The Sviss Symbolic Verifier

October 1, 2007

Abstract

Sviss is a powerful experimental platform incorporating efficient symmetry reduction into symbolic model checking. The tool consists of an extensive `C++` library for system modeling using BDDs and a rich CTL model checking engine. Applications range from communication protocols to computer hardware and multi-threaded software. For example, Sviss was one of the first symbolic tools to exploit symmetry in concurrent device driver verification, which is vital in operating system design.

1 Download

The main web page for Sviss contains exact instructions on what to download and how to unpack it. (You also find there some screenshots of working with Sviss.) After download, consult the README file in the tool's main directory for instructions on how to install it and how to test the installation on an example. Once the tool is installed, you can build your own models, as described in the next section.

2 The Build Process

In the same spirit as Murφ [MD], Sviss is by itself not an executable. Instead, the tool consists of a collection of `C++` files that are combined with source files obtained from a parser and compiled to yield an executable custom-made for a particular model. To run the tool on a model, proceed according to the following simple steps:

1. Create a new directory, say `Model`.
2. Copy the `Makefile` from the Sviss main directory into the `Model` directory. Adjust the first two lines of `Makefile` to the correct locations of the Sviss main directory (variable `SVISS`) and the `Cudd` directory (in the variable `CUDD`) that was distributed with Sviss.
3. Create a source file (as described in section 3) for your model, give it a name with extension `sv`, say `model.sv`, and place it into the `Model` directory. Now say `make`, and an executable of name `model` will be created (provided there is no error in the `model.sv` source file).

For instance, follow these steps in a UNIX-based environment:

```
cd Sviss/Examples
mkdir Model
cd Model
cp ../../Makefile .
# create source file model.sv in the new directory Model
make
./model -h # for example
```

When finished, you can say `make clean` to clean up. This will remove the executable, any backup files (`~`) and all internal object files. You can now compile a different model.

3 Creating a Model Source File

We use a Backus-Naur style grammar to describe the syntax of Sviss source files. Our notation is fairly standard: strings between two ' are literals, which must appear in the source file as is. The bar `|` separates alternatives, `[]` indicates optional elements. Curly braces `{}` are used for grouping in the grammar and are not part of the language (unless in quotes: `'{'`). The expression `{X}+` stands for at least one copy of `X`, i.e. for `X|XX|XXX|...`, whereas `{X}*` stands for any (including 0) number of copies of `X`. An *identifier* is a letter followed by a sequence of letters and digits. *text* is an arbitrary string of printable characters. Whitespace is used to delimit words in the grammar.

A Sviss source file consists of the following regions:

```
[ Parameters ]
[ Constants ]
[ Variabe Order ]
[ Global Variables ]
[ Process-local Variables ]
[ Atomic Propositions ]
  Code
```

All regions except "Code" are optional, but one of "Global Variables" and "Process-local Variables" must exist. The order of these regions must be as given above. We now describe the syntax and semantics of each region; it might be helpful to look at one of the example files that come with the Sviss distribution while reading this section.

3.1 Parameters

This region specifies parameters of the system description such as the number of processes, number of memory cells, etc. These parameters will appear as *mandatory* command line arguments in the final executable.

```
Parameters: 'Parameter' parameter_decl {',' parameter_decl}* ';'

parameter_decl: identifier [ '(' text ')' ]
```

Following the keyword `Parameter` is a non-empty and comma-separated list of parameter declarations. Each such declaration is an identifier, optionally followed by arbitrary text in parentheses. This text is printed as the "meaning" of this parameter when the executable is called with the `-h` help switch (see section 4).

3.2 Constants

Constants are integral values that must be computable at compile time. That is, the initialization of a constant may not refer to parameters or program variables.

```
Constants: 'Const' {constant_decl}+

constant_decl: identifier '=' expr ';'
```

`expr` can be a number, a constant defined before, or any combination thereof using integer arithmetic, which comprises `+`, `-`, `*`, `/` as well as `ceil_log(x)`, which computes $\lceil \log x \rceil$, and `power(b, e)`, which computes b^e. Parentheses may be used to deviate from standard precedence rules.

3.3 Variable Order

This specifies which BDD variable order to use. There are three choices: concatenated, interleaved, and dynamic (see [EW05] for an explanation of these orders). The default is concatenated.

```
Variable Order: 'Order' {'concat' | 'interl' | 'dynamic'} ';'
```

3.4 Global Variables

"Global" variables are variables that exist exactly once in the model: they are not subject to replication.

```
Global Variables: 'Global' {global_decl}+

global_decl: identifier ':' type_or_sb ';'

type_or_sb:  type_decl
  |          'SB' '(' expr ')'
```

`type_decl` is defined below. The `SB` syntax declares the global variable as *id-sensitive*: one that ranges over the index set of a particular process block [EW05]. The number given by `expr` specifies the number of the process block to which this variable refers (the first process block in the file has number 0). Like constants, `expr` must be compile-time computable.

