SABS : Spark ABStraction
A Tutorial

Victor Cacciari Miraldo

Universidade do Minho

1 Introduction

SABS is a predicate abstraction laboratory that is beeing developed at University of Minho, Portugal. Our goal is not to produce a industrial software model checker, such as SLAM [BMR01] or SATABS [CKSY05], but to have a tool to study and compare the diferent techniques (and combination of techniques) that can be used to perform the predicate abstraction of a program, in our case, a SPARK program.

This document is a both a tutorial on the usage of SABS and a (small) explanation of its implementation. Some knowledge on Predicate Abstraction and Program Verification is assumed, we refer the reader to [MLPF13] for some background on the techniques implemented by SABS.

2 Installation

SABS is obtainable through the AVIACC project website[1], *tools* section. After downloading the source, the folowing commands should configure and build SABS on most systems:

```
  cabal configure; cabal build
```

Assuming the repository is cloned under a path p, the resulting executable will be in $p/dist/build/sabs/sabs$.

Note that SABS is part of a bigger project, a Spark bounded model checker is also included in this package, for more information on SPARK-BMC check [Lou13].

3 Command Line Interface

The command line interface is pretty straight forward. The `--help` argument has the following output:

[1] `http://wiki.di.uminho.pt/twiki/bin/view/Research/Aviacc/WebHome`

```
sabs
  -i FILE        --input=FILE                       Input file
  -e PROC        --entry=PROC                       Entry Point for Abstraction
  -c             --cover-only                       Compute covers, only
                 --full-cover                       Do not simply the covers of a direct abstraction
  -b             --boolean-program                  Compute a Boolean Program, Entry point must be specified.
                 --basicblock                       Reads the program from a basicblock file
                 --bbheuristic= direct | cartesian  Set's the basic-block heuristic to be used
                 --cfheuristic= trivial | slam      Set's the controw-flow heuristic to be used
  -s ints | bvs  --solver=ints | bvs                Set's the solver to be used while abstracting
  -V             --version                          Print version
  -h             --help                             Show help
```

More options are available through annotations inside the code, such options will be presented and discussed in chapter 6.

4 Architecture

In order to understant the provided options, some knowledge of the architecture is needed. Figure 1 describes the important components. Blue labels represent input/output, gray labels are intermediate representations for data, dashed lines are used to represents plug-in components.

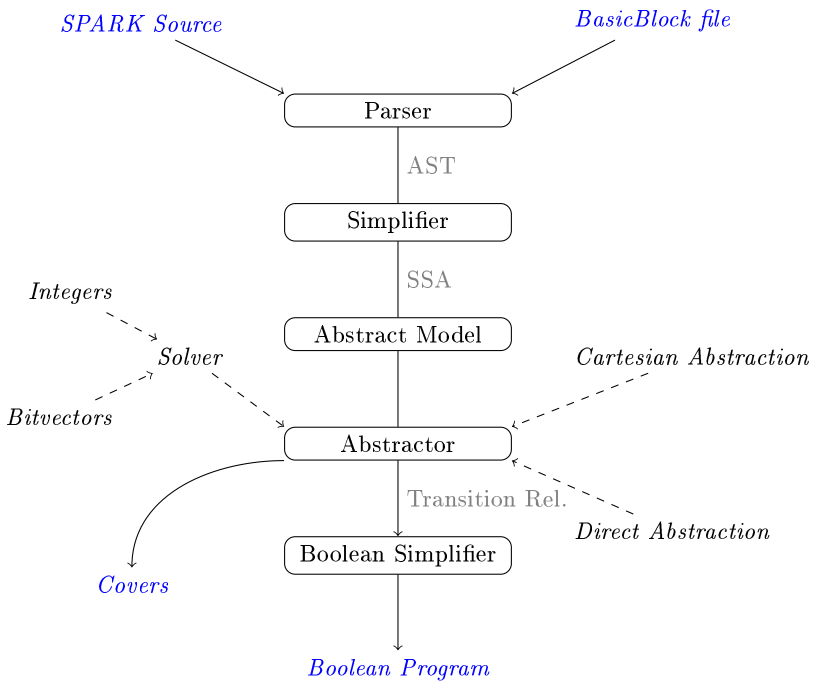

Fig. 1. SABS architecture

5 Using SABS

This chapter will present a high-level overview of the input/output options together with some output formating options. It aims at providing the user with what is needed for constructing abstractions.

