Yosys - A Free Verilog Synthesis Suite

Clifford Wolf, Johann Glaser[†]
[†]Johannes Kepler University, Austria
Institute for Integrated Circuits
clifford@clifford.at, johann.glaser@jku.at

Abstract

Most of todays digital design work is done using hardware description languages such as Verilog HDL or VHDL. HDL synthesis is used to translate that HDL code to digital circuits. Yosys is the first free and open source software for Verilog HDL synthesis which supports the vast majority of synthesizable Verilog features. Yosys is built as an extensible framework so it can be used easily as basis for custom synthesis flows and as environment for the implementation and research on new synthesis algorithms. Yosys has special emphasis on support for coarse-grain logic, making it ideal for algorithms such as logic mapping to DSP cells in FPGAs or synthesis for custom coarse-grain reconfigurable hardware.

Yosys has mature support for Verilog HDL and is able to synthesize complex real-world Verilog designs. Example design flows for fine-grain and coarse-grain architectures are presented and discussed. The availability of Yosys under a liberal open source license can greatly improve reproducibility of scientific publications, when Yosys is used as basis for reference implementations of new algorithms instead of closed-source alternatives.

1 Introduction

In modern ASIC design the use of hardware description languages like Verilog and VHDL as well as logic synthesis tools is ubiquitous. The development of these tools has a long history [7, 8] which enormously increased the productivity of designers as well as the reliability of the design process.

The synthesis process itself involves numerous complex algorithms, e.g., for optimization of the logic networks. This area of research has contributed essential techniques and features to logic synthesis [11, 5, 10, 13, 4].

Additionally, in other areas of research the concepts of logic synthesis as well as functional components of the synthesis process are an important ingredient, too. For example, the research on a methodology for the development of coarse-grain heterogeneous application-domain specific reconfigurable logic is heavily based on tight integration with the synthesis process [18, 9].

All these cases ask for a way to use widespread HDLs as input data format. This offers a well-known development process with good readability of the code and allows design reuse. Common simulation tools can be used for design verification and the research results can be formally verified to the input data using commercial equivalence checking tools.

Therefore in the next section we look for qualified synthesis tools which offer flexible and open interfaces as required for the aforementioned research topics. This gives a short overview of available commercial as well as open source synthesis tools. This is followed by a technical introduction to Yosys, which is further elaborated with three exemplary use cases. The next section gives an evaluation of the features and results of Yosys and the paper closes with conclusions and an outlook for further work.

2 State of the Art

Todays ASIC and FPGA development is completely dominated by commercial tools. These provide powerful functionality and a high quality of results. All these tools are closed source programs which do not provide interfaces for custom extensions. Further investigation showed that even intermediate data files are stored in an encrypted file format. Therefore these tools, although powerful and proven, can not be used as basis for the development of synthesis algorithms.

Besides commercial synthesis tools, several free and open source Verilog synthesis tools exist [12, 3, 13, 15]. The open source development model provides an ideal entry point for extensions, since all internals and interfaces are openly accessible. However, an evaluation [17] showed that these tools are still in a very early state and are by far not able to handle real-world Verilog synthesis tasks.

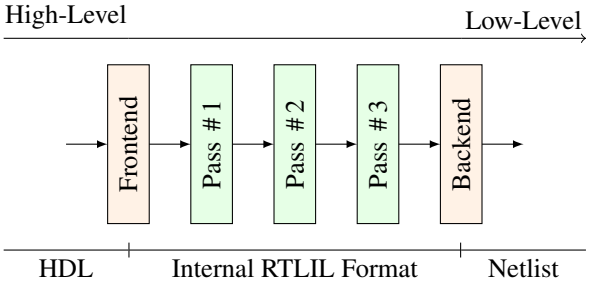

Figure 1. Simplified Yosys data- and control-flow

3 The Yosys Open Synthesis Suite

As shown in the previous section, no qualified tools with open interfaces to integrate custom synthesis algorithms are available. Therefore the new Verilog synthesis software stack Yosys was developed [16]. This section provides a brief introduction to Yosys.

