JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Random Test Case Generation and Delta Debugging for Bit-Vector Logic with Arrays

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Masterstudium

INFORMATIK

Eingereicht von:
Andreas Vida Bakk. techn., 0155987

Angefertigt am:
Institut für Formale Modelle und Verifikation

Betreuung:
Prof. Armin Biere

Mitbetreuung:
Dipl.-Ing. Robert Brummayer

Linz, Oktober 2008

Abstract

Random test case generation paired with Delta Debugging provides the programmer with an efficient way of finding and fixing code defects during software development. In this work existing and new Delta Debugging algorithms are applied to BTOR files. BTOR is the input format of Boolector, an SMT solver for bit-vector logic with arrays. To improve the quality of Boolector we developed a random DAG (directed acyclic graph) generator for the BTOR format. When the generator finds a failure inducing input, this input can be minimized with little effort using one of our implemented Delta Debugging algorithms. The defect in the source code can then be located by using the small input file in a debugging session. The input file can be appended to the set of regression tests.

This text describes a new algorithm for generating random DAGs. The algorithm has several input parameters. The most important one (maximum node count) affects the size of the generated files. In an experiment we varied this parameter to find out which setting must be used in order to find the highest portion of failure inducing inputs while spending as little running time as possible in the Boolector executable.

In another experiment we show how tweaking the algorithm's parameters influences the running time of Boolector and the satisfiability of the resulting formulas.

We show how existing Delta Debugging algorithms must be modified in order to be applied to BTOR files. After experimenting with the algorithms from the literature we discovered that an improvement is necessary that considers BTOR's specific DAG structure. We propose two new algorithms which handle cases better when failure inducing parts of the DAG are located in great depths and where the failure inducing part is very small compared to the entire size of the input file.

Keywords: Delta Debugging, Random test case generation, Random DAG generation, Random testing, Fuzz testing, Adaptive random testing, SMT, Bit-vector logic with arrays, Boolector, BTOR format, Minimization of DAGs

Contents

List of Figures

List of Tables

1 Motivation for Delta Debugging and random test case generation

Large software projects usually allow users to submit bug reports. A bug report contains the steps that need to be followed to reproduce a bug.
For example: User X of a media player submits a video file in his bug report which crashes the player. The size of the video file is 100 MB.
A smaller input file would make it a lot easier for the programmer to locate defects in the source code. Additionally the smaller file would take less space in the bug tracking system and so could be added to the program's source code in form of a regression test. Finally duplicate bugs could be identified easier if the corresponding input files were small enough. Ideally there should be a procedure to automatically reduce the size of all submitted input files. Fortunately a procedure exists and it is called Delta Debugging.

In the early stages of software development the programmer often lacks a sufficient amount of input files for testing, especially when the program introduces a new input format. Writing tests is a tedious task and these tests can at most cover all cases the programmer has thought of.

It is not easy to achieve complete path coverage with manually written tests, since the number of paths grows exponentially with the number of branches. As a consequence code that has to deal with complex input file formats will more likely contain paths that are not covered by unit or regression tests. A way to improve a software system's path coverage is random test case generation, also called fuzz testing.

Writing the generator is a one time effort for the programmer, but the amount of work put into the generator should pay off rather quickly, considering the large number of test cases a program can generate in very little time. The more test cases the generator creates, the higher is the possibility that corner cases will also be covered. The generator program doesn't have to be highly sophisticated in order to be able to generate complex output. Random test generation not only makes sense for large programs but also for small projects where often the programmer is also the tester.

1.1 Application on bit-vector logic with arrays

There are two main subjects in this work. On the one hand we will discuss a Delta Debugger for the SMT solver Boolector. Boolector decides the satisfiability of bit-vector logic formulas with arrays[1].

Previous works about Delta Debugging techniques mainly focused on syntax trees (GCC) and XML data files (Mozilla Browser)[3][4]. In order to apply these techniques to BTOR files, the algorithms involved had to be modified and 2 new Delta Debugging algorithms were invented in the progress.

