
An environment for the automatic verification of digital circuits

Author:

Javier de San Pedro Martín

Supervisors:

Jordi Cortadella Fortuny

Josep Carmona Vargas

Dept. de Llenguatges i Sistemes Informàtics

Facultat d'Informàtica de Barcelona



May 5, 2011

DADES DEL PROJECTE

Títol del projecte: An environment for the automatic verification of digital circuits

Nom de l'estudiant: Javier de San Pedro Martín

Titulació: Enginyeria Informàtica

Crèdits: 37,5

Director: Jordi Cortadella Fortuny

Codirector: Josep Carmona Vargas

Departament: Llenguatges i Sistemes Informàtics

MEMBRES DEL TRIBUNAL (*nom i signatura*)

President: Jordi Petit Silvestre

Vocal: Carina Gibert Oliveras

Secretari: Jordi Cortadella Fortuny

QUALIFICACIÓ

Qualificació numèrica:

Qualificació descriptiva:

Data:

Abstract

The aim of this project is to implement a system for the automatic verification of digital circuits written in a high-level hardware description language (Verilog), to be potentially used to assist a electronic design automation course.

It is desirable for students to be able to acquire experience with any given HDL by trying to design actual, but low complexity, circuits. However, it might not be feasible for teachers to evaluate all of the students exercises. On the other hand, giving sample test inputs and outputs is not an exhaustive method of verification that might allow hidden defects in the circuit design to go by undetected. And publishing working model circuit designs for each of the problems defeats the purpose of allowing the students to be creative in their solutions.

Therefore, we present a system where the course teachers or administrators will publish a list of *problems* (specifications for either sequential or combinational circuits – *What should the circuit do? What are its inputs? Outputs?*), along with sample correct HDL implementations of such specifications (that will not be shown to the student). Students will be able to log into the system, search for a suitable problem adjusted to their current skill level, download its specification, and try to implement it, using their favorite environment and simulation tools for the selected HDL. They might test their designs with their own test benches.

When they are comfortable enough with their design, they will be able to submit it to the system, which will model an automaton of both the student circuit and the teacher provided working one, and then compare them both, in a few seconds time. The student will then be able to see instantly if their circuit matches the specification, or if it does not. In the latter case, the system might provide an example input signals trace that causes the student's circuit to fail, at the discretion of the course administrator.

In order to reduce the scope of the work, Verilog has been selected as the HDL of choice for the project. We will not support any other HDL. Also, we will use the existing Icarus Verilog software and Jutge.org platform.

Contents

1	Background	4
1.1	Hardware Description Languages	4
1.1.1	Verilog	7
1.1.2	Icarus Verilog	12
1.2	Model checking	15
1.2.1	Symbolic Model Checking	17
1.2.2	Computation Tree Logic	18
1.2.3	NuSMV	19
1.3	Jutge.org	22
1.4	Previous work	26
2	Design	29
2.1	Goals	29
2.2	Verilog to NuSMV conversion	32
2.2.1	Analysis	32
2.2.2	Data model	37
2.2.3	Algorithm	38
2.3	Jutge.org driver	42
2.3.1	Circuit equivalence checking	45
3	Implementation	48
3.1	The tgt-nusmv converter	48
3.1.1	Data layer	49
3.1.2	Main processing	50
3.1.3	Library of Parameterized Modules	55
3.2	The Circuit Verifier Jutge.org driver	57
3.2.1	Problem preparation	57
3.2.2	Submission and correction process	59

4	Results	62
4.1	Use of the converter	62
4.1.1	Examples	63
4.1.2	Performance and resource usage	66
4.2	Sample student interaction	68
4.3	Creating a new problem	74
5	Conclusions	76
5.1	Personal conclusions	76
5.2	Cost study	77
5.3	Future work	79
5.3.1	Model readability enhancements	79
5.3.2	Additional language features	80
5.3.3	Multiple clocks	80
5.3.4	More relaxed circuit specifications	81
5.3.5	Better output messages	82
5.3.6	VHDL support	82
	Bibliography	83
	Glossary	85
A	Language support	87
B	Environment setup	89

Chapter 1

Background

1.1 Hardware Description Languages

A Hardware Description Language (HDL) is basically a computer language whose main mission is to convey the description of an electronic circuit.

Hardware description languages are older than what usually expected. It is widely regarded that the first language considered an HDL was the Instruction Set Processor (ISP) [1], in use since the early 70s. It was created as a way to describe the design of existing commercial processors, and, in fact, the PDP-11/45 Processor Handbook had such description in ISP [2]. Processors were becoming increasingly complex with time and old school schematics failed to provide a way to understand the design through all the meaningless noise.

Thus, the main goal of ISP was to allow for easier human comprehension of the design of larger processors. The language had many different layers of detail so that, by looking at the higher levels, *what* the processor did was clear to human readers. But by looking at the lower level, repetitive details, it was possible to unambiguously implement a machine identical to the one described up to the gate level.

There is no mention at all at how the actual addition is done on the counter described in listing 1.1: just an addition operator. However, clarity has improved greatly, and the intentions of the circuit designer are more clear to the reader. When the **Start** switch is pressed, since its on the right side of an *evoke* (\Rightarrow) operation, the evaluation of the action sequence at its right side is started, and the accumulator register **A** is cleared. Analogously, the counter is incremented when the **Increment** switch is pressed.


```

Register Declarations
  A\Accumulator<0:7>
Console Switches
  Start.on;
  Increment.on;
Interpreter
  Console.activity := (Start.on => A ← 0);
                   (Increment.on => A ← A + 1);

```

Listing 1.1: An eight-bit digital counter in ISP

The counter snippet is also an example of why existing imperative languages could not really be used without modification as HDLs, as seen on listing 1.2.

```

char a;
for (;;) {
  if (Start) {
    a = 0;
  }
  if (Increment) {
    a = a + 1;
  }
}

```

Listing 1.2: Same counter in C99

On the C version, detection of both the Start and Increment buttons is sequential, that is, pressing both switches at the same time has a well-defined behavior. That is not the case on real hardware, which is essentially fully parallel. Therefore, HDLs have to take into account concurrency as a core feature of the language.

In fact, on the ISP example, the ; operator is actually a parallel composition operator: **both** statements happen at the same time. Accordingly, if the operator manages to press both front console switches at the same time, both action sequences will run, and the counter will try to reset and increment itself at the same time: the result is undefined ¹.

This is a fundamental concept of HDLs. In effect, the following is also

¹A physical design implementing such counter using a multiplexer would probably have a defined behavior though (it would either Reset or Increment). However, both designs would be valid according to what was specified in the HDL.

well defined on ISP, and virtually any HDL:

```
| A ← B ; B ← A
```

On a classical imperative language, one would expect both A and B to get the original value of B . On ISP, the A and B registers contents would be swapped. If one really desires to get the imperative behaviour back, the following ISP construct would cause it:

```
| A ← B ; NEXT B ← A
```

With ISP-like languages, a computer instruction set could be described in a quite high level:

```
| Branches and Subroutines Calling :
   JMP ⇒ (PC ← D) ;
   BR  ⇒ (PC ← PC + offset) ;
   BEQ ⇒ (Z ← (PC ← PC + offset)) ;
   ...
```

Listing 1.3: Snippet from the PDP-11 manual[2]

Of course, it was soon realized that by adding a bit of computer-parsable structure to the language, it could be fed to simulation software and *simulate* the computer. In fact, the description of a processor in ISP already looked much like a software implementation of an interpreter for the instruction set from such processor – and there were already such simulators. Statements concerning the delays of gates were added to the language, so that simulation was more realistic (see the **PREVIOUS** function at [1]).

From that, the next logical step would have been for the ISP to be actually used as source to build the computer it represented, by an automated process instead of a laborious manual process. Such a process was already envisioned for at least “limited design activities” [1] in the 70s, but not for the entire circuit.

However, this changed in the 80s, with the introduction of several new, more well-defined HDLs (Abel, Verilog, VHDL, ...) as well as synthesis tools for some of them.

These days, both VHDL and Verilog share a first place in developer mind share, with many other recent additions far behind but slowly gaining ac-

ceptance (like SystemC). Most commercial synthesis tools provide support for both VHDL and Verilog, with the differences between each of them disappearing after new versions of the languages appear. Despite that, Verilog is usually regarded as a less verbose language, easier to become acquainted with than VHDL.

1.1.1 Verilog

Like ISP, Verilog was also initially designed as a simulation language by then Automated Integrated Design Systems ² in the early 80s. It was quickly marketed with the introduction of the Verilog-XL simulator in 1985. At around the same time, however, the United States Department of Defense was busy designing another HDL (VHDL) they would use to precisely describe the many ASICs they were sourcing from external companies, as reconstructing them from the technical manuals when the original company was long gone started to become unfeasible.

Since the Department of Defense planned to release the language specifications to the general public without any usage restrictions, Gateway believed designers would quickly favor it versus then proprietary Verilog. Therefore, on 1991 the Verilog language specification was also made freely available, and on 1994 it was proposed as the IEEE standard 1364.

Verilog has a syntax that is clearly modeled upon C's, except for the Pascal-influenced `Begin`, `End` used as block delimiters instead of the C curly brackets.

Structural Verilog

On Verilog, the core concept is the *module*, which can be seen as something akin to a digital circuit *component*. Much like a real world component, a module can contain other modules – even many copies of a single module, each of them called *instances*. Also, like a real world component, modules have *ports*, wires that connect the internals of the module with other components outside the module. In order to instance a component, the programmer has to indicate which wires each to connect to each of the ports in the component.

²now Gateway Design Automation, one of the largest EDA companies

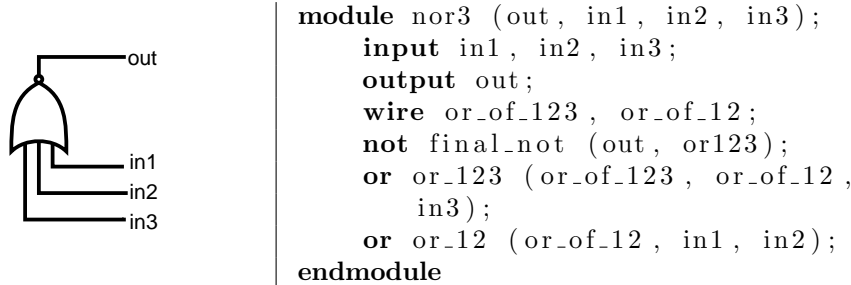


Figure 1.1: An implementation of a 3-input NOR gate using two OR and one NOT gate.

As seen on the above figure, the syntax actually resembles that of the original K&R C[3]. It is declaring a module named `nor3`, with `out`, `in1`, `in2`, `in3` as ports. Where on a K&R function declaration one would write the type information for each of arguments, on a Verilog we declare whether the ports are inputs or outputs, a restriction that will be enforced by the synthesizer to both users of the module as well as its internals.

The `nor3` module is built using one instance of the `not` module (the instance being called `final_not`), and two instances of the `or` module (called `or_12` and `or_123`), as per the following diagram:

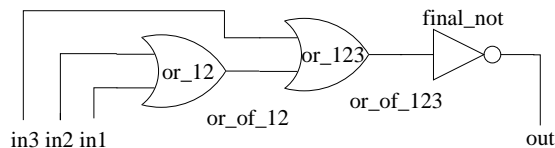


Figure 1.2: An example synthesized circuit for the NOR3 module.

The module also contains two `wire` declarations, `or_of_12` and `or_of_123`. While the first `or` has the 1st and 2nd inputs connected directly, there would be no way to connect the output of that `or` with the `or_123` that has the 3rd module input and the result of the previous `or` gate as inputs. Therefore, we need to declare an intermediate wire, internal to the module, that will connect both pins.

Buses are a way to aggregate multiple related wires so that many of them can be connected easily (same way a designer would combine them on a diagram instead of drawing many wires). On Verilog, this is done by

specifying both the least and most significant indexes of the bus in the wire declaration:

```

module bus(in , out);
    input [1:0] in; // A two-bit bus
    output [1:0] out; // Another two-bit bus

    and and1(out[0], in[0], in[1]);
    or or1(out[1], in[0], in[1]);
endmodule

```

Listing 1.4: A sample module with buses

Buses should not be confused with *arrays*, which are another kind of aggregation. Arrays are often used to implement memories, as most synthesizers allow for selection of individual items inside an array to be made with non-constant indices (i.e. using addresses coming from outside the module). Such restriction is though usually applied to buses. On the other hand, buses allow for selection of more than one item simultaneously, a feature not available while using arrays.

The language also offers an alternate syntax to implement the *nor3* module (seen at figure 1.1), where instead of explicit module instantiations, we use a shorter *continuous assignment* expression to instantiate all the required gates and realize the necessary connections:

```

module nor3 (out, in1, in2, in3);
    input in1, in2, in3;
    output out;
    assign out = ~(in1 | in2 | in3);
endmodule

```

Listing 1.5: A less-verbose implementation of a 3-input NOR

Expressions inside continuous assignments can use the standard set of C-like operators (see table 1.1 on page 10), as well as numeric literals with a specific syntax (`<size>'<base=d(ec),h(ex),b(in),o(ct)><number>` – 3'd10 is decimal number 10 represented using 3 bits).

Care must be taken while using continuous assignments as a very short expression can be synthesized into thousands of gates, e.g. an arithmetic division of 64-bit buses. Moreover, undriven nets (not connected to the output of any gate) have the value `x` (unknown), which is propagated along each operand using such value as input, causing problems if not accounted for.

Operator	Description	Example
[]	Bit access (to individual bits from a bus)	<code>wire = bus[2];</code> <code>bus2 = bus[1:0];</code>
{ }	Bit concatenation (aggregates individual wires to create a bus)	<code>bus = {wire1,</code> <code>wire2};</code>
+ - * / %	Unsigned arithmetic (can work on buses; output is a bus of the same width)	<code>bus = bus1 + bus2;</code>
> >= < <=	Unsigned relational (can work on buses but returns a single bit)	<code>wire = bus1 <=</code> <code>bus2;</code>
! &&	Logical (treat buses as <i>true</i> if any of its bits is 1; return a single bit)	<code>flag = flag1 </code> <code>flag2;</code>
~ & ^(<i>xor</i>)	Bitwise operations	<code>busres = bus1 </code> <code>bus2;</code>
== !=	Equality tests	<code>wire = bus1 ==</code> <code>bus2;</code>
& ~& ~	Reduction (works on each of the bits of a vector, returning a single bit)	<code>nand = ~& bus;</code>
<< >>	Logical shift	<code>buso = busi << 3;</code>
<<< >>>	Arithmetic shift	<code>buso = busi >>> 1;</code>
? :	Conditional (ternary operator)	<code>res = cond ?</code> <code>iftrue : iffalse;</code>

Table 1.1: List of Verilog operators[4]

Behavioral Verilog

Verilog is divided into two subsets: *structural* and *behavioral* Verilog. All of the above examples have been about structural Verilog, which virtually maps 1:1 to a gate-level description of a circuit. The other subset, behavioral, greatly simplifies the design of sequential circuits. While designers could model such circuits using flip-flop modules alone, Verilog offers an higher-level way to do it, the **always** block (called an *always process* in technical terminology).

An **always** block contains a set of instructions that are to be executed continually and perpetually. Commonly, this set of instructions starts with a list of events to wait for (like the rising edge of the `clk` signal in listing 1.6) plus a block of instructions to be executed when the event is signaled.

```
module dff (clk, q, d);
    input clk, d;
    output q;
    reg r;
    always @(posedge clk) begin
        r <= d;
    end
    assign q = r;
endmodule
```

Listing 1.6: A simple D flip-flop implementation

Nevertheless, this is not mandatory, and an **always** process could have no conditions to wait on – thus always executing continually –, or could even have multiple secondary sets of events to wait after the primary one is signaled.

Assignments (\leq) can appear on those instructions, but only *regs* can appear on the left side. All Verilog operators (see table 1.1) can be used on the right side – only bit access and concatenation can be used on the left side.

Similarly to ISP (see section 1.1 on page 4), such assignments are done in parallel: firstly, all of the right sides are evaluated, then the results written.

Like ISP, Verilog has a way to force an assignment to be done sequentially, by using the $=$ operator instead of \leq . However, this is usually frowned upon as it makes synthesis harder and the synthesized circuit larger.

Apart from the assignment instruction, control structures like **if** and **while** are found in the language, with the usual C syntax (except the use of **Begin** and **End** instead of curly braces). Of special interest is Verilog's **case** control structure,

Non synthesizable Verilog

Verilog has also many other features that do not map nicely to physical circuits, and are thus usually used to enhance the use of Verilog as a simulation language. For example, the **initial** statement, which runs code as soon as the simulation is started; or **delays**, which prolong the time between the evaluation of the right part and the record of the result on the left part and can be used on the same places the syntax allows *wait for event* (**@**) controls

(it is, after all, an event – x time has passed).

