Intelligent Interactive Online Tutor for Computer Language Teaching

Dr Simon Moore
Senior Lecturer
Computer Laboratory
University of Cambridge, UK
simon.moore@cl.cam.ac.uk
http://www.cl.cam.ac.uk/users/swm11

Mrs Kate Taylor
Research Associate
Computer Laboratory
University of Cambridge, UK
kate.taylor@cl.cam.ac.uk
http://www.cl.cam.ac.uk/users/ksw1000

ABSTRACT

We present the case for an intelligent interactive online tutor to teach computer languages, with a particular focus on the Verilog hardware description language. This system allows the detailed syntactic and semantic components to be presented in byte sized chunks with the student's understanding checked and reinforced via problem solving. A key challenge has been the provision of human like feedback to erroneous solutions to encourage and assist the learning process.

Keywords

Context sensitive error messages, Interactive Tutor, On-line learning, Prolog, State machines, Verilog.

1. INTRODUCTION

Teaching computer languages requires the student to understand many detailed syntactic and semantic forms as well as more general concepts, such as object orientation or parallelism. Whilst we have found that general concepts are well taught in lectures, the large numbers of important syntactic and semantic details soon overload most students. We observe that languages can be better taught in a small tutorial group format which allows improved information presentation and interactive problem solving.

However, small tutorial groups are expensive when one has a large class and even experienced supervisors do not spot all of the students' mistakes. We believe that the automated tutor system presented in this paper is a better delivery mechanism than lectures and has comparable benefits to a tutorial approach without being so labour intensive.

This paper focuses on teaching the Verilog hardware description language, though we believe that many of the techniques described could be applied to other programming languages. Verilog is taught as part of the existing Electronic Computer Aided Design (ECAD) course. This course includes a challenging hands-on hardware/software co-design component which is undertaken in supervised laboratory sessions. Our system – the Intelligent Verilog Compiler (IVC) – replaces part of the ECAD lecture course which has been taught in its current form for six years to all second year Computer Science students. These sessions remain unchanged as we introduce the IVC system to help with evaluation of the success of the use of an on-line tutor instead of the information being presented in four lectures.

The students have already learned Java and ML in their first year of study. During these sessions, the students develop Verilog progams to run on Field Programmable Gate Arrays (FPGAs) which provide connections to Light Emitting Diodes (LEDs) and an infra-red receiver. Semantic errors in their code may mean that LEDs do not light up correctly, though this output may be faster than the human eye. To debug effectively, students then use a simulation environment in the Quartus tool from Altera to analyse the circuit's behaviour over time. This simulation is very different from being able to add debug print statements to their Java code, and is another skill to learn in the Laboratory sessions.

The IVC is a web-based tutorial and a collection of programming exercises which are compiled by the Icarus Verilog compiler and then interpreted by the IVC. The IVC provides semantic and syntactic support to the student venturing into Verilog for the first time.

The explainer component of the IVC provides context-sensitive explanation of the compiler error messages that the student generates, together with context-sensitive help drawn from the tutorial pages and a lint-like check.

Once compiled, the students' programs are simulated and the underlying state models are

displayed and commented on to assist in validating the students' design against the specified problem.

The advantages of the IVC compared with other on-line tutorials available now are:

- ability to present information relevant to the task the student is doing ;
- ability to give feedback on the student's progress in learning Verilog via short revision questions;
- ability to provide part programs for the students which are then compiled and checked for correctness and completeness, with an English explanation of any difficulties as well as a graphical representation of the program they have written;
- ability to generate context sensitive help text at runtime by generating an English explanation of syntactic errors made in the programming exercises.

The advantages of IVC over going to the lectures or reading a book are:

- the ease of access to information;
- a clear focus on what the student needs to design code and test a program;
- the ability to work at the student's own pace in their own environment.

2. CONSTRUCTING THE IVC

2.1 Gathering requirements

The Intelligent Verilog Compiler (IVC) has been constructed using a prototyping evaluation lifecycle. Requirements were gathered from a questionnaire given to students who were taught in the traditional four lecture style whilst they completed the practical sessions.

The requirements analysis for the IVC was conducted from three viewpoints: teaching, learning and observing. These draw on the underlying structuralist versus phenomenological cognitive approaches.

Dr Moore provides the teaching viewpoint and the present pedagogical approach to teaching ECAD and Digital Electronics. This includes what constitutes good style and program structure, and further development of the core computer science skills of developing algorithms and data structures. The course is delivered in the framework of the Computer Science course which develops the skills of algorithm design and testing within several programming language courses.

The learning viewpoint is obtained by an on-line survey capturing the phenomenological or first person view of the students doing the practical ECAD laboratory work. There is little published work on how to successfully go from the theory to the actual survey. The design of the survey is based on [1] and [2]. This is augmented with Mrs Taylor's experience of learning the area from a background equivalent to a student without prior knowledge of Digital Electronics but with a knowledge of a variety of third generation and declarative languages.