SABS accepts input in two forms: (A) a SPARK package body file or (B) a BasicBlock (bb) file. A bb-file consists in a list of variable declarations, a basic-block and instrumentation code. For instance:

Listing 1.1. Sample BasicBlock file

```
x : INTEGER;
y : INTEGER;

x := 3;
y := y + 1;
--% assert y <= x;
--% assert y mod 2 = 0;
--% assert y >= 0;
--% assert x >= 0;
```

In order to input a bb-file to SABS we should invoke it with the option `--basicblock` . If we wish to input a SPARK package body we just need to specify the entry point of the abstractor, that's done with the option `-e` . The simplest way to invoke SABS, then, is:

```
sabs  --basicblock -i myfile.bb
```

or

```
sabs -i mypackage.adb -e myfunc
```

For the rest of this document, I'll use the file tutorial.bb to refer to the code presented in listing 1.1.

5.1 Producing Boolean Programs

The most direct usage of SABS is to produce a boolean program of a given piece of code. The option `-b` or `--boolean-program` does just that. For instance, to produce a boolean program out of our tutorial program the following command would suffice:

```
sabs --boolean-program -i tutorial.bb
```

5.2 Producing Covers

In order to produce a boolean program, we have to produce a transition relation first, and then simplify it into a boolean program. It's possible to get this intermediate output from SABS with the flag `-c` or `--cover-only` . Obviously, the output will depend on the abstraction used (direct or cartesian) and as of the current version, this option is only allowed when the input is a bb-file.

Covers for Cartesian Abstraction Running the command:

```
sabs -i tutorial.bb --basicblock --bbheuristic=cartesian --cover-only
```

will output:

```
//b1 : (y <= x)
//b2 : ((y mod 2) = 0)
//b3 : (y >= 0)
//b4 : (x >= 0)
>>> 1
ON : F
OFF: (!b3 & !b4) | (b1 & !b3)
UND: (!b1 & b3) | (b3 & b4)
>>> 2
ON : (!b1 & b2 & !b4) | (!b1 & b2 & b3) | (b1 & b2 & !b3) | (b1 & b2 & b4) | (b2 & !b3 & !b4) | (b2 & b3 & b4)
OFF: (!b2 & !b3 & !b4) | (!b2 & b3 & b4) | (!b1 & !b2 & !b4) | (!b1 & !b2 & b3) | (b1 & !b2 & !b3) | (b1 & !b2 & b4)
UND: F
>>> 3
ON : (b1 & b2 & !b3) | (b2 & !b3 & !b4)
OFF: (!b1 & b3) | (b3 & b4)
UND: (!b2 & !b3 & !b4) | (b1 & !b2 & !b3)
>>> 4
ON : F
OFF: (!b3 & !b4) | (!b1 & !b4) | (!b1 & b3) | (b1 & !b3) | (b1 & b4) | (b3 & b4)
UND: F
[ 128 solver calls ]
```

The given output is, in fact, the disjunction $S^{wp(P,E(l))}$ for every literal l, where P is the basic block in the input file. Let's analyze the covers associated with the variable $b1$:

```
>>> 1
ON : F
OFF: (!b3 & !b4) | (b1 & !b3)
UND: (!b1 & b3) | (b3 & b4)
```

The *ON* part corresponds to $S^{wp(P,E(b_1))} - UNDET$, *OFF* is $S^{wp(P,E(\neg b_1))} - UNDET$ and *UND* is equivalent to the UNDET-cover, formaly $S^{wp(P,E(b_1))} \cap S^{wp(P,E(\neg b_1))}$. Note that we have `ON : F` in our example, and this does not look like a cover. It's the simplification of the empty cover. SABS will try to simplify boolean expressions (where applicable) before giving the output. We can skip this process by providing the flag `--full-cover` , this may produce very large outputs, though!

Now, what does that output mean? It's very simple to interpret it. For instance, whenever ON (OFF) holds before the execution of the basicblock, we can ensure that b_1 will be true (false) after the execution. If UND holds, on the other hand, we can't say anything about b_1, it can be both true or false.

Covers for Direct Abstraction Running the command:

```
sabs -i tutorial.bb --basicblock --bbheuristic=direct --cover-only
```

