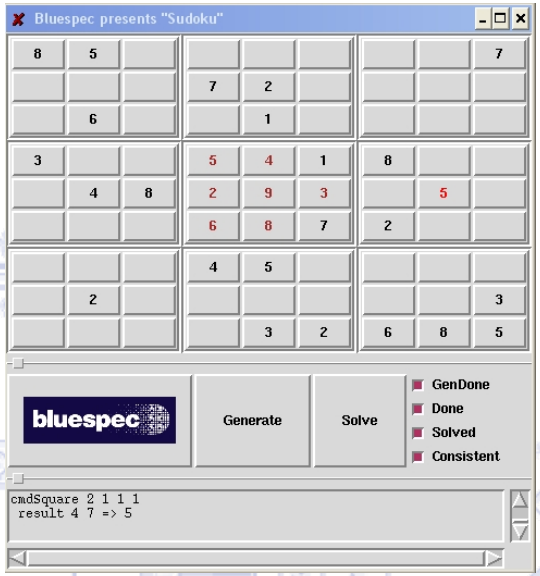
bluespec

ESL SYNTHESIS

Sudoku: Revolution #9, or #4, or #16, ...

Jef Newbern, Steve Allen and Rishiyur Nikhil

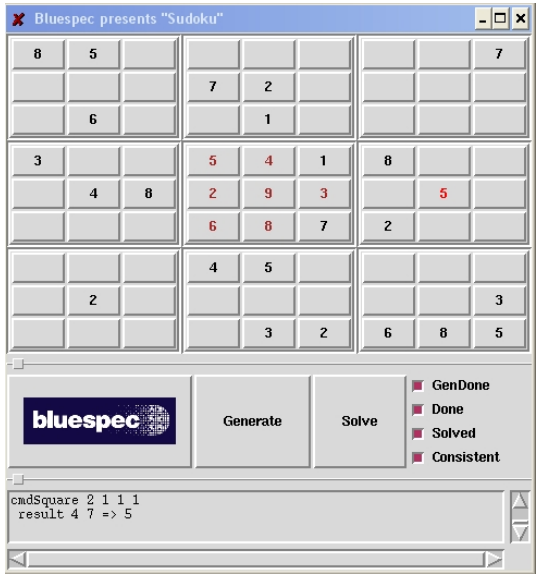


www.bluespec.com

Filters

- Do you know how to play Sudoku?
- Could you write a program to solve Sudoku puzzles?
- Could you write one that doesn't use backtracking?
- Could you write one in your favorite Hardware Description Language and synthesize it?
- Could you write a puzzle generator as well as a solver?

BluDACu demo

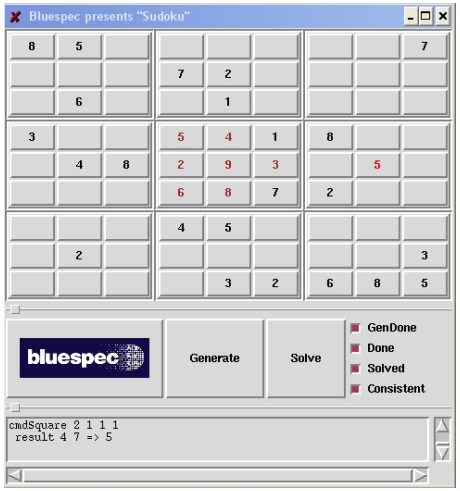


bluespec

BluDACu: system architecture

Running on host workstation

TCL/TK GUI



C/C++ with socket communications

Three different execution platforms:

Verilog sim

Sudoku Testbench (BSV + C/C++)

Sudoku Generator and Solver (BSV)

