Towards Automating Simulation-Based Design Verification using ILP

Kerstin Eder, Peter Flach, and Hsiou-Wen Hsueh

Department of Computer Science, University of Bristol
MVB, Woodland Road, Bristol BS8 1UB, UK
Kerstin.Eder@bristol.ac.uk, Peter.Flach@bristol.ac.uk,
hsueh@cs.bris.ac.uk

Abstract. Increasing the productivity of simulation-based semiconductor design verification is one of the urgent challenges identified in the International Technology Roadmap for Semiconductors. The most difficult aspect is the generation of stimulus for functional coverage closure. This paper introduces a new Coverage-Directed test Generation (CDG) feedback loop which applies Inductive Logic Programming (ILP) to selected tests and coverage data to induce rules that can be used to automatically direct stimulus generation towards outstanding coverage. The case study documented in this paper shows a significant reduction of simulation time when ILP-based CDG is compared to random test generation. This is an exciting and promising new application area for ILP.

1 Introduction

ILP has been used to support scientific discovery and knowledge synthesis in a wide range of practical domains [19] such as protein structure prediction, mutagenicity prediction and pharmacophore discovery. Even the very process of scientific hypothesis generation and experimentation has been automated using ILP-based learning in a closed loop environment [12]. The main advantage of ILP over propositional learning is the expressive power resulting from a first-order representation. This allows learning results to be represented in a declarative format which is comprehensible to domain experts, without losing the ability to automatically process the learning results.

This paper aims to introduce the reader to a promising new application area for ILP, namely simulation-based semiconductor design verification, and demonstrates the potential that ILP has to offer in the context of functional coverage closure.

Verification of industrial designs still relies heavily on simulation; it can take up to 70% of the entire design effort [1]. Traditionally, design verification environments are based on a testbench [3] which is the code used to generate a valid input sequence to a design, called a test, drive this test into the design and then observe and check the design's response. Simulators are used to execute testbenches. The increasing complexity of real-world semiconductor designs makes exhaustive simulation prohibitive; in most cases the sun would burn out before even a fraction of the test cases can be simulated [20]. In reality, tight time-to-market constraints force verification engineers to be selective with respect to the tests they run to gain confidence in the functional correctness of a design. The verification plan specifies the scenarios that must be verified

before a design can be manufactured. It is the task of the verification engineers to create tests that fully cover these scenarios, often within a very short timeframe.

Recent advances in simulation-based verification have established coverage-driven verification methodologies which are essentially feedback loops that automate a large part of the simulation-based verification process. A pseudo-random stimulus generator at the front-end generates valid input stimulus according to a set of parameters or constraints, called directives, which bias test generation towards the scenarios of interest to verification. At the back-end a coverage analysis tool collects and analyses the coverage obtained from running these tests to check the effectiveness of the directives and to identify coverage closure targets such as rarely covered events as well as coverage holes. Coverage results are used to help engineers focus the next round of stimulus generation on these coverage targets.

Generating stimulus to increase functional coverage is a key challenge in simulation-based verification. Closing a functional coverage model is by no means trivial in complex industrial designs. For example, in state-of-the-art microprocessors, the subtle effects of issuing multiple instructions, out-of-order execution and aggressive pipelining can make it very difficult if not impossible to see what sequence of instructions to drive in order to reach a specific functional coverage scenario (coverage task). This is particularly difficult when the signals involved in the specification of the functional coverage task are related to micro-architectural features of the design.

In practice up to 90% of coverage tasks can be reached via *biased pseudo-random tests* which are automatically generated based on a set of user-defined directives. However, even supplying the directives requires significant engineering skill and is often only accomplished through many trial-and-error runs. At the end engineers resort to writing *directed tests* by hand aiming to cover the missing cases. Consequently as much as 90% of a verification team's time and resources can be spent on closing the remaining 10% coverage manually. Figure 1 shows the typical long flat-tailed curve when plotting the coverage rate achieved by random simulation (y-axis) against the number of simulation runs (x-axis). The data for this figure originates from our case study and is representative for many industrial verification projects. It clearly shows that the number of simulations necessary to obtain the last few coverage tasks is excessive in comparison to the number of simulations needed to get most of the coverage. This is one reason why verification has become the dominant cost in the design process and many verification projects run over time and budget. Verification, if not done properly, can cost a company its reputation and potentially put people at serious risk. If it takes too long the product will miss its market window which results in loss or significant decrease of the market share (and hence profits). The latest version of the International Technology Roadmap for Semiconductors [1] calls for more *automation* in the process of functional coverage closure to reach verification targets faster and with less engineering effort.