The main goal of Yosys is the synthesis of Verilog HDL to logically equivalent netlists. Unlike commercial synthesis tools for ASIC or FPGA design, it is (currently) not intended to handle timing information, perform STA (static timing analysis) or consider constraints in Yosys. Some steps in the synthesis process utilize external tools, e.g. combinational logic minimization and fine-grain technology mapping is handled by Berkeley ABC [5]. This allows to provide the core feature of logic synthesis with the maximum of openness and extensibility.

3.1 Yosys Data Flow

Yosys is a very modular program. Its data flow is illustrated in Fig. 1. *Frontends*, such as the Verilog frontend, read HDL source and transform it to RTLIL[1], the internal format used by Yosys. *Passes*, such as various technology mappers, operate on this RTLIL data. Each pass either transforms the RTLIL data (perform optimizations, etc.) or analyzes it (e.g. evaluate a logic net with a given input). As all passes operate on the same data structure, i.e. RTLIL, they can be combined in any arbitrary order. *Backends*, such as the Verilog backend, are finally used to write the resulting netlist to an output file.

3.2 Yosys Scripts and Extensions

Yosys is controlled using synthesis scripts. Each command in a synthesis script executes a frontend, pass, or backend. Custom synthesis flows are created by writing new synthesis scripts. Additionally, new functionality can be added by developing new frontends, backends, and especially passes. This extensibility is one of the key advantages of Yosys. The Yosys manual contains documentation on the C++ APIs for developing such extensions, as well as example code.

An exemplary Yosys script with descriptive comments is shown in Fig. 2. The `opt` pass is particularly frequently used in this script. It performs some simple optimizations and is used to clean up the internal representation of the design between other passes.

[1]The "Register Transfer Intermediate Language", basically a netlist format with a set of built-in cell types.

```
# read design using verilog frontend
read_verilog mydesign.v

# analyze design hierarchy
hierarchy -check -top mytop

# map always-blocks to RTL netlists
proc; opt

# optimize FSM state encodings
fsm; opt

# map design to the built-in
# logic-level cell library
techmap; opt

# write verilog netlist
write_verilog synth.v
```

Figure 2. Example Yosys Synthesis Script

4 Flow Examples

This section covers three exemplary Yosys flows. The first example is a straight-forward ASIC synthesis flow in which the logic is mapped to a very simple CMOS cell library. The second example is an FPGA flow in which the logic is mapped to LUTs. The third example is a coarse-grain flow in which the logic is mapped to coarse-grain cells.

4.1 ASIC Cell-Based Synthesis

For the synthesis targeting an ASIC process, a cell library with standard cells is given and the synthesis tool shall ultimately map all logic to the cells specified in this library. In this example the cell library from Fig. 3 is used.

```
library(demo) {
  cell(NOT) {
    area: 3;
    pin(A) { direction: input; }
    pin(Y) { direction: output; function: "A'"; }
  }
  cell(NAND) {
    area: 4;
    pin(A) { direction: input; }
    pin(B) { direction: input; }
    pin(Y) { direction: output; function: "(A*B)'"; }
  }
  cell(NOR) {
    area: 4;
    pin(A) { direction: input; }
    pin(B) { direction: input; }
    pin(Y) { direction: output; function: "(A+B)'"; }
  }
  cell(DFF) {
    area: 18;
    ff(IQ, IQN) { clocked_on: C; next_state: D; }
    pin(C) { direction: input; clock: true; }
    pin(D) { direction: input; }
    pin(Q) { direction: output; function: "IQ"; }
  }
}
```

Figure 3. ASIC Cell Library (in Liberty format) which provides a NOT, NAND, NOR, and a D-flip-flop gate

The synthesis script (Fig. 4) is similar to the generic script from Fig. 2. The only difference (besides formatting) is that two additional passes have been added to the end of the script. The `dfflibmap` pass extracts all flip-flop cells from the cell library and replaces flip-flops in the design with the ones from the library, adding additional logic as necessary. Finally, the `abc` pass in `-liberty` mode uses the external tool Berkeley ABC [5] to efficiently map the remaining combinational logic to cells from the cell library.