There is no `typedef` construct in Sviss; pre-defined types are as follows:

```
type_decl: 'bool'
  |        'enum' '{' id_list '}'
  |        'range' expr '..' expr
  |        'range' '[' expr ']'
  |        'array' '[' expr ']' 'of' type_decl
  |        'record' '{' record_decls '}'
```

`bool` declares a variable of range $\{0, 1\}$. `id_list` is a comma-separated list of identifiers. The first `range` declaration defines a range of the form from..to inclusively. The second defines a range from 0 to `expr` $- 1$. For example, `range 0..9` and `range[10]` define the same type. The types `array` and `record` have the obvious meaning.

3.5 Process-local Variables

"Process-local variables" are specified once, but replicated for each process of the block they belong to.

More precisely, there can be any number of "process blocks". A process block declares a set of processes that are interchangeable. Processes from different blocks are not interchangeable.

```
Process-local Variables: {processblock}+

processblock: {'Clique' | 'Ring'} '[' expr ']' {local_decl}+

local_decl: identifier ':' type_decl ';'
```

A `processblock` can be either of the `Clique` (all processes in the block are pairwise interchangeable) or the `Ring` type (processes in the block can be rotated, such as in the dining philosopher's example). The `expr` specifies the number of processes in the block. Note that this expression need *not* be a compile-time constant: it may (and usually does) refer to parameters. `type_decl` is as above.

The distinction between a clique and a ring is relevant only as far as symmetry reduction is applied. To model a system with replicated components *without* the intent to exploit symmetry, one may use either of the two `processblock` types or simply arrays.

3.6 Atomic Propositions

... are only *declared* in the `Atomic Propositions` region of the grammar; their *definition* is part of the `Code` region.

```
Atomic Propositions: 'Proposition' {identifier ';'}+
```

3.7 Code

The purpose of `Code` is to define the atomic propositions ("APs") (declared in the `Atomic Propositions` region) and the transition relation R, by means of `C++` code. The user must define a function with the signature

```
UBDD R(const StateSpace& S);
```

at the global scope, and an analogous function for each AP declared under the `Proposition` keyword. The argument `StateSpace& S` is needed to be able to refer to program variables; see an example file about how to do this. The `Code` region is not examined by the Sviss parser, but just copied verbatim into a file and passed to the `C++` compiler. Thus, the user is free to define any function or procedure that they see fit.

Both the APs and the transition relation R are declared of type `UBDD`, a form of BDD specialized for the purposes of this tool. Each UBDD is a Boolean expression that stands for a set of states (if it defines an AP) or a set of transitions (if it defines R). The functions described in figure 1 are available from the library that comes with the tool; note that some of these functions are only suitable when defining APs, some only when defining R.

In order to allow the modeling of a transition relation, each program variable exists twice internally: once for the current-state value, once for the next-state value. Transitions are specified as pairs of current-state and next-state values of variables. For example, a transition that is due to the assignment `y:=y-1` is modeled using the BDD returned by the following function call:

```
UBDD::dec(S.y(), S._y());
```

`S` is the interface to the state space, i.e. to the program variables. `S.y()` returns the bit vector that stores the current-state value of variable `y`, which is to be decremented. `S._y()` returns the bit vector that stores the next-state value of the same variable, which is to receive the computed value.

4 Using the Executable

Every executable compiled as described in section 2 has a `-h` switch, which causes a lot of information about the usage of the verifier to be printed in quite some detail. Once you have read the instructions in this document, the `-h` switch should suffice for quick reference. Use it without any other arguments to the executable:

```
./model -h
```

4.1 Command Line Syntax

Since each executable is custom-made for a particular model, the exact form of the command line arguments varies depending on the model, but it roughly looks like this:

```
./model -h n l -s -spec <val> -alg <val> -iter
```

The syntax of this command line format, which is also used when you request help using `-h`, is as follows.