The most demanding aspect regarding full automation of the existing coverage-driven feedback loops is the *automatic* generation of the directives for functional coverage closure. Coverage-directed stimulus generation (CDG) techniques [18] aim to achieve exactly this. Feedback-based CDG integrates machine learning into the feedback loop (which is depicted in Fig.2) in order to automatically generate new directives that bias stimulus generation towards producing tests which target specific coverage tasks. Ma-

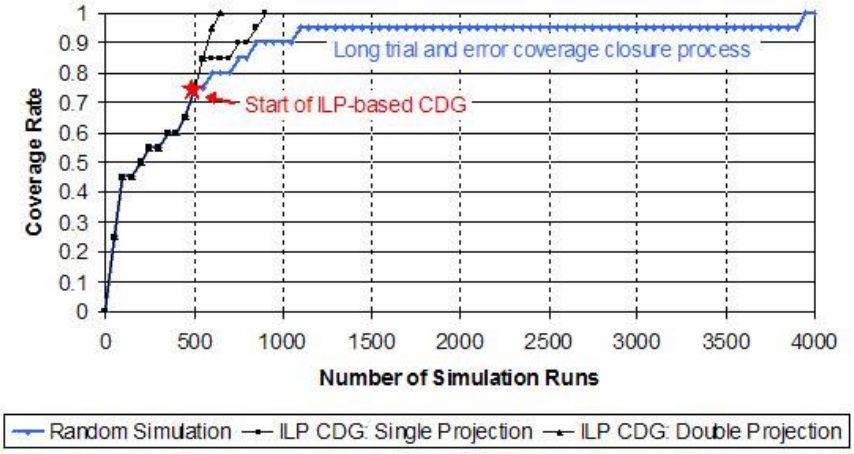

Fig. 1. Coverage progress for random simulation compared to ILP-based CDG

chine learning techniques employed in this context include Bayesian networks, evolutionary techniques such as genetic algorithms and genetic programming as well as Markov chains. The underlying assumption is that the learning mechanism can identify, from existing tests and coverage, how best to bias stimulus generation such that the resulting tests can reach outstanding coverage tasks. As a result, the curve in Fig.1 should climb significantly faster than random simulation thus saving a large number of simulation runs and hence verification effort.

In contrast to other machine learning applications where the measure of success is achieving a very high accuracy of the learning output resulting in a large lift when comparing system performance with and without learning, this application is slightly different in that the number of examples to learn from is variable and depends on when the learning is kicked off during simulation. From a machine learning viewpoint, the later in the simulation phase learning is started the more examples are available, hence a higher accuracy can be expected. Conversely, the earlier learning is started the fewer examples are available, resulting in a lower accuracy. From a verification viewpoint, however, the earlier the curve starts to *climb faster than random simulation* the more verification effort can be saved. [1] These two conflicting interests need to be traded off carefully with the verification interests dominating in this context. For example, the lift achieved in our case study, although in machine learning terms not impressive, was good enough to save a significant number of simulations as shown in the two steeper curves in Fig.1.

This paper introduces a novel CDG technique based on an inductive machine learning method that discovers relational information from structured data. Inductive Logic

[1] Finding the best starting point for learning is a challenging optimisation problem which requires further research. A second experiment in which learning commenced after 400 simulations produced curves that climbed much slower than the two steep ILP-based CDG curves in Fig.1 (but still faster than random simulation).

Programming (ILP) is applied to tests and their related coverage in order to induce general rules which describe the characteristics of these tests. The resulting rules can be used directly as directives, to obtain tests that are structurally similar to the examples presented to the learning system. Coverage closure can be automated by applying rule learning to clusters of a target coverage task and combining the resulting rules to obtain directives for test generation. As the tests and associated coverage are supplied to the ILP system in a declarative representation, the induced rules are also declarative and in principle human readable. This gives engineers an insight into the knowledge discovered by the ILP system and is also an excellent basis for automatic translation of these rules into test generation directives.

A case study demonstrates the fundamental principles of ILP-based CDG in two steps. The first step evaluates the consistency and reliability of the generated directives for existing coverage in a rediscovery experiment. The second step documents the results of the application of a novel cluster-based coverage closure method.

This paper is organised as follows. Section 2 reviews coverage models and existing CDG approaches. Section 3 introduces the fundamental principles of ILP-based directive generation. The experimental framework and results of our case study are presented in Section 4. Further research and conclusions are discussed in Section 5.

2 Background

This section reviews coverage models and existing CDG approaches.

2.1 Coverage Models

To measure the coverage of simulation test suites, coverage models are generally classified into *structural* and *functional*. Structural coverage is focused on measuring which parts of the design source code have been exercised during simulation at various levels of detail ranging from statement down to expression coverage. Structural coverage helps verification engineers to see which code parts have not been verified. It is inherently weak, however, in determining whether the design is functionally correct.

The most tricky bugs to find often reside in functional *corner cases* of the design which involve multi-cycle scenarios and high degrees of concurrency. To ensure these are covered during verification, experienced engineers define functional coverage models based on the specification, the design and often also the implementation. This makes functional coverage models inherently user-defined and application-specific [22]. Designing meaningful functional coverage models requires significant design knowledge, experience and engineering skill.