The observationalist viewpoint was provided by Dr Williams as a teacher but with no previous involvement or knowledge of Verilog and hence fewer preconceptions of what is easy or hard. Dr Moore designed the laboratory sessions for October 2003. He and the other demonstrators who supervise the laboratory sessions provide the structuralist or third person view of the laboratories based on the task that the students have to complete.

The first version of the system, IVC 1.1 was used to conduct user trials for the same cohort of students three months later. This was primarily a revision exercise for the students, but collected very valuable data on programming mistakes and the strategies students used to resolve coding and semantic errors. The students reported back using an informal email, but these very often followed the structure of the original questionnaire.

2.2 From Requirements to Teaching Points

Research undertaken in Cambridge on on-line conferences supporting postgraduate courses [3] suggest four clusters of pedagogies correlated with students' grades. A pedagogy comprises the strategies, techniques and approaches that are used to pass on information to students. Postgraduates have a better developed style of learning, but we use these clustering ideas to look for similar groupings within our student cohorts.

2.3 Defining the pedagogy

The four existing lectures provided the initial list of teaching points to define what the pedagogy must include. What strategies and techniques and approaches would actually work with a web-based delivery must be defined iteratively.

The 2004 survey of the students' experience of doing the laboratory sessions having learned Verilog from four lectures provided the requirements for the construction of the IVC1.1 prototype. The IVC 2.0 system is the full delivery of the pedagogy based on the feedback from the students' experience of IVC1.1.

The teaching points were extended by looking at common programming errors mentioned in the 2004 laboratory survey, and those captured by logging the students' use of IVC1.1. These were analysed and divided into the following three categories:

Conceptual: the student has missed the distinctions between behavioural and structural Verilog, between the different types of assignment used, the four valued logic and how the flow of control and timing are handled.

The resemblance between Java and Verilog lulls the novice into thinking that it will be straightforward to apply algorithms they already know. We hypothesise that the primary conceptual difference is that Verilog is inherently parallel though sequential behaviour can be described, whereas Java is inherently sequential and it requires more effort to write in a parallel manner.

Although Verilog arose from the same need to abstract away from circuit designs to allow larger designs to be created by more people, in a similar way to C arose from Assembler, it retains closer links to the netlists than C does to Assembler. Students need to appreciate this close relationship to underlying circuits. For example, continuous assignment is a conceptual step which students either get straight away, or which they find hard to grasp. One possible explanation, which we hope this research will clarify, is whether this has a direct link to their knowledge of the underlying digital electronics.

Syntactic: the student understands the concepts but cannot quite remember the syntax, or the student has all the right constructs but not necessarily in all the right places. This leads to student comments along the lines of "What is wrong with my assign statement", to which the answer is "Nothing at all: it is simply in the wrong place".

Verilog has more restrictions on nesting the statements and expressions than Java, which the students already know. Many assumptions from Java simply do not work for a Verilog program. For example, there are far fewer defaults used by the Java compiler, whereas the Verilog compiler will not report an error in a declaration but merely assign a default type, and often then allow assignments to the variable that are unexpected by the student.

The common mistakes identified from the questionnaire are categorised into three groups: statement based, block based and module based. The IVerilog compiler performs best on statement-based errors, but without some context, it is impossible to provide a helpful message for block or module errors. This is the area that the IVC explainer focuses on.

Semantic: the student understands the syntax but has not quite met the specification for the programming problem giving a state machine that is partially correct.

2.4 Web page design

2.4.1 Order of presentation

The Verilog language constructs are presented to the student in the traditional bottom-up manner, starting with variables and ending with higher level constructs. The students have not at this stage studied compilers in depth, so Backus Naur format (BNF) is explained and used alongside an English explanation.

2.4.2 Checking understanding

The language basics are reinforced by frequent multiple choice questions. The wording of these questions is carefully designed to cover the common mistakes and misconceptions identified in the questionnaire and user trials. Figure 1 below shows an example.