Embedded C/C++ with socket communications

Bluesim

Embedded C/C++ with socket communications

EVE FPGA box

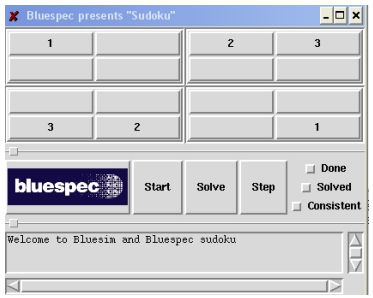
Embedded BSV communication transactors

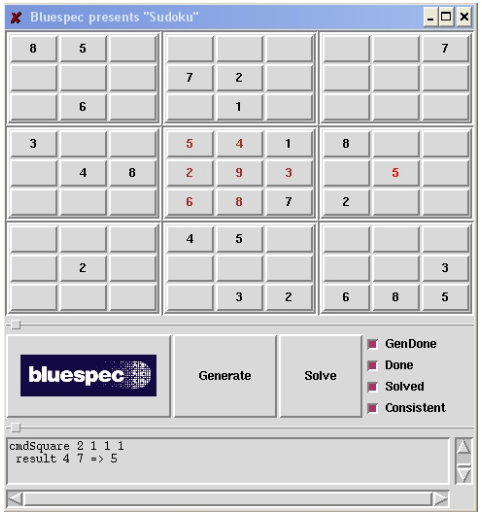
Note: C/C++ can be embedded in BSV, for SW control, early SW/HW codesign, verification, etc.

bluespec

BluDACu fully parameterized for size

- Generator and solver
  - 4x4, 9x9, 16x16, …
- Terminology: *order* 2, 3, 4, …





bluespec

Cell values

- Each cell value is 9-bit mask, representing remaining candidates for that cell
  - Initially 9'b111111111
  - Gradually eliminate candidates until 1 bit remains
    - If reaches 9'b000000000, puzzle is inconsistent

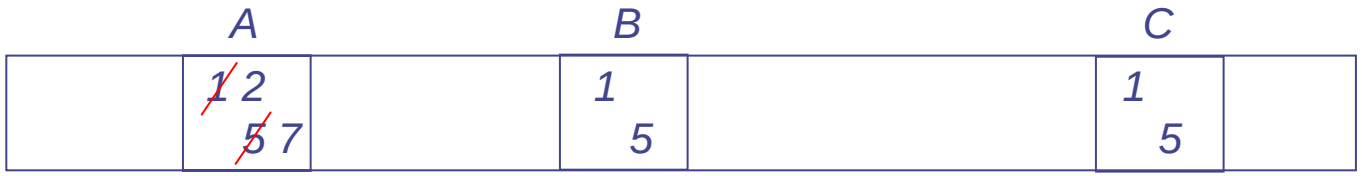
```
typedef    Bit#(TSquare(order))    Cell#(order);

// Create a cell with a given solution x
function Cell#(order) given (UInt#(n)  x)
   provisos (Mul#(order, order, size), Log#(size, index_bits),
                  Log#(TAdd#(1,size), n));
   Index#(order) idx = unpack(pack(x-1)[valueOf(index_bits)-1:0]);
   return (1 << idx);
endfunction
```

The Sudoku grid

```
// Grid values
typedef Vector#(nr, Vector#(nc, t))                    Grid#(nr, nc, t);

// Grid of registers containing Cell values
typedef Grid#(TSquare#(o), TSquare#(o), Reg#(Cell#(o)))  SudokuRegGrid#(o);

// Indexes of cells
typedef UInt#(TLog#(TSquare#(o)))                      Index#(o);

// Indexes of boxes ("rank and file index")
typedef UInt#(TLog#(o))                                RFIndex#(o);
```