One way of specifying a functional coverage model is outlined in [13]. Their models contain a semantic description (story) detailing the purpose of the verification task, the list of attributes mentioned in the story, the set of all possible values (domain) for each attribute and a list of restrictions on the permitted combinations in the Cartesian product, or cross-product, of the attribute domains. The overall size of the coverage space associated with such a functional coverage model is the product of all domain cardinalities. The elements in the cross-product of the attribute domains are referred to as *coverage tasks*. Each coverage task can be represented as an n-tuple of values

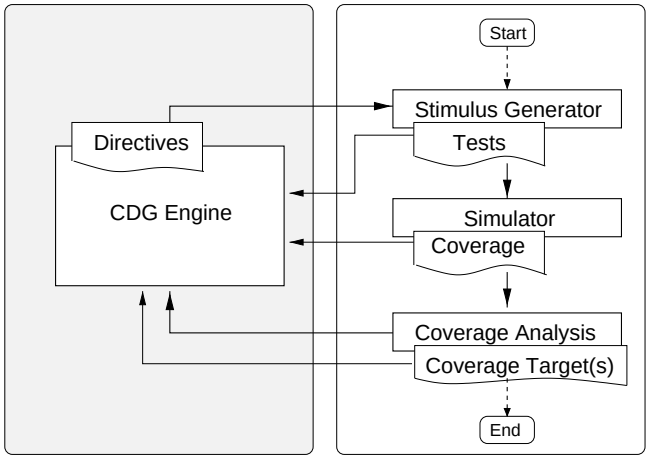

Fig. 2. CDG feedback loop

from the attribute domains, where n is the number of attributes in the coverage model. The restrictions identify which coverage tasks are legal and hence need to be covered during verification. Uncovered legal tasks are referred to as *coverage holes*. The process of constructing tests to cover a hole is called *coverage closure*. In this paper we mean cross-product based coverage models when referring to functional coverage.

2.2 Feedback-based Coverage-Directed test Generation

Coverage-Directed test Generation (CDG) is a technique that aims to automate the generation of simulation stimulus based on coverage information [18]. There are two main approaches towards CDG: one is by construction using formal methods and the other is based on feedback. The approach introduced here can be classified as a novel feedback-based CDG technique with a CDG engine that embeds Inductive Logic Programming.

The feedback-based CDG framework [8] is shown in Fig.2. It is built around a state-of-the-art testbench automation environment that contains a stimulus generator, a simulator and coverage analysis. Coverage targets are identified and the coverage analysis results are then fed into the CDG engine together with existing tests and coverage data to generate directives that bias the random stimulus generator towards achieving the target coverage tasks. The CDG engine can be realized with different techniques.

Early approaches [21] focused mainly on genetic algorithms (GAs) that learn specific test cases, such as sequences of assembly code instructions, directly. This required an explicit encoding of the target instructions within the representation on which the GAs worked. Results were application-specific and lacked generality. In [4] another GA for automatic bias generation is presented. This approach generates biases for an industrial instruction stream generator. Its main drawbacks are the architecture-specific encoding of the representation on which the GA works. The approach has also been transferred to a hierarchical test generation framework to target statement and path coverage [24].

A more flexible Genetic Programming (GP) technique that generates machine code test programs for design verification, called μGP, was developed in [6]. It directs test

generation towards maximising code coverage with the goal to generate a set of test programs that achieves maximum statement coverage. The test sets generated with μGP are smaller and yield higher statement coverage than randomly generated tests [7]. The approach requires a syntactical description of the microprocessor's assembly language in the form of an instruction library. The internal representation on which the μGP core works is generic and test programs can be generated for any given instruction library. The main limitation is that the approach only targets structural, *i.e.* code-based, coverage models, rather than functional coverage models based on cross-products, which in practice are far more difficult to close.

In [9, 5] a coverage-directed test generation approach based on Bayesian networks is presented. It models the relationship between the test directives and coverage tasks via a Bayesian network, where general knowledge regarding the design's operation taken from a domain expert is encoded in the network structure. This approach targets cross-product functional coverage models and has resulted in a significantly improved coverage rate achieved in a shorter time frame. An advantage of the approach is that it can discover diverse directives that all target the same coverage task. However, the design and training of an appropriate Bayesian network is required; this includes the identification of the network structure that models the joint probability distribution based on the directives to the test generator, the coverage model, as well as expert domain knowledge. In practice, very few if any verification engineers have these skills.

An approach based on Markov models that contain user-specified templates for instruction sequence generation has been developed in [23]. The Markov model's parameters are adjusted to settings that stress certain activities of interest to verification through an iterative design-activity directed feedback loop. A simple Markov model was extended by introducing a cache and some dependency variables to propagate directive dependencies further than one step. This approach approximates the correlation of directive parameters over several instructions. However, it is weak in controlling the actual distance of dependency.

In summary, although existing feedback-based CDG approaches achieve promising results, they have shortcomings which have so far prevented them from being widely used for functional coverage closure. The major limiting factors are the requirement of specialised encodings or models on which the algorithms work, the need for non-verification expertise to set up and maintain the environment, or the use of coverage models other than functional coverage.