- An argument of the form `-h` is a *switch*. Switches are of type $\{false, true\}$ and always have default value *false*. They are set to *true* by simply mentioning them on the command line.
- An argument of the form `-spec <val>` is a *keyword* argument. Such an argument is set by specifying the name, including the `-` at the beginning, followed by its value.
- An argument of the form `n` is a *keyless* argument. Such an argument is set by specifying the value in place of the argument name.
- All arguments are optional, i.e. they have a default value. The default values of arguments can be learned using the `-h` switch.
- Switches and keyword arguments can be set in any order. Keyless arguments must be set in the order they appear in the command line.
- Values of arguments may be abbreviated if unambiguous. Argument names may not be abbreviated.

```
UBDD(); // create new UBDD
UBDD UBDD::Zero(); UBDD UBDD::False(); // both: empty set of states (or transitions)
UBDD UBDD::One (); UBDD UBDD::True (); // both: full  set of states (or transitions)

bool A==B; // set equality

// Boolean operations
UBDD !A;        // negation    (set complement)
UBDD A & B;     // conjunction (set intersection)
UBDD A | B;     // disjunction (set union)
UBDD A.Eqv(B);  // A equiv B (distinguish this from set equality above)
UBDD A.Exor(B); // exclusive or
UBDD A.Diff(B); // (A & !B) | (B & !A), known as symmetric difference

UBDD A.IfThen(B); // same as (!A) | B, same as A implies B, same as "if A then B"
UBDD A.Ite(B,C);  // same as (A & B) | (!A & C), same as "if A then B else C"

bool A.zero(); bool A.empty(); // true iff A is empty (false)
bool A.one (); bool A.full (); // true iff A is full  (true)
bool A.subset(B);              // true iff A subset B

// for the following, "bits", "x" and "y" should refer to the bits representing a var-
// iable of appropriate type. These bits are obtained using the StateSpace& S argument:
// UBDD::atMost(S.tok(), n - 1)
// returns the set of states in which the (global) variable tok is <= n-1 (i.e. < n)
UBDD UBDD::greater(const vector<short>& bits, long l); // {s: <bits> >  l}
UBDD UBDD::equal  (const vector<short>& bits, long l); // {s: <bits> == l}
UBDD UBDD::less   (const vector<short>& bits, long l); // {s: <bits> <  l}
UBDD UBDD::atMost (const vector<short>& bits, long l); // {s: <bits> <= l}
UBDD UBDD::atLeast(const vector<short>& bits, long l); // {s: <bits> >= l}

UBDD UBDD::greater(const vector<short>& x, const vector<short>& y); // {s: x >  y}
UBDD UBDD::equal  (const vector<short>& x, const vector<short>& y); // {s: x == y}
UBDD UBDD::less   (const vector<short>& x, const vector<short>& y); // {s: x <  y}
UBDD UBDD::atMost (const vector<short>& x, const vector<short>& y); // {s: x <= y}
UBDD UBDD::atLeast(const vector<short>& x, const vector<short>& y); // {s: x >= y}

UBDD UBDD::dec(const vector<ushort>& x, const vector<ushort>& y); // {s: y = x - 1}
UBDD UBDD::inc(const vector<ushort>& x, const vector<ushort>& y); // {s: y = x + 1}

// The remaining only make sense when defining transitions.

UBDD UBDD::invariant(const vector<ushort>& bits); // variable in bits is invariant

// the following are methods of the StateSpace class:
UBDD S.invariant        (short p, short sbn); // p            in block sbn invariant
UBDD S.invariant_but    (short p, short sbn); // all except p in block sbn invariant
UBDD S.invariant_sbn    (short sbn)         ; // all          in block sbn invariant
UBDD S.invariant        ()                  ; // all processes invariant
UBDD S.invariant_globals()                  ; // all global variables invariant
```

Figure 1: Functions available to the user when creating APs and R

Argument values that contain spaces or other characters that may be subject to interpretation of your shell must be protected appropriately. For example, in a UNIX-like environment the call

```
./model 10 5 -spec 'INV_FS !bad' -alg DYN -iter
```

is syntactically correct and sets the first keyless parameter to 10, the second to 5, `spec` to `INV_FS !bad`, `alg` to `DYNAMIC` and `iter` to *true*. Switches `-h` and `-s` are (per default) set to *false*.

4.2 Meaning of Command Line Arguments

Most systems modeled using Sviss have at least one parameter, such as the number of processes. These parameters are specified using the `Parameter` keyword in the source file and must be given a value at the command line. Parameters show up as keyless arguments of the command line.

The remaining arguments, all of which are optional, have the following meaning:

`-s` (if present) print information about the parsed state space.

`-spec <val>` (if present) `val` is assumed to be a CTL specification (see section 4.4), evaluated, and the results are displayed appropriately. After that, the control is returned to the calling operating system shell. If `-spec` is missing, the tool will read the model and then prompt for a specification. The latter form of submitting a specification is useful when several properties are to be verified of the same model.

`-alg <val>` (if present) `val` is assumed to be an algorithm name; possible values are `NOSYMM`, `ORBIT`, `MULTIPLE`, `DYNAMIC`; default value is `NOSYMM`. These names stand for (1) ignoring symmetry, (2) symmetry reduction using the orbit relation [CEFJ96], (3) symmetry reduction using multiple representatives [CEFJ96], (4) dynamic symmetry reduction [EW05]. Remember that the algorithm names may be abbreviated if unambiguous.