Tactics

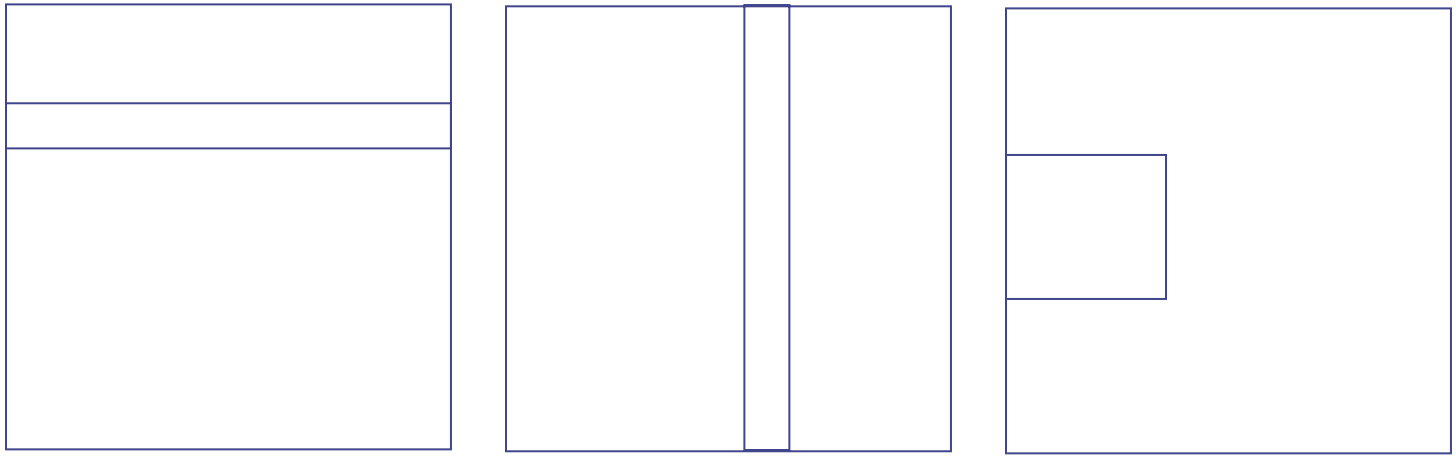
- Instead of backtracking, the solver repeatedly applies *tactics.* Example of a tactic:
  - *repeated_2_set*: If, in this cell A's row, two other cells B and C contain the same "2-set" { *j, k* }, i.e., symbols *j* and *k* are the only remaining possibilities in cells B and C, then *j* and *k* can be eliminated from this cell A (because *j* and *k* must be in the cells B and C, even though we may not yet know which one goes in B and which one goes in C)



bluespec

Tactics parameterized over Groups

- The same tactic, described for a row, can also be applied to a column or a 3x3 box—i.e., more generally, a *constraint group* of 9 cells



```
typedef  Vector#(TSquare#(o), Cell#(o))    Group#(o);

// Determine all values which are possible in a group
function Cell#(order) possibles (Group#(order) g)
   provisos (Add#(1,_,SizeOf#(Cell#(order))));
   return fold(\| , g);
endfunction
```