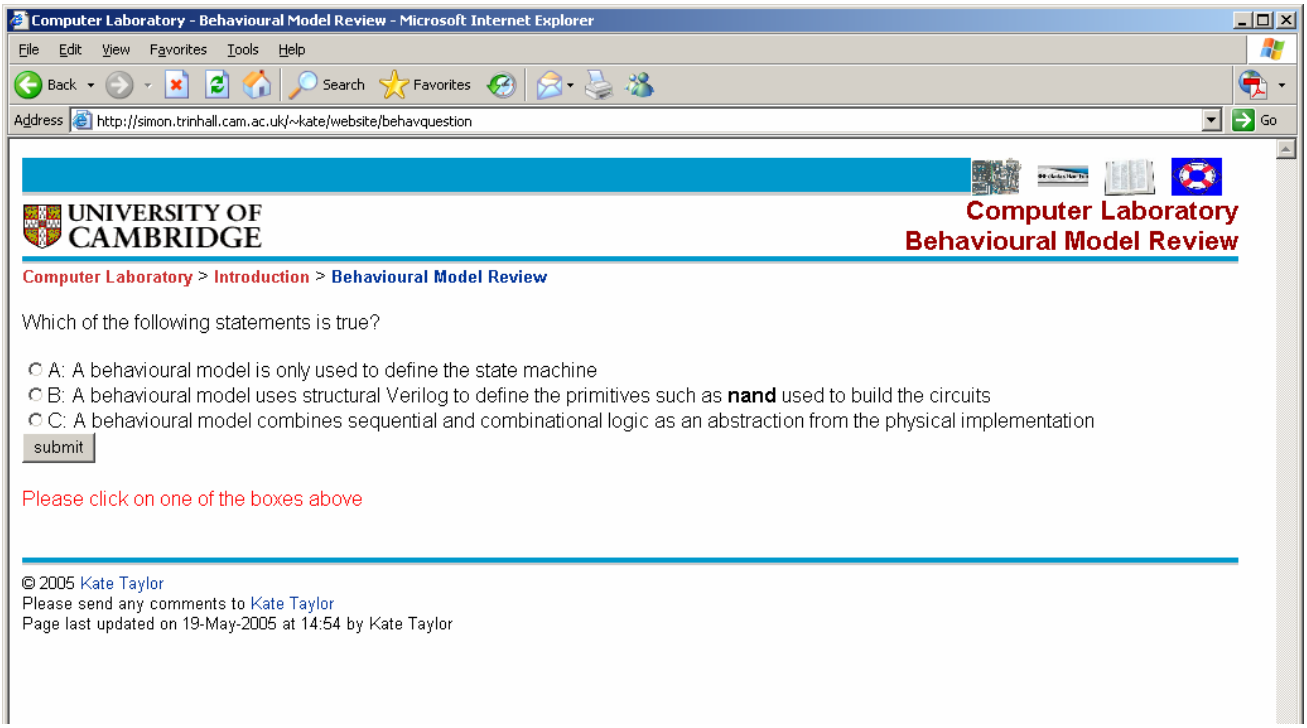

Figure 1 Example of Reinforcement Questioning

The language basics are then reinforced by a number of programming examples; an up-down counter, a multiplexer, an electronic die and a simple traffic light controller.

2.5 BNF as the basis for web pages

Structuring the English text and the structure of the web pages themselves around the BNF grammar of Verilog focuses the student on what can be written where. There are many parallels between the teaching of human languages and the teaching of computer languages: the most relevant to the IVC being the necessity of learning phrases and then larger grammatical constructs, and the importance of practice.

2.6 Help text generation

The help text generation relies on the IVC "understanding" how terms are related to one another. This is done via a semantic web of hypernomic and hyponomic links to model "is a" relationships and meronymic and holonymic links to represent "is a part of".

For example, a declaration has a datatype and a variable name. This is represented as data type and variable name both being meronyms of declaration. Getting an error in the declaration may be due to a misunderstanding of data types of variable naming conventions as well or instead of a misunderstanding of declarations.

The semantic web is used to adjust the level of explanation required and to provide background information to help when the wrong kind of construct is being used, for example when an assignment is in the wrong place in the program.

Use of a semantic web combines the Model Tracing and Constraint Based Tutoring approaches discussed in [4]. The heart of the semantic web is based on the grammar of Verilog itself, as the BNF definition of a language is also hierarchical.

At present, information is provided about the construct itself and its parent and sibling constructs only.

2.7 Context sensitive compiler error messages

Compilers have a limited view of the source code they are viewing as they are based on the BNF of the language construct currently being parsed. In many cases, “parse error at line 6” is the best that can be done. “Error in conditional assignment” is slightly more informative but not much help to the novice programmer.

The IVC explainer aims to provide informally phrased explanations similar to what would be provided by a more experienced programmer leaning over your shoulder: “oh, that means you have used <= where you should have used =”.

2.8 Lint-like checks

The well established Lint [5] tool provides lexical and syntactic analysis of code with extra sanity checks such as variables being initialised before use. We use a similar approach to identify some of

the common mistakes which fall into this category. Many of these are already incorporated into the Icarus Verilog (IVerilog) [6] which uses a similar informal style of presentation.

For example, the IVC lint can spot incorrect range definitions for buses and memories which are syntactically correct, so compile, but do not make sense. Because Verilog does not complain about such errors at compile time, the student is left with code that inexplicably does not run on the Field Programmable Gate Arrrays.