3 ILP-based Coverage-Directed Stimulus Generation

The directive generation approach introduced here differs from existing feedback-based CDG approaches by its use of inductive learning from examples, in particular Inductive Logic Programming (ILP), in the CDG engine. ILP [14] is a declarative inductive learning method. It requires a set of factual examples E and some relational background knowledge B. ILP will find a single (or multiple) hypothesis H in terms of the relations given in B such that (ideally) every positive example in E is covered by H and no negative example in E is covered. In this context B, E and H are represented as definite

logic programs [15]. The next section provides a more detailed introduction regarding the application of ILP within a CDG framework.

3.1 Method to Learn Rules for Test Generation

To learn rules suitable for test generation, the examples E for ILP learning are the tests, which are initially randomly generated, together with their respective coverage data. Using a first-order logic concept description language each test can automatically be translated into a relational representation of a sequence of instructions together with a test identifier and the coverage data associated with this test by analogy with the encoding of Michalski-style trains in [16]. The background knowledge B describes the general structure of these tests and relationships between test components such as register use, re-occurrence of registers, *e.g.* as destination or source, specific sequences of instructions, relative distance between dependencies, instruction classes etc. The learning task is to find hypotheses H which represent general rules describing the characteristics of a test to target a given coverage task. Provided there are enough instances to learn from, at the end of the learning process the ILP system returns a set of rules containing at least one rule for each coverage task presented to the system.

The learning task described above produces a set of rules which give rise to directives that can be used to generate tests structurally similar to the original examples.[2] These tests achieve the same amount of coverage as was originally obtained, but with increased accuracy, a smaller number of tests and hence far fewer simulation runs than with biased-random generation alone. [3] In addition, the rules give insight into the structure of the existing test suite and can thus be used to analyse test diversity.

To construct tests that reach coverage holes the learning task needs to be changed to find rules for *coverage clusters* which share a degree of similarity with the target coverage hole. The underlying assumption here is that the *directive* to target the coverage hole shares a degree of similarity with the directives used to approach similar coverage tasks. Learning is most effective when the selected clusters have a high coverage rate. Existing coverage data clustering techniques which can in principle be applied for ILP-based coverage closure are discussed in [13]. The rules returned by the learning system for each cluster are then combined to form a directive to target the coverage hole. This technique can also be used to generate tests that increase the coverage rate of rarely covered tasks via a different execution path compared to existing tests.

3.2 Integrating ILP into the CDG Framework

The entire CDG feedback loop with ILP-based learning is shown in Fig.3. Input to

[2] Note that the learning output is not, as in traditional machine learning applications, used for the *classification* of tests into those which do or don't reach a given coverage task, but instead for the automatic *construction* of tests to target a given coverage task, which is achieved by generating tests that satisfy the constraints imposed by the rules in the learning output.

[3] From discussions with engineers we learned that this can already be valuable in practice, *e.g.* for rediscovering directives when subtle changes to the design turn out to have a major effect on the coverage of tests, often rendering existing test suites completely invalid. Because these

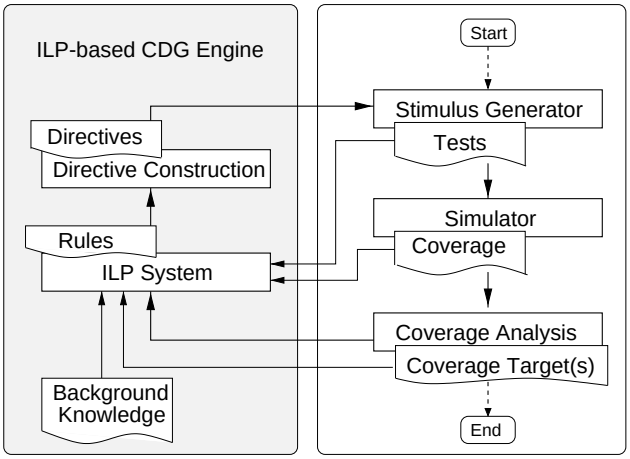

Fig. 3. CDG feedback loop with ILP-based CDG engine

the ILP system, such as tests and associated coverage as well as coverage targets, is sourced directly from the data existing within the standard verification flow. In practice, the background knowledge can be provided in pre-defined application-class specific libraries. Alternatively, to make background knowledge acquisition more flexible and user-friendly, a domain-specific template-based declarative input language could be designed for verification engineers, which is automatically translated into the first-order logic representation used for ILP-based learning. Defining the background knowledge requires as much expertise as is necessary to efficiently use the features of a state-of-the-art test generation environment such as [2].

The ILP system returns a set of rules for selected coverage tasks or clusters. The final directive construction stage combines these rules to obtain directives that target the coverage holes or rarely covered tasks selected for closure.

4 Preliminary Study

This section demonstrates the key aspects of the ILP-based CDG technique on an example microprocessor Design Under Verification (DUV).

