 s, 1 proof obligation] error
ex2.bpl(7,3): Error BP5001: This assertion might not hold.
Execution trace:
    ex2.bpl(4,3): anon0

Boogie program verifier finished with 0 verified, 1 error
```

The reason is that after x is assumed to be 1, x is havoced: setting it to an arbitrary value of type `int`. We are therefore trying to proof that $x = 3$ for any x, which is clearly not true. Usually, the `havoc` statement is used in conjunction with `assume` for induction invariants, setting variables to arbitrary values adhering to the given conditions.

Another way non-determinism can be introduced in Boogie is to use a wildcard $*$ in an `if`-statement:

```
procedure main()
{
  var x,y: int;
  if(*)
  {
    y := -x;
  } else
  {
    y := x;
  }
  assert y >= 0;
}
```

In this example, a non-deterministic choice between the branches is simulated. In essence, this means that no additional assumptions can be made through the IF condition itself. Therefore, the assertion $y \geq 0$ must be fulfilled by both branches without any additional properties. The corresponding SMT formula looks as follows:

$$(y_1 = -x_1 \rightarrow y_1 \geq 0) \wedge$$
$$(y_1 = x_1 \rightarrow y_1 \geq 0)$$

Clearly, this formula is not satisfiable. Running BOOGIE on this example confirms this. The missing constraint of the previous `if` example was indeed necessary to proof its correctness. The addition of non-determinism is however a powerful, useful modeling tool in many instances: usually, it is used to proof loops correct that run for an undetermined amount of iterations.

3.2.2 Loops and Loop Invariants

More interesting examples crop up when loops are considered. Traditional WHILE loops with boolean constraints can be written as known from C. Consider the following program:

```
procedure main()
{
  var i,n: int;
  assume(i == 0 && n >= 0);
  while(i < n)
  {
    i := i + 1;
  }
  assert(i == n);
}
```

While we expect the assertion to hold, BOOGIE is not able to proof it:

```
ex4.bpl(9,1): Error BP5001: This assertion might not hold.
Execution trace:
    ex4.bpl(4,1): anon0
    ex4.bpl(5,1): anon3_LoopHead
    ex4.bpl(5,1): anon3_LoopDone

Boogie program verifier finished with 0 verified, 1 error
```

The problem is that it is not a priori clear how often the loop will be executed, since this depends on the value of n. The assumption $n \geq 0$ does however not fully specify n. What we need is a loop invariant I: I has to hold before the loop, be inductive, and imply the assertion $i = n$ after the loop has ended. These

three conditions refer to the ones reported by BOOGIE (`anon0`, `anon3_LoopHead`, `anon3_LoopDone`). More specifically, we need the following:

$$(i = 0 \wedge n \geq 0) \rightarrow I(i, n) \text{ (precondition)},$$

$$(I(i, n) \wedge i' = i + 1) \rightarrow I(i', n') \text{ (inductiveness)}$$

and

$$(I(i, n) \wedge \neg(i < n)) \rightarrow i = n \text{ (postcondition)}.$$

Having such an invariant I would be sufficient to proof the assertion.

Generally, it is quite tough (even undecidable) to find such a loop invariant. The ultimate goal of this exposition is to use the assistance of automated tools to find the loop invariant, which we will do in section 3.3. We are however also able to manually provide a loop invariant. For now, let us therefore provide such a loop invariant for the given example:

```
1  procedure main()
2  {
3    var i,n: int;
4    assume(i == 0 && n >= 0);
5    while(i < n)
6    invariant i <= n;
7    {
8      i := i + 1;
9    }
10   assert(i == n);
11 }
```

Clearly, $i \leq n$ holds before the loop, as this is directly implied by the assumption. Also, since prior to every iteration $i \leq n$ holds, $i + 1 \leq n$ is true as well. Thus, we say that the invariant is inductive. And finally, since after the loop $\neg(i < n) \wedge (i \leq n)$ holds, $i = n$ is implied by the loop invariant. Using the loop invariant, BOOGIE is now able to proof the assertion correct.

3.2.3 Preconditions and Postconditions

BOOGIE supports procedures in order to improve modularity. A procedure can be annotated with preconditions and postconditions. Preconditions are defined using *requires* and must hold prior to calling the procedure. They are required to imply the postconditions. Postconditions are defined using *ensures*. These features can be used in order to decouple verification of procedures in inter-procedural analysis, improving modularity and smaller individual proof obligations. Consider for example the following procedure declaration:

```
1  procedure Add(x: int, y: int) returns (n: int)
2    requires x >= 0 && y >= 0;
3    ensures n == x + y;
4  {
5    n := x + y;
6  }
```

We require that x and y are greater than 0 and ensure that n equals $x + y$ after the procedure call, which is clearly true. Illustrating what happens under the hood, we can imagine that the procedure is translated to the following equivalent statements for verification:

```
havoc x;
havoc y;
assume(x >= 0 && y >= 0);
n := x + y;
assert(n == x + y);
```

And indeed BOOGIE proves this correct. The procedure requires x and y to be larger than 0, and if that is the case, guarantees that n will be the sum of both after the procedure call. Now imagine we have another procedure that calls *Add*:

```
procedure main()
{
  var n: int;
  var x: int;
  var y: int;
  x := 3;
  y := 7;
  call n := Add(x, y);
  assert(n == 10);
}
```

And indeed, BOOGIE can proof this correct. Again, illustrating what happens under the hood: since we know from the proof of the procedure itself, that the precondition implies the postcondition, the precondition of the function now needs to be asserted (verified), while the postcondition can then be assumed:

```
procedure main()
{
  var n: int;
  var x: int;
  var y: int;
  x := 3;
  y := 7;
  assert(x >= 0 && y >= 0);
  assume(n == x + y);
  assert(n == 10);
}
```

Consider now the following, slightly changed method call, adding -3 and 7:

```
procedure main()
{
  var n: int;
  var x: int;
  var y: int;
  x := -3;
  y := 7;
  call n := Add(x, y);
  assert(n == 4);
}
```

By looking at the procedure *Add*, we can see that it indeed calculates $-3 + 7 = 4$ and $n = 4$ will hold. But since BOOGIE verifies both procedures independently, it is not able to proof this. The precondition assertion for the procedure call in *main* will fail, since we required $x \geq 0$ in *Add*:

```
ex6.bpl(15,3): Error BP5002: A precondition for this call might not hold.
ex6.bpl(2,3): Related location: This is the precondition that might not hold.
Execution trace: ex6.bpl(13,5): anon0
```