2.9 Representing state models for programs

Discovering how students think about their programs is harder to measure directly. Feedback from the demonstrators and from students themselves from the laboratory sessions in October 2004 and the user trials of IVC 1.1 suggest that there are three groups of student.

The able student has quickly assimilated the syntax of Verilog and can program competently in Java and ML taught in the first year has probably already an intuitive understanding of the notion of state.

Others may understand the language but have not yet noticed the abstraction to a state model. They know intuitively what variables are needed to store data, but may use more than are really necessary. A third group still struggle to turn a specification into an abstract design, having no clear idea how to divide the problem into data and processing.

This correlates with the three levels of student identified in Section 2.2. Learning how to program is a different skill from learning the syntax of a language.

Another new concept for a student used to initialising variables in Java is the idea of self-starting circuits. Because Verilog is a language to define circuits, initialisations to data are replaced by processing in the Verilog modules converting these unsafe states to a safe state. This corresponds to a default clause in the case statement used in both Verilog and Java. Creating designs that are self-starting is part of the good practice that the model checker is seeking to support.

3. SUPPORT WHEN CODE DOES NOT COMPILE

3.1 Explaining compiler error messages

The compiler used by the IVC is the Icarus Verilog compiler. A Perl program examines each error in turn, looking back at the previous statement or block as required for that particular error. The explanation provides a top level description of the problem. The original error message is displayed to help the student understand when that particular error can occur

3.2 References back to the tutorial

The error messages from the IVerilog compiler are scanned for language keywords which are passed to a knowledge base implemented in Prolog [7]. Prolog is a declarative logic programming language based on predicates which can be true or false, and is a de facto standard in generating expert systems and knowledge bases.

Within the knowledge base, each web page is referenced to the language keywords in a hierarchy of usage and composition reflecting the BNF of the language. This links the semantic web to the pieces of HTML that are to be displayed to the student. The glossary entry for the keyword and the BNF definition are combined into the teaching page together with further pages providing the initial description, the first example and a more complicated example.

Having these pieces of HTML separately stored means that the explanations can provide very specific links to each kind of information that the student needs rather than simply displaying the whole tutorial page back again. This supports the stressed student, who does not want to read the whole page when they actually just need one particular part of the explanation.

It also allows future versions of the IVC to be able to support different levels of user expertise to tailor what they read in the tutorial web pages. A confident student does not need to see the early examples, whereas a mystified student faced with syntax errors certainly does.