For example, this functionality is used to implement a module that simulates both the reset and clock signals:

```

module clkgen(clk , rst);
  output clk , rst;
  reg clk , rst;
  // Alternate the clock every simulation cycle
  always #1 clk <= ~clk;
  initial begin
    clk <= 0;
    rst <= 0;
    // Delay 50 simulation cycles , then raise rst
    #50 rst <= 1;
    // Delay 50 extra sim. cycles , then clear rst
    #50 rst <= 0;
  end
endmodule

```

Listing 1.7: A clock generation simulation-only module

Many extensions have been made to the language since its IEEE standardization on 1995. On 2001, the more modern ISO C-like syntax for declaring modules was adopted by the IEEE standard (as on listing 1.8), along with signed arithmetic operators and many additions to the system *tasks* library (which could be described as a set of functions and procedures useful in a simulation environment, like `fprintf`).

```

module nor3 (output out , input in1 , input in2 , input
  in3);
  assign out = ~(in1 | in2 | in3);
endmodule

```

Listing 1.8: The nor3 module in even shorter Verilog-2001 accepted syntax

1.1.2 Icarus Verilog

Icarus Verilog is one of the many available simulation and synthesis tools for the Verilog language. It is free and open source software, licensed under the GPL and openly developed under the guidance of its original designer, Stephen Williams, who started it around the 1990s. [5]

The design of the application is heavily modular. The frontend application (*iverilog*) accepts a backend as a command line configuration parameter. The frontend application will virtually only parse the source code. The backend has to do any synthesis work or code generation if required. Therefore, Icarus Verilog is both able to handle synthesis work, for example by using the *fpga* backend, or simulation, by using the *vvp* backend (a lower-level simulation program that is also included with the Icarus Verilog distribution).

The process used by the software is well-defined:

1. *Source preprocessing*: done by *ivlpp*. Handles all the Verilog language preprocessor statements.
2. *Source parsing*: done by the *ivl* frontend using the well-known *Flex* and *Bison* parser generators. Generates an abstract syntax tree.
3. *Design elaboration*: the syntax tree is converted into an *elaborated design*. This is a tree-like representation of the program itself, not the source. A *Module* root class would have each of its ports, wires, *always* and *initial* blocks as direct children.
4. *Functors*: from this point on, the process is guided by the user selected backend, which contains a set of *functors* – small graph filters, whose input is a graph and each generate a slightly modified graph. For example, a synthesis backend might want to use the *synth2* functor, that, among many other functionalities, generates a D flip-flop for each assignment inside an *always* block with a inferred clock signal, then removes such assignment from the tree.

On the other hand, a simulation backend might want to use the *cprop* functor, which, as its name indicates, implements a constant propagation optimization pass.

5. *Code generation*: the backend is once again called after all processing has been done so that it can massage the resulting graph into an appropriate format. If all the synthesis functors have been called, the graph could now be printed in order to get a gate-level diagram of the design.

Backends, called *targets* in Icarus Verilog, are composed of both a text file indicating the functors to be run as well as the actual executable code (in the form of a standard shared object or dynamic link library) to run at the code generation stage. A public C API is available so that the targets can explore the elaborated design tree (after functors have been applied): it is a simple object-oriented set of accessor functions to *opaque pointers* to nodes

from the graph.

```

/* Opaque pointer to a builtin logic/gate device. */
typedef struct ivl_net_logic_s *ivl_net_logic_t;
/* All of the builtin logic types. */
typedef enum ivl_logic_e {
    IVLLO_AND      = 1,
    IVLLO_NAND     = 6,
    IVLLO_NOR      = 8,
    IVLLO_NOT      = 9,
    IVLLO_OR       = 12,
    ...
} ivl_logic_t;
/* Gets the type of a logic device. */
extern ivl_logic_t ivl_logic_type(ivl_net_logic_t net);
/* Gets the device connected to a certain pin of this
 * logic device. */
extern ivl_nexus_t ivl_logic_pin(ivl_net_logic_t net,
    unsigned pin);

```

Listing 1.9: A brief snippet of the public target API

Since this project implements a Icarus Verilog target, this API will be used extensively.

While the current stable release of Verilog, at the time of writing this document, is 0.9.3, from the 0.9.x series of releases, we have decided to use 0.8.7 for this project instead because:

- The 0.8.x series is an evolution of the older Icarus Verilog codebase, which has mostly working synthesis support. However, it lacks support for most features from the newer language standards as well as some less used functionality from the current standard [6]. There will be no new features developed for this branch, only bugfixes. In spite of that, there is still some unofficial work done on it.
- The 0.9.x series are an huge improvement on language support and performance over the older series. Unfortunately, compatibility with the existing 0.8.x targets was broken, as well as the synthesis features, which did not work as well as with the 0.8 series.
- The future 0.10.x was initially expected to have mostly new features, like SystemVerilog, Analog Verilog or even VHDL support in the frontend. Nonetheless, no new work in the synthesis backend due to lack

of interest. However, the recent increasing interest in the 0.8.x series might change that.

A typical simulation session with the Icarus Verilog toolchain might be the following:



Figure 1.3: An example simulation session with Icarus Verilog

1.2 Model checking

When testing a complex concurrent system, for example, a large circuit designed using Verilog, and save for basic human evaluation which is slow and prone to errors, virtually the only other method that has been used is simulation, as described on the previous section.

However, simulation has its own share of problems. It is not exhaustive; it largely depends on the skill of whoever writes the test cases, which means it still largely requires human intervention. Thus, it is a lengthy process, and does not usually achieve 100% coverage of the system functionalities under test.

This is not desirable; we want to develop a system that is as automated as possible.

There is an alternate way. A circuit can be modeled as a finite state machine; after all, a idealized logical digital circuit has a few elements storing actual state information, like latches and flip-flops. All those elements compose the state variables of the model, and from the combinational logic we can deduce the rules that control transitions between states.

After such state machine model has been created, and since it is a finite model, we can verify certain properties. For example, we can ask if on all possible states, property P holds. Or if it holds only on states that can be directly reached from state S , and property $\neg P$ holds on all others, where such property can be, in an example related to the state machine show on figure 1.4, ‘is the counter zero?’.

We can easily implement a system checking such properties by representing the state machine as a graph like the one in 1.4, and from then on,

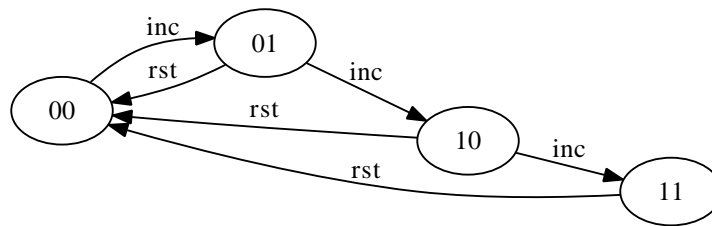


Figure 1.4: A two-bit counter modeled as a state machine

enumerating all the states, checking if property P applies to all of them. Enumerating all the states directly reachable from state S as well as indirectly reachable is just a matter of examining the graph connectivity.

If the system can reach a state T where property P does not hold but *should* according to what was described by the user, it is also easy by the checker to emit the path it had to take to reach from the initial state to T . In consequence, the user not only knows the system does not satisfy the properties it was asked for, but also the chain of states that brings it to the error condition.

Thus, one could check correctness of a system by specifying the right set of properties or specifications to check for, and then run the automated model checker. As a subbranch of such verifications, we find *equivalence checking*, where the correctness of a system is asserted by comparing it with a system that is known to be correct.

Even so, the checker system has to build a graph with all of the states of the system, which means the problem quickly gets out of hand as the number of the states increases. On a digital circuit, the maximum number of states is usually 2^n where n is the accumulated size of all the state-storing elements in the circuit (flip-flops and latches).

As a consequence, while 2 bits of information means the system has 4 states (as the counter described on figure 1.4), 16 bits of information means the system has 65, 536 states, and more than four billions for a mere 32 bits of information – the well-known powers of two sequence, appropriately named the *state explosion* problem.

1.2.1 Symbolic Model Checking

On 1992, *SMV*[7] was introduced, which pioneered *Symbolic Model Checking*: an approach to model checking that was to solve the state explosion problem by using boolean formulas to represent sets and relations between states and avoid ever having to construct the entire state graph.

This was mostly accomplished by the use of *Binary Decision Diagrams*. Those structures are used to represent boolean functions (of the kind $\{True, False\}^k \rightarrow \{True, False\}$, i.e. functions that work over a vector of boolean values and return a single boolean result, like AND, OR, ... gates) in an efficient form.

For example, if we were to build a (non-optimized) decision diagram for a classical AND-gate (defined as $and : a, b \rightarrow a \wedge b$):

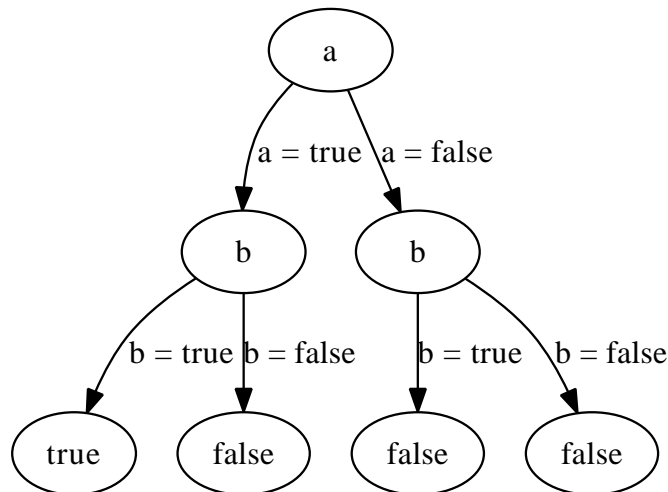


Figure 1.5: Binary decision diagram for the AND function

It can be seen that evaluating the *and* function from the diagram is just a matter of starting from the root and taking the proper path. However, the interesting aspect of BDDs is that they can be reduced into *Shared Reduced Ordered Binary Decision Diagrams*, like the one on figure 1.6.

After reduction, the diagrams are more efficiently stored. However, doing the actual reduction is a hard problem by itself (in truth, NP-Complete[8]), as it might require finding the proper ordering of all the decision branches so as to be able to create the minimal canonical version of it.

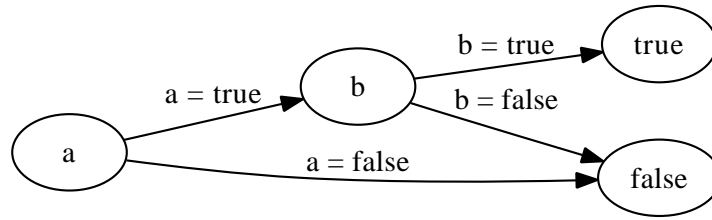


Figure 1.6: Reduced binary decision diagram for the AND function

A state model can be represented with boolean functions – whether the system is in a certain state or not can be modeled as vector of booleans, and in fact, if we are modeling a logical circuit, the entire state information from the sequential circuit is actually this vector of booleans. With that in mind, transitions between states can be modeled by a boolean function that takes two such state boolean vectors (and the current input state), and returns a boolean value indicating whether from the first state and with that input the system will jump to the second.

Kenneth L. McMillan’s PhD. thesis[7] concentrated on developing a working model checker (*SMV*) that was capable of handling many more states than traditional checkers of the era – by using BDDs – and introduced algorithms to perform many operations on boolean functions represented as BDDs, like the union and intersection operations, allowing much of the property checking that was previously done on the state graph to be calculated without such an expensive data structure.

1.2.2 Computation Tree Logic

When defining properties to be checked on a state model, it was proposed that a formal logic system – like the ones used by philosophers – could be used to represent properties such as the ones described on section 1.2: ‘do all the states reachable from S hold property P ?’.

Computation Tree Logic (CTL) is one of the existing temporal logics, whose usage was introduced by Pnueli[9] in 1977 and which we will use extensively during this project as it is enough for our requirements.

On CTL, apart from the usual logical operators (\wedge , \vee , \neg , \dots), one can find a series of operators formed from combining a set of affixes (see table 1.2) forming a finite set of operators with a well defined meaning.

Prefix	Meaning	Description
A	<i>Always</i>	For all the transitions from the current state $S \dots$
E	<i>Exists</i>	At least in one transition from the current state S \dots
Suffix	Meaning	Description
G	<i>Globally</i>	$\dots P$ is true in all reachable states
X	<i>neXt</i>	$\dots P$ is true in all directly reachable states
F	<i>Finally</i>	$\dots P$ is true in a reachable state

Table 1.2: The main affixes forming CTL operators

For example, a specification consisting of **AG** P would mean that the specification we want to check is that “*for all the transitions from the current state S , P is true in all reachable states*”, where S , the current state, is the initial state. Thus, we want to check that P holds for *all* the states.

On the other hand, **EF** $(P \vee Q)$ would mean that “*At least in one transition from the current state S , $P \vee Q$ is true in a reachable state*”. That is, from the initial state, there is at least one path to a state T where either property P or Q will be true.

Many more advanced combinations can be created using those operators. For instance, **AG** $(P \wedge (\text{EX } Q))$ would mean that for all states, property P holds and a way exists to directly reach a state where property Q also holds.

One could think for example that P is a safety enforcement property (‘the system is not stuck’ and that Q would mean ‘the system is off’. We would be enforcing that there is no state where the system is stuck and that from every state we can immediately reach a state where the system is off – after model checking, we can now be sure we would be able to shut such a complex system down.

A description on how to implement CTL checking in a symbolic model checker can be found at [7].

1.2.3 NuSMV

NuSMV[10] is a descendant of the original SMV tool developed in 1992 by Kenneth McMillan[7]. Thus, it is a Symbolic Model Checker implementation as described on section 1.2.1. One of the best known and fastest model checkers, it is under continuous development, and since the release of version

2 of the program, also open source software under the copy-left *LGPL 2* license.

The latest stable version as of the writing of this report was 2.5.2 .

Like most other model checking tools, it defines its own language by which to define the finite state model. The language is much more featured than a simple states list plus a dump of the state transitions, as the easiness of how complex systems where converted into finite state models was greatly priced.

On the NuSMV language, like on Verilog, the core concept is the module. Inside modules, *state variables* can be find – those represent the actual state information, and are defined under the **VAR** section inside the NuSMV module.

```

MODULE counter(i, rst)
VAR
  c : 0..3;
ASSIGN
  init(c) := 0;
  next(c) := case
    rst : 0;
    i : c + 1;
    TRUE : c;
  esac;

```

Listing 1.10: The model whose state graph is show on figure 1.4

In the example on listing 1.10, we have defined a counter model with a state variable *c* whose type is a integer in the range between 0 and 3 inclusive. This is syntactic sugar for hiding the actual state machine: NuSMV will convert this type into raw boolean variables representing the states behind the scenes.

NuSMV has a handful of built-in types, listed in table 1.3.

State transitions are represented in the **ASSIGN** sections, with two clauses. The **init**(*var*) := *value* statement assigns the initial state for the state variable – otherwise, it is left undefined, and the checker will have to consider all states as potential initial states. In the example, the initial state for the counter is zero.

The **next**(*var*) := *expr* statement is the one that models the actual transition, by indicating that the value of *var* in the next state will be the

Type	
<code>boolean</code>	
<code>a..b</code>	(ranged integer)
<code>item1, item2, ...</code>	(enumeration)
<code>word[n]</code>	(boolean vector of size n)
<code>array a..b of</code>	(vector of any type)

Table 1.3: NuSMV language built-in types

result of evaluating *expr* – which can reference *var* so that the value in the current state can be used (as well as the values of all other state variables and inputs).

In the example, one of NuSMV’s most complex operators is used: the **case** switch. It checks each of the conditions on the left side, and returns a value that is the result of evaluating the right side of the first condition that is true.

Thus, the model will return to the initial (zero) state when **rst** is true, increment the counter when **i** is true and do not switch state at all if none of the previous conditions was true.

We have referenced the **rst** and **i** identifiers, which are from *input* parameters of the module – declared on the module header, and must be assigned actual values when instantiating the module.

To understand how those work, we must first know how NuSMV handles modules during the loading of the state machine model: NuSMV *merges* all of the module instantiations into a single flat one, as seen on the following snippet:

<pre> — Source MODULE main() VAR b: boolean; i: 3..4; instance1: mymodule(b); instance2: mymodule(i); MODULE mymodule(in) VAR t: boolean; ASSIGN next(t) := in; </pre>	<pre> — Flat MODULE main() VAR b: boolean; i: 3..4; instance1_t : boolean; instance2_t : boolean; ASSIGN next(instance1_t) := b; next(instance2_t) := i; </pre>
--	---

Thus, formal module arguments get textually replaced with the actual argument, like if modules were a convoluted form of C-like macro expansion. While this means we have polymorphic modules for free, whose arguments are dynamically typed, care must be taken not to do type-specific operations on the formal parameters if we can not guarantee callers will always pass in the correct type (in the above snippet, a type error will appear on `next(instance2_t)` because `i`, an integer, cannot be casted implicitly into a boolean.

NuSMV also offers one last piece of syntactic sugar: the `DEFINE` statement. This acts much like an actual macro, except it is scoped to the module where it was defined. Defines are sometimes used as a output parameters, as parent modules can reference to such macros via the usual `module.name` notation.

1.3 Jutge.org

From the Jutge.org homepage [11]:

“Jutge.org is an educational online programming judge where students can try to solve more than 600 graded problems using 20 different programming languages. The verdict of their solutions is computed by the Judge using exhaustive data sets run with time, memory and security restrictions. Moreover, instructors can use it to create their own courses, attaching documents, creating lists of problems, assignments, contests and exams, as well as roasting their students and tutors.”

The software was created by Jordi Petit, Salvador Roura, and other researchers from the *Technical University of Catalonia* (UPC), initially as a way to help the evaluation tasks for the UPC Programming Contest, where around 200 contestants enrolled.

However, the idea of using the Jutge.org codebase to change the way the existing UPC introductory programming course was handled soon appeared.