```
4
5 Boogie program verifier finished with 1 verified, 1 error
```

In order to proof this, we could simply drop the precondition $x \geq 0$.

3.3 Invariant Synthesis

A special application for BOOGIE is the automated search for loop invariant. As we have seen, loop ivariants play an important role. However they are hard to write manually There are several approaches possible. Here you will get a insight of three learning algorithm for this task: *Decision Tree learner*[2], *Houdini* and *Sorcar*[4]. For quick use of the tools you can insert example loops in this website created by Dr. Daniel Neider. All three algorithms operate in thhe so-called Horn-ICE framework[2], which we sketch next.

3.3.1 Horn ICE

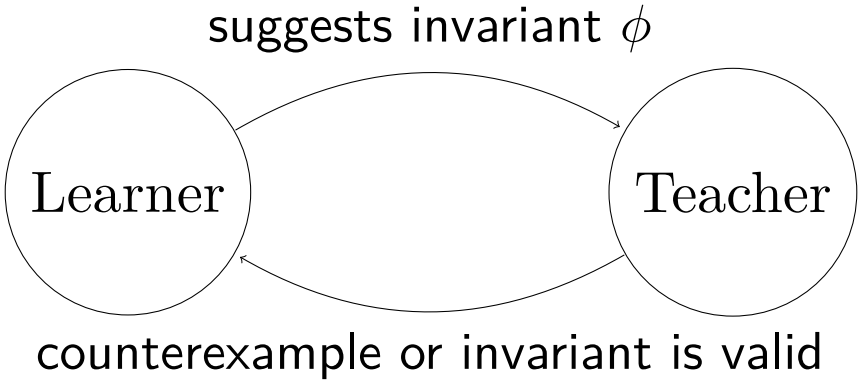


Figure 3.1: Learner-Teacher-Model

All this Algorithm are implemented in the Horn ICE Framework which is a general learning framework for inductive invariant. For the sake of simplicity, let us assume that our program has a single loop and we need to synthesize a single loop invariant. The horn-ICE framework is more flexibl, though and can handle multiple annotations at different places in the code. The invariant that will be tried to learn are one arbitrary formula over a set of predicates P over program varibales.

the horn ice framework implements a learner and a teacher entity and operates in a loop. The learner suggests a formula ϕ to the teacher who knows the implementation of the loop. It puts the invariant in a Boogie program which represent the loop. Then Boogie says if the invariant proves the loop or gives back a counterexample. This counterexample can come in three different types:

- If the pre-condition α of the program does not imply ϕ, then the teacher returns a *positive counterexample* $c \in C$ such that $c \vDash \alpha$ (c satisfies α) but $c \nvDash \phi$.
- If ϕ does not imply the post-condition β of the program, then the teacher returns a *negative counterexample* $c \in C$ such that $c \vDash \phi$ but $c \nvDash \beta$.
- If ϕ is not inductive, then the teacher returns a *Horn counterexample* $(\{c_1, c_2, ..., c_n\}, n) \in 2^C \times C$ such that $c_i \vDash \phi$ for each $i \in \{1, ..., n\}$, but c $\nvDash \phi$. This can be understood as $(c_1 \wedge c_2 \wedge ...c_n) \rightarrow c)$

The learner might adjust his invariant depending on the counterexample and gives it to the teacher. If no invariant can be found that is consistent to all counterexample the learner finishs the learning attempt. This could mean that the pre- and postcondition requested on the loop are faulty. If the suggested invariant holds for the program the teacher also say so. But there could also hold other maybe shorter invariants.

3.3.2 Decision tree learner

The decision tree learner can learn a invariant that is a arbitrary boolean formula over the set of pradicates P. In addition, it accepts terms (e.g. x+15) for which it synthesize predicates automatically.

Decision tree

For this algorithm the different configuration of the loop program will be considered as data points *d*. A data point can be understood as vectors over the configuration of the single variables and temporary results of the this variables.
This algorithm uses a binary decision tree to store given counterexample.
The *internal nodes* represent a decision over a data point *d* which holds in this decision node or not.
The *leaves* can be assigned to *true*, *false* or *unlabeled*. A path from the root to a true leaf represents a set of datapoints decisions that form a conjunction for invariant suggestion ψ_T. ψ_T is the disjunction over each path from root to a true leaf. This invariant will be given to the teacher if it is not contradicting to the received counterexample.

Algorithm

The algorithm itself starts with a single unlabeled leaf as decision tree. It is associated to each occurring datapoint.
Each step the learner searches a unlabeled leaf and checks if has only pure positive and unsigned datapoints or pure negative and unsigned datapoints associated with it. If this is the case this leaf it will be assigned *true* for pure positive or *false*. Otherwise the leaf get replaced be a internal node which a leaf for the positive data points and a the other leaf for the other data points.

Example

For demonstration let us look on this program:

```
1  function {:existential true} inv(x : int) : bool;
2
3  procedure main()
4  {
5
6    var x: int;
7
8    assume x == 0;
9
10   while (x < 4)
11   invariant inv(x);
12   {
13     x := x + 2;
14   }
15   assert x == 4;
16
17 }
```

Before executing this,two new keywords should be mentioned.
existential true tells Boogie that this function is to be infered/learned. It is usually used in conjunction with the invariant statement.
And as a reminder *invariant* enforce that the invariant is checked for the pre-,postcondition and inductiveness condition. So lets execute the decision tree learner with this program. It is recommended to change the directory to */Boogie/Binaries/*. You call the invariant syntheses with:

```
mono Boogie.exe /trace /nologo /noinfer /contractInfer /mlHoudini:./hice-dt
 /learnerOptions:"-b" ../../benchmarks/ownExample/loopInvariants2.bpl
```

The important parameter is *-mlHoudini:./hice-dt* determine that the horn-ICE framework will be chosen. *contract Infer* allows the invarianten syntheis by algorithms and *noInfer* disables heuristics of Boogie. There are two *lerarnerOptions* for *hice-dt*: with *-b* the decision tree learner will be used as the algotithm, *-h* run the first houdini algorithm pre-phase.
As easily seen the invariant should be $\phi = x == 0 || x == 2 || x == 4$ The Output shows you the different suggested invariant for *inv(x)*.

