How to use the SWORD library

Jean Christoph Jung

December 7, 2009

This document describes the usage of *SWORD* – a decision procedure for the SMT bit-vector logic. It is shown, how SWORD can be utilized as C++ library in existing projects. Furthermore, it is described how to develop and integrate new modules. For a more detailed treatment of the major concepts behind SWORD, we refer to the respective literature.[1]

1 Creating and Solving QF_BV Instances

SWORD supports quantifier free bitvector logic (see the description of the logic QF_BV at SMTLIB[2] for details). To create and solve instances the SWORD library provides a couple of public functions, namely

- `addVariable` and `addConstant` to define new bit-vector variables and constants, respectively,
- `addOperator` to define operations over terms,
- `addAssertion` to assert terms for *all* following solve processes,
- `addAssumption` to assert terms for *one* following solve process,
- `solve` to solve the resulting instance, and
- `getVariableAssignment` to obtain assignments of variables in case the instance is satisfiable.

In the following, the creation of a SWORD instance is described by means of a running example. Assume the task is to factorize 18 into two 8-bit variables x and y, i.e. solve the equation

$$x \cdot y = 18$$

First, SWORD has to be instantiated. This is simply done by:

[1] For example at the SWORD homepage www.informatik.uni-bremen.de/agra/eng/sword.php

[2] `www.smtlib.org`

```
sword * solver = new sword();
```

Afterwards, using the resulting SWORD object all variables and terms are constructed by calling the respective functions mentioned above. Each function returns an object of type `PSignal` which is used as unique identifier for the term created by the call.

In the example, first of all the variables are defined. To this end, the bitsize and the name of the variables (optional) are declared in the parameters:

```
PSignal x = solver->addVariable(8, "x");
PSignal y = solver->addVariable(8, "y");
```

The same way constants including their bit-size (8) and their value (18) are specified.

```
PSignal eighteen = solver->addConstant(8, 18);
```

Therewith, the atomic elements of the considered equation have been created. To construct the terms, the `addOperator` function is applied. This function takes the operator type as first argument. More detailed information on how to create all terms possible in QF_BV with SWORD can be found in the header files `include/swordOpcode.h` and `include/libsword.h`.

After the operator type, the terms that the operation is applied to, are passed. With that as a basis, the equation from the running example can be created in two steps:

```
PSignal mult = solver->addOperator(MUL, x, y);
PSignal eq   = solver->addOperator(EQUAL, mult, eighteen);
```

Finally, the resulting term is asserted in the solver.

```
solver->addAssertion(eq);
```

Now the problem can be solved:

```
bool result = solver->solve();
```

In the running example the created instance is satisfiable so that the Boolean variable `result` evaluates to `true`. In this case, the model can be obtained using the `getVariableAssignment` function with the identifier of the respective variable:

```
vector<int> solutionX = solver->getVariableAssignment(x);
vector<int> solutionY = solver->getVariableAssignment(y);
```

The respective calls return a vector that contains a satisfying assignment for the given variable. For example, `solutionX` contains the assignment of variable x, where `solutionX.size()==8` and `solutionX[0]` is the value of the least significant bit of x. Possible values in the solution are `SWORD_DONTCARE`, `SWORD_TRUE`, and `SWORD_FALSE`. To check if exactly one of the factors in the running example is assigned to a negative number (in 2-complement), the following function calls are sufficient:

```
bool one_negative = (solutionX.back()==SWORD_TRUE
                   ^ solutionY.back()==SWORD_TRUE);
```

Using the functions introduced so far, any QF_BV instance can be created and solved. Besides that, also the definition of assumptions is possible. Assumptions are temporarily asserted terms, that are dropped after every call to `solver->solve()`. They can be added to an instance by the function `addAssumption` in the same way like `addAssertion`. In particular, assumptions enable incremental usage of SWORD as the following example shows.

```
PSignal zero   = solver->addConstant(8, 0);
PSignal x_gt_0 = solver->addOperator(SGT, x, zero);
PSignal y_gt_0 = solver->addOperator(SGT, y, zero);

sword->addAssumption (sword->addOperator (AND, x_gt_0, y_gt_0));
if (sword->solve())
  cout << "found␣solution␣with␣x>0␣and␣y>0\n";

sword->addAssumption (sword->addOperator (NOR, x_gt_0, y_gt_0));
if (sword->solve())
  cout << "found␣solution␣with␣x<=0␣and␣y<=0\n";
```

This code in the snippet checks if there are solutions with both x and y assigned to positive values and both x and y assigned to negative values. The complete code is also available in the file `doc/example1.cpp` of the SWORD v1.1 package.