bluespec

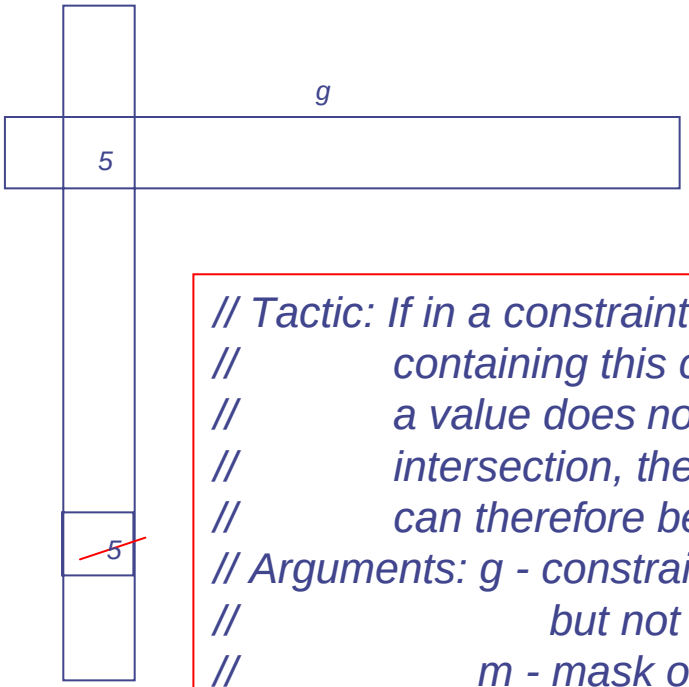
Parameterized Tactics

- All tactics are collected into a module

```
interface Tactics#(numeric type order);
   method Cell#(order)  elim_other_singletons   (Group#(order) g, Index#(order) n);
   method Cell#(order)  process_of_elimination  (Group#(order) g, Index#(order) n);
   method Cell#(order)  forced_in_intersection   (Group#(order) g, Mask#(order) m);

   // Tactic: If a value appears in a 2-set which is repeated twice in the
   //          group (excluding this cell), then it can be eliminated from this cell.
   // Arguments: g - constraint group containing the cell
   //                n - index of this cell in the group
   method Cell#(order)   repeated_2_set  (Group#(order) g, Index#(order) n);

   method Cell#(order)  hidden_pair  (Group#(order) g, Index#(order) n);
endinterface
```

Parameterized tactics

- With suitable use of higher-order help functions, tactics can be as short as one line of code




```
// Tactic: If in a constraint group which intersects a constraint group
//         containing this cell (but which does not itself contain the cell),
//         a value does not appear in the portion outside of the
//         intersection, then it must appear within the intersection and
//         can therefore be eliminated from this cell.
// Arguments: g - constraint group intersecting a group containing the cell
//                but not containing the cell itself
//            m - mask of the portion of g outside of the intersection

method Cell#(order) forced_in_intersection(Group#(order) g, Mask#(order) m);
  return possibles (maskN (g,m));
endmethod
```

bluespec

Parameterized types, library functions, higher-order functions

- Higher-order functions have functions as arguments or results, and are hence an extreme case of parameterization

```
typedef Bit#(TSquare#(order)) Cell#(numeric type order);

// Determine if a cell has only 2 possibilities
function Bool is2set(Cell#(order) c);
   return (countOnes(c) == 2);
endfunction: is2set


// compute a mask over a constraint group that
// identifies the 2-sets in the group

let two_set_mask = map ( is2set, g );
```

Outer loop: apply tactics repeatedly across all cells, using BSV FSM facilities

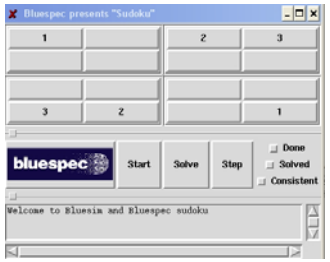
```
Stmt tactic_sequence =
  seq
    while (True)
      seq
        action
          found_inconsistent <= False;  all_cells_complete <= True; made_some_progress <= False;
        endaction
        for (r <= 0; r < fromInteger(valueOf(size)); r <= r + 1)
          seq
            for (c <= 0; c < fromInteger(valueOf(size)); c <= c + 1)
              seq
                apply(… singleton tactic to row r …);
                apply(… singleton tactic to column c …);
                apply(… singleton tactic to box containing row r and column c …);
                … similarly, apply tactics Elimination, Pairs, HiddenPairs …
              endseq
          endseq
          await(!results.notEmpty());
          if (found_inconsistent || all_cells_complete || !made_some_progress)  break;
        endseq
  endseq;

FSM controller <- mkFSM(tactic_sequence);
```

BluDACu: composable state machines

- For a legal Sudoku puzzle (unique solution) tactics can be applied in any order, and even in parallel
  - ➔ In the outer loop FSM
    - Statements can be reordered
    - The *seq/endseq* compositions can be replaced by *par/endpar* compositions

- But note the implication:
  - Each such change can have dramatic impact on control logic (because of shared resources accessed in parallel)
    - All of this control hardware is generated automatically
  - These choices affect area, clock speed, power
    - These are fundamental choices in microarchitecture exploration!