4.1 Experimental Setup

The DUV is a five-stage pipelined Superscalar DLX [11] with four independent execution units: Branch Resolve Unit (BRU), Arithmetic Logic Unit (ALU), Multiply Divide Unit (MDU), and Load Store Unit (LSU). At the entrance of each execution unit, *i.e.* between the decode and the execution stage, is a buffer pipeline register, called Reservation Station, which is used when the data for an instruction is not yet available to

test suites have been generated with an iterative adjustment to the directives, and the history has been lost, automatically rediscovering the directives would save engineering effort.

enable the processor to fetch the next instruction. The processor uses a Reorder Buffer, which is a ring buffer with five entries, to ensure in-order-termination of instructions.

The functional coverage model evaluates the utilisation of the reservation station of the Superscalar DLX in conjunction with data dependencies between the instruction waiting in the reservation station and the instruction in the reorder buffer that provides the data. In particular this model monitors that the reservation station for each pipeline unit is used by an instruction that is waiting for data on either one of its source registers, and this data is provided by an instruction that occupies any one of the entries in the reorder buffer. Hence the coverage model consists of the following three attributes which are listed together with the set of all possible values for each attribute: Pipeline Unit (PU) of the utilised reservation station which can take values from {alu, mdu, lsu, bru}, Source Register Location (SRL) which can be one of {rs1, rs2} and Reorder Buffer Location (RBL) which can be from {0, 1, 2, 3, 4}. The full size of this coverage space is 4 x 2 x 5 = 40 coverage tasks. However, constraints imposed by the instruction set architecture and the implementation result in a reduction of the coverage space to the 20 legal coverage tasks shown in the first column of Table 1.

The ILP System used in this experiment is Progol [17]. Test programs together with their coverage are translated by an automatic procedure into the logic programming language Prolog on which Progol works. This translation is based on the three types `task`, `test` and `instr` which define valid instances of identifiers denoting coverage tasks, tests, *i.e.* instruction sequences, and instructions, as well as a fixed set of mnemonics denoting opcodes and a set of register identifiers. The sequence of instructions is then described using the following set of relations:

```
cover(task,test_id).
has_instruction(test_id,instr_id).
is_followed_by(instr_id,instr_id).
instr_has_opcode(instr_id,mnemonic).
instr_has_rs1(instr_id,reg).
instr_has_rs2(instr_id,reg).
instr_has_rd(instr_id,reg).
```

A translation of a test program fragment that contains three instructions and covers task (alu,rs1,3) is given in Fig.4.

The background knowledge provides Progol with an important aid in the learning process because Progol searches for hypotheses by generalising an example in terms of the background relations. An example of such a relation is given below. The relation `same_rd_rs1_d1(I1,I2)` specifies under which conditions the destination register of an instruction `I1` is being reused as first source register by the instruction `I2` which immediately, *i.e.* in distance `d1`, follows `I1`.

```
same_rd_rs1_d1(I1,I2) :-
   is_followed_by(I1,I2),
   instr_has_rd(I1,R),
   instr_has_rs1(I2,R).
```

A set of 161 relations similar to the one above has been used as background knowledge for the experiments.

Because Progol uses mode-directed inverse entailment to guide the process of generalisation from examples, a set of mode declarations needs to be provided for the relations which are used in the learning process. The mode declarations constrain the

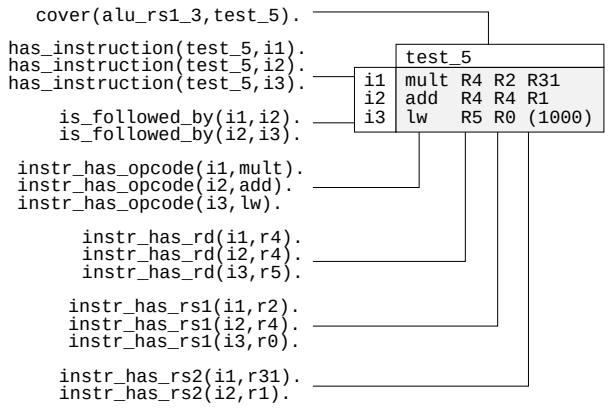

Fig. 4. Test program fragment and corresponding Prolog representation

search, and essentially establish a structural template for the output rules. A total number of 156 mode declarations constrained the ILP search space in the two experiments carried out. Representative examples are given below.

```
modeh: cover(#task,+test)
modeb: has_instruction(+test,-instr)
modeb: has_opcode(+instr,#opcode)
modeb: same_rd_rs1_d1(+instr,-instr)
modeb: same_rd_rs2_d1(+instr,-instr)
```

4.2 Rules for Existing Tests and Coverage

The first part of the experiment aims to show that, given a set of pseudo-randomly generated tests together with their coverage, and under the assumption that there are enough tests to learn from for each coverage task, it is possible to induce rules that correctly characterise the features of tests to target the achieved coverage. Successful completion of this experiment confirms the correctness of a fundamental principle of ILP-based CDG. It also validates the actual ILP setup including the data encoding and shows whether the background knowledge is fit for purpose. This initial step can be compared to the rediscovery step described in [10].

The learning was started with the test data available after 500 simulation runs, when the pseudo-randomly generated tests covered 15 of the 20 coverage tasks which equates to an overall coverage rate of 75%. Only 57 out of the 500 randomly generated tests were successful in adding to coverage. The number of successful tests for each coverage task is given in the second column of Table 1. The total number of induced rules after ILP learning is summed up in the third column.