Before Jutge.org, students were evaluated by their ability to write algorithms in a pseudo-code partially defined by the course staff. Since it was thought that a programming course should be graded by letting students actually program, in front of a computer, the evaluation model was changed with more frequent tests and an automated evaluation system behind the scenes allowing students to get feedback from their code as soon as possible.

Up to 250 simple C++ introductory problems were created for the use with the Judge.org system [12].

Later on, Judge.org was made available to a greater audience by publishing it on the www.judge.org site. Thus, the current system was born, used widely at more than six programming courses from UPC [11]. It is also freely available to use by both students from other colleges – with a public set of problems from some of the UPC courses –, as well as interested instructors willing to allow their students use the Judge.org system.

Use of the web application requires prior registration – name, email address, and birth year. This way, users get personalized views after logon, with the system remembering problems they have already successfully solved, their favorite compiler, enrolled courses, and so on. However, for quick checks there is a demo account everyone can use.

Verdict	Description
Accepted (<i>AC</i>)	Program is correct and output matches that of known-good solution.
Wrong Answer (<i>WA</i>)	The program output does not match the known-good solution.
Execution Error (<i>EE</i>)	The program crashed or was too slow.
Compilation Error (<i>CE</i>)	The compiler failed to build the program.
Internal Error (<i>IE</i>)	Verification failed in some unspecified way.

Table 1.4: The important Judge.org verdicts

The Judge.org system is written mostly using the PHP³ language. However, some maintenance scripts are written in Python⁴. The system uses public, open source compilers to support each of the many available languages (for example, it uses the GNU Compiler Collection⁵ to support C and C++).

The main web server uses the Apache⁶ HTTP Server running under Ubuntu⁷. Corrections (which might involve running completely untrusted

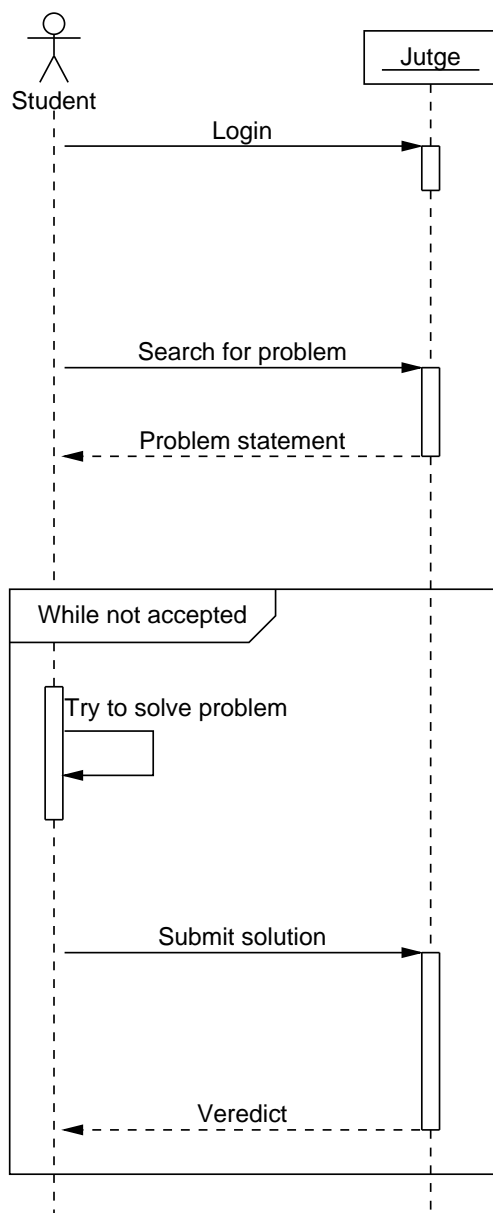
³<http://www.php.net>

⁴<http://www.python.org/>

⁵<http://gcc.gnu.org/>

⁶<http://www.apache.org/>

⁷<http://www.ubuntu.com/>



Judge.org
The Virtual Learning Environment for

Login

Email:

Password:

Log in

Proceed to register?
Try a demo account?
Forgot your password?

Homepage

Welcome to *Judge.org*

Judge.org is an e-learning environment with more than 600 graded problems, their solutions in C, C++, Java, JavaScript, Python, Perl, PHP, Ruby, and Haskell, memory and search algorithms, and courses, attached to them, as well as forums, blogs, and a chat.

Problems

By course Search All

Show all **Hide all**

Algorithms

- Backtracking - Subsets
 - P12828 Zeros and ones
 - P45965 Zeros and ones
 - P18957 Subsets (1)

Problem P29017_en: Printing stacks

Statement Submit Submissions Annotation Forum

Problem files

Statement

Printing stacks

Write a procedure

```
void print_from_top_to_bottom (stack<int>& S);
```

that prints a line (end line included) with the elements of S separated by spaces.

Also, write a procedure

```
void print_from_bottom_to_top (stack<int>& S);
```

Submit your solution.

*Source file:

*Compiler: G++

Comment:

Post an Email: No

Submit

Verdict: Accepted
Compiler: G++
Time: 2011-04-08 10:10:10

Figure 1.7: A typical Judge.org student interaction

executable code coming from the student) are made in virtual machines, of which there might be several (up to four) per physical machine. Each virtual machine is rebooted after a few corrections are made, so that potentially unwanted state information – like compiler generated temporary files – is deleted. Submissions are evenly distributed between each of the virtual machines.

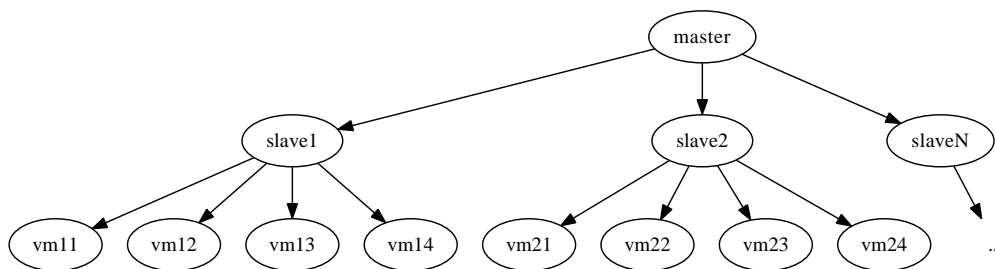


Figure 1.8: The Jutge.org architecture.

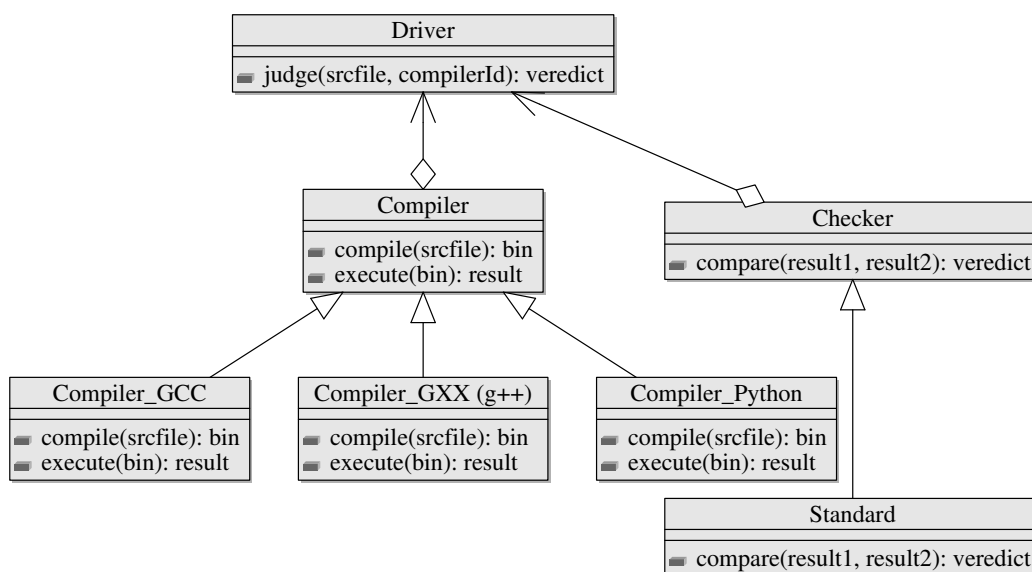


Figure 1.9: Jutge.org’s *Standard* driver high-level class diagram

So far, Jutge.org was mostly used for imperative languages, like C, C++, Java, Python, or Perl. The support for each of these languages is abstracted under what is called a *Compiler* – a Python script that wraps the necessary scripts to launch the compiler and execute the compiled program. After the program is executed, its results are compared using a special *Checker* – that

can ensure the execution results of the student submitted program match that of the teacher made program. The entire process is wrapped under what is called a *Driver*, as seen on figure 1.9.

Compilers can thus reuse the testing functionality by providing the generic layer with the execution results (its standard output). At the same time, we can reuse most of the existing web user interface while discarding the code that assumes programs are executable and its textual output is to be compared – our circuit models are not.

Thus, it makes sense for Judge.org to be used in this project.

1.4 Previous work

While there has been work done in the context of this project, we do not know of any public similar web environment. However, we will briefly look at similar Verilog formal verification tools.

Automatic formal verification of Verilog is not a new concept. In fact, there are already many available commercial solutions that implement formal equivalence checking, as listed on Xilinx’s ⁸ commercial support site [13] for interested customers:

<i>Synopsys Formality</i>	http://www.synopsys.com
<i>Cadence Conformal</i>	http://www.cadence.com
<i>Prover eCheck</i>	http://www.prover.com

Obviously, a commercial solution was not of interest since we would not be able to adapt it to our needs. All of the solutions are more oriented to verification as part of the development process during the different stages of a typical Field-Programmable Gate Array (FPGA) implementation flow, while on the other hand our solution must concentrate on a single implementation stage and should instead verify equivalence between different HDL implementations made by different authors.

While we did not have access to any of these tools, it is expected that such tools can do equivalence checking of multi-million gate designs *in a fraction of the time* needed for simulation[13] – from days or weeks to mere hours –

⁸Xilinx is one of the biggest FPGA manufacturers, with a large stake on the electronic circuit design tools market

while also providing 100% testing coverage.

Confluence

Among the fewer free tools, specially interesting is *InFormal*, a open source tool developed by *Confluent* (no longer in existence) as part of their *Confluence* HDL language.

Confluence was a new HDL that would fix many of the limitations in Verilog and VHDL. However, it was not fully intended to replace Verilog, and in fact, one of the features Confluence had was its ability to emit Verilog code from a hardware description on its own language.

This was done by the `fnf` tool[14], which would work on the output of the Confluence synthesizer (thus, it would work on a list of nets) and from that output a structural Verilog equivalent – in a very low level fashion, as all of the high-level information that could be used to build a easier-to-read behavioral Verilog equivalent was already lost in the synthesis process.

In order to be able to do different kinds of formal verification, `fnf` was also able to write a NuSMV model file the same way it could output a Verilog source file: from a netlist.

Considering that and the fact that a conversion process from the Icarus Verilog synthesized output and the format used by `fnf` exists, this software also presented a way to transform Verilog source files into NuSMV models, which is what was packaged under the *InFormal* name as a way to aid formal verification of existing designs.

However,

- The software is no longer being maintained and is hard to find, and it worked only with older versions of NuSMV and Icarus.
- Since it works on a netlist level, the NuSMV models it generated would be hard to read as many of the high-level information that cannot be conveyed into a netlist is lost.

For example, all Verilog instantiations would be combined into a single large module in the destination NuSMV file, with all of the state variables merged in between with random names. Even the simplest of the circuits would generate into a enormous source file virtually impossible to understand by a human.

As verification of what the converter does is clearly one of our goals, our project shall integrate more with Icarus itself so that the system is able to access more information about the original Verilog source file, thus producing more readable NuSMV models that can be understood by a human, and even reused into other NuSMV models.

Chapter 2

Design

2.1 Goals

The main objective of the project, as stated on the abstract, will be to implement a software system for the automatic verification of circuits written in Verilog, comparing their behavior to that of a known-good circuit using model checking.

This requires:

1. A system where teachers can store problem definitions and known-good circuits and students can download problem statements and upload their own answers in Verilog.
2. Automatic conversion of Verilog source files into equivalent state machine models.
3. Verification of whether the known-good model and the student's model behave equally.

We have decided that:

- We will use the existing Jutge.org environment as the website where teachers will upload problems and students will get its statements from, as well as the system where students will submit their solutions.

As per section 1.3 on page 22, Jutge.org has well-tested support for a very similar use case (exchanging Verilog circuits for computer programs in imperative languages) and is modular enough for the requirements of this project.

- NuSMV (see section 1.2.3 on page 19) will be used to perform the symbolic checking of the models.
- For the conversion of Verilog source files into models, we will develop our own Verilog to NuSMV solution, using Icarus Verilog (section 1.1.2 on page 12) as frontend. Icarus will do the parsing and initial synthesis of the design, greatly simplifying the work to be done.

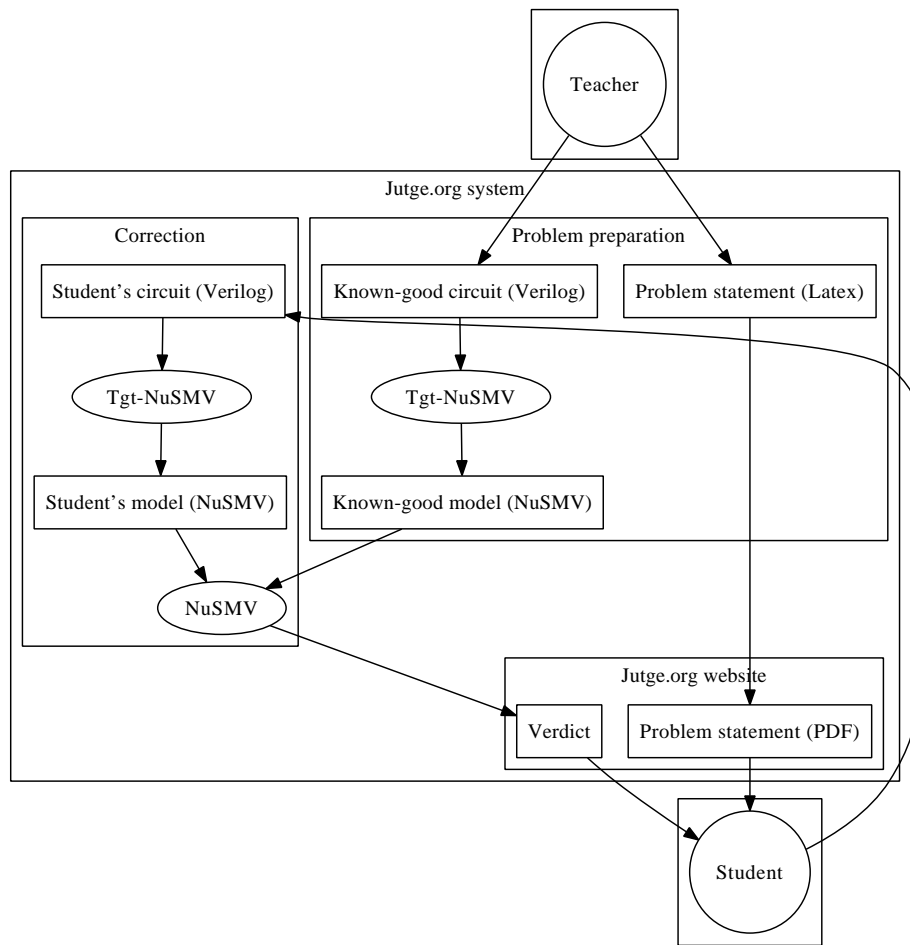


Figure 2.1: A brief look at the entire process

Following is thus the resolved list of project objectives:

1. Implement a Icarus Verilog 0.8 synthesis target that generates a NuSMV model (*tgt-nusmv* item on figure 2.1).

- The most common features of the synthesizable subset from the Verilog language should be supported. We believe such a subset will allow for the most interesting problems to be solved.
 - The generated models should be readable to improve verifiability by a human. Ideally, it might keep a bit of the structure from the original Verilog source file.
 - The converter must be reasonably fast so that it can be used interactively, in a typically configured server machine.
 - The generated models should also be able to be validated automatically in a reasonable time. The converter should avoid statements or NuSMV features that cause delays in the validation.
2. Implement a Jutge.org driver to allow for the upload and verification of Verilog circuits using the models generated by the previous tool (the *correction* system as seen on figure 2.1), as well as the necessary problem preparation routines required for the correct function of the driver (*problem preparation* on figure 2.1).
 - The system should be fully integrated under the existing Jutge.org infrastructure. All involved components should be able to work on its servers, and the end user interface should be similarly usable.
 - Human intervention in the entire student submission process should be kept to a minimum.
 - Allow for integration with other languages in the future, like VHDL.
 - The system will be available online so it has to be safe and reliable.
 3. Create a small set of sample problems to test the system.
 - A *problem* should consist of a brief statement and a known-good circuit to formally compare it to student submitted circuits.
 4. Write the required project documentation and the project report.

The project work is clearly structured in two separate parts: the conversion process, and the integration with the existing Jutge.org infrastructure. We will keep this distinction for the rest of this report.