BluDACu parameterized for size



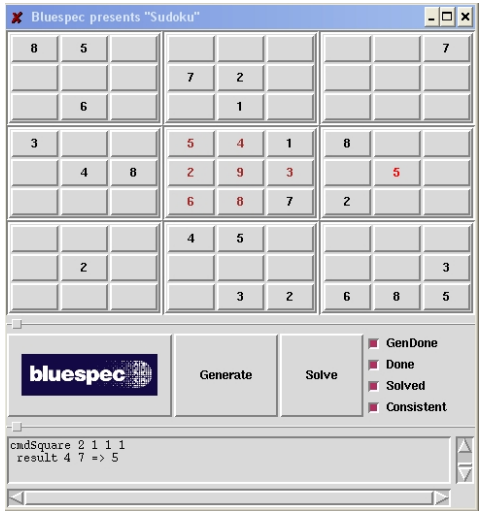
- Generator and solver
  - 4x4, 9x9, 16x16, …
  - Terminology: *order* 2, 3, 4, …

- By simply changing instantiation line:

  SudokuGenerator#(2) generator <- mkSudokuGenerator(); ***TO***

  SudokuGenerator#(3) generator <- mkSudokuGenerator()

  all of the following change:
  - The width of each cell register
  - The number of cell registers in each row and column
  - The width of method arguments and values in the interfaces
  - All of the functions for accessing and modifying Sudoku grids
  - The logic for each tactic
  - The number of tactic applications
  - The sequence of states in the solver FSM
  - The sequence of states in the generator FSM
  - The logic to place a given in the grid



bluespec

Completeness

- Sudoku puzzles range in difficulty from “very easy” to “diabolical”
- A solver is only as strong as its repertoire of tactics
  - Current tactic set solves “medium” difficulty puzzles
  - Will get stuck if given harder puzzles
- Adding a new tactic, as we discover them, has a *profound* impact on the HW control logic
  - But it’s trivial to do this in the BSV program, because all this control logic is automatically regenerated

Puzzle generator

- Just uses solver, in a brute-force manner (backtracking!). Use BSV FSM for:

Reset to blank puzzle (in all cells, all 9 digits are still candidates)

While (not done)
  Randomly pick a cell, randomly choose one of its remaining candidates, kill remaining candidates
  Apply the solver:
    done if solved,
    continue loop if stuck,
    reset to blank puzzle if choice led to inconsistency/contradiction

- Note: therefore, produces legal puzzles that it can solve

BluDACu effort

- Sudoku Generator & Solver (Jeff Newbern)
  - *Developed in less than one man-week*
  - *(The tactics were developed earlier, over a longer period of time — Nikhil)*
  - *1044 lines of BSV for solver and generator*
    - *Compare: 926 lines for C code with similar tactics (solver only; unparameterized 9x9 only)*
- GUI (Steve Allen)
  - *Developed and integrated with generator and solver in less than one-man week*
- Running on EVE (Steve Allen)
  - *Developed and running in less than one man-week*

The final word: why?

- It's a novelty, of course
  - Sudoku solvers are quite easily implemented in SW on a small processor (unless speed or power were an issue)
  - Non-backtracking version not so easy, in SW or HW
- But there are some serious messages:
  - With BSV, you can do some things in hardware that you might not have imagined doing in hardware because of the degree of difficulty (with obvious implications for performance and power)
  - BSV can match and exceed many software languages in expressive power and abstraction
    (essential for correctness, rapid development, rapid architecture exploration, and reuse)
  - BSV can be combined easily with C/C++, Tcl/Tk, ...
    (essential for HW/SW codesign)
  - BSV can be run easily on FPGA HW accelerators
    (essential for early SW development, verification)

bluespec

BSV source code, paper at:
www.bluespec.com/products/BluDACu.htm

(ask if you also want the Tcl/Tk GUI)

Also, Google for the “BluDACu Song”
somewhere on George Harper’s blog

(sung to the tune of “Suzie Q”)

Thanks!

bluespec