4.2 FPGA LUT-Based Synthesis

When synthesizing for FPGAs, the logic portion of the design must be mapped to LUT (lookup-table) cells. After the `techmap` pass (see the synthesis script in Fig. 5) the logic portion of the design is represented using the Yosys' internal cell library that consists of AND, OR, XOR, NOT, and MUX gates. In the previous example, the `abc -liberty` pass was used to map this netlist to gates in the target cell library. Here the `abc -lut` N pass is used to map the logic to N-input LUT cells, again using Berkeley ABC.

```
# read design
read_verilog counter.v

# high-level synthesis
hierarchy -check -top counter
proc; opt; fsm; opt; techmap; opt

# mapping registers to ASIC cells
dfflibmap -liberty asic_cells.lib

# mapping logic to ASIC cells using Berkeley ABC
abc -liberty asic_cells.lib; opt

# write netlist
write_verilog asic_synth.v
```

Figure 4. ASIC Synthesis Script

```
# read design
read_verilog counter.v

# high-level synthesis
hierarchy -check -top counter
proc; opt; fsm; opt; techmap; opt

# mapping logic to LUTs using Berkeley ABC
abc -lut 4; opt

# map internal cells to FPGA cells
techmap -map fpga_cells.v; opt

# write netlist
write_verilog fpga_synth.v
```

Figure 5. FPGA Synthesis Script

In this example, the design should be synthesized to LUT1, ..., LUT4, and FDRE cells, as used by the 7-Series Xilinx FPGAs[2]. The `abc -lut` N pass resulted in internal Yosys LUT cells `$lut` and positive-edge flip-flops `$_DFF_P_`. The `techmap -map` pass is used to replace these with the appropriate Xilinx cells. This pass reads a Verilog file that contains implementations for internal cell types and replaces these internal cells with the given implementation. Figure 6 shows the Verilog file used for this cell mapping.

Note that the Xilinx FDRE cell actually has synchronous

[2]The Xilinx 7-Series CLB contains 6-input LUTs that can optionally be grouped to 7- or 8-input LUTs, dedicated carry logic, and contains dedicated memory resources (in M-slices). For simplicity we only use LUT1, ..., LUT4, and FDRE cells in this example.

```
module  \$_DFF_P_ (D, C, Q);
  input D, C;
  output Q;

  FDRE fpga_dff (
        .D(D), .Q(Q), .C(C),
        .CE(1'b1), .R(1'b0)
  );
endmodule

module \$lut (I, O);
  parameter WIDTH = 0;
  parameter LUT = 0;

  input [WIDTH-1:0] I;
  output O;

  generate
    if (WIDTH == 1) begin:lut1
      LUT1 #(.INIT(LUT)) fpga_lut (.O(O),
        .I0(I[0]));
    end else
    if (WIDTH == 2) begin:lut2
      LUT2 #(.INIT(LUT)) fpga_lut (.O(O),
        .I0(I[0]), .I1(I[1]));
    end else
    if (WIDTH == 3) begin:lut3
      LUT3 #(.INIT(LUT)) fpga_lut (.O(O),
        .I0(I[0]), .I1(I[1]), .I2(I[2]));
    end else
    if (WIDTH == 4) begin:lut4
      LUT4 #(.INIT(LUT)) fpga_lut (.O(O),
        .I0(I[0]), .I1(I[1]), .I2(I[2]),
        .I3(I[3]));
    end else begin:error
      wire TECHMAP_FAIL;
    end
  endgenerate
endmodule
```

Figure 6. Models for Yosys' internal FPGA cells (in Verilog) which are used to map to Xilinx Virtex 7 series FPGA cells.

reset and clock-enable pins. They would be a perfect match for the example design in Fig. 7. However, in this simple example flow, these pins stay unused. The next example will demonstrate how the `extract` pass can be used to improve synthesis results in such situations.

4.3 Coarse-Grain Synthesis

The main use-case of coarse-grain synthesis is mapping designs to reconfigurable logic which provides coarse-grain cells that operate on bit-vectors instead of single-bit signals. This can be (optional) accelerator cells such as DSP cells in modern FPGAs, or the cells of a complete heterogeneous coarse-grain architecture.

```
module counter (clk, rst, en, count);
   input clk, rst, en;
   output reg [3:0] count;

   always @(posedge clk)
      if (rst)
         count <= 4'd0;
      else if (en)
         count <= count + 4'd1;
endmodule
```