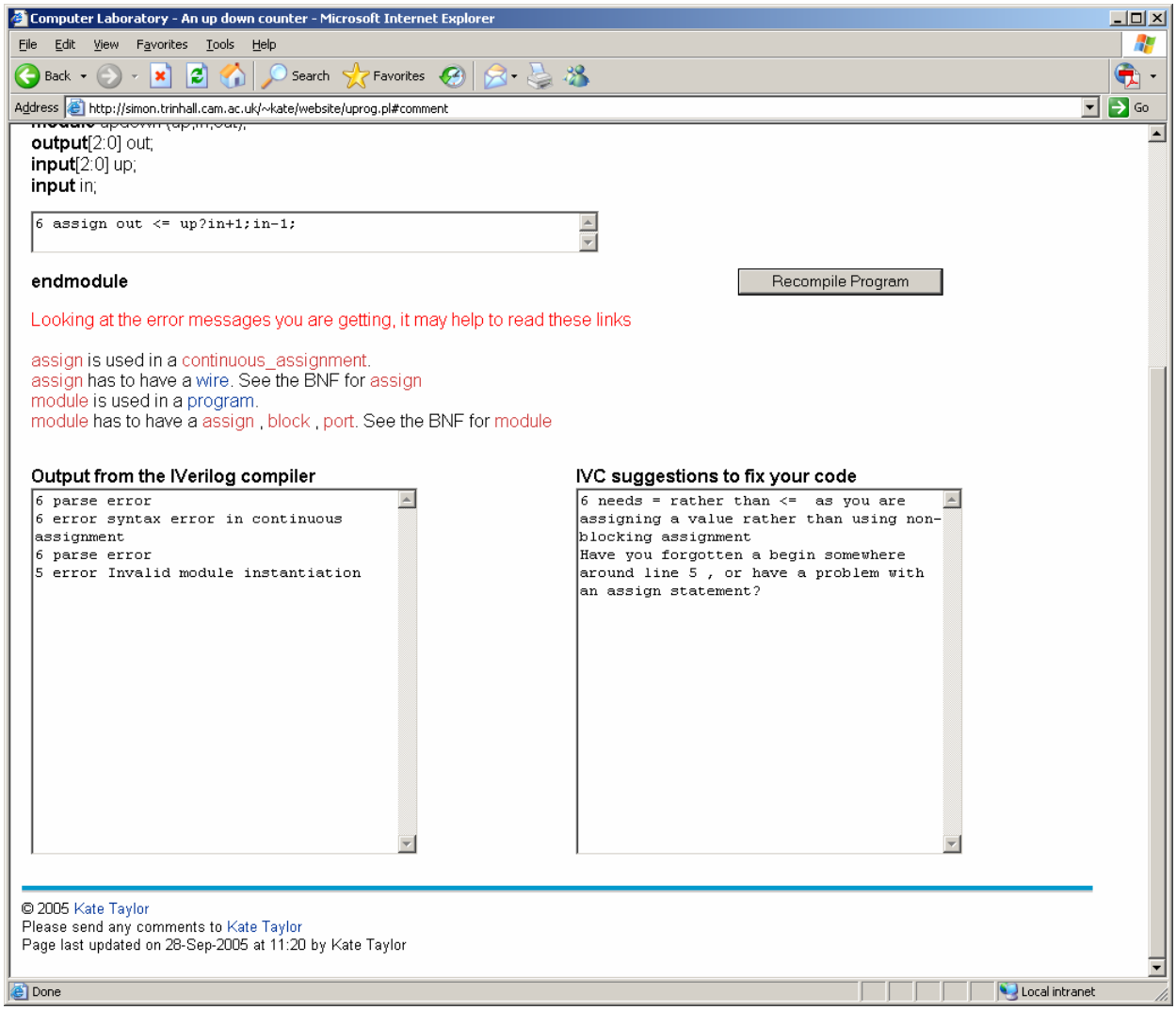

Figure 2 Context sensitive explanation of compiler messages and help text provision

The IVC displays relevant links embedded within a structural explanation of the language constructs, as shown in the central red text in Figure 2, part of which is reproduced below, with links are shown underlined.

`Assign is used in a continuous assignment`

`an assign has to have a wire. See the BNF for assign`

The information about the BNF for the assign statement is given informally in the explanatory sentences. The first line provides information about the concept of continuous assignment: did the student realise that this is what they were doing? Were they really trying to write an assignment within an always block?

The second line deals with what an assign statement needs to have: in this example the error is with using <= instead of = in the assignment. This may be a typographical error, or may reflect that the student has not realised the implications of the different assignment types. The help text generated by the IVC is designed to help the student in either case in the same way that a demonstrator would.

The IVC cannot at present use previous knowledge of the student or conduct a conversation with them in the way that a human could, but it can use its knowledge of the structure of the Verilog language to provide more than just appropriate links back into the tutorial.

4. SUPPORT WHEN CODE DOES NOT WORK

4.1 State space restrictions

The problem specifications are presented to the student in English, with the correct state diagram. The model checker provides the state diagram together with comments on how closely the student's program matches the problem specification.

The state diagram is generated by restricting the student to a template of variable declarations and partial output assignments defining the state space, leaving the student to produce the state space contents. A Prolog file relates errors in the state space to an English explanation of the probable cause relating directly back to the problem definition.