will output:

```
//b1 : (y <= x)
//b2 : ((y mod 2) = 0)
//b3 : (y >= 0)
//b4 : (x >= 0)
```

```
Produced AT:
!b4 & !b3 & b2 & !b1  -->  b4 & !b3 & !b2 & b1
!b4 & !b3 & b2 & b1  -->  b4 & !b3 & !b2 & b1
b4 & !b3 & b2 & b1  -->  b4 & !b3 & !b2 & b1
!b4 & !b3 & !b2 & !b1  -->  b4 & !b3 & b2 & b1
!b4 & !b3 & !b2 & b1  -->  b4 & !b3 & b2 & b1
b4 & !b3 & !b2 & b1  -->  b4 & !b3 & b2 & b1
!b4 & b3 & b2 & !b1  -->  b4 & b3 & !b2 & !b1
b4 & b3 & b2 & !b1  -->  b4 & b3 & !b2 & !b1
b4 & b3 & b2 & b1  -->  b4 & b3 & !b2 & !b1
!b4 & b3 & b2 & !b1  -->  b4 & b3 & !b2 & b1
b4 & b3 & b2 & !b1  -->  b4 & b3 & !b2 & b1
b4 & b3 & b2 & b1  -->  b4 & b3 & !b2 & b1
!b4 & b3 & !b2 & !b1  -->  b4 & b3 & b2 & !b1
b4 & b3 & !b2 & !b1  -->  b4 & b3 & b2 & !b1
b4 & b3 & !b2 & b1  -->  b4 & b3 & b2 & !b1
!b4 & !b3 & !b2 & !b1  -->  b4 & b3 & b2 & b1
!b4 & b3 & !b2 & !b1  -->  b4 & b3 & b2 & b1
b4 & b3 & !b2 & !b1  -->  b4 & b3 & b2 & b1
!b4 & !b3 & !b2 & b1  -->  b4 & b3 & b2 & b1
b4 & !b3 & !b2 & b1  -->  b4 & b3 & b2 & b1
b4 & b3 & !b2 & b1  -->  b4 & b3 & b2 & b1
[ 112 solver calls ]
```

The output is very different than the one produced by the cartesian method. This is due to the fact that direct abstraction produces an assignment table (AT), such assignment table can be seen as a transition relation. SABS outputs this transition relation interpretation.

Let's take the first line of the produced AT as an example. Such line is in fact, telling us that:

$$\neg E(b_4) \wedge \neg E(b_3) \wedge E(b_2) \wedge \neg E(b_1) \wedge L_P \wedge E(b_4) \wedge \neg E(b_3) \wedge \neg E(b_2) \wedge E(b_1)$$

is SAT, where L_P is the logic encoding of the suplied basic block P. That is, there exists a transition from the state $(\neg b_4, \neg b_3, b_2, \neg b_1)$ to the state $(b_4, \neg b_3, \neg b_2, b1)$, in the abstract model of P.

Theoretically, we would want to construct the projection of such transition relation in respect to one variable, that's a cover for the given variable. By means of a shell script, called `pi` , we can extract such projections.

To find out which pre-states will, certainly, turn b_1 on, or off, we could runthe command:

```
        sabs -i tutorial.bb --basicblock --bbheuristic=direct --cover-only | pi 1
```

which would return

```
pi  b1 = !b4 & !b3 & b2 & !b1 , !b4 & !b3 & b2 & b1 , b4 & !b3 & b2 & b1 , !b4 & !b3 & !b2 & !b1 , ...
pi !b1 = !b4 & b3 & b2 & !b1 , b4 & b3 & b2 & !b1 , b4 & b3 & b2 & b1 , !b4 & b3 & !b2 & !b1 , ...
```

Since we're not producing actual covers here, the flag `--full-cover` makes no sense, and will be silently ignored if the user had it suplied. Note that the `pi` tool is a simple script, and it is not installed automatically, you should create a file and add it to the search path. The code for `pi` is available as anex 1.2

6 Annotations

It's possible to control SABS behavior from inside the code, annotating each source file (both SPARK packages and bb-files) with "set" options. In this chapter we'll discuss each of those options.

6.1 Controw-Flow Options

In case a SPARK package body is given as input, SABS will start deconstructing the code. This code may have controw-flow instructions, and we can control how SABS handle those options through:

$$\%\mathit{set}\ \text{cfheuristic} \ := \ \text{slam} \mid \text{trivial}\ ;$$

slam controw-flow heuristic will abstract each controw flow conditions c with $*$, adding an assume statement before the next branch that guarantees that we continue through that branch only when c (or $\neg c$, in the `else` case) would be valid in the actual program. For instance:

```
if c then
  -- ...
else
  -- ...
end if
```

```
if (*) {
  assume (G(c))
  // ...
} else {
  assume (G(!c))
  // ...
}
```

The G function is defined in [BMR01].

trivial controw-flow heuristic will add every controw-flow predicate to our set E of predicates, therefore, every controw flow predicate will be abstracted by a (single) boolean variable.