Figure 7. Simple Example Verilog Design

```
# read design
read_verilog counter.v

# high-level synthesis
hierarchy -check -top counter; proc; opt

# mapping coarse-grain cells
extract -map coarse_cells.v; opt

# write netlist
write_verilog coarse_synth.v
```

Figure 8. Coarse-Grain Synthesis Script

```
(* extract_order = 1 *)
module MACRO_INC(in, out);
  input [3:0] in;
  output [3:0] out;
  assign out = 4'd1 + in;
endmodule

(* extract_order = 2 *)
module MACRO_DFF(clk, rst, en, d, q);
  input clk, rst, en;
  input [3:0] d;
  output reg [3:0] q;
  always @(posedge clk)
    q <= rst ? 4'd0 : en ? d : q;
endmodule
```

Figure 9. Coarse-Grain Cell descriptions (in Verilog) which are used to replace subcircuits of the full design.

While the logic gates targeted in ASIC synthesis (and used as intermediate representation when packing logic to LUTs for FPGA synthesis) are of equal or smaller granularity than the operators available in Verilog, coarse grain cells often implement functions of larger granularity than the operators available in Verilog. An essential requirement for that is that the RTL netlist which keeps multi-bit operators is directly available for technology mapping, contrary to common synthesis tools, which create a bit-level netlist very early before any custom actions can be performed.

For example, a coarse-grain cell library might contain a combined multiply-add cell that implements the function $y = ab + cd$. In this case, three cells in the RTL netlist (two multipliers, one adder) must be mapped to only one cell in the cell library.

The `techmap` pass used in the previous examples does the opposite of coarse-grain mapping: It replaces single cells in the input netlist (e.g., a 4-bit adder) by netlists of multiple cells (e.g. logic gates building a chain of half- and full-adders).

On the other hand, the `extract` pass is designed for coarse-grain mapping: A set of coarse-grain cells is given by their netlists. Within the netlist of the current design, isomorphic occurrences of these sub-netlists are identified and replaced by the respective coarse-grain cells. The algorithm for this is based on the Ullmann Algorithm for Subgraph Isomorphisms [14].

For the example coarse-grain synthesis flow, the design from Fig. 7 is used. It is processed using the script given in Fig. 8. The `extract` pass uses a small cell library (Fig. 9) with coarse-grain cells for incrementing a 4 bit vector by one (`MACRO_INC`) and a 4 bit wide D-type flip-flop with synchronous reset and enable (`MACRO_DFF`). Figure 10 shows the resulting RTL netlist with the matches – as found by the `extract` pass – marked with frames.

4.3.1 Designing Reconfigurable Coarse-Grain Architectures

The design of reconfigurable coarse grain architectures is split in two phases: In the pre-silicon phase, the actual architecture is designed and optimized for the application domain. This consists of the design and selection of coarse-grain cells and the design of the reconfigurable interconnect. The post-silicon phase is analogous to FPGA development: a given logic design is mapped to the reconfigurable coarse-grain architecture.

One of the most challenging tasks in the pre-silicon phase is the identification of design partitions which are suitable for coarse-grain cells. In the case that the application domain is specified by a set of example designs, frequent subgraph mining can be used to identify common patterns in the RTL representation of the example designs. The `extract` pass has limited support for frequent subgraph mining and can therefore be also used in

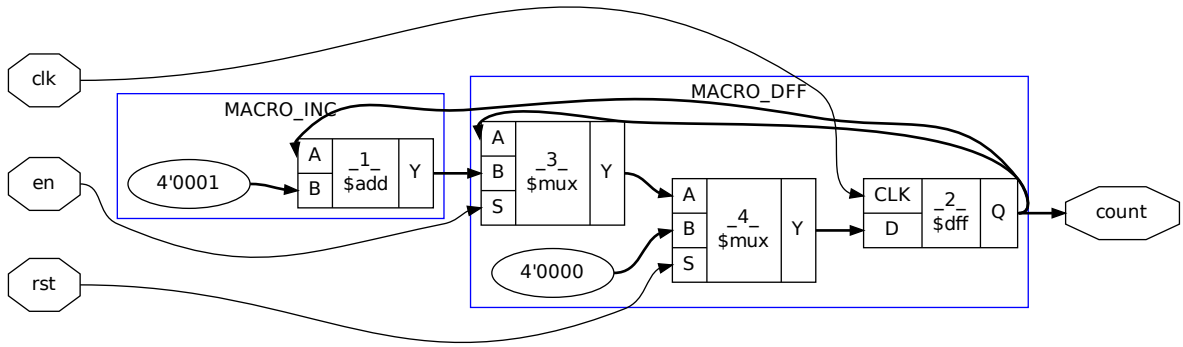

Figure 10. RTL Netlist with Coarse-Grain Cell Matches. Generated with GraphViz and the Yosys `show` command.

the analysis of the application domain in the beginning of the design process.