To give the reader an example of the learning output, below is one rule which shows the characteristics of tests that reached coverage task (alu,rs1,2).

```
cover(alu_rs1_2,Test_ID) :-
   has_instruction(Test_ID,I1),
   has_instruction(Test_ID,I2),
   instr_has_opcode(I2,alu),
   same_rd_rs1_d1(I1,I2).
```

Note that due to the declarative nature of ILP, the above rule can easily be translated into natural language: "*The test must contain an instruction with an opcode of alu type, and the destination register of the instruction preceding the alu type instruction is used as the first source register by the alu type instruction.*"

The rules obtained from ILP-learning were then used directly as directives for test generation, *i.e.* the test generator was given the task to generate a sequence of instructions that satisfied the constraints contained in the rule body. To evaluate the accuracy of the resulting tests, the average number of successful tests generated either pseudo-randomly or on the basis of the ILP induced rules was compared. The fourth and fifth column of Table 1 contain these numbers which are now termed the *hit rate*. For example, the hit rate of the ILP-based directive for the coverage task (alu,rs1,0) given in Table 1 is 15%, which means on average 15 out of 100 tests generated to satisfy the rule body also covered the task in the rule head. In comparison, when the tests are generated without specific constraints on average 14 out of 1000 tests reached that coverage task. From the results we computed the lift[4] which is shown in the last column of Table 1.

The results show a significant lift (of more than 8) for seven out of the twelve coverage tasks for which rule learning was successful. No rules were generated for (mdu,rs1,0), (mdu,rs1,2), and (mdu,rs1,3), because rule learning is a generalisation process which only works well when there are two or more instances to learn from. For the two coverage tasks (alu,rs2,2) and (mdu,rs1,4) tests generated from the directives obtained after rule learning perform worse. The tests used as examples for learning these two cases show orthogonal aspects in that they represent completely different approaches to reach the same coverage task. For this reason the ILP system did not find a rule that accurately generalised the given tests. The problems encountered can be resolved in practice by increasing the number of examples to learn from. In the majority of cases, however, the results show that the ILP system has successfully generalised the patterns in the existing tests and that test generation from the induced rules gives a higher hit rate.

4.3 Rules for Conceptually New Tests

The second part of the experiment aims to generate directives that target the remaining five coverage holes by learning from related coverage clusters. A simple syntax-based clustering technique which is easily automated is projection [13]. Given a target tuple (coverage task) in a coverage space, projection replaces one or more of the attribute values in the target tuple by a wildcard. In this experiment two types of projection were used. Single projection aggregates successful tests from one dimension of the coverage task, leaving two values in the tuple, *e.g.* (alu,rs1,*). Similarly, double projection aggregates tests from two dimensions, leaving only one value in each tuple, *e.g.* (alu,*,*).

[4] The lift was computed as the ratio of the hit rate of tests generated based on the ILP induced directives over the hit rate of tests generated pseudo-randomly.

Coverage Task (PU,SRL,RBL)	Covered by Tests	Number of Rules	Random Hit Rate	ILP Hit Rate	Lift
(alu,rs2,4)	2	2	0.4%	7%	17.5
(alu,rs2,1)	4	4	0.8%	13%	16.25
(alu,rs2,3)	4	3	0.8%	13%	16.25
(alu,rs1,1)	4	2	0.8%	12%	15.00
(alu,rs1,4)	7	5	1.4%	17%	12.14
(alu,rs1,0)	7	4	1.4%	15%	10.71
(alu,rs1,2)	8	4	1.6%	13%	8.13
(alu,rs2,0)	3	1	0.6%	1%	1.67
(alu,rs1,3)	8	5	1.6%	2%	1.25
(bru,rs1,2)	2	1	0.4%	0.4%	1
(mdu,rs1,4)	2	1	0.4%	0.1%	0.25
(alu,rs2,2)	3	2	0.6%	0.1%	0.17
(mdu,rs1,0)	1	-	0.2%	-	-
(mdu,rs1,2)	1	-	0.2%	-	-
(mdu,rs1,3)	1	-	0.2%	-	-
(mdu,rs1,1)	0	-	0%	-	-
(bru,rs1,0)	0	-	0%	-	-
(bru,rs1,1)	0	-	0%	-	-
(bru,rs1,3)	0	-	0%	-	-
(bru,rs1,4)	0	-	0%	-	-
Total	57	34	-	-	-

Table 1. Coverage and learning results after 500 simulation runs

The learning was again started with the test data available after 500 simulation runs. For each coverage hole a set of rules was collected from first applying single and then double projection to the data before learning was performed. Test generation directives were then constructed manually from these sets of rules by simply conjunctively combining rule bodies and resolving conflicts via the introduction of disjunctions which are interpreted as random choice during test generation. A remaining challenge is to fully automate this process of directive construction.

The same method was applied to the five rarely covered tasks from the first part of the experiment to see whether coverage could be increased. Table 2 shows the results obtained after test generation from the so constructed directives for the five coverage holes in the upper half and the five rarely covered tasks in the lower half.