2.2 Verilog to NuSMV conversion

The most important part of this project is the software converting a Verilog circuit into a NuSMV model. That is, convert something like this:

```

module counter(clk , rst , i , r);
    input clk , rst;
    input i;
    output reg [1:0] r;

    always @(posedge clk)
        if (rst)
            r <= 0;
        else if (i)
            r <= r + 1;
endmodule

```

Listing 2.1: A 2 bit counter with reset

Into something like this, which is very similar to the counter example in the NuSMV tutorial[15], albeit slightly optimized and with a reset signal:

```

MODULE counter(i , rst)
VAR
    value : word[2];
ASSIGN
    next(value) := case
        rst : 0;
        i : value + 1;
        TRUE : value; — Default case
    esac;

```

Listing 2.2: Manually generated model for circuit in listing 2.1

2.2.1 Analysis

During the first part of the analysis, we will evaluate the work Icarus Verilog is doing before invoking our target, as described at section 1.1.2 on page 12, assuming its input source file is the one on listing 2.1.

After the design elaboration stage, without any synthesis done at all (just parsing and initial elaboration), the generated graph (figure 2.2 on page 33) looks very much like an Abstract Syntax Tree (AST) – a tree representing the

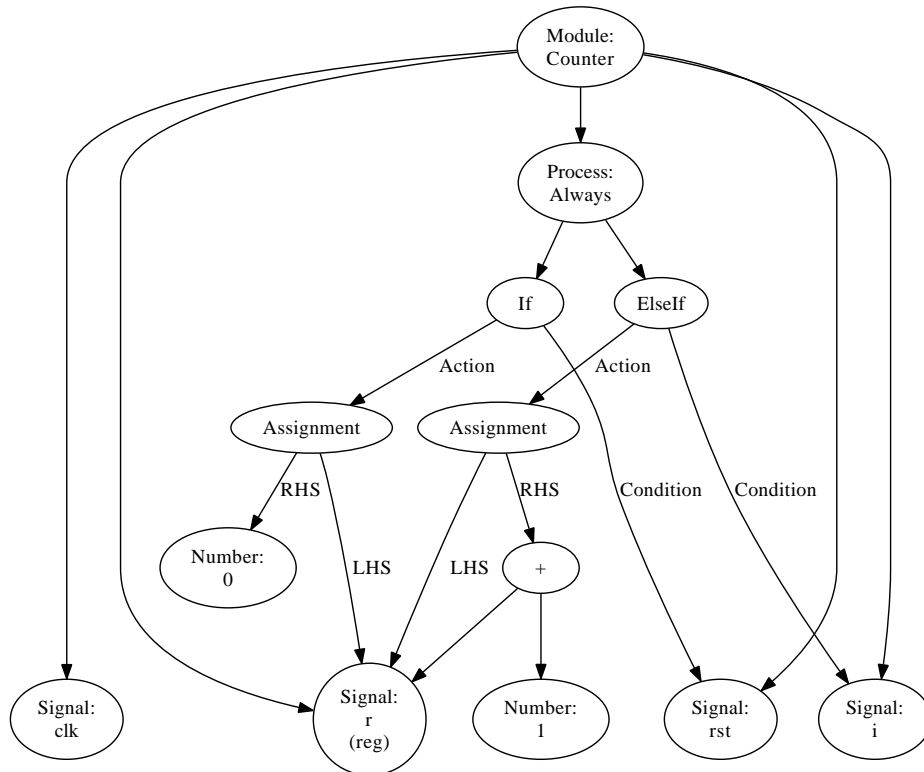


Figure 2.2: Counter module elaborated design (without synthesis)

syntactic structure of the source code – and far from a real circuit. Working from this stage would require to handle all kinds of Verilog processes, primitives and operators by ourselves, which would be a huge amount of work.

Therefore, we will also configure Icarus Verilog do synthesis before invoking our target.

After enabling the *synth2*, *synth* and *syn-rules* functors, the graph that comes out from the synthesis process (figure 2.3 on page 34) looks quite different – resembling an actual circuit.

It is actually an *hypergraph*, where edges can connect an unlimited number of nodes – the same way a single cable can connect multiple electronic components – but represented as a multigraph, with each hyperedge being represented as a new node (a *nexus*) where all of the vertices the hyperedge connected are instead connected to the nexus.

- The ellipse shaped nodes are **signals** – the name Icarus Verilog gives

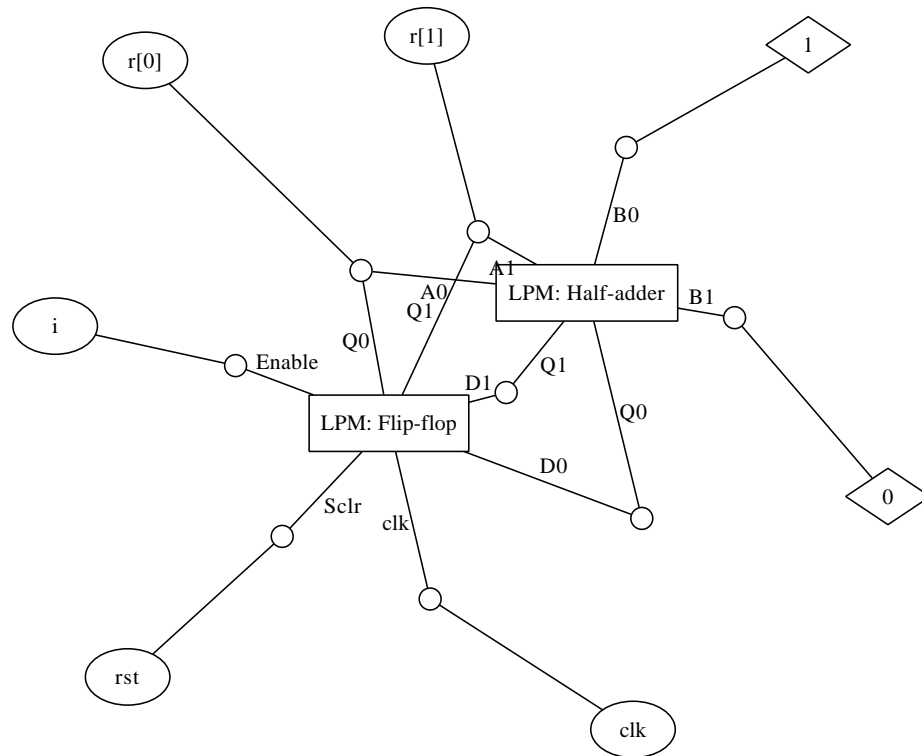


Figure 2.3: Counter module elaborated design (with synthesis)

for named nets on a module. All of the input and output ports in the original module appear as signals on the design graph.

Therefore, it is possible to accurately know what each input and output port is connected to.

- The box nodes are parametrized modules (from a *Library of Parametrized Modules* or *LPM*, thus usually known as **LPMs**). This is a quite important concept. The library contains already synthesized modules that do from the simplest of the tasks (like a multiplexer, or a decoder) to encapsulate some of the more difficult components (multipliers, divisors...). All sequential circuits synthesized by Icarus are based on two key LPMs: the *D flip-flop* and the *D Latch*.

The modules are said to be *parametrized* in that each of them has a set of configurable parameters – for example, a half-adder LPM will have a *width* configurable parameter that specifies the size of its operands. As many LPMs as required can be instantiated for any given design.

LPMs are expected to be synthesized by the code generation stage because it is assumed an usual FPGA will have some builtin components that will match with those of the Library. In which case, it is clearly desirable to use those versus a fully synthesized version that might expand to hundreds of gates. As NuSMV has lots of operands that nicely match some of the LPMs, this is also useful to us.

See table 2.1 on page 42 for the full list of LPMs used in Icarus Verilog 0.8.

- The diamond shaped nodes are **constants**. Since this a digital circuit, there is only two of them: 1 and 0. If a port from a component is always set to 1, or to 0, it will be connected to the adequate constant node.
- Not appearing in the example graph – **logic gates**. Those are like LPMs except they are simpler and only have one output port (but can have 1, 2, or more input ports, as required).

As they store no information, logic gates can be represented with truth tables.

- The small circles are the **nexus**. Those are circuit interconnection points. Each of the edges connected a to a nexus convey that the correspondent electronic component pins are all of them interconnected via a cable.

These vertices are the way Icarus Verilog represents hyperedges on the elaborated design graph.

To continue with the example, we can find two LPMs on figure 2.3, which for clarity we have created a gate-level diagram of it on figure 2.4.

1. A half-adder, LPM.ADD, with three two-bit ports:

Name	Description	Connected to
$A1, A0$	First operand	To the current value of the counter (r).
$B1, B0$	Second operand	To digital 0, 1 respectively.
$Q1, Q0$	Result	To the input (D) of the flip-flop.

It is obvious that the half-adder is the component that increments the counter value – adding 01 to it.

2. A D flip-flop, LPM_FF, with the following ports:

Name	Description	Connected to
<i>clk</i>	Clock	To the <code>clk</code> signal.
<i>Enable</i>	Chip Enable	To the <i>i</i> signal. Therefore, the flip-flop only gets set as long as <i>i</i> is raised.
<i>Sclr</i>	Synchronous clear: for every clock cycle this pin is active, the flip-flop resets to zero.	Connected to the <code>rst</code> input port of the main module.
<i>D1, D0</i>	Flip-flop input	To the output of the half-adder module.
<i>Q1, Q0</i>	Flip-flop output	To one of the inputs from the half-adder, as well as the <code>r</code> signal.

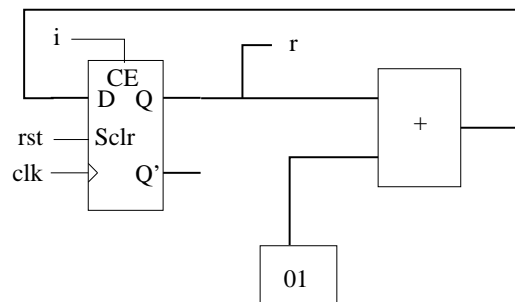


Figure 2.4: Gate-level diagram of the synthesized counter module

The amount of features a target would have to handle at this level is much smaller than with previous stages: the size of the library of parametrized modules size is reasonably small, and there are not many other kinds of nodes to be handled.

In fact, one important aspect stands up at this point from the elaborated design: in a sequential circuit, *all of the circuit state information will be stored in either Flip-flop or Latch LPMs*, easing the task of extracting this information in order to build the state machine model.

Thus, this is the point at the pipeline where the converter implemented in this will start its work.

2.2.2 Data model

In order to improve readability, we would like the modules in the generated models to map 1:1 to Verilog modules (including input and output ports). Therefore, we need a data model to store the module definition and port structure, as well as store instances of both other modules, *LPMs* and logic gates.

This model will be built from information given by Icarus, and will be used to perform fast lookups of required data – owners of signals, all instances of a module, etc – which will be required during the conversion process.

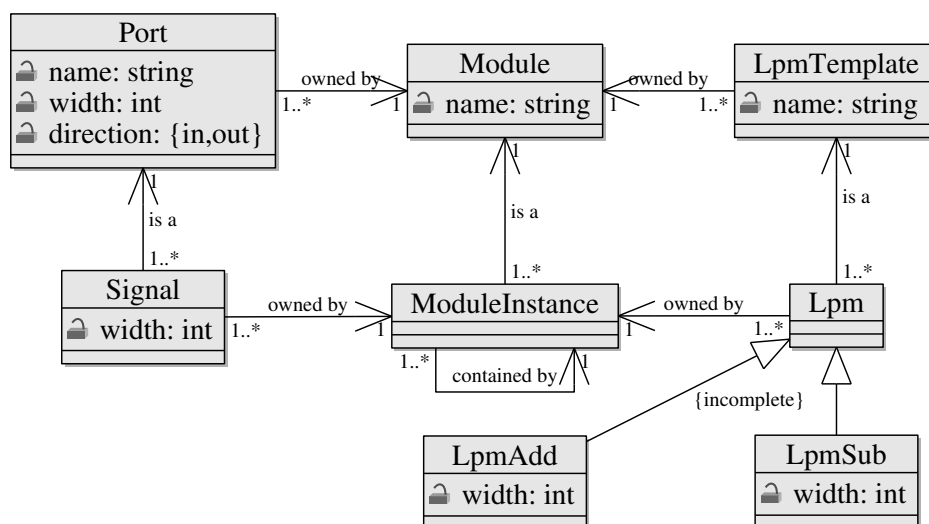


Figure 2.5: High-level class diagram of the data model

As seen on figure 2.5, we will keep the **Module** as the most important component of the model. However, we will keep a distinction between a definition **Module** and its potentially multiple instances **ModuleInstances**.

A module definition will have links to the definitions of its ports (**Port**), its referenced LPM definitions (**LpmTemplate**), and to the module definitions referenced as part of the instance declarations inside it. Analogously, a module instance (**ModuleInstance**) will link to the actual port instances (**Signals**, as seen on the elaborated design graph), LPM instances (**Lpm**), and each of the instantiated submodules.

Lpm is an abstract class. Each of the different LPMs from the library will have its own specialization in order to handle the differences between port configurations among the different LPMs.

The actual logical connections between *LPMs*, instances, etc. will be known by keeping references from our data model to the Icarus elaborated design graph.

2.2.3 Algorithm

With all in mind, we can start building the NuSMV model.

For each Verilog module m in the input source file:

1. Create a new NuSMV module m' , whose identifier will be the same as the name of m .
2. Enumerate the input ports of m – those will be the input variables of the NuSMV module m' ¹.
3. Enumerate all the child module instances of m and owned *LPMs* and instantiate them as state variables in the NuSMV module m' .

When writing the definition of module m' , we have to provide the actual input parameters for each module instantiated by m . In the synthesized Verilog module m , connections were made for each of the ports (both input and output) that appear on the elaborated design graph.

Therefore, to generate expressions for the actual parameters the algorithm will need to traverse the elaborated design graph, starting from the input port edge until we find a device that is driving this signal and that we can express as a NuSMV expression. For example, a constant value (a literal), or a input signal of the module m – for which we can generate the expression “`signalname`” since we can safely assume NuSMV will know how to evaluate it as the signal name will be a valid identifiers in the scope of the module m' . We just introduced them as input parameters of m' in the previous step.

A more detailed explanation of the above process is as follows.

Assuming that:

- m is the Verilog module instance from which we are currently trying to create its equivalent NuSMV module m' .

¹Clock signals are handled separately – see section 5.3.3 on page 80

- a is the edge in the design graph representing the input signal corresponding to the parameter we want to generate an expression for.

The expression generation algorithm we will use is:

- Let A be the set of all the other edges a 's nexus is directly connected to (not a itself). Abstractly, A is now the set of all edges that used to represent a single hyperedge from the elaborated design graph.
- If any b in A is:
 - Connected to a *const*/literal.
 - An input signal of m .
 - An output signal from any of the direct child instances of m (including *LPMs*).

Then we are done. We know that the expression is either a literal or a trivial one in the form of `childmoduleinstance.port` or `inputsignalname`, both being valid in the scope of m' .

- If any b in A is connected to the output of a logic gate l , then the result is an expression of the form $x_1 \text{ op } x_2 \text{ op } \dots \text{ op } x_n$ where
 - x_i is the result of recursively applying this algorithm but with a being the node correspondent to input i of l .
 - op is the NuSMV operator *equivalent* to l (i.e. if l is an *and* gate, the equivalent operator would be `&`).
 - Otherwise, this port was not connected to any component driving it. This is most probably a mistake in the input Verilog circuit, so the system should warn the user.
- Enumerate all the output ports and add them as NuSMV defines in the `DEFINE` section of m' .

Use the same expression generator algorithm as described above to construct the correct expressions for each of the output ports, but using this output port as starting edge a .

The elaborated design graph of the Verilog module that is the source of the NuSMV module generated in listing 2.3 can be seen in figure 2.3.

As an example, we will describe the steps made by the algorithm for the conversion of this module:

```

MODULE counter(i, rst) — The input ports
VAR
  — Child module instances and LPMs
  ff: lpm_ff(add.Q, i, rst);
  add: lpm_add(ff.Q, 0b2_01);
DEFINE
  — Output ports
  r := ff.Q;

```

Listing 2.3: Sample of module generated using the above rules

1. The algorithm creates the NuSMV module `counter` from the Verilog module `counter`, the input parameters being the input ports of the Verilog module.

```
| MODULE counter(i, rst)
```

2. It also creates the `ff` and `add` instantiations, as they are instanced by the source Verilog module.
3. The `ff` instance of `lpm_ff` has three input ports:
 - *D*²: If we follow the D edge on graph in figure 2.3, we see that it is connected directly to the Q output of the half-adder module. This is the output port of a direct child of `counter`, and thus, according to the algorithm, we generate the expression `add.Q` and use it as actual parameter for the `ff` instance.
 - *Enable*: connected to the `i` signal, which is an input port of `counter`. Therefore, the expression is `i`.
 - *rst*: connected to the `rst` input port. The expression is `rst`.

After all the actual parameters have been determined, the state variable definition is written into the NuSMV file:

```
|         ff: lpm_ff(add.Q, i, rst);
```

²In the design, both D and Q are a two bit buses. Since D0 and D1 are symmetrical, the same reasoning works for any of them. For simplicity we will consider them both as single-digit only.

4. The `add` instance of `lpm_add` has two input ports:
- *A*: by the elaborated design graph, we can see that there are two other ports with connectivity to this signal: `Q` of `ff`, and `A` of `add`. The latter is discarded as it is not a literal, input of `counter`, output of a child of `counter`, or a logic gate – it is an input of a child module, `add`. The former is accepted as it is an output port of the child module `ff`.
Thus, the algorithm chooses `ff.Q` as expression.
 - *B*: connected to a *const*. We convert the literal into the appropriate NuSMV literal expression: `0b2_01`.