2 Use of modules

Besides the QF_BV operations of SMTLIB, SWORD additionally provides the possibility to define user-specified operations in terms of *modules*. Each module is defined over (bit-vector) variables and provides functions enabling dedicated strategies for decision making (`decide`) and propagation (`propagate`). Summarily, `decide` is called whenever the solver asks the module for a decision, and `propagate` is called whenever the value of a variable constrained by the module is changed. A detailed description of this concept can be found at SWORD's homepage.

This section briefly describes how to define new modules with own strategies. Therefore, the example from the previous section is continued and extended by a cardinality constraint: A solution for $x \cdot y = 18$ has to be determined where the number of bits in x assigned to `true` is smaller than a given number n. A straight-forward solution to this problem using the QF_BV operations is possible by extracting all single bits from x, adding all of them up, and constrain the sum to be less than n. However, this can be done more elegant by a dedicated SWORD module.

To this end, a new class that inherits from `SwordModule` has to be created. For the example additionally two fields to store the maximal number n of bits allowed to be assigned to `true` and the variables involved in the modules are created.

The variables are of type `Lit`, which is SWORD's internal representation of Boolean literals.

```
class CardinalityLessThan : public SwordModule {
  ...
  const std::vector<Lit> _vars;
  const unsigned _maxOnes;
}; // class CardinalityLessThan
```

A SWORD module needs to be initialized with the current solver object. Furthermore, the constructor gets the variables considered by the module and parameters of the respective constraint (in the example the value of n). With the function `useVariables`, the module informs the solver object about the variables (of type `Lit`) it is using. This ensures, that each time the value of one of the variables is changed (due to decision or propagation), the respective propagate routine of the module is called to update the current status and to deduce further assignments or to detect conflicts, respectively.

```
CardinalityLessThan(sword * swd, PSignal signal, unsigned maxOnes)
  : SwordModule(swd)
    , _vars(signalToLiterals(signal))
    , _maxOnes(maxOnes)
  {
    useVariables(_vars);        // inform the solver to use all _vars
  }
```

Finally, the respective functions for decision making (`decide`) and propagation (`propagate`) can be overwritten.

The decision function of a module assigns a value to one of its variables. The function is called when the solver wants the module to make a decision and returns a literal representing the respective decision. In the case that all variables are already assigned an undefined literal is returned. Different strategies can be thereby applied for different types of modules. For the running example, a possible strategy could be to assign a free variable to `false` in order to not exceed the number of bits assigned to true:

```
virtual Lit decide () {
  for (unsigned i = 0; i < _vars.size(); ++i) {
    if (isFree(_vars[i]))      // check if variable is still free
      return ~_vars[i];        // set _vars[i] to false
  }
  return l_Undef;              // no free variable in this module
}
```

The function `propagate` is called when a variable of the module was changed. Then, it is checked if the current status is conflict-free and if further assignments can be deduced. In case of a conflict, a reason for the conflict has to be returned, i.e., a set of assignments that are not valid in the module. The following code snippet demonstrates the propagate function by means of the running example:

```
virtual Clause* propagate () {
  conflict_set_t reason;
  for(unsigned i = 0; i < _vars.size(); ++i) {
    if ( getValue(_vars[i]) == l_True )
      reason.push_back( _vars[i] );
  }
  if ( reason.size() >=  _maxOnes ) {
    return makeConflict(reason);
  } else {
    return NULL;
  }
}
```

In the for-loop the number of true literals in vector `_vars` is counted. If this number exceeds the maximal number of ones, a conflict is found. In this case, the reason of the conflict are all assignments to `true` in the module. These assignments are collected in `reason` and returned using the method `makeConflict`. If the module is in a consistent state, `NULL` is returned.

Similarly, if a module makes an inference, this can be propagated in the solver with `inferLiteral (inferedLiteral, reason)`. Again, the reason is a set of literals that imply the infered literal in the module. For example, if the maximal number of bits is assigned to `true`, all other literals are infered to be `false`. This is done with the following code:

```
if (reason.size() + 1 == _maxOnes) {
  for (unsigned i = 0; i < _vars.size(); ++i) {
    if ( getValue(_vars[i]) == l_Undef )
      inferLiteral(~_vars[i], reason);
  }
}
```

Having defined the module it is asserted with:

```
unsigned max = 4;
SwordModule * card = new CardinalityLessThan (solver, x, max);
solver->addAndAssertModule (card);
```

Then the instance can be solved as shown above with `solver->solve()`. Again the instance is satisfiable and the model is either $x = 2, y = 9$ or $x = 9, y = 2$.

The complete code of the example is also available in the file `doc/example2.cpp` of the SWORD v1.1 package.