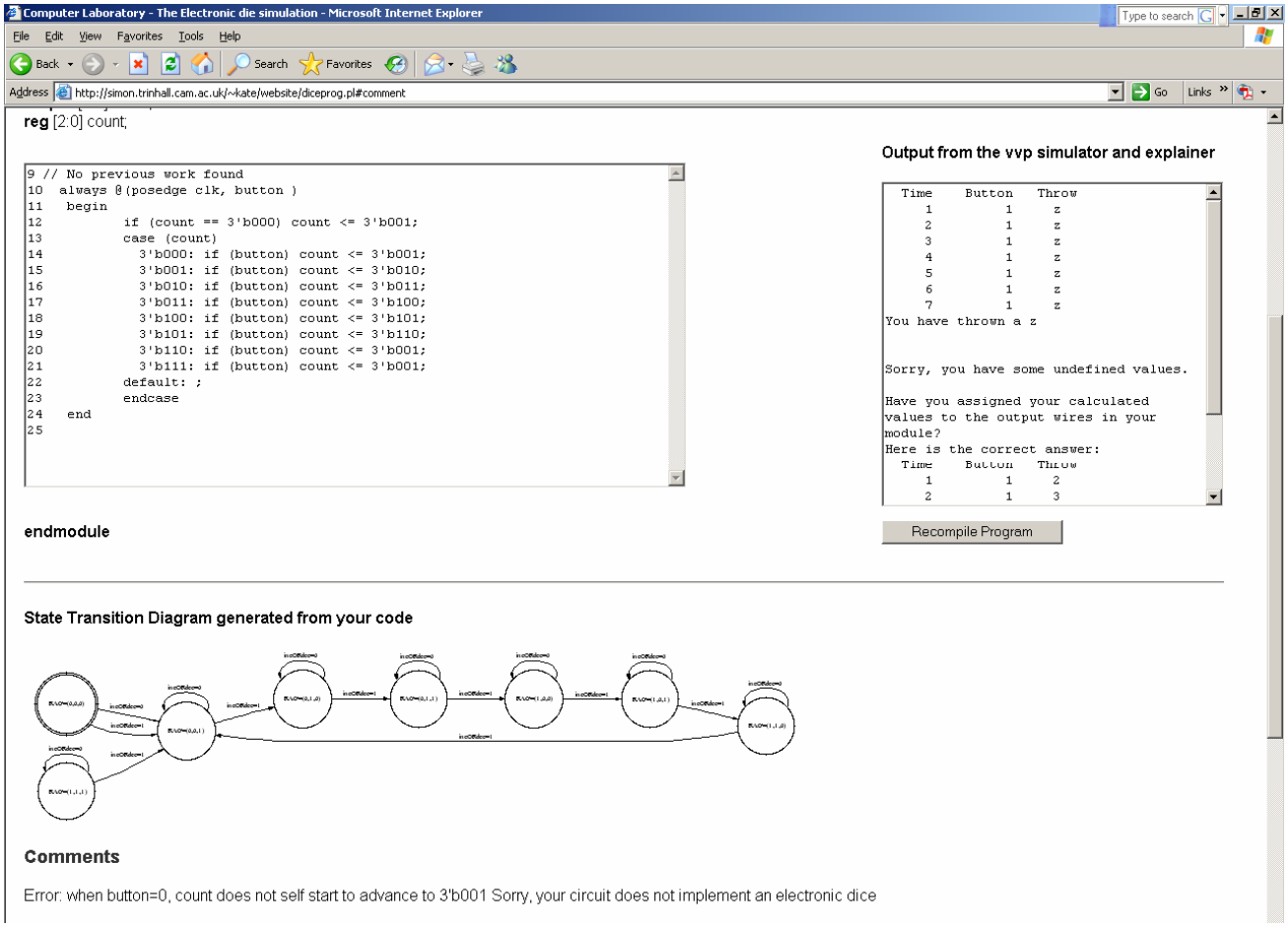

Figure 3 Identifying semantic errors

Another Prolog predicate is used to check that the all the necessary states are reachable and that the circuit will self-start into a correct state if started in an incorrect one. For example, a three bit state representing a traffic light needs move to the safe state of 100 (red) if the circuit is in an incorrect state such as 111 (all lights on).

4.2 Discrete event simulation

A discrete event simulation is provided based on the vvp component of the IVerilog compiler. This generates a simulation trace output in a text format, shown in the middle right hand window in Figure 3 above. The simulation explainer helps pick up wiring errors that produce unknown values on wires.

In this case, the student has forgotten to assign the calculated values to the wires. This generates an unknown "x" value which produces an error in the state space which the model checker spots as an error in the transitions.

The simulation provides another explanation of the error which is more familiar to the student used to adding print statements to code that does not work. It identifies which wire is unconnected and hence providing unknown "x" values.

4.3 Model checking in action

Figure 3 above shows the model checker spotting an error in wiring where a calculated value has not been "wired" in to an output variable.

```
When button = 0 count does not advance
to 3'b001.  Sorry, your code does not
implement an electronic die.
```

Figure 4 below shows an example where the simulation trace is unhelpful because the incorrect program stops producing new output once the electronic die it represents gets stuck on the value "2". The model checker, on the other hand, scans all of the state space and is able to identify incorrect behaviour, thereby pointing the student directly to the problem.

It is extremely likely that the student has cut and pasted the lines in the case statement but forgotten to edit the line afterwards. We have all been there.