The second part of this work is a random test case generator, which was designed specifically for the BTOR format. A new algorithm had to be thought up because there was no other program available before this project that could randomly generate BTOR files i.e. bit-vector logic with arrays including array conditionals and extensionality.

In the course of this project the two aspects random generation and Delta Debugging were unified in a graphical user interface for presentation purposes. This program will be referred to as the Boolector Debugger.

Moreover the generator and Delta Debugging algorithms can also be used from the command line for better scripting and automation.

1.2 Development of the Boolector Debugger

At the beginning of the project a very basic implementation of a Delta debugger (deltabtor by Armin Biere) already existed in form of a C program.

The generator and Delta Debugging algorithms were all implemented in Java from scratch. Boolector Debugger was developed to incorporate both aspects. It can graphically visualize BTOR files and it offers a bug search mode, which repeatedly generates random BTOR files and writes them to the standard input of a newly generated process using a pipe. The standard output and standard error streams of this process are searched for the strings “segmentation” and “assertion”, as these usually indicate a failure. The executable name and arguments can be freely chosen, it doesn’t necessarily have to be a Boolector executable. Boolector Debugger uses pipes during Delta Debugging too.

Additionally a few shell scripts were written to perform experiments with the new and modified algorithms. One of them produces gnuplot output files for visualization. Other scripts perform cross checks with results from Z3[13] using Wine[14].

The generator, Delta Debugging algorithms and Boolector were all developed in parallel. Every time the BTOR format changed, the other programs had to be modified too.

1. Original version of BTOR
2. Support for array conditionals and array equality (extensionality) added.
3. *nego* no longer allowed. Array node declarations always contain width and length of the array. *acond* needs to be used for array conditions instead of *cond*.
4. New nodes: *next* and *anext*. These are not supported by the Delta Debugger and generator, because development was already finished when *next* and *anext* were introduced.

Figure 1.1: BTOR format evolution

Before the development of the Delta Debugger a problem specific to Boolector had to be solved. The already existing Delta Debugging algorithms from the literature cannot be applied to the BTOR format directly because sub-graphs of a Boolector DAG (directed acyclic graph) cannot just be truncated. Instead other ways had to be found to prune them in a similar fashion.

The existing algorithms didn't always result in a satisfying file size reduction. Because of this two new algorithms are proposed in this text. The first is called changeref algorithm. Although this algorithm produces small results in a reasonable amount of time, its worst case performance of $O(n^2)$ is not really satisfying. As a consequence an extended version of the algorithm was invented. Let's call it the changeref+ binary search algorithm.

1.3 Text overview

The following chapter will deal with random test case generation for Boolector. For this reason the BTOR format will be discussed briefly. The next chapter sums up the experiments performed with the generator. One chapter is dedicated to Delta Debugging. There we explain Delta Debugging in general, algorithms from the literature and our new approaches. Another chapter thoroughly analyzes the performance of each Delta Debugging algorithm. Appendix A contains information how to install and use the presented tools of this work.

2 Random test case generation for Boolector

The generator in Boolector Debugger is used to automatically create random BTOR files. The program has evolved with the BTOR format and supports versions 2 (default) and 3 (with option -newformat) from figure 1.1. Its main goal is to find segmentation faults and assertion failures. It can also handle endless loops. This can be achieved by running Boolector with a given timeout. Unfortunately the generator cannot make a precise statement about the satisfiability of a generated formula. This disadvantage can be partially compensated by cross-checking:

1. Run Boolector several times, each time with different command line arguments
2. Run other SMT solvers using the same input
3. Compare the results

2.1 Boolector's BTOR format

Before we discuss the generator's algorithm we take a look at the format we want to generate. The summary here contains only definitions important to our generator. For a complete overview see [2]. The input language of Boolector was designed with simplicity in mind, so that a parser can be written easily. The format is plain text with each line describing exactly one node of the graph. Comments starting with a "`;`" character tell the parser to ignore the rest of the line. Every line of the file follows this pattern: Node number, node type, width in bits of the node, constant value/name of a variable node/ number of child nodes. A minus sign in front of a child node number indicates negation of this child. A node has to be defined before it can be referenced by another one. The two most basic nodes are variable (*var*) and array nodes (*arr*). Variables have a fixed bit-width. Arrays have a fixed size 2^n (with $n \in N$) with each element having the same given bit-width.