`-iter` (if present) prints statistics during iterations of fixpoint computations.

As an example, the command line supplied at the end of section 4.1 causes two parameters to be set to 10 and 5, respectively, the model to be built and an invariant formula to be verified using dynamic symmetry reduction. The invariant is `bad`, statistics about completed fixpoint iterations will be printed.

If the `spec` argument does not appear on the command line, the user will be prompted by the tool to provide a specification; upon this occasion it is also possible to supply an algorithm and the `iter` switch. After evaluation, the prompt is repeated. If the `spec` argument *is* set on the command line, *all* arguments must be set on the command line; there is no prompt later on to change anything.

4.3 How to Model Check using Sviss

Sviss technically is a CTL evaluator. That is, the tool accepts a CTL specification (where "CTL" is defined in section 4.4) and evaluates it into a set of states. For example, in order to check whether $M, init \models \mathsf{AG}\, good$, the user should not just type `AG good`: this merely returns the set of states from which only `good` states can be reached. What we really want to know is whether the initial states all fall into this set, i.e. whether the set represented by the formula `init & !(AG good)` is empty.

If the computed set is empty or equal to the entire state space, the user is so notified. If it is neither, the user is given the choice to have the set enumerated. Sviss has support for enumerating sets succinctly in a hybrid (between explicit and symbolic) notation. In addition, the user can ask the tool to compute the *number* of states in the set represented by the formula. Obviously, the last two features only make sense in systems whose size allows such information to be computed and printed succinctly, e.g. small instances of parameterized systems.

Sviss also has invariant checking capabilities, both using forward or backward search through the state space. This is different from standard CTL evaluation in that the result of an invariant check is either the answer that the invariant holds, or it is the answer that it does not, followed by a printout of an error trace.

At the end of an evaluation, statistics is shown about the computation times and BDD sizes accumulated so far.

4.4 Syntax of the Specification Language

Properties are specified in SVISS in a dialect of CTL [EC82] that adds past-time and invariant temporal operators to the language. The reader is assumed to be familiar with CTL; we only list here the names of the temporal operators as written in our tool and the additional features available.

constants: `False`, `True` (note: capital initial)

state space: `States` prints, in compact form if possible, the BDD for the entire state space. This differs from the BDD for `True` if some variables have a domain whose size is not a power of 2. For example, if an `enum` with three values is declared, two bits will be reserved. Since the valuation `11` is not valid (only `00`, `01` and `10` are), it is not part of the state space and not printed by `States`.

atomic proposition: an identifier as defined in the `Atomic Propositions` section of the input file

Boolean connectives: `! | & == =>` for: not, or, and, equivalence, implication

future-time: `EX`, `AX`, `EF`, `AF`, `EG`, `AG`, `EU`, `AU` with their standard CTL meaning

past-time: `EY`, `AY`, `EP` with the meanings: existential previous time, universal previous time, existential past. For example, the formula `EP init` represents the states reachable from `init`: from these states, there exists a past along which eventually `init` is true. `EY` and `AY` are forward image operators; `EP` can be used for forward reachability.

search: `FS`, `BS` as forward and backward search operators, respectively. These operators can syntactically be used wherever the other CTL modalities can. `FS` and `BS` compute the same result as `EP` and `EF`, respectively, but they use *frontier set optimization* and are occasionally more efficient.

equivalence mappings: `alpha`, `orbit`: these operators can also be used wherever any CTL modality can. They map the argument set of states to the set of representative states, and to the set of symmetry equivalent states, respectively.

All above operators can be viewed as predicate transformers: they take a set (or sets: `EU`, `AU`) of states as input and produce a new set of states. There are a few operators that do not compute a set of states. These operators can be used only at the top-level, i.e. they may not appear in the scope of any other operators, including the propositional ones. These are `INV_FS`, `INV_BS`, `R` and `size`. The first two operators start an invariant checking procedure using forward or backward search, respectively. The result is either a confirmation of the invariance property, or a failure statement, followed by a counter example trace. The constant expression `R` enumerates the transitions the system can take, in a way that reflects the variable and process block layout of the system. This is only useful for small transition systems. The last operator computes the number of states represented by the argument set (which may take long if the BDD for the set is large, so use with caution).

References

[CEFJ96] Edmund Clarke, Reinhard Enders, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design (FMSD)*, 1996.

[EC82] Allen Emerson and Edmund Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming (SOCP)*, 1982.

[EW05] Allen Emerson and Thomas Wahl. Dynamic symmetry reduction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005.

[MD] Ralph Melton and David Dill. *Murφ Annotated Reference Manual, rel. 3.1.* `http://verify.stanford.edu/dill/murphi.html`.