```
1.
function {:existential true} {:inline} inv(x: int) : bool
{
  true
}
Prover Time taken = 0,009853
Added: inv:5: negative
2.
function {:existential true} {:inline} inv(x: int) : bool
{
  false
}
Added: inv:0: positive
3.
function {:existential true} {:inline} inv(x: int) : bool
{
  x <= 0
}
Prover Time taken = 0,004695
Added Horn clause: 1 => 2
```

The first 3 rounds should look like above. After 8 rounds it stops with the invariant:

```
(-1 < x && x <= 0) || (-1 < x && 0 < x && 1 < x && x <= 2)
|| (-1 < x && 0 < x && 1 < x && 2 < x && 3 < x && x <= 4)
```

. This formula is equivalent to the invariant we came up with.

3.3.3 Houdini

The new version of the Houdini[4] Algorithm tries to learn conjuncts as a invariant. Therefore it can only work with boolean variables. Such a invariant ϕ is of this form

$$\phi = p_{i_1} \wedge p_{i_2} \wedge ... \wedge p_{i_n} \text{ with for each } i \in \{1, ..., n\} : p_{i_j} \in P$$

The learner has 3 sets to saves all three kind of counterexample $S = (S_+, S_-, S_H)$. The learner always suggest the biggest conjunct $X \subseteq P$ that is consistent with S through this loop:

1. start with X=P
2. remove for a positive counterexample the predicates which contradict the positive counterexample.

3. for a Horn Counterexample $(c_1 \wedge c_2 \wedge ...c_n) \rightarrow c)$ add c to S_+. Here all predicates c_i satisfy the invariant but c does not.
4. Repeat Steps 2 and 3 until a fixed point is reached.
5. At the end it checks if X is consistent with $S\_$. If yes the invariant is found, else there exists no conjunct invariant.

Because with each counterexample the learner can eliminate at least one predicate it has a runtime of $O(|P|)$.

Example

As an example for Houdini we can not take the same program because the invariant for *Houdini* and *Sorcar* may only contain boolean parameter. We also need to come up with predicates that are literals for the conjunctive invariant. Let us observe the following example:

```
function {:existential true} inv1 (x0 : bool,x1 :bool,x2 :bool,
 x3: bool, x4: bool, x5: bool) : bool;

procedure main()
{

  var x: int;

  assume x == 0;

  while (x < 4)
  invariant inv1(x<4,!(x==1),!(x==3),x<5,(x==0||x==2||x==4),x>0);
  {
    x := x + 2;
  }
  assert x == 4;

}
```

It is the same loop as the decision tree learner, so that the invariant is still valid. six predicate are suggested here for the invariant. The fifth predicate would be a invariant on its own. But it is also valid to add some overlapping predicates to it. For the execution again go into *Boogie/Binaries* and call this time:

```
mono Boogie.exe /trace /nologo /noinfer /contractInfer /mlHoudini:../../sorcar/src/sorcar
/learnerOptions:"-a horndini" ../../benchmarks/ownExample/loopInvariantsBool.bpl
```

There might occur some exception after the first round of the Horn ICE framework with only */mlHoudini:./sorcar*. Therefore it is better to use the whole relative path to the algorithm For more information of the learneroption remove the *horndini* and add /help. You will get a overview over the learneroptions. Now the output:

```
Verifying main:
function {:existential true} inv1(x0: bool, x1: bool, x2: bool, x3: bool, x4: bool, x5: bool) : bool
{
  true
}
Added: inv1:0,1,1,0,0,1: negative
Prover Time taken = 0,092246
Calling ../../sorcar/src/sorcar -a horndini ../../benchmarks/ownExample/loopInvariantsBool.bpl
Total learner time was 00:00:00.0050370
Verifying main:
```

```
function {:existential true} inv1(x0: bool, x1: bool, x2: bool, x3: bool, x4: bool, x5: bool) : bool
{
  x0 && x1 && x2 && x3 && x4 && x5
}
Added: inv1:1,1,1,1,1,0: positive
Prover Time taken = 0,003857
Calling ../../sorcar/src/sorcar -a horndini ../../benchmarks/ownExample/loopInvariantsBool.bpl
Total learner time was 00:00:00.0043710
Verifying main:
function {:existential true} inv1(x0: bool, x1: bool, x2: bool, x3: bool, x4: bool, x5: bool) : bool
{
  x0 && x1 && x2 && x3 && x4
}
Prover Time taken = 0,005973
Added Horn clause: 2 => 3
Calling ../../sorcar/src/sorcar -a horndini ../../benchmarks/ownExample/loopInvariantsBool.bpl
Total learner time was 00:00:00.0032880
Verifying main:
function {:existential true} inv1(x0: bool, x1: bool, x2: bool, x3: bool, x4: bool, x5: bool) : bool
{
  x1 && x2 && x3 && x4
}
Number of positive examples:1
Number of negative examples:1
Number of Horn clauses:1
verified
```

The usual Houdini start invariant can be seen in the second round where all predicates are assumed to be true. After each round you can inspect the current suggestion for the invariant. Note that *true* demands no predicate to be *true* for being a valid invariant an this would allow any evaluation of the corresponding predicate. After the first positive counterexample $x > 0$ gets removed because $x == 0$ is possible. The horn clause counterexample can only be understood with the next invariant suggestion because the right-hand side of the implication is treated as a positive counterexample. After the 4 round of the horn-ICE framework a invariant is found. This invariant has the lagest number of predicates.

3.3.4 Sorcar

The Sorcar algorithm[4] is a improvement of the Houdini algorithm. A disadvantage of Houdini is that it is not property driven and so it might not regard all asserts of the program. The newer algorithm solves this problem. Furthermore, Sorcar uses the negative counterexample for adjustment of the invariant.
Sorcar starts in the procedure `Sorcar-Iterative` with the empty set of predicates as a invariant. It calls the passive step where it collects counterexample and extend the invariant in the Relevant-Predicates Function by at least one new predicate. After a fix point is reach but the invariant doesn't verify the program the Horn-ICE Framework calls the starting method `Sorcar-Iterative` again.
The worst case amount of rounds of the `Sorcar-Iterativ` is also linear limited with $2 * n$. So it might need up to twice as much runtime but the invariant tends to be more general.

Example

Here we can use the same example as for Houdini. Only the learneroption changes to*/learnerOptions:"-a sorcar"*.

The output shows a improvement of the sorcar algorithm:

```
Verifying main:
function {:existential true} inv1(x0: bool, x1: bool, x2: bool, x3: bool, x4: bool, x5: bool) : bool
{
  true
}
Added: inv1:0,1,1,0,0,1: negative
Prover Time taken = 0,085295
Calling ../../sorcar/src/sorcar -a sorcar ../../benchmarks/ownExample/loopInvariantsBool.bpl
[0, 0, 1, 1, 0, 0, 1] -> 0
Total learner time was 00:00:00.0045960
Verifying main:
function {:existential true} inv1(x0: bool, x1: bool, x2: bool, x3: bool, x4: bool, x5: bool) : bool
{
  x0 && x3 && x4
}
Prover Time taken = 0,004576
Added Horn clause: 1 => 2
Calling ../../sorcar/src/sorcar -a sorcar ../../benchmarks/ownExample/loopInvariantsBool.bpl
Total learner time was 00:00:00.0036420
Verifying main:
function {:existential true} inv1(x0: bool, x1: bool, x2: bool, x3: bool, x4: bool, x5: bool) : bool
{
  x3 && x4
}
Number of positive examples:0
Number of negative examples:1
Number of Horn clauses:1
verified
```

The learner can starts with no predicate selected for the invariant. It reacts then to a negative example. And even after the third round a different invariant with less predicates is found. This was easy to expect here because sorcar includes until it is consistent and even the fifth predicate alone is a invariant.

References

[1] Montgomery Carter et al. “SMACK software verification toolchain”. In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2016, pp. 589–592.

[2] P Ezudheen et al. “Horn-ICE learning for synthesizing invariants and contracts”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), p. 131.

[3] K. Rustan M. Leino. “This is Boogie 2”. June 2008.

[4] “Sorcar: Property-Driven Algorithms for Learning Small Conjunctive Invariants”. Preprint provided by Daniel Neider.

4 GRASShopper

4.1 What is the tool about?

GRASShopper is a verification tool for heap-manipulating programs against user-provided specifications.[2] The tool further checks that memory accesses are valid (the respective memory locations are allocated) and no memory is leaked. Specifications are formulated using seperation logic enriched with a decidable theory of singly linked lists.[1] Under the hood, GRASShopper uses the SMT solver Z3.

4.2 What can it be used for?

The tool is suited to verify the memory safety of programs using data stuctures like lists or trees (graphs are probably out of reach). Both iterativ and recursive implementations can be verified.

4.2.1 How can you use the tool?

The intended workflow is to write the methods in the GRASShopper language first and then find the right specification until pre- and postconditions hold. By default, GRASShopper will check the specifications of all procedures of the input: `grasshopper.native <file.spl>`. To verify only one method of the file, add a parameter `-procedure <method>`. With `-v`, GRASShopper will print a more verbose description of the verification result. Finally, GRASShopper can produce C code for the program: `grasshopper.native <file.spl> -compile <file.c>`. For more options, run `grasshopper.native --help`.

4.3 Language Basics

The language is–much like the Boogie IVL–a simplified version of C. Unlike Boogie however, it has dynamic memory and allows the definition of new datastructures with the keyword `struct`. Structures are referenced by pointers and allocated dynamically using `new`; deallocation is done with `free`. Pointers are type checked, i.e. no unsafe casting is allowed. Further, no arithmetic is allowed on pointers. Pointers can however be invalid, i.e. when `null` or when the referenced memory was deallocated.

As said above, specifications use separation logic (see [3] for the original paper). Seperation logic is an extension of Hoare logic that can express properties of the heap. Where in Hoare logic one would only speak about the state of the program, separation logic also considers the state of the heap.

Since GRASShopper uses a strict interpretation of separation logic, it is possible to speak of the memory footprint of a formula and thus avoid the rather cumbersome formal definition. E.g. the statement `acc(x)` states that the structure referenced by the variable `x` is allocated in memory. Formally, `acc(x)` is satisfied when the heap is valid only at `x`, i.e. no other memory must be allocated. Thus, the memory footprint of `acc(x)` would be just `x`.

The classical operators of separation logic are seperating conjuction (written as `&*&` in GRASShopper) and separating implication (`-**`). A separating conjunction states that both sides are true on separate portions of the current heap. A separating implication states that, when the left hand side holds on an extension of the current heap, the right hand side has to hold on the extended heap. In other words, a separating conjunction is satisfied when both sides are satisfied and their memory footprints are disjoint. The memory footprint of the separating conjunction would then be the union of the footprints of the two sides. A separating implication is satisfied when it is satisfied as implication and the memory footprint of the left hand side is a subset of the footprint of the right hand side. The footprint of the implication would then be the footprint of the right hand side without the footprint of the left hand side.

The strictness of GRASShopper might be best seen when comparing separating conjunction (`&*&`) and usual conjunction (`&&`). Where separating conjunction requires that the footprints are disjoint, usual conjunction will demand that the footprints are identical. Since some statements like `Btwn` and `Reach` have an empty footprint, one would usually combine them with `&*&` instead of `&&` since the latter would require that the other side have empty footprint as well. Although this strict interpretation can result in rather technical specifications, it will make memory leaks explicit and can further be used to impose further restrictions on the interface of a method.

The statement `emp` has empty footprint, i.e. no memory is allocated at all. The operator `x.p |-> y` states that `x` is allocated in memory and the field `p` of `x` points to `y`. One can probably write down an equivalent statement using only `acc` and `&*&`.

Besides plain separation logic, GRASShopper implements a decidable theory of singly linked lists. The central statement of this theory is `Btwn(p, x, y, z)` stating that `z` can be reached by following the `p` field of `x` and `y` lies on the path. There is also an analogous statement that requires `y` to not lie on the path. The statement `Reach(p, x, y)` stating that by following the `p` entries, one can reach `y` from `x` is thus equivalent to `Btwn(p, x, x, y)`.

Moreover, the operator `acc` can be used with sets: `acc({x: T :: c})` where `T` is the name of a structure states that the entire set $\{x \in T \mid c\}$ is allocated. Under the hood, the set will be interpreted as something like $\forall x \in T.c \rightarrow \text{acc}(x)$, i.e. first-order logic.

4.4 Example

A node of a singly linked list could be defined like so:

```
1 struct Node {
2   var next: Node;
3 }
```

The following predicate describes a segment of a singly linked starting at `x` where the last element has to point to `y`:

```
1 predicate lseg(x: Node, y: Node) {
2   acc({ z: Node :: Btwn(next, x, z, y) && z != y }) &*& Reach(next, x, y)
3 }
```

The first part `acc()` states that all elements of the segment except for `y` have to be allocated. I.e. lists will be `null`-terminated. The second part is needed to ensure that the segment is connected. Note the use of `&*&` instead of `&&`. The latter would force the list to be empty.

Singleton lists can be created as follows:

```
procedure create_sl()
  returns (lst: Node)
  requires emp
  ensures lseg(lst, null)
{
  lst := new Node;
  lst.next := null;
}
```

The empty list would be represented by a `null` pointer.

To dispose of a list:

```
procedure dispose_sl(lst: Node)
  requires lseg(lst, null)
  ensures emp
{
  while (lst != null)
    invariant lseg(lst, null)
  {
    var curr := lst;
    lst := lst.next;
    free(curr);
  }
}
```

Note the use of `emp` to state that the heap has been cleared.

To concatenate two singly linked lists, one has to first find the end of the first list:

```
procedure concat_sl(a: Node, b: Node)
  returns (res: Node)
  requires lseg(a, null) &*& lseg(b, null)
  ensures lseg(res, null)
{
  if (a == null) {
    return b;
  } else {
    var curr := a;
    while (curr.next != null)
      invariant acc(curr) -** lseg(a, null)
    {
      curr := curr.next;
    }
    curr.next := b;
    return a;
  }
}
```

By adding a pointer to the tail, the traversal of the first list would become obsolete. We need the following predicates for tailed lists:

```
// tail list segment
predicate tlseg(x: Node, y: Node, z: Node) {
  (lseg(x, y) &*& y.next |-> z)
}

// tail list (null terminated)
predicate tlist(x: Node, y: Node) {
  tlseg(x, y, null)
}
```

Note that tailed lists cannot be empty anymore since we need to specify the tail.

Concatenation now becomes very simple:

```
procedure concat_tl(lst1: Node, tail1: Node, lst2: Node, tail2: Node)
  returns (lst3: Node, tail3: Node)
  requires tlist(lst1, tail1) &*& tlist(lst2, tail2)
  ensures tlist(lst3, tail3)
{
  tail1.next := lst2;
  lst3 := lst1;
  tail3 := tail2;
}
```

Methods for removing or inserting elements are however complicated as they now have to get head and tail pointers right.

References

[1] Ruzica Piskac, Thomas Wies, and Damien Zufferey. "Automating separation logic using SMT". In: *International Conference on Computer Aided Verification*. Springer. 2013, pp. 773–789.

[2] Ruzica Piskac, Thomas Wies, and Damien Zufferey. "Grasshopper". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2014, pp. 124–139.

[3] John C Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2002, pp. 55–74. URL: `http://www.cs.cmu.edu/~jcr/seplogic.pdf`.