Typically a BTOR file describes a DAG with exactly one *root* node of bit-width one. Although the BTOR format allows multi-rooted files, Boolector Debugger can only handle files with a single root.

Leaf nodes of the DAG can be variables, arrays and constants. Constant values (*const*) can be encoded in base 2, 10 or 16 with 2 being the default. The other bases are indicated by suffixes (*constd* and *consth*).

There are arithmetic, logic and shift operators. An if-then-else operator (*cond*) exists for both bit-vectors and array nodes. Slice, signed- and unsigned-extension nodes allow changes in bit-width.

Arrays can be read from, written to and they can be compared using an equality operator.

2.1.1 Node type overview

Table 2.1 shows input and output path counts for all node types known to the generator. Additionally note that:

- The prefixes s and u indicate signed and unsigned operators.
- The suffix o stands for overflow.
- A *slice* node needs two integer arguments: The indices of the higher and lower border bit. Its resulting bit-vector is cut out of the input vector with border bits inclusive.
- *sext* and *uext* both need one integer argument: The bit-width of the resulting vector.

2.1.2 BTOR example

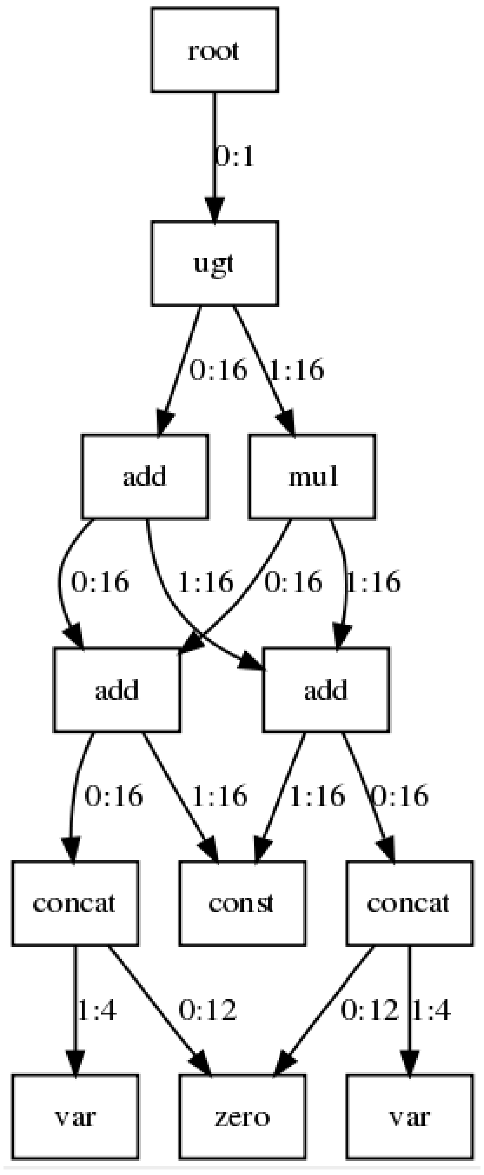
Let a and b be bit-vectors of length 4 and let us assume that $a \geq 0010$ and $b \geq 0010$. Suppose we want to know whether the following theorem holds:

$$ab \geq a + b$$

If the formula is a tautology then its negation must be a contradiction. We negate the formula and write it down as a BTOR file (figure 2.1). A call to Boolector yields the result unsatisfiable. Therefore we have proven the theorem.

```
1 var 4
2 var 4
3 zero 12
4 constd 16 2
5 concat 16 3 1
6 concat 16 3 2
7 add 16 5 4
8 add 16 6 4
9 add 16 7 8
10 mul 16 7 8
11 ugt 1 9 10
12 root 1 11
```