A more detailed description of how to use Yosys for designing a heterogeneous coarse-grain reconfigurable architectures is given in [9].

4.3.2 Challenges in Coarse-Grain Synthesis

The most challenging problem in coarse-grain synthesis is that there usually are many ways of expressing the function that can be implemented by a coarse-grain cell in HDL code. For example, there are simple symmetries that can be handled by the `extract` pass automatically, such as commutative operators. Note that the expression "`count + 4'd1`" in Fig. 7 corresponds to the expression "`4'd1 + in`" in Fig. 9. The constant and the signal have swapped places. The `extract` pass is equipped with a set of rules, which specify for example that the add-operator is commutative and thus still matches the circuits.

Another common challenge in coarse-grain synthesis are cells of different bit-width. For example, a 16-bit ALU cell is also able to perform 12-bit arithmetic. Therefore it should also be mapped to smaller arithmetic operators. The backend for the `extract` pass does support variable bit widths for the matches, but at the moment of writing this feature is not exposed through the `extract` command.

Even with this features, it is still possible to describe functionality in a design that is equivialent to a coarse-grain cell, but is not described by means of a circuit that is isomorphic to any of the cell descriptions passed to the `extract` command for that cell type. Therefore, designing for a coarse-grain synthesis flow usually involves iteratively adding additional cell descriptions for cases that have not been covered before.

In complex architectures it might be more suitable to use a synthesis script that uses multiple runs of the `extract` pass, that first map to intermediate cell types before mapping those to the final cell library, or combinations of alternating runs of `extract` and `techmap`.

Another challenge is choosing the right substitutions when different conflicting possibilities exist. This is a complex problem[3]. Yosys simply applies the substitutions as they are found, without considering if and how this would effect other possible substitutions. Therefore it is recommended to run `extract` for the most beneficial substitutions first. The order in which the possible substitutions are evaluated can be set using the `extract_order` attribute in the cell library file (see Fig. 9).

These challenges are, of course, less pressing when the coarse-grain cells are optional accelerator cells (such as DSP cells in modern FPGAs), than when the entire target architecture is coarse-grain and therefore all logic in the design must be mapped to the coarse-grain cell types.

5 Evaluation

The correctness of Yosys' Verilog implementation has been tested and verified using the following three methods:

1. A simple test suite with more than 200 small Verilog modules was simulated with testbenches to verify the correctness of the synthesis results.

2. A collection of more than 6000 auto-generated Verilog modules that systematically test a wide range of Verilog features for combinational logic was synthesized with Yosys as well as with three different commercial synthesis tools. The Yosys netlists were successfully checked for logical equivalence with the netlists generated by the commercial tools using Yosys's built-in SAT solver [6]. This test has also revealed some bugs in the commercial tools used as reference. Majority voting was used to determine the correct behavior if the commercial tools produced non-equivalent results.

3. A small collection of real-world designs ranging from 300 logic gates to 40.000 logic gates, including the OpenRISC 1200 CPU [2] and the OpenMSP 430 CPU [1]. The correctness of the synthesis results for this designs has been verified using a commercial formal verification tool for logical equivalence checking.

[3] For general graphs this is an NP-hard problem that can only be solved with heuristics for larger inputs, but in the special case of trees it can be solved efficiently using dynamic programming [11, page 509ff].

Especially the last test shows that Yosys is feature-complete enough to handle many real-world Verilog designs [16]. However, there are some features in Verilog-2005 that have not been implemented in Yosys so far, for example multi-dimensional arrays and initialization of ROM using the `$readmemb` and `$readmemh` system tasks.

6 Conclusions and Future Work

The research on synthesis algorithms as well as other digital logic design issues can greatly benefit from HDLs like Verilog or VHDL as input data format. This requires an RTL synthesis tool with open and flexible interfaces to connect the implementations of the actual research topic.