The resulting state variable is defined as follows:

```
|      add: lpm_add( ff.Q, 0b2_01 );
```

5. In the `DEFINE` section, we list the output ports of the `counter` module. There is only one output port, `r`, which, per the elaborated design graph, is connected to both `add.A` and `ff.Q`. As in the previous step, the expression generation algorithm decides to use `ff.Q`.

```
|      r := ff.Q;
```

Note thus that the modules generated by this algorithm will follow a set of rules:

- Each of the Verilog module input ports will be mapped to a NuSMV module parameter.
- Each Verilog module output port will be mapped to a NuSMV module definition/alias. Thus, parent modules can refer to those by using the very readable `instancename.port` syntax that was referenced in the above algorithm.

Assuming the `lpm_ff` and `lpm_add` modules are already defined somewhere else (which they can entirely be done manually, considering the entire library consists of a handful of such modules – see table 2.1), the model on listing 2.3 is already a perfectly working equivalent of that we shown on listing 2.2 on page 32.

Ergo, the conversion process is done.

LPM	Description
ADD	Full/Half-Adder
CMP_EQ,GE,GT,NE	Comparators
DECODE	Decoder
DEMUX	Demultiplexer
DIVIDE	Divider
FF	Flip-flop
LATCH	Latch
MOD	Modulus
MULT	Multiplier
MUX	Multiplexer
SHIFTL,SHIFTR	Shifters
SUB	Subtractor
RAM	Memory

Table 2.1: List of LPMs

2.3 Jutge.org driver

The other half of this project is to build a Jutge.org *driver* for Verilog circuit verification, as stated on the goals. By building a driver, we will be able to leverage the work done by the Jutge.org authors and obtain a complete web frontend that satisfies our requisites – and we can concentrate on the actual verification part instead of creating Web frontend code managing problem lists, queuing, submitting solutions, etc.

In fact, the input to the driver will be the student’s raw submitted code, and the output will be a *verdict* that the Jutge.org frontend will properly format and display to the student.

Thus, the external user interaction with the system will be exactly that of the original Jutge.org (see figure 1.7 on page 24), albeit with the necessary modifications to problem presentation (as for obvious reasons the statements from circuit problems will be formatted differently) as well as results viewing (since we have both different potential verdicts along with different diagnostics for each verdict).

The driver will, from two Verilog source files, one coming from the student, and the other a known-good circuit coming from the teacher, run the Verilog-to-NuSMV converter on both source files to get two NuSMV models. Then, the two NuSMV models will be combined so that a unique state machine with a single “outputs match” output is created, and the NuSMV

application will be run on this model to perform the verification using CTL logic. Depending on the result of each of these stages, a verdict is given back to the student.

See figure 2.1 on page 30 for a visual representation of the internal workflow.

We will call this driver **cv**, which stands for *circuit verifier*. The class structure of it will be mostly identical to that of the existing Judge.org driver, *std* – whose class diagram can be seen on figure 1.9 on page 25.

Unlike the *std* driver, which has to compare the outputs after running both the student and the known-good programs, our driver will always use NuSMV to check if both models match and has no need to perform any additional checking on them. Therefore, all of the different *Checker* classes have been removed from the structure.

On the other hand, we might want to add support for future languages (and thus synthesizers) in the future, so we decided to add an abstraction similar to the *Compiler* class from the original driver: the *Synthesizer* class, with one single realization – the one for Icarus Verilog 0.8, which we will call *IVL08*. The class diagram detailing all the relationships can be seen on figure 2.6.

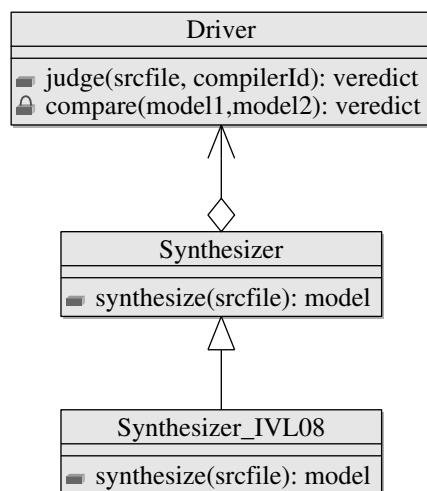


Figure 2.6: *cv* driver architecture overview

The process for the main *judge* function shall be (assuming the selected compiler is *IVL08*):

1. Launch Icarus Verilog with `tgt-nusmv` as target for both circuit files. Use result error codes to check if there was a synthesis error in the student file. If there was, verdict is *Compilation Error*: output the compilation error messages.

We have now built the equivalent NuSMV model for both circuits.

2. Ensure that both the the student's and the known-good circuit externals interface is consistent: that is, both have the same inputs, outputs, module names, etc. If they are not consistent, verdict is also *Compilation Error*. An error message should tell the student what the correct interface is.
3. Combine both generated circuit models into a single model so that
 - All inputs by the same name are connected to the same source in both models (both the known-good and the student circuit will received the same stimulus during verification).
 - A comparator on the output will output whether the output from both circuits matches or not, as in the following figure:

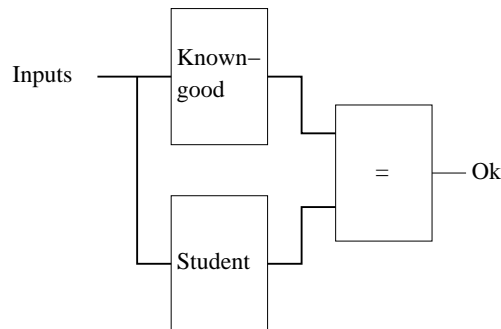


Figure 2.7: Modeled circuit

The definition of what exactly is considered a match is left open depending on the problem definition, as described on the next section.

4. Use NuSMV to check if, in the combined circuit model built above, there is a sequence of input signals that will cause *Ok* to be zero.

If such input sequence exists, verdict is *Wrong Answer* – emit the full sequence that causes the student circuit to produce an output that is not equal to that of the known-good.

5. Otherwise, verdict is *All Correct*.

Hence, our project has four potential causes for emitting a negative verdict – notwithstanding internal errors – which we have mapped into three existing Jutge.org verdicts ³ to minimize the amount of work to do on the Web frontend. Also, as per the goals, our project will give different information back to the student for certain verdicts as listed on table 2.2.

Verdict	Description	Extra information
Accepted	Source file was synthesized <i>and</i> the synthesized circuit matched the required specifications.	–
Compilation Error	Source file was not valid Verilog. <i>or</i> The interface of the Verilog module did not match that of the known-good one.	List of error messages from the synthesizer. List of differences from the known-good interface.
Wrong Answer	Source file was synthesized, but resulting circuit did not match problem specifications.	Example of input signals sequence causing the circuit to give an output that does not match the output the known-good circuit would give.

Table 2.2: All possible verdicts from the circuit verifier driver

2.3.1 Circuit equivalence checking

We have seen that as part of the evaluation process, both generated NuSMV models have to be combined into a single one that can be used to check whether the two original models describe circuits that behave identically when given the same set of inputs.

This combination is one important step of the process, as the construction of the circuit and verifier module as envisioned on figure 2.7 on page 44 will determine how we actually define whether a circuit is *equivalent* to the known-good one.

³A list of all Jutge.org possible verdicts can be seen at table 1.4 on page 23.

Within this project, we will use a trivial comparator as verifier that will ensure that all of the outputs with the same name are always equal to those from the known-good circuit. This comparator module model will have to be generated from the known-good Verilog source file (in order to get port names, directions, and widths). An example is shown in listing 2.4.

```

MODULE teacher_half_adder(a,b)
  — The known-good circuit model

MODULE student_half_adder(a,b)
  — The student submitted one

MODULE main() — The comparator module
VAR
  a: word[2]; — Input port A
  b: word[2]; — Input port B
  t: teacher_half_adder(a,b);
  s: student_half_adder(a,b);
DEFINE
  ok := (t.r = s.r);
  — ok is only true if r the output from both
  — teacher and student modules are equal.

```

Listing 2.4: Example manually generated comparator NuSMV module

Missing from the previous example is the actual specification, to be checked using NuSMV. We would obviously like to ensure that `ok` is always true for every possible state. Such a specification is good enough for purely combinational circuits, and is very easy to develop using CTL:

```
| SPEC (AG ok);
```

However, such a specification would not be acceptable for sequential circuits because it can not be assumed that outputs will be equal on the initial states, before a reset has been signaled. Nonetheless, we can trivially reduce the specification to take this into account by forcing that the `ok` signal has to be true for every possible combination of inputs *only after* at least one cycle following the falling edge of the designated reset signal:

```
| SPEC (AG (rst_negedge → AG ok));
```

The `rst_negedge` flag can be modeled by a simple module with one bit of state that stores the status of `rst` at the previous state. Such module can be implemented manually and all the driver will have to do is to add its definition to the generated source file as required.

The specifications described on this section mean that the system will guarantee, from a *correct* student circuit, that all of its outputs will exactly be those of the known-good one for each clock cycle after reset. Less restricted specifications will be discussed in section 5.3.4 on page 81.

Chapter 3

Implementation

From the two larger parts of the project described in the previous chapter, the Verilog to NuSMV converter was started first because it was a required dependency which was expected to take most of the available time.

When the converter was reasonably finished and well-tested, work was started on the Judge.org driver.

The development process of both components will be documented in this chapter.

3.1 The `tgt-nusmv` converter

The language of choice for this part was *C++*. A language compatible with *C* calling conventions was mandated due to the target module having to be loaded by the Icarus Verilog (and also having to use Icarus *C* Target API). Having object oriented support was a plus.

Icarus Verilog itself also being coded in *C++* and using the *Standard Template Library (STL)*, the build and runtime dependencies of the converter are a subset of those from Icarus itself. No external dependencies were deemed required apart from the usual *STL* containers.

Any Icarus target, being a shared object that will be loaded at runtime, has to export two required publicly visible function symbols: `target_design` – the entry point, where the target module receives the handle for a start node in the elaborated design graph – and `target_query`, which Icarus can call to query the capabilities of the target. Currently, it is only used to ask for the target plugin version number.

Backends also need to define the list of functors that will be applied to

the elaborated design before actually passing control to the `target_design` function. As we argued on the design, it is best for our converter to start from a synthesized design. Therefore, we will use the same functors as the other Icarus synthesis backends. The entire list along with the shared object path containing the target module have to be specified in a config file in

```
/usr/local/lib/ivl-0.8/backendname.conf.
```

```
functor:synth2  
functor:synth  
functor:syn-rules  
functor:cprop  
functor:nodangle  
-t:dll  
flag:DLL=nusmv.tgt
```

Listing 3.1: nusmv.conf file

When our generator is finally invoked by Icarus, we can divide the work it does into three large steps:

1. Walk the elaborated design graph and build from it a database whose model was detailed on section 2.2.2.
2. *Process* the design.
3. Write the output NuSMV file.

The implementation is actually pull-driven instead of push. This means values, expressions and lists are calculated only as soon as the the last step during conversion needs them – that is, when the NuSMV file is being written, in the same order they appear in the output file. The only exception are checks required for the proper emission of certain warnings like the one for when multiple clock signals are involved in the circuit. Those constraints are forcefully checked during the processing stage.

3.1.1 Data layer

During the design, it was decided that a small data set from information obtained by scanning the elaborated design graph would be built, so that some operations that would otherwise need to traverse the graph again could be performed faster by having the results cached in such data set. The initial model of this data set was described in figure 2.5 on page 37.

This database does not need persistence of any kind. Therefore, the abstract model classes were implemented as C++ classes. Relationships between instances of items in the database were implemented using *STL* containers, guided by the performance requirements of the operations the database was required for. Such requirements and the resulting implementation decisions are detailed in the table 3.1.

It was also determined that several times we would need to map an opaque Icarus handle into a signal, LPM or module instance from our database. For this reason, a singleton class named `DB` that would store *handle* \rightarrow *instance* dictionaries for each class of object was envisioned.

For simplicity and to ease the memory management problems that tend to plague C programs, we decided that the `DB` class would be the *owner* of all the object allocations made as part of the data layer described in this section – deallocation of an object happening only when its own `DB` dictionary entry is removed.

The construction of the actual database happens by walking the elaborated design graph, starting from the root module instance, noticing all the handles involved – each time a new signal, LPM, or module instance is found, the `DB` object is queried, a new entry being created for it if it did not exist previously. Thus, after no new paths are left to be explored in the graph, the database contains all entries for all reachable objects in the circuit.

3.1.2 Main processing

Filtering

During the process stage, our implementation currently applies a few *filters* before starting to write the NuSMV file. Filters were thought to be analogous to functors from Icarus Verilog, except for being run inside our backend which means they can use all of the database information described on the previous section.

Only two such filters were implemented: one that ensures all flip-flops share a single clock source, and one that generates a prototype of the root module in a `.txt` file. Both are specializations of a generic `Filter` class (see figure 3.1), and both are optionally enabled with command line arguments that will be seen on section 4.1.

Operation	Example functionality requiring it Data structure used
Get a list with all input ports from a module	When building a NuSMV module definition, to lookup the signals whose expressions need to be generated. Add a vector of strings with all input port names to the Module class.
Get a list with all output ports from a module	When building a NuSMV module definition list of DEFINES . Add a vector of strings with all output port names to the Module class.
Get a sample instance of a given module	In order to build a definition for such module, a sample instance is needed. Keep a list of the instances in the Module class, then grab the first one when requested.
Find a instance signal by name	When building a NuSMV module definition, to lookup the signals whose expressions need to be generated after the names have been retrieved from the module definition. The ModuleInstance \leftrightarrow Signal relation was implemented as a map with the signal names as keys.
Test if a given signal is owned by a module instance	When generation an expression, to check if a certain signal is owned by a module that is a child of the current one. The ModuleInstance \leftrightarrow Signal relation implementation was extended with an owner field in the Signal class.
Get a list of the childs of a instance	When generation an expression, to check if a certain signal is owned by a module that is a child of the current one. The ModuleInstance \leftrightarrow ModuleInstance relation was implemented as a list of child instances in the parent one.

Table 3.1: Functionality required and data structures used

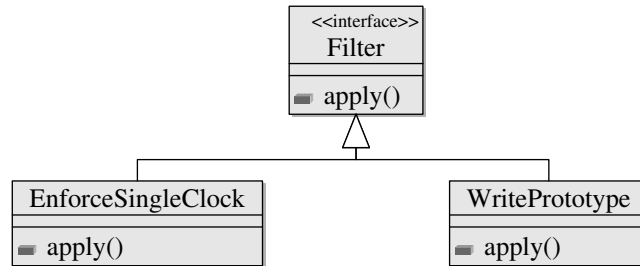


Figure 3.1: Filters class diagram

Since we keep a list of all LPM instances in the database, the `EnforceSingleClock` filter only has to enumerate all flip-flop LPMs and check if the nexus their clock signals are connected to is the same.

NuSMV module definitions

After filters are applied, the implementation starts writing the NuSMV destination file, by instantiating the `Writer` class with the output file stream as a construction parameter.

Initially, each module in the database is processed according to the algorithm described in section 2.2.3 on page 38. This process is guided by trying to *fill in the blanks* in the following module definition template:

```

MODULE modulename(inputport1name, inputport2name, ...)
VAR
    childinstance1name = childinstance1module
        (childinstance1inputport1expr,
         childinstance1inputport2expr, ...);
    childinstance2name = childinstance2module (...);
    ...
    childlpm1name = childlpm1templatename
        (childlpm1inputport1expr,
         childlpm1inputport2expr, ...);
    childlpm2name = childlpm2templatename (...);
    ...
DEFINE
    outputport1name := outputport1expr;
    ...
  
```

- *modulename* is the module name of the generated NuSMV module. It comes from the source Verilog module that is being converted. How-

ever, the sets of valid identifiers as defined in the Verilog and NuSMV language definitions, while overlapping, do not exactly match. For this reason, care is taken by the implementation to *sanitize* Verilog identifiers not valid in NuSMV.

For example, `SPEC` (NuSMV is case-sensitive) is a valid identifier in Verilog, but not in NuSMV because it is a reserved language keyword. The implementation fixes this by suffixing such identifiers with a `#`, a character that is not allowed in Verilog identifiers (so that unfortunate identifier collisions are prevented). For this task, a list of reserved NuSMV keywords was embedded in the converter source code.

- NuSMV requires us to name the formal arguments to a function on its header. As per the algorithm, those are the input ports of the source Verilog module (*inputport1name, ...*).
- Each of the child instances of the source Verilog module is mapped to a NuSMV state variable (`VAR`), with the name of the variable the same of the instance name given in Verilog¹: *childinstance1name, ...*

As per the algorithm, the actual parameters passed to the instantiations of those NuSMV modules correspond to expressions generated by traversing the elaborated design graph.

- Each of the LPMs instanced by the module are also mapped to NuSMV state variables like if they were child modules. In fact, they are child modules because as we will see later each LPM will have its own NuSMV module definition.
- After the state variables, we alias certain expressions to identifiers (*outputport1name, outputport2name, ...*) scoped to the module being generated.

The Eval module

When an expression needs to be generated as per step 3 of the algorithm referenced in 2.2.3, the `Eval` module is called, which takes a starting point of the graph as parameter and starts the exploration from it.