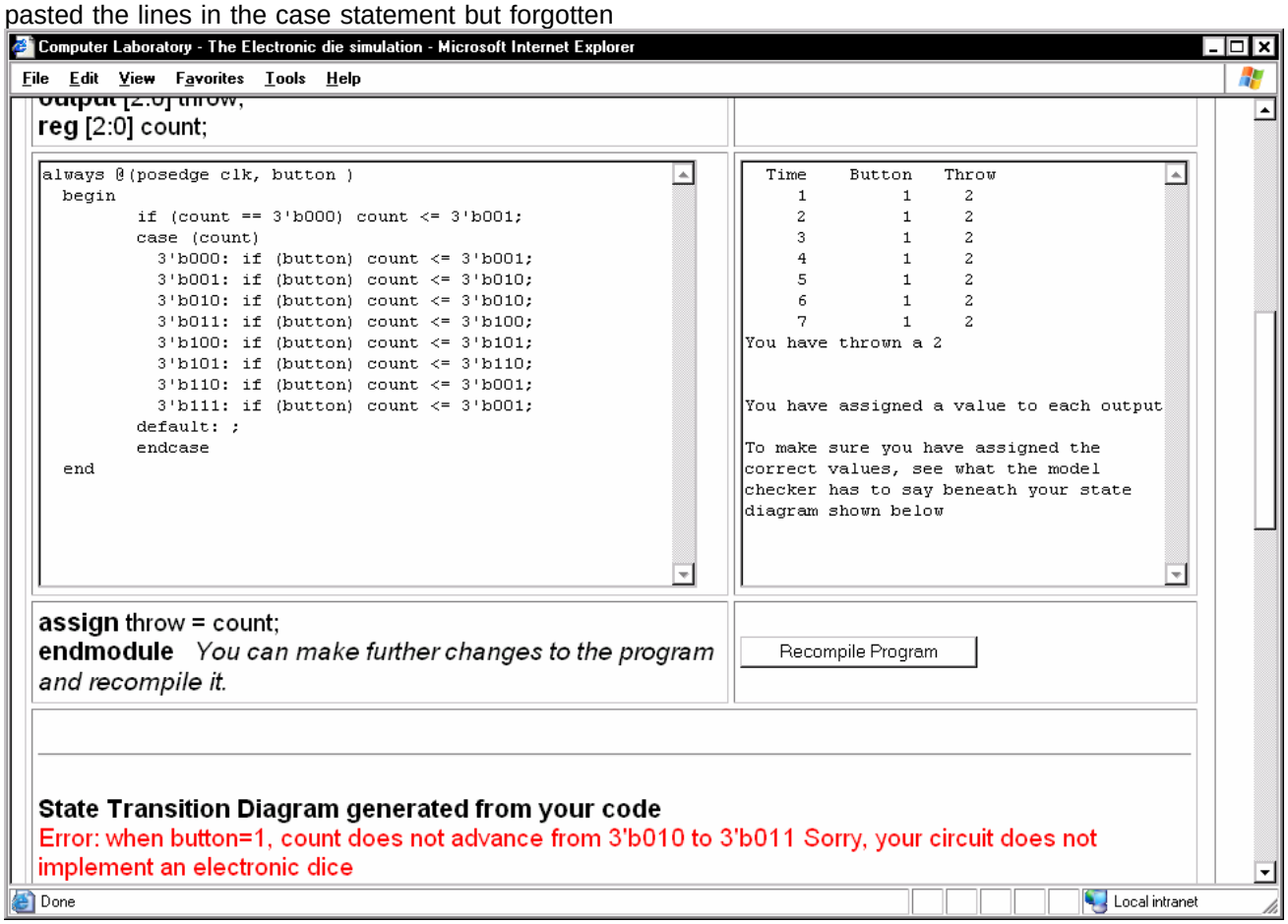

Figure 4 Identifying state machine errors

Another common error occurs when the student has forgotten to assign the calculated value to the output bit representing the amber traffic light. This generates an unconnected "z" value which prevents the student's code from working. However, there is no error in the state space.

Having both representations encourages the student to recognise that a trace table and a state transition diagram are different views on to the same concept.

Understanding which variables are needed to represent an algorithm and what processing is needed to achieve the necessary state is a skill the students have already started to learn with Java and ML.

5. RESULTS

5.1 Prototyping of IVC 1.0

At the time of writing, version 1.1 of the IVC has been evaluated by students who already knew the language from having taken the lecture course six months before. Their feedback has shaped the ideas above, and provided confirmation of the advantages of having all resources to hand to make the learning of the language as painless as possible.

5.2 Live use of IVC 2.0

Our research hypotheses are as follows: having used the IVC instead of attending four lectures.

1: Students do complete the tutorial before attempting the first laboratory exercise;
2: Students take less time to finish the first and second laboratory exercise;
4: Students answer examination questions better
5: More students attempt the examination questions

At the time of writing, we are able to provide results to 1 and 2 above. Achieving 2 is taken as a measure that the students have learned basic Verilog.

Only one student out of a class of eighty four chose not to use the IVC.

The following questions from the IVC questionnaire completed after its use in October 2005 show how the students reacted to the intelligent components of the IVC:

10. The intelligent compiler also provides links back to relevant parts of the tutorial to help solve syntax and usage queries. How did you use these links?

35% did not use them
11% read the BNF
14% answered the question from the generated text without following the links
40% read the generated tutorial page link

11 Once your code compiled, the intelligent compiler provides a text simulation and model checks your code to generate a state transition model. These next questions ask how useful these were. How did you spot the errors in your program?