This paper introduces Yosys, which is the first free and open-source logic synthesis tool that supports most of the synthesizeable subset of Verilog-2005. It is capable of synthesizing Verilog to a logically equivalent netlist which can be further used for common fine-grain logic optimization. Contrary to commercial synthesis tools, Yosys also provides functionality for coarse-grain synthesis tasks as well as flexible and open interfaces, which establishes its vast extensibility. In addition to synthesis-related features, Yosys also provides some features for digital circuit analysis, such as a built-in SAT solver.

Future work includes more synthesis passes, driven by the requirements of applications that use Yosys as well as additional interfaces to external tools, similar to the interface to ABC. Future work also includes more features on the analysis-side, such as a more powerful SAT-based formal equivalence checker and commands for reverse-engineering circuits. It is also planned to create a reference FPGA flow that is capable of using dedicated DSP and memory resources on modern FPGA architectures. Finally, a couple of additional small utility commands for modifying and analyzing designs within Yosys are planned for the near future.

Yosys is meant to be used as framework for testing new algorithms in the context of a full-featured synthesis flow. The Yosys source code is well-structured and the individual commands are well separated within this structure. We hope that others will also find this framework useful and contribute interesting new commands to it.

Yosys is free software under the terms of the ISC license and is available at `http://www.clifford.at/yosys/` and `https://github.com/cliffordwolf/yosys`.

References

[1] openMSP430 CPU. `http://opencores.org/project,openmsp430`.

[2] OpenRISC 1200 CPU. `http://opencores.org/or1k/OR1200\_OpenRISC\_Processor`.

[3] Parvez Ahmad. HDL Analyzer and Netlist Architect (HANA). Verison linux64-1.0-alpha (2012-10-14), `http://sourceforge.net/projects/sim-sim/`.

[4] Armin Biere, Johannes Kepler University Linz, Austria. AIGER. `http://fmv.jku.at/aiger/`.

[5] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. HQ Rev b5750272659f, 2012-10-28, `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[6] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[7] Daniel D. Gajski, editor. *Silicon Compilation*. Addison-Wesley Publishing Company, Inc., 1988.

[8] Daniel D. Gajski. *Principles of Digital Design*. Prentice Hall, Upper Saddle River, NJ, 1997.

[9] Johann Glaser and Clifford Wolf. Methodology and Example-Driven Interconnect Synthesis for Designing Heterogeneous Coarse-Grain Reconfigurable Architectures. In Jan Haase, editor, *Advances in Models, Methods, and Tools for Complex Chip Design — Selected contributions from FDL'12*. Springer, 2013. to appear.

[10] MVSIS group at Berkeley studies logic synthesis and verification for VLSI design. MVSIS: Logic Synthesis and Verification. Version 3.0, `http://embedded.eecs.berkeley.edu/mvsis/`.

[11] G. D. Hachtel and F. Somenzi. Logic Synthesis and Verification Algorithms, 1996.

[12] Peter Jamieson and Jonathan Rose. A Verilog RTL Synthesis Tool for Heterogeneous FPGAs, 2005.

[13] The VIS group. VIS: A system for Verification and Synthesis. Version 2.4, `http://vlsi.colorado.edu/~vis/`.

[14] J. R. Ullmann. An Algorithm for Subgraph Isomorphism. *J. ACM*, 23(1):31–42, January 1976.

[15] Stephen Williams. Icarus Verilog. Version 0.8.7, `http://iverilog.icarus.com/`.

[16] Clifford Wolf. Design and Implementation of the Yosys Open SYnthesis Suite. Bachelor Thesis, Vienna University of Technology, 2012.

[17] Clifford Wolf. Evaluation of Open Source Verilog Synthesis Tools for Feature-Completeness and Extensibility. Unpublished Student Research Paper, Vienna University of Technology, 2012.

[18] Clifford Wolf, Johann Glaser, Florian Schupfer, Jan Haase, and Christoph Grimm. Example-Driven Interconnect Synthesis for Heterogeneous Coarse-Grain Reconfigurable Logic. In *FDL Proceeding of the 2012 Forum on Specification and Design Languages*, pages 194–201, 2012.