One of the issues that were brought early during implementation is that sometimes there can be two plausible expressions for a given entry point. For instance, in the following Verilog source file:

¹Verilog allows for unnamed instances: Icarus gives autogenerated names for those.

```

module evaldual(i, o);
    input i;
    output o;
    passthrough p(i, o);
endmodule
module passthrough(i, o);
    input i;
    output o;
    assign o = i;
endmodule

```

Trying to generate an expression for the output port `o` of the `evaldual` module generates two semantically valid expressions:

- `o := i`
- `o := p.o`

Technically, the latter expression is more correct. Yet, our current implementation, when given such a choice, *always* decides in favor of the simplest one, defined as the one that was found to have the least amount of circuitry (nodes in the elaborated design graph) between.

This is implemented simply by discarding results during exploration with an higher distance from the start node than the current best result.

In the case at hand, the simplest option is obviously the first one, so the generated NuSMV file ends up like this:

```

MODULE evaldual(i)
VAR
    p : passthrough(i);
DEFINE
    o := i;

MODULE passthrough(i)
DEFINE
    o := i;

```

which looks as if the converter had applied a trivial optimization (it is indeed what it has done) by removing an unneeded connection to the `passthrough` module – yet it did not remove the now unrequired module definition.

3.1.3 Library of Parameterized Modules

During analysis (section 2.2.1) we found out that Icarus has a library of already synthesized modules that it can use to avoid having to synthesize this common functionality each time (for example, the library contains half-adders, flip-flops, etc.). During the design, we decided we would create NuSMV module definitions for each of these modules manually, as there is not many of them and more importantly there is no Verilog source for them.

In the normal conversion process, after all of the module definitions have been written into the NuSMV file, the implementation starts writing the definitions for all the LPMs that have been used in the previous module definitions. This is obviously desirable versus simply dumping all the LPM module definitions we have in the library, and is implemented simply by having a global dictionary of `LpmTemplates` in the data layer where the key is the actual definition text of the involved LPM.

This way, each time the system needs to instantiate an `Lpm` object, it creates its definition, but then checks if the same definition (and thus, the same LPM) was not in the `LpmTemplates` dictionary. If it was, the `Lpm` instance just references that `LpmTemplate` object. If it was not present, a new unique `LpmTemplate` is created that will be emitted later on the NuSMV file.

In the original class diagram (figure 2.5 on page 37), we described multiple specializations of the `Lpm` class for each kind of LPM. This is because each module has different port configurations. Also, we simplify the implementation of the generation of the LPM module definition by using polymorphism. The `Writer` class just asks the `Lpm` object for its definition, and each specialization will take care of loading and returning the right definition.

Also, many modules have *parameters* (i.e. width for a half-adder). Each specialization will properly take care of reading its instantiation parameters from Icarus and to consider them when creating the correspondent definition.

As an example, one specializazion is `LpmAlu` that implements most arithmetical (adders, subtractors, multiplier, ...) and logical (shifters, ...) with the following template:

```

MODULE lpm(Data, DataB)
— Library: <name of the operation> <signedness>
DEFINE
  Q := Data <equivalent NuSMV operand> DataB;

```

For instance:

```

MODULE lpm(Data , DataB)
  — Library: Adder
DEFINE
  Q := Data + DataB;

```

Note that NuSMV handles the width automatically due to type inference, so there is no need for the converter to generate different definitions of this LPM for different widths.

D Flip-flop

There are two special LPMs in the library: flip-flops and latches. During analysis, we found that for a synthesized circuit, all state information is stored on instances of these two LPMs. Therefore, those two modules are the only ones in generated NuSMV design that will have state variables.

The Icarus definition for a D Flip-flop includes all the following functionality:

- Asynchronous clear (**Aclr**)
- Asynchronous set (**Aset**) to a value that is preset in the flip-flop construction.
- Synchronous clear (**Sclr**)
- Synchronous set (**Sset**)
- Chip Enable signal (**Enable**)

Those features are modeled based on the module described in listing 3.2. A few `integer` \rightarrow `word`[*width*] type casts have been omitted for clarity.

Since our converter currently only supports one clock source for all flip-flops, the implementation of the asynchronous and synchronous functionality is exactly identical. However, the distinction is kept for potential future expansion (see section 5.3.3 on page 80).

```

MODULE lpm(Clk, Data, Aclr, Aset, Sclr, Sset, Enable)
— Library: D Flip-flop
VAR
  Q : word[\langle width \rangle];
ASSIGN
  next(Q) := case
    Aclr : 0;
    Aset : \langle presetValue \rangle;
    Sclr : 0;
    Sset : \langle presetValue \rangle;
    Enable : Data;
    TRUE : Q;
  esac;

```

Listing 3.2: Idealized full template instantiation of a D Flip-flop

Gated D Latch

The implementation of the latch is much simpler, again, due to the assumptions the converter makes about the clock.

```

MODULE lpm(Gate, Data)
— Library: Gated D latch
VAR
  Q : word[\langle width \rangle];
ASSIGN
  next(Q) := Gate ? Data : Q;

```

Listing 3.3: Idealized full template instantiation of a gated D latch

3.2 The Circuit Verifier Jutge.org driver

3.2.1 Problem preparation

For performance reasons – not having to do the `tgt-nusmv` conversion every time a student submits a design –, models have to be prepared on the teacher’s machine before being uploaded to the Jutge.org platform. This is done by a script called `cvproblems.py`, analogue to one that was already existing on the Jutge.org servers (`problems.py`) that did a handful of similar tasks like updating the expected outputs from the test vectors using a sample implementation provided by the teacher and running \LaTeX on the problem

statement sources.

During design (section 2.3), we determined that we would need a way to check whether the interfaces of the root modules from both the student and the known-good circuit matched. This was implemented by adding a special mode into the Verilog to NuSMV converter that would also write the prototype of such root module into a special file – a task that is very easy to do for the converter, because of the database it creates as a natural part of the conversion process.

Since, as said, we do not want to run the converter for the known-good problem each time, we also generate this prototype information during problem preparation.

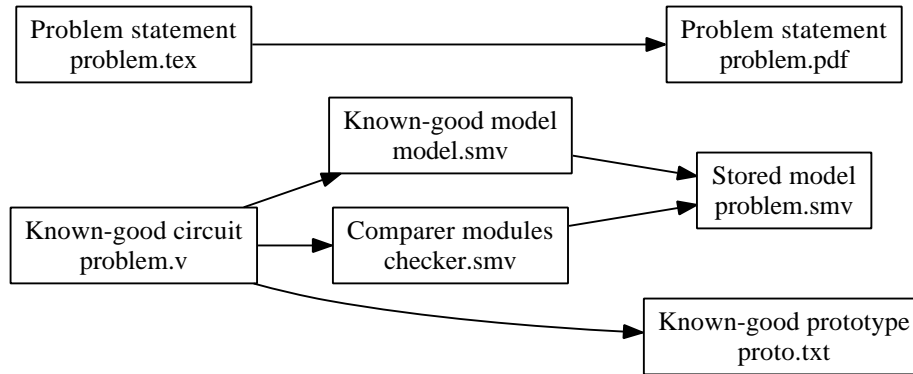


Figure 3.2: Input and outputs for the problem preparation process

The second task to be done during problem preparation is to create the comparer module (as shown on figure 2.7 on page 44).

By reading all the output port names from the prototype information, a very simple Python script can create the comparer module definition.

In order to decide whether the circuit is a combination or sequential one, so that it can determine if the specific CTL specification that must consider reset signals into account has to be written, the implementation reads the prototype information and checks for the presence of either the clock (*clk*) or reset (*rst*) signals. If any of them is detected, we assume it is a sequential circuit and use the extended specification as well as the `reset_detect` module that was described during the design and implemented manually as seen on listing 3.4.

```

MODULE reset_detect(rst)
VAR
    old_rst: word[1];
ASSIGN
    init(old_rst) := rst;
    next(old_rst) := rst;
DEFINE
    negedge := bool(old_rst & !rst);

```

Listing 3.4: `reset_detect` module definition

The comparer module as well as the `reset_detect` are then merged with the known-good model, which is then stored in the Judge.org system along with the problem statement and the known-good module prototype (see figure 3.2). Combining this stored NuSMV file with the student's NuSMV model will result in a valid NuSMV input file.

3.2.2 Submission and correction process

As mentioned in section 1.3, the correction process in Judge.org is handled by what is called a *driver*. While Judge.org does not require drivers to be written in any specific language (communication between the main system and drivers is done via temporary files), the original *std* driver was written in Python and therefore we will use Python for the implementation in order to keep consistency.

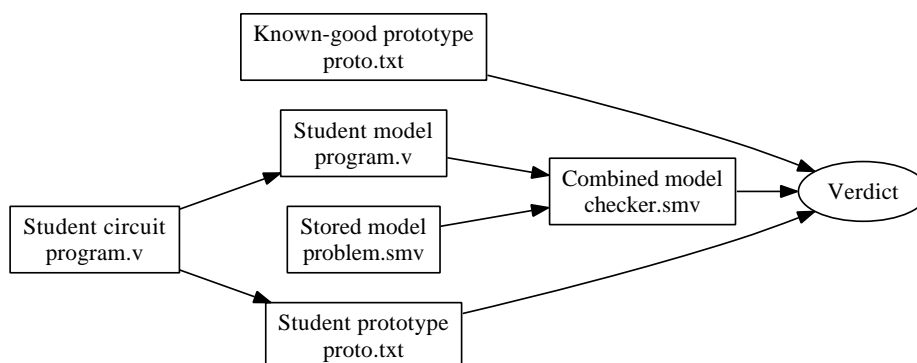


Figure 3.3: Input and outputs for the correction process

The actual steps the driver implementation runs every time a student uploads a solution are:

1. `/usr/local/bin/iverilog-0.8 -t nusmv-user
-pwrite-prototype=proto.txt -penforce-single-clock=1
-pmodule-prefix=s_ program.v -o program.smv
2> compilation1.txt`

This runs the conversion process of the student's circuit into a model. The interface of the detected root module is also dumped to `proto.txt`, while error messages, if any, are written in `compilation1.txt`.

2. `diff ../problem/proto.txt proto.txt`

This checks for differences between the known-good and the student's root module interfaces. The exit code is used to check if *diff* found any differences.

3. `cat program.smv ../problem/solution.smv > checker.smv`

Both the known-good NuSMV model (including all the necessary comparer modules definitions) and the student model files are combined into a single one.

4. `NuSMV checker.smv > verification.txt`

NuSMV is run on the combined file. Exit code and parsing of its standard output are used to check if all specifications were satisfied.

NuSMV generates the following output when a specification is true:

```
| — specification AG (rd#.negedge -> AG equal#) is true
```

And generates a countertrace when a CTL specification is false, an example of which can be seen on listing 3.5.

Mapping a trace generated by NuSMV on a model generated by the converter implemented in this project back into a Verilog trace is trivial, as the NuSMV structure accurately follows that of the Verilog source modules. Virtually, the only differences between a NuSMV trace and a trace that would be understandable to someone who only knows Verilog is to remove the decorations (#) from the identifiers, and convert NuSMV-style literal constants (0ud1_0) into Verilog ones (1'd0).


```
— specification AG (rd#.negedge -> AG equal#) is false
— as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  clk = 0ud1_0
  i = 0ud1_0
  rst = 0ud1_1
  ...
-> State: 1.2 <-
  rst = 0ud1_0
  rd#.negedge = TRUE
-> State: 1.3 <-
  ...
```

Listing 3.5: NuSMV generated counterexample

This is all handled by the Python script, which generates such a Verilog-style trace and passes it back to the web interface via a file named `errorreport.txt`.

Chapter 4

Results

4.1 Use of the converter

The Verilog to NuSMV converter is an Icarus Verilog backend. In order to invoke it, Icarus must be launched with the following arguments:

```
$ iverilog-0.8 -t nusmv -o <outputfile.smv> <inputfile.v>
```

The `-S` argument, used to indicate whether synthesis is wanted or not, is ignored by the *tgt-nusmv* converter, as it always enables synthesis.

The backend has several other optional arguments:

Argument	Effect
<code>-penforce-single-clock=1</code>	Causes <i>tgt-nusmv</i> to enforce that all flip-flops share a single clock source and edge. See section 5.3.3 on page 80 for the rationale behind this functionality.
<code>-pfatal-warnings=1</code>	Causes <i>tgt-nusmv</i> to stop if any warnings were found during conversion. For example, unconnected nets will by default emit a warning but will otherwise be considered as grounded signals by the converter – which creates circuit models that have a clearly defined behavior when the real ones would not. If more strict evaluation is required, this flag will cause the conversion process to fail when any such unconnected net is found.

Table 4.1: *tgt-nusmv* optional flag arguments

Argument	Effect
<code>-pwrite-prototype=<file></code>	Writes the prototype of the root module from the source circuit into <code><file></code> . This was implemented so that the <i>cv</i> driver did not had to do parsing on its own.
<code>-pmodule-prefix=<string></code>	The converter will try to name generated NuSMV modules the same as the original Verilog modules those came from. If this is not desired, setting this option will cause all of the generated module names to be prefixed with <code><string></code> .

Table 4.2: *tgt-nusmv* other optional arguments

Icarus Verilog forces the source circuit to have a single root module, that is, at least one Verilog module that is not instantiated by any other module in the source file. If there is more than one (not really a established good practice), you can specify which one you want to use with the `-s rootmodule` argument. Icarus might discard modules that are not used by the selected module or its instantiated children modules.

4.1.1 Examples

We tested the converter with around a hundred samples coming from a selection of Verilog learning books, as well as a handful of hand-made tests that exploited certain patterns we found interesting.

The following is a very simple Verilog implementation of a two-bit ALU (from [16]):

```

module ALU(Op, A, B, Z);
  input [1:0] Op;
  input [1:0] A, B;
  output reg [1:0] Z;

  parameter ADD = 'b00,
             SUB = 'b01,
             MUL = 'b10,
             DIV = 'b11;

  always @(Op or A or B)
    case (Op)

```

```

        ADD : Z = A + B;
        SUB : Z = A - B;
        MUL : Z = A * B;
        DIV : Z = A / B;
    endcase
endmodule

```

Along with the NuSMV model the converter tool generated:

```

MODULE ALU(A, B, Op)
VAR
    _s7 : lpm_0 ((A[1:1]) :: (A[0:0]),
                (B[1:1]) :: (B[0:0]));
    _s10 : lpm_1 ((A[1:1]) :: (A[0:0]),
                (B[1:1]) :: (B[0:0]));
    _s12 : lpm_2 ((0b1_0_) :: (0b1_0_) ::
                (A[1:1]) :: (A[0:0]),
                (0b1_0_) :: (0b1_0_) ::
                (B[1:1]) :: (B[0:0]));
    _s16 : lpm_3 ((A[1:1]) :: (A[0:0]),
                (B[1:1]) :: (B[0:0]));
    _s4 : lpm_4 ((Op[1:1]) :: (Op[0:0]),
                (_s7.Q[1:1]) :: (_s7.Q[0:0]),
                (_s10.Q[1:1]) :: (_s10.Q[0:0]),
                (_s12.Q[1:1]) :: (_s12.Q[0:0]),
                (_s16.Q[1:1]) :: (_s16.Q[0:0]));
DEFINE
    Z := (_s4.Q[1:1]) :: (_s4.Q[0:0]);

MODULE lpm_0(Data, DataB) — Library: Adder
DEFINE Q := Data + DataB;

MODULE lpm_1(Data, DataB) — Library: Subtractor
DEFINE Q := Data - DataB;

MODULE lpm_2(Data, DataB) — Library: Multiplier
DEFINE Q := Data * DataB;

MODULE lpm_3(Data, DataB) — Library: Divider
DEFINE Q := Data / DataB;

MODULE lpm_4(Sel, D0, D1, D2, D3)
— Library: 8-to-2 Multiplexer
DEFINE
    Q := case
        Sel = 0ud2_0 : D0 ;
        Sel = 0ud2_1 : D1 ;
        Sel = 0ud2_2 : D2 ;

```

```

        Sel = 0ud2_3 : D3 ;
    esac ;

```

It can be clearly seen that `_s4` (a synthesizer generated name) is the multiplexer that chooses between the different ALU functions (the `Op` bus being the selector key). The output of the ALU module (`Q`) is the output of the multiplexer, and its inputs are the outputs of the four arithmetic LPM instances.

The model is easily verified to be accurate by plain eyesight. Note however that Icarus Verilog instantiated a 4-bit multiplier (`_s12`) when a 2-bit would have been enough. Yet, since the upper bits are ignored, the result is truncated and thus matches the semantics of the original circuit.

The above was a purely combinatorial circuit. We also show a sample of a sequential circuit, from [16]:

```

module SyncFlipFlop (ClkB, Reset, Set, CurrentState,
    NextState);
    input ClkB, Reset, Set;
    input [1:0] CurrentState;
    output reg [1:0] NextState;

    always @(negedge ClkB)
        if (!Reset)
            NextState <= 0;
        else if (!Set)
            NextState <= 2'b11;
        else
            NextState <= CurrentState;
endmodule

MODULE SyncFlipFlop(ClkB, CurrentState, Reset, Set)
VAR
    _s2 : lpm_0 (ClkB, (_s15.Q[1:1]) :: (_s15.Q[0:0]),
        !(Reset));
    _s15 : lpm_1 (!(Set), (CurrentState[1:1]) ::
        (CurrentState[0:0]), (0b1.1) :: (0b1.1));
DEFINE
    NextState := (_s2.Q[1:1]) :: (_s2.Q[0:0]);

MODULE lpm_0(Clk, Data, Sclr)
    — Library: D flip-flop (with Synchronous clear)
VAR

```

```

    Q : word[2];
ASSIGN
    next(Q) := case
                bool(Sclr) : 0ud2_0;
                TRUE : Data;
            esac;

MODULE lpm_1(Sel, D0, D1)
    — Library: 4-to-2 Multiplexer
DEFINE
    Q := case
        Sel = 0ud1_0 : D0 ;
        Sel = 0ud1_1 : D1 ;
    esac;

```

In this case, the synthesized flip-flop has both a synchronous clear – `Sclr` and synchronous preset that is implemented by multiplexer at the input of it. The rest of the flip-flop features that were seen in listing 3.2 are automatically removed from the generated NuSMV flip-flop module.