It is encouraging to see that all hit rates have increased significantly for both the coverage holes and the previously rarely covered tasks. Interestingly, in this experiment double projection performs better than single projection for coverage hole closure. This might indicate that, as more dimensions are projected out, more instances are supplied to the ILP system to learn from, which in turn induces rules that give rise to directives with higher accuracy. On the other hand single projection outperforms double projection for the previously rarely covered tasks. This might indicate that when tests that reach the target coverage tasks do exist, double projection introduces more noise into the rules than single projection. Further research is needed to better understand these results.

Coverage Task (PU,SRL,RBL)	Random Hit Rate after 500 Random Tests	5000 Random Tests	ILP Hit Rate with Single Projection	Lift	ILP Hit Rate with Double Projection	Lift
(mdu,rs1,1)	-	0.32%	1.00%	3.13	4.13%	12.91
(bru,rs1,0)	-	0.08%	1.00%	12.50	2.38%	29.75
(bru,rs1,1)	-	0.18%	1.00%	5.56	3.63%	20.17
(bru,rs1,3)	-	0.22%	4.00%	18.18	6.38%	29.00
(bru,rs1,4)	-	0.06%	0.50%	8.33	1.86%	31.00
(alu,rs2,2)	0.60%	0.56%	10.00%	17.86	8.33%	14.88
(mdu,rs1,0)	0.20%	0.16%	2.00%	12.50	1.50%	9.38
(mdu,rs1,2)	0.20%	0.20%	9.00%	45.00	5.00%	25
(mdu,rs1,3)	0.20%	0.28%	1.67%	5.96	6.75%	24.1
(mdu,rs1,4)	0.40%	0.30%	2.00%	6.67	2.50%	8.33

Table 2. Results for coverage holes (top half) and rarely covered tasks (bottom half)

Figure 1 from the Introduction section compares the coverage progress of this experiment to random simulation. It shows that test generation from single and double projection methods reached full coverage in 367 and 108 simulations respectively (after the initial 500 pseudo-randomly generated tests). In total, test generation from single and double projection methods used 867 and 608 simulations to reach full coverage, compared to 3914 simulations based on pseudo-random test generation alone. In this case study, the ILP learning started from the 57 successful tests collected in the first 500 simulations (based on pseudo-randomly generated test). This was sufficient to reach full coverage within a maximum of another 400 tests to simulate.

To open this interesting application up for the ILP community the 500 randomly generated tests used in the experiment and the resulting coverage have been made available on `http://www.cs.bris.ac.uk/~eder/ILP_CDG/`. This site also contains further information on the encoding of these tests and the Progol setup including background knowledge and mode declarations.

To repeat the entire experiment the complete feedback loop is needed. It includes the DUV as well as the test generator, the simulator and the coverage analysis component. Except for the DUV, which we obtained using references in [11], these are commercial products which require licenses which some of the EDA vendors offer via their higher education programmes. More information on the setup of the feedback loop used here can be obtained by contacting the authors.

5 Conclusions

This paper shows how ILP can be applied in the context of functional coverage closure as part of the CDG engine in a standard CDG feedback loop. The strength and promise of ILP-based CDG have been demonstrated in a two part experiment. In the first part rules to be used as test generation directives have been induced from existing tests and coverage. Test generation from these rules achieved a higher hit rate on their target coverage tasks than was possible with random test generation. The second part of the

experiment presents a coverage closure methodology, whereby learning is applied to selected projections of a coverage hole and learning output is then combined to obtain a directive that targets this coverage hole.

Clearly, the example case study is small and a larger industry-based trial will be undertaken shortly on a realistic-sized processor. However, the results obtained provide an interesting and encouraging starting point for further work to establish ILP-based CDG alongside existing learning-based techniques.

Various aspects of the ILP-based CDG methodology would benefit from further research. With a focus on learning, one research direction is to explore the use of clustering methods that are more sophisticated than purely syntactic projection for ILP-based learning. It is anticipated that ILP techniques can be applied to identify semantically meaningful coverage clusters. The development of a kernel-based method to define a meaningful distance metric in this context is one of our next research goals. Another research task is to establish a user-friendly background acquisition methodology for ILP-based CDG. This is a key requirement for acceptance of this methodology in practice. We also intend to experiment with ILP systems that use a more descriptive induction approach compared to Progol, which in practice is mostly used for classification tasks.

In summary, this paper pioneers a methodology that makes CDG an exciting new application area for ILP. Although there is more work to be done before ILP-based CDG is mature enough to be integrated into practical verification environments, it is clear that ILP-based CDG has important advantages compared to other learning based CDG techniques. First, it seamlessly integrates into an existing verification flow without the need for encodings or models that are outside the expertise of a professional verification engineer; the learning input can be sourced from existing tests and coverage data directly via automatic translation procedures. Second, due to the declarative representation of data, ILP-based CDG requires intuitive input from the verification engineer at setup (for the background knowledge) and no non-verification expertise is needed. Application-specific libraries to cover the background knowledge (and the respective mode declarations if Progol is used) can in principle be provided to reduce the engineering input even further. Third, ILP-based CDG is fully automatic and can target user-defined functional coverage models. In addition, the transparency of the resulting directive rules, which are declarative and hence intuitively human readable, gives verification engineers an insight into the knowledge gained. This is one of the key strengths of ILP compared to other learning methods and gives ILP-based CDG a strong competitive edge.