9% used the model checker comments
47% used the state transition diagrams
29% used both the above
9% used the simulation and model checker
6% just looked at code again

Figure 5 below shows the time taken to complete the laboratory exercises (ticks) for 2004, without the IVC, and this year, with the IVC.

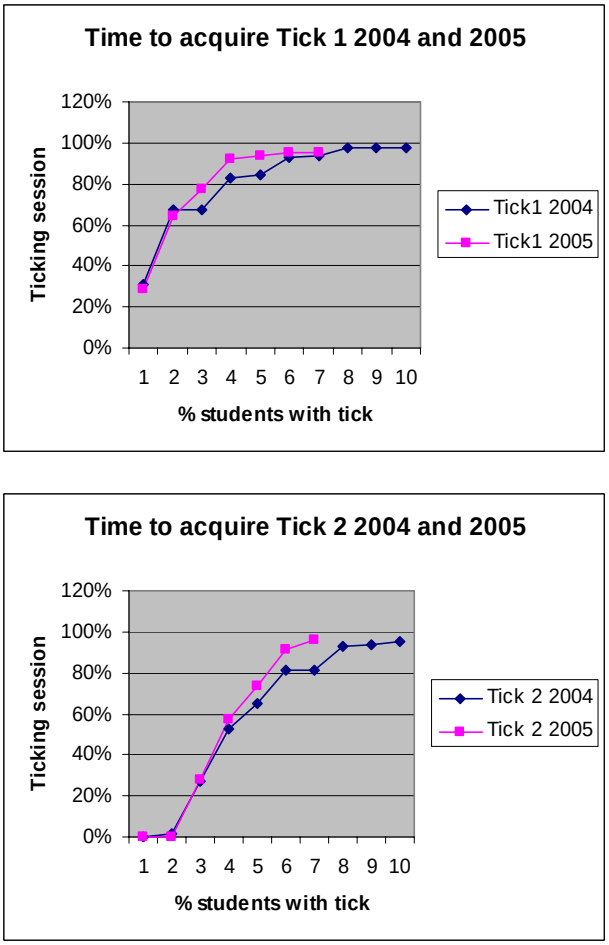

Figure 5 Tick Acquisition after IVC use

6. CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this paper we have demonstrated the advantages of the use of an English explanation, the language grammar, graded examples, feedback on student progress and the opportunity to try small programming examples in the protected environment of a context-sensitive compiler and model checker.

Over 85% of the students used the visual representation of the state machines. 65% of students used the context sensitive help, with 14% being able to answer their query merely by the structure of the generated text. This represents a good take-up of the two intelligent components of the IVC.

We have demonstrated a significant improvement in success in completing the laboratory exercises.

We have shown that adding an understanding of Verilog syntax and the semantics of simple programming exercises to a compiler can vastly improve the support given to novice Verilog programmers in a university setting.

6.2 Future work

We hope that examination results in the summer will also reflect the positive results from this term's survey. We hope to publish further papers in due course based on the quantitative and qualitative analysis of the students' progress with the IVC and the laboratory sessions.

The current IVC system has been built very much as a proof of concept. We wish to extend the framework to cover a broader range of intelligent context sensitive explanations of both syntax and semantic errors, based on the structure of the language and the problem space to be solved. We believe other semantic checks apart from the representation of state may be possible using concepts analogous to the invariants and pre- and post-conditions in conventional third generation languages.

We wish to be able to support different levels of user expertise to tailor what they read in the tutorial web to what they have already seen. User levelling and history capabilities would extend what further intelligence can be added to the intelligent complier and model checker. We also wish to generalise the

framework to allow a broader range of subjects to be taught.

7. ACKNOWLEGEMENTS

Many thanks are due to the Cambridge MIT Institute (CMI) for funding this work, and to Dr Williams at the Centre for Applied Research in Educational Technology (CARET) for assisting with evaluation.

Thanks are also due to the second-year students of the Computer Science course at Cambridge who have evaluated and used the IVC during its development over the last two years. Their enthusiasm and interest has been invaluable.

8. REFERENCES

[1] Peter McDonald, *The Nature of Algorithm Understanding: A Phenomenographic Investigation*. pmcd@cs.rmit.edu.au, School of Computer Science and Information Technology, RMIT University, Melbourne, Vic., Australia, 3000

[2] CS378 Stanford

http://www.stanford.edu/class/cs378/cs378-topics.html

[3] Mehanna *The pedagogies of e-Learning* Proceedings of the Networked Learning Conference 2004

[4] Billingsley W, Robinson P Towards an Intelligent On Line Textbook for Discrete Mathematics, University of Cambridge

[5] Johnson S *Lint, a C Program checker* Computer Science Technical Report 65 Bell Laboratories December 1977

[6] Icarus Verilog http://www.icarus.com/eda/verilog/

[7] SWI-Prolog

Http://www.swi-prolog.org