4.1.2 Performance and resource usage

During the goals we established that it was within our interest to perform a system that is fast enough to be used interactively.

We were able to run a part of the testsuite composed of 42 Verilog source files totaling 420 lines in less than 0.26 seconds¹, all while using a maximum of 9 Kibibytes of system memory – including time spent reading and writing disk files. Intuitively, the converter can indeed be used interactively.

it is expected, however, that the time used by both the converter and the model checker will increment exponentially as the number of bits of state in the logical circuit increases, specially for the latter.

A 128 bits RAM, instantiated in a few lines in Verilog, would innocently be converted into a model that has over 300 *undecillion* possible states (2^{128}).

Such a model would obviously not be fully verifiable by a model checker, unless some higher-level simulation technique is applied. This is usually known as the *state explosion* problem.

¹Measured on a 3.33Ghz Intel Clarkdale core

To check for this, we have designed a very simple RAM implementation in Verilog with 8 bit word sizes, and tried to do equivalence checking for different numbers of words.

```

module ram(clk , iv , ia , oa , ov);
    // Clock signal
    input clk;
    // Write address , Read address
    input [7:0] ia , oa;
    // Value to write
    input [7:0] iv;
    // Read value
    output [7:0] ov;

    reg [7:0] m [ramSize:0];

    assign ov = m[oa];

    always @(posedge clk)
        m[ia] <= iv;
endmodule

```

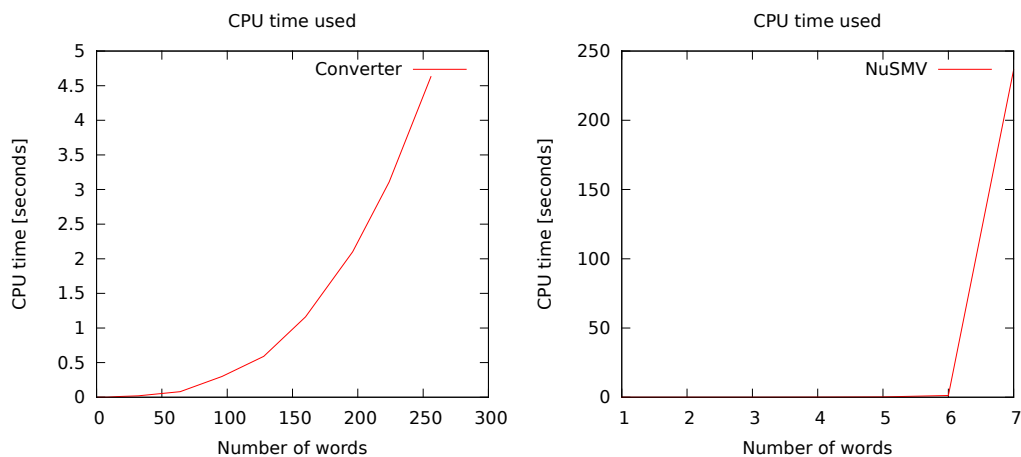


Figure 4.1: CPU time by individual components

As seen on figure 4.1, NuSMV's time increases much more quickly than that of the converter (note the different scales used in the figure). It is expected, as the converter's job pales in comparison of the algorithmic costs of symbolic checking.

The time spent by NuSMV explodes as soon as the total state size approaches 64 bits, because that is the word size of the computer the test was

run on, and certain optimizations NuSMV is otherwise able to do are disabled. As a matter of fact, by default, NuSMV will not accept, for a single state variable, words larger than the word size of the computer it is being run on.

While we could workaroud this during the test, we had to cancel the execution after half an hour had passed. More than one minute is way too much for interactive use and thus it was not felt necessary to continue with the testing.

Since the Jutge.org computers are 32 bit, it is to be expected that only checking for circuits with up to 32 bits of state will be reasonable.

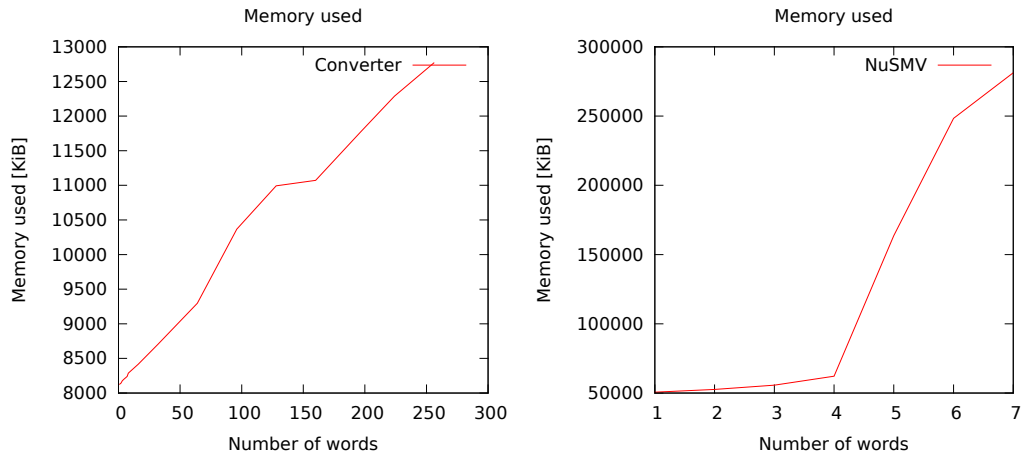


Figure 4.2: Memory usage by individual components

As for memory usages, plotted on figure 4.2, they are clearly not a concert for the normal usage of our project. NuSMV starts using up to a few hundred megabytes on the larger ones, but by then the CPU times required to perform the checking on such big models already makes their verification virtually impossible on lesser powerful machines.

4.2 Sample student interaction

In this section we will evaluate the external use of the system by an student, and compare it to that of the original Jutge.org platform as shown on figure 1.7 on page 24.

The first step a student has to do is to login.

Jutge.org
The Virtual Learning Environment for Computer Programming

Login

Email:

Password:

[Log in](#)

[Proceed to register?](#)
[Try a demo account?](#)
[Forgot your password?](#)

Homepage

Welcome to *Jutge.org*, the Virtual Learning Environment.

Jutge.org is an education online program with more than 600 graded problems using a time, memory and security restrictions. More features include: creating your own courses, attaching documents, creating exams, as well as roasting their students and

For students:

Figure 4.3: Login

After login, the student is able to enroll into courses, and list the problems available for each course. However, there is not yet a HDL course created on the system, so we will skip this interaction and assume the student already knows the problem identifier or URL.

Once a problem is selected, we can download the statement (as a PDF) and any other attached files, if any.

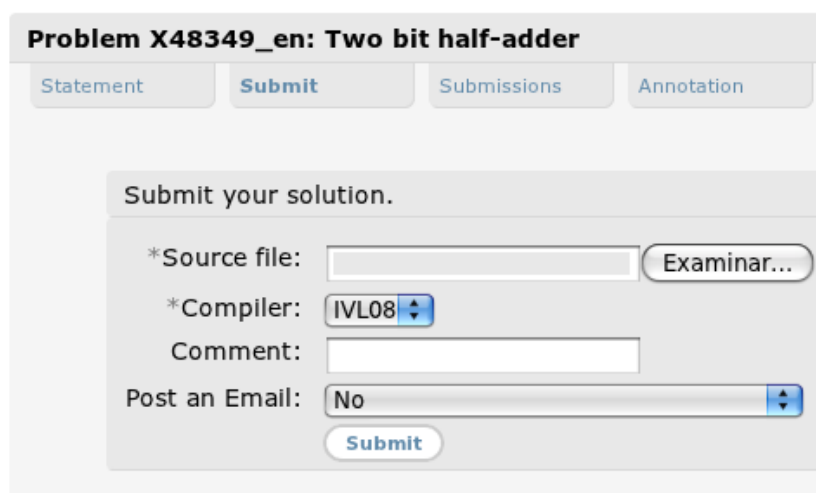
The screenshot shows the Jutge.org website interface. At the top, the logo 'Jutge.org' is displayed with the tagline 'The Virtual Learning Environment for Computer Programming'. A navigation sidebar on the left includes links for Profile, Documentation, Courses, Lists, Problems, Submissions, Documents, Exams, and Favorites, along with a Log out button. The main content area is titled 'Problem X48349_en: Two bit half-adder' and features tabs for Statement, Submit, Submissions, and Annotations. Under 'Problem files', there are icons for a PDF document, a printer, a folder, and a 3D cube. The 'Statement' section includes a 'Hide Image' button and the title 'Two bit half-adder'. The problem description states: 'Implement a two bit half-adder without carry-out. The root module must be called `suma`.' It also defines 'Input' as two bit sized inputs a and b , and 'Output' as a two bit output r . At the bottom, 'Problem information' lists the author as Javier de San Pedro Martín, the generation date as 2011-04-12 11:37:31, and the copyright as © Jutge.org, 2006–2011, with the URL http://www.jutge.org.

Figure 4.4: Reading the problem statement

After carefully reading the statement, the student writes its proposed implementation. For testing purposes, we have deliberately created invalid Verilog code (`--!` is not an operator):

```
module suma(a, b, r);  
  input [1:0] a;  
  input [1:0] b;  
  output [1:0] r;  
  
  assign r = a --! b;  
endmodule
```

that we will submit to the system.



Problem X48349_en: Two bit half-adder

Statement Submit Submissions Annotation

Submit your solution.

*Source file: Examinar...

*Compiler: IVL08

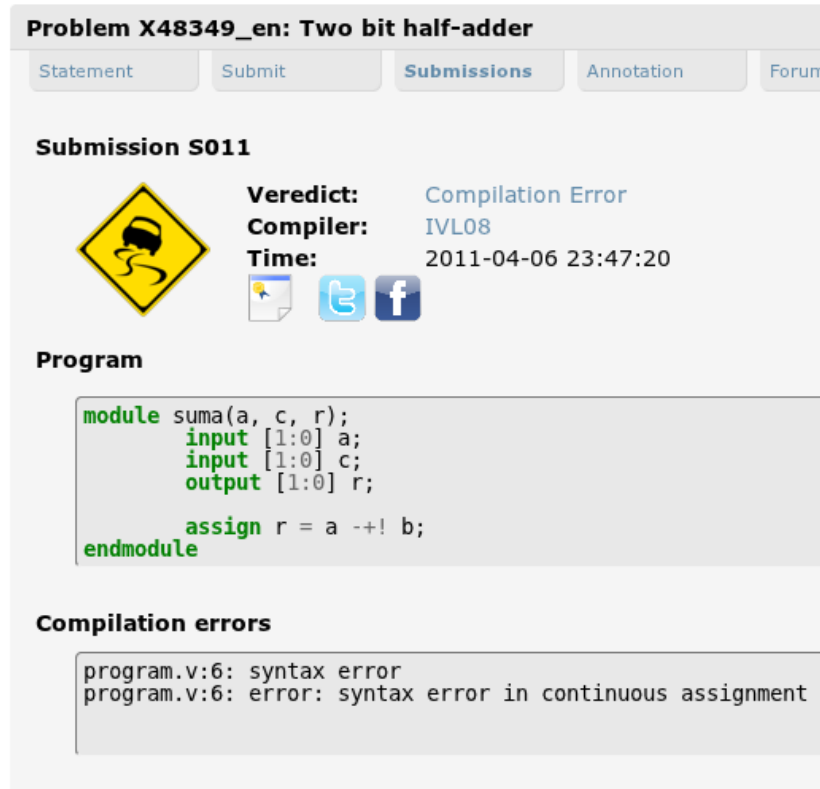
Comment:

Post an Email: No

Submit

Figure 4.5: Submit form

The synthesizer detects the invalid submitted design and a few seconds after submission, we get a *Compiler Error* verdict.



The screenshot shows a web interface for a problem titled "Problem X48349_en: Two bit half-adder". It has tabs for "Statement", "Submit", "Submissions", "Annotation", and "Forum". The "Submissions" tab is active, showing "Submission S011". A yellow warning icon is displayed next to the verdict "Compilation Error". The compiler used is "IVL08" and the submission time is "2011-04-06 23:47:20". There are social media icons for a document, Twitter, and Facebook. The "Program" section contains the following Verilog code:

```
module suma(a, c, r);
  input [1:0] a;
  input [1:0] c;
  output [1:0] r;

  assign r = a -+! b;
endmodule
```

The "Compilation errors" section shows the following messages:

```
program.v:6: syntax error
program.v:6: error: syntax error in continuous assignment
```

Figure 4.6: Compiler error

If we try to submit a circuit that produces wrong output:

```

module suma(a, b, r);
  input 1:0 a;
  input 1:0 b;
  output 1:0 r;

  assign r = a - b;
endmodule

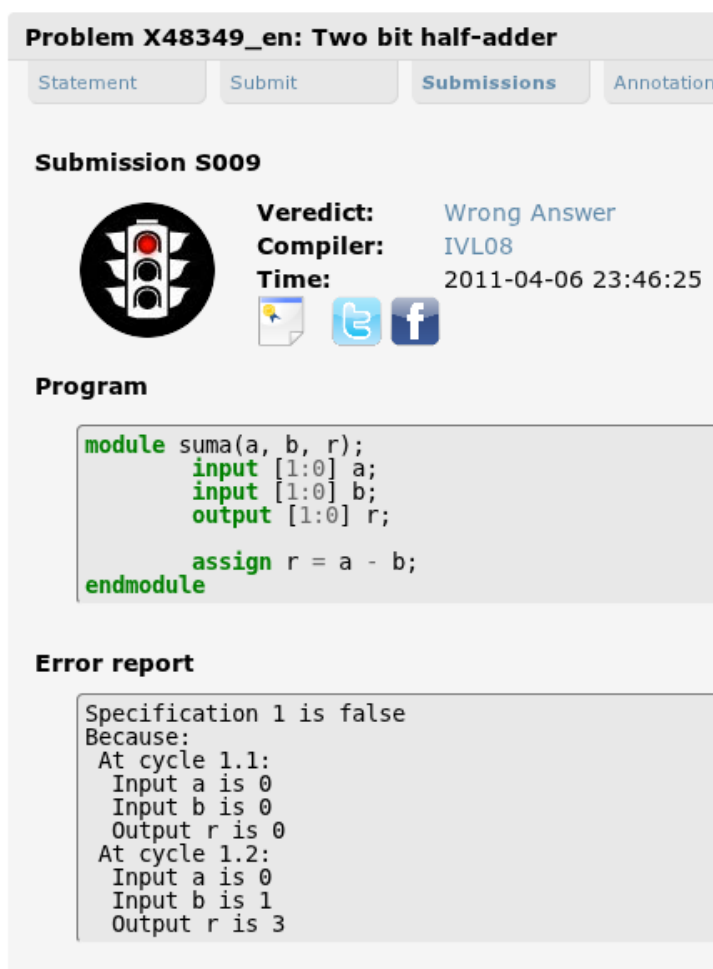
```


NuSMV detects the invalid output and creates a different verdict:

Problem X48349_en: Two bit half-adder

Statement Submit Submissions Annotation

Submission S009

 **Verdict:** Wrong Answer
Compiler: IVL08
Time: 2011-04-06 23:46:25



Program

```

module suma(a, b, r);
  input [1:0] a;
  input [1:0] b;
  output [1:0] r;

  assign r = a - b;
endmodule

```

Error report

```

Specification 1 is false
Because:
At cycle 1.1:
  Input a is 0
  Input b is 0
  Output r is 0
At cycle 1.2:
  Input a is 0
  Input b is 1
  Output r is 3

```

Figure 4.7: Wrong answer

We not only get the error message, but also an example of inputs that cause our circuit to fail.

If we submit the proper answer, we rightfully get an *Accepted* message:

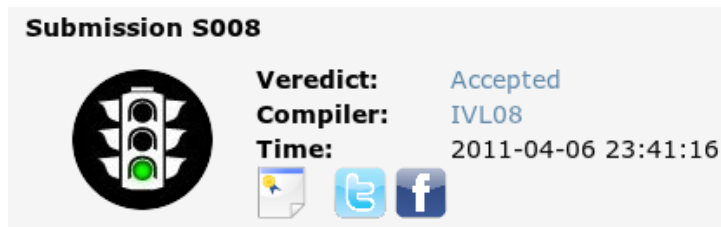


Figure 4.8: Accepted

4.3 Creating a new problem

The process to add a new problem to a course is yet to be fully automated – we expect a reasonable way to do it in the future.

However, Judge.org uses a standardized directory structure for problems. Each problem has its own directory, and once the full structure is created, adding the problem is a matter of a few minutes of work.

