Software Model Checking

Exercises

Assignment 2

Georg Weissenbacher
(credit: Henning Goes)
184.747

What you have to do

1. Translate a small C program into a transition relation
2. Write simple a bounded model checker
    - Input: A transition relation and an unwinding depth N
    - Output: Is there a bug within N steps of the transition relation?

Encoding the transition relation

The encoding is based on *SMTLib* syntax:

```
(program
  (state (x Int))
  (init (= x 0))
  (transition (= x_ (+ x 1)))
  (property (< x 10))
  )
```

Encodes: $I(x) = (x = 0), T(x, x') = (x' = x + 1), P(x) = x < 10$

See `http://smtlib.org` for the syntax & semantics of expressions & types.

How to talk to an SMT solver

User/Application	**Solver**
"I want to talk about the following variables: x which is an integer and c which is boolean."	"Ok."
"The formula $x > 5 \wedge c$ holds."	"Ok."
"Can you find a solution?"	"Yes, I can!"
"What is the value of x?"	"It's 20!"

How to talk to an SMT solver

User/Application	Solver
(declare-fun x () Int) (declare-fun c () Bool)	
"The formula $x > 5 \wedge c$ holds."	"Ok."
"Can you find a solution?"	"Yes, I can!"
"What is the value of x?"	"It's 20!"

How to talk to an SMT solver

User/Application	**Solver**
(declare-fun x () Int)	
(declare-fun c () Bool)	
(assert (and ($>$ x 5) c))	
“Can you find a solution?”	“Yes, I can!”
“What is the value of x?”	“It’s 20!”

How to talk to an SMT solver

User/Application	**Solver**
(declare-fun x () Int)	
(declare-fun c () Bool)	
(assert (and ($>$ x 5) c))	
(check-sat)	sat
“What is the value of x?”	“It’s 20!”

How to talk to an SMT solver

User/Application	**Solver**
(declare-fun x () Int)	
(declare-fun c () Bool)	
(assert (and ($>$ x 5) c))	
(check-sat)	sat
(get-value (x))	((x 20))

SMTLib – Basics

The SMTLib format is based on *s-expressions* (made famous by *lisp*). An expression can be:

- A constant: 10, `true`, etc.
- A function application: $(f\ arg_1 \ldots arg_n)$

For example, the expression $x = y \cdot z$ would be represented as:
`(= x (* y z))`

SMTLib – Types

SMTLib supports various types:

- Booleans: `Bool`
- Natural numbers: `Int`
- Bitvectors of length *n*: `(_ BitVec n)`
- Arrays with index type *i* and element type *e*: `(Array i e)`

Two ways to introduce new variables:

- Introduce a fresh unconstrained variable:
  `(declare-fun f (Int Bool) Int)`
  declares *f* to be an uninterpreted function with two arguments.
- Define an alias for a function:
  `(define-fun f ((x Int) (y Int)) Int (* x y))`
  defines *f* to be an alias for the multiplication function.

SMTLib – Theories

SMTLib supports multiple theories which provide you with function symbols you can use:

- Core: Boolean constants, `and`, `or`, `not`, etc.
- Bitvectors: Constants: `(_ bv5 32)`, `bvadd`, `concat`, etc.
- Integers: Integer constants, `+`, `*`, etc.
- Arrays:
  - `(store a i e)` puts element *e* into array *a* at index *i* and returns the new array.
  - `(select a i)` retrieves the element at index *i* from array *a*.