6.2 Basic Block Options

Whenever sabs reaches a basicblock in the code it has two options, read all assignments together (default behaviour) or read and abstract one by one. This can be controled by adding a line in the beginning of our file:

$$\%\mathit{set}\ \text{bbgroup} \ := \ \text{true} \mid \text{false}\ ;$$

After determining the basic block, SABS will continue to abstract such block. Two abstractors are available. A cartesian abstractor and a direct abstractor. The cartesian abstractor can also receive an aditional parameter regarding the maximum cube size to be considered.

$$\%\mathit{set}\ \text{bbheuristic} \ := \ \text{direct} \mid \text{cartesian} \mid \text{cartesian} <\text{int}>\ ;$$

6.3 Solver Options

The solver to be used can also be changed, both by CLI or by annotations. Note that annotations take precedence over command line parameters. There are two solvers available, an unbound integers and a bitvector solver. To change such option:

$$\%set\ \text{solver} \ := \ \text{ints} \mid \text{bvs} \ ;$$

6.4 Optimization Switches

During the computation of a cover, some optimizations may take place. We allow the user to turn those optimizations on or off. It is important to note that the applicable optimizations depends on the abstraction method beeing used.

Direct Abstraction allows us to detect invalid states through:

$$\%set\ \text{optimization} \ := \ \text{invstate} \ ;$$

An invalid state is any state $(b_1, \cdots, b_n)$ such that $\bigwedge_{1 \leq i \leq n} b_i$ is *unsat*. For example, consider the program $x := 10$; with predicates $\pi_1 \equiv x \geq 0$ and $\pi_2 \equiv x \leq 100$. The state $\neg\pi_1 \wedge \neg\pi_2$ is impossible to be reached by the original program, therefore should be ignored (with some care, more on this problem can be found on [MLPF13]).

This is implemented by first determining which post-states of the current program are *not* invalid, and later on calculating which pre-states lead to each post-state. Note that for each invalid post-state that is left behind we're saving 2^n solver calls, where n is the number of predicates in E.

6.5 Cartesian Abstraction

The Cartesian method offers us a different kind of optimizations. We can generate the possible states in a smart way. Let's recall the definition of S^{ψ}_{V}.

$$S^{\psi}_{V} = \bigvee \{c \in C_V \mid \ \ \models E(c) \rightarrow \psi\}$$

Let's consider that we generate cubes in order of increasing size, during the generation of a cube $c \in S^{\psi}_{V}$, two formulas can be checked to stop the generation of cubes bigger than c:

Prime Implicants. If $c \rightarrow \psi$ then for every cube c' such that $c \subset c'$ can be safely ignored, since $c' \rightarrow \psi$ too. We just need to add c to our set. Using this optimization we'll generate the prime implicants of our formula only.

Prune Impossible. If $c \rightarrow \neg\psi$ then for every cube c' such that $c \subseteq c'$ can be safely pruned, since $c' \rightarrow \neg\psi$, therefore it's impossible for such cubes c' to imply ϕ.

Those optimizations can be turned on (together or individually) by:

$$\%set\ \text{optimization} \ := \ \text{priming, pruning} \ ;$$

A Pi

Listing 1.2. pi tool

```
#! /bin/bash

#returns a "cover", the variable is specified for
fieldsep='\n'
ret_cover="old"
covers="undefined"

function readStdin() {
  buffer=""
  while read line
  do
    end=$(echo $line | grep "solver calls")
    if [ -z "$end" ]
    then
      buffer="${buffer}${line}\n";
    fi
  done
  covers=$buffer
}

#get's the projection for the variable given as parameter $1.
function projection() {
  matchBi="(\$2 ~ \".*b$1.*\") && (\$2 !~ \".*!b$1.*\")"
  matchNBi="(\$2 !~ \".*[^!]b$1.*\") && (\$2 ~ \".*!b$1.*\")"
  awkProg="BEGIN { pbi = \"pi  b$1 = \"; pnbi = \"pi !b$1 = \"; }
                 { if ($matchBi)
                      pbi = pbi \", \" \$1;
                   else if ($matchNBi)
                      pnbi = pnbi \", \" \$1;
                 }
           END   { printf \"%s$fieldsep\", pbi;
                   printf \"%s\n\",   pnbi;
                 }"

  echo -e $covers | grep "\-\->" | awk -F "-->" "$awkProg"
}

# Calculates the projection of a transition table, for a given variable.
if [ $# -ne 1 ]
then
  echo "Usage: assignment_table_producer | pi <bool_var>"
  exit
else
  readStdin
  if [ "$1" == "all" ]
  then
    n=`echo -e $covers | grep "^//" | wc -l`
    echo $n
    for var in $(seq $n); do
      projection $var
    done
  else
    projection $1
  fi
fi
```

References

BMR01. Thomas Ball, Todd Millstein, and Sriram K. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI'01*, pages 203–213, 2001.

CKSY05. Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005.

Lou13. Cláudio Lourenço. *A Bounded Model Checker for SPARK Programs*. PhD thesis, Informatics Department, Minho University, Portugal, 2013.

MLPF13. Victor C. Miraldo, Cláudio B. Lourenço, Jorge S. Pinto, and Maria João Frade. Experimenting with predicate abstraction. In *INFórum 2013*, 2013.