1. The directory created for a problem is by convention called *name-of-the-problem.pbm*.
2. Each problem must have two files inside this directory. The first one is `handler.yml` and must contain the following, which indicates to the system this problem is to be handled by the `cv` driver developed on this project.

```
handler: cv
```

3. The second one is `problem.yml`, which must contain the problem name, and author contact information at a minimum.

```
email: author@example.com
title: Two bit half-adder
author: Example Author
```

If localization is necessary, additional `problem.*.yml` files can be created, where `*` is an standard ISO 639-1 code for the language (for instance, `problem.en.yml`).

4. The problem statement can be written, in multiple languages, into `problem.*.tex`.
5. The known-good circuit has to be written into a `solution.v` file.
6. Finally, `cvproblems.py` must be run within the problem directory. It generates the PDF files that will be displayed to the student (see figure 4.4), as well as the model files required by the driver by analyzing the `solution.v` file. If there are any synthesis errors on the known-good circuit, those will be shown now.

Chapter 5

Conclusions

We have developed a system for the exhaustive verification of circuits that is simple to use and works for a large set of simple Verilog circuits.

The converter component satisfies its requisites: it can be used interactively on a modern computer with reasonable complexity circuits, and the models it generates are at least understandable by humans. It is conceivable that models generated by this converter could later be reused as part of the construction of another NuSMV model.

The developed system has been successfully integrated with the existing Jutge.org platform in a way that does not substantially modify the user interaction patterns.

These patterns have been tested by thousands of students that the Jutge.org platform has been in use by the Technical University of Catalonia, and thus we can reasonably say that it is easy to use as we required in the project goals.

However, it is evident that there are many areas for expansion available for this project. A few will be detailed in section 5.3.

5.1 Personal conclusions

This project represented my first contact with the HDL world. Thus, I spent the first few months of the project getting used to the the Verilog language and the specifics of HDL design.

Also, thanks to my project's supervisors I had access to a Xilinx FPGA which, while not being strictly required for the development of this project, allowed me to view the later stages of a typical HDL design cycle and test

my Verilog designs in hardware, which helped me get enthusiastic about the language.

There are only two courses in the official informatics engineering curriculum from the Barcelona School of Informatics that even mention Verilog, and none of them are compulsory, while it is clear that it is a topic that is been gaining importance for the last decade up to the point I strongly believe basic knowledge of the area should be mandatory for any engineering degree minimally related to electronics.

Thus, anything that could aid in the development of a HDL course is in my opinion a very interesting contribution.

This project is one small step in this direction, as it should help interested students increase the speed by which they learn the language allowing them to solve and test exercises in a more efficient manner. I am fully confident that the system developed in this project, within its limitations, will work well enough for at least introductory Verilog exercises.

During the project, I learned many languages far from the traditional imperative models – like Verilog and NuSMV –, and I was introduced to electronic design automation as well as model checking, all along while testing the most used open source software system in both areas.

5.2 Cost study

The topic for this project was known by January 2010. The first months where spent learning the Verilog language itself, as well as getting used to the specifics of FPGA based development.

During Summer 2010 I was unfortunately not able to work on the project due to external work reasons.

Coding on what would become the current `tgt-nusmv` converter started on late September 2010. On the other hand, the converter was finished by March 2011. By then, design on the Jutge.org integration had started.

Using the frequency of commits to the version control system used during development of the converter module (*Git*), we can estimate with reasonable accuracy the number of hours employed for the development of this project.

Month	Work done	Man-hours
February 2010	Learning Verilog	40 h.
March 2010		10 h.
April 2010	Converter design and prototyping	40 h.
October 2010		100 h.
November 2010		100 h.
December 2010	Converter implementation	40 h.
January 2011		40 h.
February 2011		100 h.
March 2011	Jutge.org integration	140 h.
April 2011	Finishing this report	100 h.
		710 h.

Table 5.1: Time study

Assuming a fixed labor rate of 30€ per man-hour, the total human cost of the project is estimated at 21300€.

As for the tools used during the project, software-wise:

Software	Cost
Gentoo GNU/Linux operating system	Free
Icarus Verilog 0.8.7	Free
NuSMV 2.5.2	Free
T _E X Live 2008	Free

Table 5.2: Software cost study

The computer used to run those tools is valued at 800€. A less powerful computer and thus cheaper might also have been usable for the project. Yet, we will use this figure as an approximation.

	Cost
Engineering costs	21300€
Software costs	0€
Hardware costs	800€
Total project cost	22100€

Table 5.3: Total costs

5.3 Future work

Some of the features that do not currently work in the tgt-nusmv converter, as seen on the start of this chapter, would be quite easy to implement, but have been deemed outside of the scope of this project.

5.3.1 Model readability enhancements

While readability of the generated models was one of the tgt-nusmv objectives, there is still some work that could be done to greatly improve it with minimal work. For example, a trivial Verilog module with a single continuous assignment of a two-bit bus:

```

module bus(input [1:0] i, output [1:0] o);
    assign o = i;
endmodule

```

Generates the following model:

```

MODULE bus(i)
DEFINE o := (i[1:1]) :: (i[0:0]);

```

Note that it generates two bit addressing operations as well as one concatenation.

The cause of such behavior is that the *Eval* module (section 3.1.2) works on a bit per bit basis, calculating the expressions for each of the individual bits in any given port, and then concatenating all of the generated expressions – taking bit order into account – to obtain the entire bus expression.

Ideally, the generator would have simplified the entire module to:

```

MODULE bus(i)
DEFINE o := i;

```

This could be implemented by having the Eval module decide, when generating such bus expression, if the expression generated for the current bit is *equivalent*¹ to the expression generated for the previous bus. Merging them into one single direct access without any addressing or concatenation

¹defined as being exactly identical as the previous one except the last addressing operation, which will select bit $n + 1$ if the current expression selected n

operations when possible.

However, this will neither benefit nor hinder the efficiency of the generated models in any way, because NuSMV will decompose them again and work on a bit per bit basis.

5.3.2 Additional language features

As per the Language support table (appendix A on page 87), some features of the language are not implemented.

Most of the unimplemented features come from limitations in the Icarus 0.8 tool, and from those, a big majority are constructions that make no sense in a synthesis environment (and therefore, they are out of scope for our project). Notwithstanding that, there are a few elements that are not allowed for synthesis in Icarus 0.8 yet we found desirable:

- **Repeat/For loops** that can be unrolled at synthesis time because they are either trivial or use fully constant values.
- **Inferring multibit latches.** Latches are usually inferred on Verilog when any branch on a behavioral model flow allows for a `reg` to keep its value between clock cycles. While Icarus correctly creates single bit latches, it failed to create either a multibit latch or multiple latches for a bus. This restriction was partially lifted for the scope of this project by altering the Icarus source code, but a more generic solution is desirable.

Many of those limitations may be removed in future versions of Icarus Verilog, still, `tgt-nusmv` will require modifications in order to work with such future versions.

On the other hand, `tgt-nusmv` also has language limitations that could be improved, like support for named scopes.

5.3.3 Multiple clocks

The current implementation of `tgt-nusmv` does not handle more than one common clock signal for all inferred flip-flops. It assumes all flip-flops are always triggered on each simulation/verifier cycle.

This means it will (gracefully) fail to generate a model for the following circuit:

```

module mff(input clk1 , input clk2 , input i ,
           output reg a , output reg b );
    always @(posedge clk1) a <= i ;
    always @(posedge clk2) b <= i ;
endmodule

```

or even the following one, because each of the inferred flip-flops triggers on a different clock edge:

```

module mff2(input clk , input i , output reg a , output
reg b );
    always @(posedge clk) a <= i ;
    always @(negedge clk) b <= i ;
endmodule

```

It would be easy for `tgt-nusmv` to allow all of those special cases and write proper models for them (in fact the feature had been implemented into `tgt-nusmv` at one point during development). Yet it adds additional complexity to the verification process. Models grow bigger and verification time increases, for a feature that is rarely used on the simple designs we expect students to work with. Also, problems arise in the verification of the interaction of the different clock signals – how do we model them? A trivial answer is to model them like a Verilog simulator would do: by having a “implicit” clock signal whose frequency is the least common multiple of all the involved clock signals, and use it to simulate each of the required extra clock signals from the implicit one. However, while easy to model, that is not how real clock signals on physical hardware interact – but it might be enough for less accurate requirements.

Any future work willing to expand into this area will need to answer these questions first.

5.3.4 More relaxed circuit specifications

The problem preparation scripts, responsible for generating the model for the known-good circuit as well as the state machine that will embed both models and check whether outputs match will currently always require that outputs match during all clock cycles. Otherwise, the circuit will be considered invalid.

This works well enough for combinational and very simple sequential circuits. It is obviously desirable to allow for more relaxed specifications consenting a circuit to have undefined outputs – for a fixed amount of cycles, or

until a `ready` signal fires – after a change in any of the inputs.

Since the system stores NuSMV models, it is already possible to write the specifications with the full power from any of NuSMV supported logics (CTL, LTL, ...). Nonetheless, it is enticing the automated tool would do that. Using bounded model checking[15], we might even be able to ensure the outputs from a student’s circuit become valid in less than or equal number of cycles than the known-good circuit.

5.3.5 Better output messages

Given the traces the verification process generates on failure, it should be trivial to generate a *VCD*² file from them and then plot it visually using any existing waveform viewer tool.

5.3.6 VHDL support

VHDL is the other *big* language within the HDL world, and it is obviously desirable to allow students program in this language if they desire so.

It was one of the objectives of this project to allow for inclusion, in the future, of support for other languages. Thus, most of the components are prepared to do so: the Jutge.org integration scripts allow for the definition of new *Synthesizers*, and the fact that we used NuSMV itself as the language problem specifications are defined in would allow for problems to be defined in VHDL if required.

However, any alternative language support would require the implementation of a converter from that source HDL to NuSMV. This would be by itself a project with a complexity level similar to this one. It is also worth mentioning that there are plans to integrate VHDL support in Icarus Verilog for the 0.10 series.

²Value change dump, a Verilog standardized format for waveform data exchange

Bibliography

- [1] M.R. Barbacci, G.E. Barnes, R.G. Cattell, and D.P. Siewiorek. ISP: A language to describe instruction sets and other register transfer systems. Technical report, Dept. of Computer Science, Carnegie-Mellon University, 1978.
- [2] G. Bell, R. Cady, H. McFarland, B. Delagi, J. O’Laughlin, R. Noonan, and W. Wulf. A new architecture for mini-computers – the DEC PDP-11. In *AFIPS SJCC*, volume 36, 1970.
- [3] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.
- [4] Cadence Design Systems, Inc. *Verilog-XL Reference Manual*, 3.4 edition, 2002.
- [5] Icarus Verilog homepage. <http://www.icarus.com/eda/verilog/>.
- [6] Stephen Williams. *Release Notes for Icarus Verilog 0.8*.
- [7] Kenneth McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie-Mellon University, April 1992.
- [8] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is NP-Complete. *Information Processing Letters*, 5:15–17, 1976.
- [9] Amir Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.
- [10] NuSMV home page. <http://nusmv.fbk.eu/>.
- [11] The Jutge.org homepage. <https://www.jutge.org>.
- [12] Compilar i bufar globus! *Dept. of Llenguatges i Sistemes Informàtics monthly newsletter*, 2nd issue, March 2008.

- [13] Xilinx, Inc. *Formal Verification - Frequently Asked Questions*, May 2010.
- [14] Tom Hawkins. *FNF tool man page*.
- [15] Roberto Cavada, Alessandro Cimatti, Gavin Keighren, Emanuele Olivetti, Marco Pistore, and Marco Roveri. *NuSMV 2.5 Tutorial*.
- [16] J. Bhasker. *Verilog HDL Synthesis: A Practical Primer*. Star Galaxy Publishing, 2nd edition, 1999.

Glossary

AST Abstract Syntax Tree. 32

CTL Computation Tree Logic. 18, 43

driver Encapsulated component part of Judge.org platform, handles the actual student code evaluation task. 26, 42

elaboration In Icarus Verilog, elaboration is the process where the syntax tree is converted into a graph of interconnected components instantiated as required by the design's semantics. 13, 32

equivalence checking The use of model checking to validate if two systems are identical from an external point of view. 16

FPGA Field-Programmable Gate Array. 26

functor In Icarus Verilog, a functor is a program that has an elaborated design as input and modifies it in a specialized way. 13

HDL Hardware Description Language. 4, 26

ISP Instruction Set Processor. 4

LPM Library of Parametrized Modules. 34, 55

nexus In a Icarus Verilog elaborated design graph, a nexus is a kind of node which conveys that all nodes with direct connectivity to it are electronically interconnected. 35

root module The root module in a Verilog circuit is the one that is not instanced in any other module in the circuit; thus, when there's only one, it encompasses the entire circuit. 50, 58, 63

STL Standard Template Library. 48

target In Icarus Verilog, a target is a loadable plugin that handles the actual synthesis, optimization and code generation stages. Also known as a *backend*. 13, 36, 48

Appendix A

Language support

Most of the Verilog language is supported except a few not very often used or not synthesizable constructs. The following is a list of the constructs that are known to work:

Lexical conventions
Operators, White Space, Comments, Numbers, Identifiers, Keywords.
Data Types
Value set, Registers, Nets, Vectors, Implicit Declarations, Net Types, Memories, Integers, Parameters.
Expressions
Binary, Arithmetic, Relational, Equality, Logical, Bit-Wise, Reduction, Conditional, Shift and Concatenation Operators; Net and Register bit Addressing, Memory Addressing; Expression Bit Lengths.
Assignments
Continuous, Procedural.
Gate and Switch Level Modeling
Gate and Switch Declaration Syntax, Implicit Net Declarations; AND, NAND, NOR, OR, XOR, XNOR, NOT gates.
Behavioral Modeling
Procedural assignments; Always, Conditional (<code>if</code>) and Case Statements.
Hierarchical Structures
Modules, Top-Level Modules, Module Instantiation, Input and Output Ports.

The following constructs will **not** be processed by the current implementation:

Data Types	
Strings	Strings are not usually synthesizable.
Strenghts	Wire strenghts are ignored.
Gate and Switch Level Modeling	
Delays	Delays are ignored, as most synthesizers do.
BUFIF1, BU-FIF0, NOTIF1, NOTIF0 gates	Those are not implemented. While it would not be hard to implement them within the current code base, wires with multiple drivers were not tested, which would limit its usefulness.
Behavioral Modeling	
Looping Statements	While looping constructs are usually synthesizable if they can be unrolled at synthesis time (e.g. constant number of times in a repeat statement), the Icarus Verilog version used had no support for them.
Initial Statement	Not generally synthesizable.
Named Block Statement	Not supported, rarely used.

Any remaining language feature not mentioned here is not supported.

Appendix B

Environment setup

During the deployment of this project, we had to install several pieces of software into the virtual machines used by the Judge.org environment (see figure 1.8 on page 25 for a description of the architecture). The software used, its license, where the source was downloaded and how it was build and installed will be documented in this section for completeness.

Icarus Verilog 0.8.7

- Downloaded from:
`ftp://icarus.com/pub/eda/verilog/v0.8/verilog-0.8.7.tar.gz`
- GPLv2 license.
- Requires GNU Make, GCC, flex, bison and gperf.
- Unpacked, built and installed under `/usr/local` as follows:

```
$ tar xzf verilog-0.8.7.tar.gz
$ cd verilog-0.8.7
$ ./configure --prefix=/usr/local
$ make -j 2
$ sudo make install
```

- Note that a few patches where applied to the unpacked Icarus Verilog source. They might not be required, but we used them during our tests. Those can be found under the `verilog-0.8.7-patches` directory.
 - `build-fixes` implements changes that where required to build Icarus Verilog under `gcc 4.4`.

- `latches` adds minimal support for synthesis of latches with width > 1 .
- `anachronism` fixes minor warnings that appear when running with the `stub` driver that was heavily used during development.

Those patches can be applied by running the following snippet in the Verilog source directory:

```
| $ patch -p1 < /path/to/patchfile
```

NuSMV 2.5.2

- Downloaded from:
http://nusmv.fbk.eu/NuSMV/download/getting_src-v2.html
- Main library under the LGPLv2.1 license; some components under the BSD-style Cudd license.
- MiniSat2 support was enabled. It is not strictly required for the project, but potentially desirable in future modifications, as it improves NuSMV's bounded model checking performance. The MiniSat2 library has to be downloaded and unpacked as part of the NuSMV setup. Recent versions of the library are released under a permissive BSD-style license.

The version we used was fetched from:

```
http://minisat.se/downloads/minisat2-070721.zip
```

- Unpacked, built and installed under `/usr/local` as follows:

```
$ tar xvf NuSMV-2.5.2.tar.gz
$ cd NuSMV-2.5.2
$ cp ../minisat2-070721.zip MiniSat/
$ cd MiniSat
$ ./build.sh
$ cd ../cudd-2.4.1.1
$ make
$ cd ../nusmv
$ ./configure --prefix=/usr/local
$ make -j 2
$ sudo make install
```

tgt-nusmv

- Developed as part of this project.
- GPLv2 license.
- Built and installed as follows:

```
$ make -j 2
$ sudo make install PREFIX=/usr/local
# Required by the driver
$ sudo cp /usr/local/lib/ivl-0.8/nusmv.conf
    /usr/local/lib/ivl-0.8/nusmv-user.conf
```

cv driver

- Developed as part of this project.
- GPLv2 license.
- Installed as follows (on master computer only):

```
$ make
# Requires proper Judge.org credentials in
# .netrc file.
$ make push
```