References

1. International Technology Roadmap for Semiconductors, Design Chapter, 2005 Edition. Available at `http://public.itrs.net/`.
2. A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv. Genesys-Pro: Innovations in test program generation for functional processor verification. *IEEE Design & Test of Computers*, 21(2):84–93, March-April 2004.
3. J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2nd edition, 2003.
4. M. Bose, J. Shin, E. M. Rudnick, T. Dukes, and M. Abadir. A genetic approach to automatic bias generation for biased random instruction generation. In *CEC2001: Congress on Evolutionary Computing*, pages 442–448, May 2001.

5. M. Braun, S. Fine, and A. Ziv. Enhancing the Efficiency of Bayesian Network Based Coverage Directed Test Generation. In *IEEE International High-Level Validation and Test Workshop (HLDVT)*, 2004.
6. F. Corno, G. Cumani, M. S. Reorda, and G. Squillero. Evolutionary test program induction for microprocessor design verification. In *ATS2002: IEEE Asian Test Symposium*, pages 368–373, Guam (USA), November 2002.
7. F. Corno, E.Sanchez, M. S. Reorda, and G. Squillero. Automatic Test Program Generation: A Case Study. *IEEE Design & Test of Computers*, 21(2):102–109, March-April 2004.
8. S. Fine, M. Levinger, and A. Ziv. Apparatus and method for coverage directed test. Patent Number: US2004249618, 09 December 2004. IBM (US).
9. S. Fine and A. Ziv. Coverage directed test generation for functional verification using Bayesian networks. In *DAC2003: 40th Design Automation Conference*, pages 286–291, Anaheim, California (USA), June 2003.
10. P. Finn, S. Muggleton, D. Page, and A. Srinivasan. Pharmacophore Discovery using the Inductive Logic Programming System PROGOL. *Machine Learning*, 30:241–273, 1998.
11. J. Horch. Entwurf eines RISC-Prozessors in der Hardwarebeschreibungssprache VHDL. Technical report, Technische Universitaet Darmstadt Institut fuer Datentechnik, 1997.
12. R. D. King, K. E. Whelan, F. M. Jones, P. G. K. Reiser, C. H. Bryant, S. H. Muggleton, D. B. Kell, and S. G. Oliver. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature (letters to nature)*, 427:247–251, January 2004.
13. O. Lachish, E. Marcus, S. Ur, and A. Ziv. Hole analysis for functional coverage data. In *DAC2002: 39th Design Automation Conference*, New Orleans, Louisiana, USA, June 2002.
14. N. Lavrac and S. Dzeroski. *Inductive Logic Programming. Techniques and Applications*. Ellis Horwood, New York, 1994.
15. J. W. Lloyd. *Foundations of Logic Programming.* Springer, second edition, 1987.
16. D. Michie, S. Muggleton, D. Page, and A. Srinivasan. To the international computing community: A new East-West challenge. Technical report, Oxford University Computing laboratory, Oxford,UK, 1994.
17. S. Muggleton. Inverse Entailment and Progol. *New Generation Computing*, 13(3-4):245–286, 1995.
18. G. Nativ, S. Mittermaier, S. Ur, and A. Ziv. Cost evaluation of coverage directed test generation for the IBM mainframe. In *ITC2001: International Test Conference*, pages 793–802, October 2001.
19. D. Page and A. Srinivasan. ILP: A Short Look Back and a Longer Look Forward. *Journal of Machine Learning Research*, 4:415–430, 2003.
20. C.-J. H. Seger. An introduction to formal verification. Technical Report 92-13, UBC, Department of Computer Science, Vancouver, B.C., Canada, June 1992.
21. J. Smith, M. Bartley, and T. Fogarty. Microprocessor design verification by two-phase evolution of variable length tests. In *Proceedings of the 1997 IEEE International Conference on Evolutionary Computing*, pages 453–458. IEEE Press, 1997.
22. S. Ur and A. Ziv. Off-the-shelf vs. custom-made coverage models, which is the one for you? In *Proceedings of the 7th International Conference on Software Testing, Analysis and Review (STAR)*, May 1998.
23. I. Wagner, V. Bertacco, and T. Austin. StressTest: An automatic approach to test generation via activity monitors. In *DAC2005: 42nd Design Automation Conference*, pages 783–788, Anaheim, California (USA), June 2005.
24. X. Yu, A. Fin, F. Fummi, and E. M. Rudnick. A Genetic Testing Framework for Digital Integrated Circuits. In *Proceedings of the International Conference on Tools with Artificial Intelligence (ICTAI)*, page 521ff. IEEE, 2002.