For a full list of all theories and logics, see `http://smtlib.org`

Translating C-programs

```
void sum(int n) {
  int i = 0;
  int r = 0;
  while(i<=n) {
    r = r + i;
    i = i + 1;
  }
  assert(r < 100);
}
```

Translating C-programs

Step 1: Label program locations

```
void sum(int n) {
  int i = 0;
  int r = 0;          //0
  while(i<=n) {       //1
    r = r + i;        //2
    i = i + 1;        //3
  }
  assert(r < 100);    //4
}
```

Translating C-programs

Step 2: Model the transition relation using a program counter

```
void sum(int n) {
    int i = 0;
    int r = 0;               //0
    while(i<=n) {            //1
        r = r + i;           //2
        i = i + 1;           //3
    }
    assert(r < 100);         //4
}
```

$pc = 0 \Rightarrow i' = 0 \wedge r' = 0 \wedge n' = n \wedge pc' = 1$
$pc = 1 \wedge i \leq n \Rightarrow r' = r \wedge i' = i \wedge n' = n \wedge pc' = 2$
$pc = 1 \wedge i > n \Rightarrow r' = r \wedge n' = n \wedge pc' = 4$
$pc = 2 \Rightarrow r' = r + 1 \wedge i' = i \wedge n' = n \wedge pc' = 3$
$pc = 3 \Rightarrow r' = r \wedge i' = i + 1 \wedge n' = n \wedge pc' = 1$
$pc = 4 \Rightarrow pc' = 5$
$pc = 5 \Rightarrow pc' = 5$

See `tests/sum.l` for the complete solution.

Exercise 1

Translate the following program:

```
void gcd(int starta,int startb) {
  int a = starta, b = startb;
  int t;
  while(b!=0) {
    t = b;
    b = a % b;
    a = t;
  }
  assert(starta % a == 0);
  assert(startb % a == 0);
}
```

MathSAT

We're using *MathSAT 5* as the SMT solver for this exercise:

- Download from `http://mathsat.fbk.eu/`
- Available for Linux, Mac and Windows.

Getting the code

The code template is hosted on GitHub:

- `git clone https://github.com/hguenther/smc.git`
- This makes it easier to distribute bug fixes
- If you don't want to install git:
  - Goto `https://github.com/hguenther/smc`
  - Click "Download ZIP"

Compiling the code

Code is compiled using *CMake*.

- Makes it possible to use Makefiles (Linux) or Visual Studio (Windows) or your favorite (supported) build-tool.
- Get *CMake* from `http://cmake.org`

For Makefiles:

```
mkdir build
cd build
cmake ..
make
```

See `cmake --help` for a list of other generators available for your platform.

Integrating MathSAT with CMake

It might be tricky to let CMake know about MathSAT:

- You can tell CMake where to look for MathSAT by using:

  ```
  cmake . -DMATHSAT_PREFIX=/path/to/mathsat
  ```

- Please contact me if there are problems:
  `weissenb@forsyte.at`

Creates an abstraction of the MathSAT C-interface.

- Initialize it by giving it the name of the logic we are using:

```
IMathSAT iface("QF_LIA");
```

- Provides simplified functions: *check_sat* checks if the instance is satisfiable etc.

program.h

Provides a simple parser for the transition relation format:

- Creating the parser:

```
int fd = open("myfile",O_RDONLY);
Program prog(fd);
```

- *vars*: A mapping of variable names to variable types.
- *inits*: A vector of initial conditions.
- *trans*: A vector of expressions forming the transition relation.
- *props*: A vector of properties that have to be checked.

All types and expressions are unparsed S-expressions.

parser.h

Provides a way to parse expressions and types into SMTLib format.

- Initialization:

```
IMathSAT iface("QF_LIA");
SMTLibParser<IMathSAT> parser(iface);
```

- *add_named_term*: Tell the parser that a certain name is bound to a given term:

```
msat_term x = ...;
parser.add_named_term("x",x);
```

- *parse_term*: Parse an S-expression into an SMTLib expression.
- *parse_type*: Parse an S-expression into an SMTLib type.

bmc_exercise.h

This is the template for the "BMC" class that you have to complete.

- A few helper functions should give you an idea of how to structure your code.
- To complete the exercise, you have to implement:
  - *extend*(): This function adds a new state to the vector of states and asserts that it is the successor state of the previous last state.
  - *check*(): Uses the SMT solver to find out if the BMC instance is solvable.

Solution Submission

- All solutions are due **February 28, 2021**.
- Please contact me if you need more time.
- Submit using the TUWEL system.
- Plan enough time for setting up everything.