(a) Plain text representation



(b) Graphical representation

Figure 2.1: Example BTOR DAG. Graphical representation was created using Boolector Debugger

Operator name	inputs	width of inputs	width of output
udiv, *sdiv*, *add*, *sub*, *urem*, *srem*, *smod*, *mul*	2	n	n
usubo, *ssubo*, *sdivo*, *uaddo*, *saddo*, *smulo*, *umulo*, *nego*	2	n	1
ulte, *slte*, *ult*, *slt*, *ugt*, *sgt*, *eq*, *ne*, *ugte*, *sgte*	2	n	1
or, *implies*, *iff*, *xor*, *xnor*, *and*, *nand*, *nor*	2	1	1
slice	1	n	$1 <= x < n$
uext, *sext*	1	n	x > n
sll, *srl*, *sra*, *rol*, *ror*	2	2^n, n	2^n
const, *consth*, *constd*, *zero*	0	0	n
read	2	m, n	m
write	3	m, n, m	m
redxor, *redand*, *redor*	1	n	1
not, *neg*	1	n	n
concat	1	m, n	m + n
cond	3	1, n, n	n
root	1	1	0
array, *var*	0	0	n

Table 2.1: Operators of Boolector

2.2 Requirements for the test case generator

The generator should be able to discover bugs that are hard to find using conventional unit tests. Without performing experiments it's hard to tell whether a lot of small generated examples contain more bugs than fewer more complex ones. Keeping this in mind the size of the DAG produced by the generator should be configurable.

The results should be reproducible. In other words given the same random seed and input parameters the generator should always produce exactly the same output file.

An exhaustive generation of all DAGs given certain input parameters is not our goal for now, as it would be hard to achieve even if we would fix the bit-width for all nodes and arrays. The generator should be fast enough to produce lots of random samples so that we can cover important cases with a high probability.

2.3 Algorithm for generating random DAGs

The BTOR format knows around 60 (depending on the format version) different node types. Each node type has a fixed number of input parameters. The bit-widths of each parameter depend on the node type. The generator has to consider the rules of each node type in order to generate a syntactically correct graph.

The implemented algorithm uses bottom up generation, as with this approach a DAG structure can more easily be created. Using top down generation one would need 2 steps. The first steps generates a tree and in the second step references would need to be changed in order to create a DAG.

The algorithm starts with an empty data structure and then adds a fixed number of variable and array nodes.

A loop adds nodes one by one until the maximum graph depth or the maximum node count are reached. The node type and its bit-width are randomly chosen.

Depending on the node type several parameters have to be determined. For a *slice* node low and high values are determined. These define which part of the bit-vector will be cut out. Node types *uext* and *sext* also need a numeric parameter, which defines the bit-width of the result. This number is stored in the low field of the node. When a value has been chosen for the width of the first child, this can also affect the width of other inputs or the output. For example choosing the width of the first child for an *add* node would imply the width of the other input and the output. To keep the algorithm simple all node types were grouped considering the relevant properties for the generator. For example: *add* and *mul* are in the same groups because the same constraints apply to the bit-widths of their inputs and their output. This way of handling the different node types resulted in 16 different groups (see rows of table 2.1)

For each array type node the array width is randomly chosen using the pool of existing array nodes.

When the node's type and its children's bit-widths have been determined child nodes can be selected one by one.

In order to accomplish this we check if there are matching nodes considering bit-width and array type. Depending on the setting of DAG tendency nodes without a parent are considered to have priority. If nodes with matching array type and bit-width are found, one of them is chosen randomly avoiding slice, uext and sext nodes. If no matching nodes were found then one is chosen randomly anyway, but a node of type slice, uext or sext is selected and inserted between it and the child to match the bit-width.

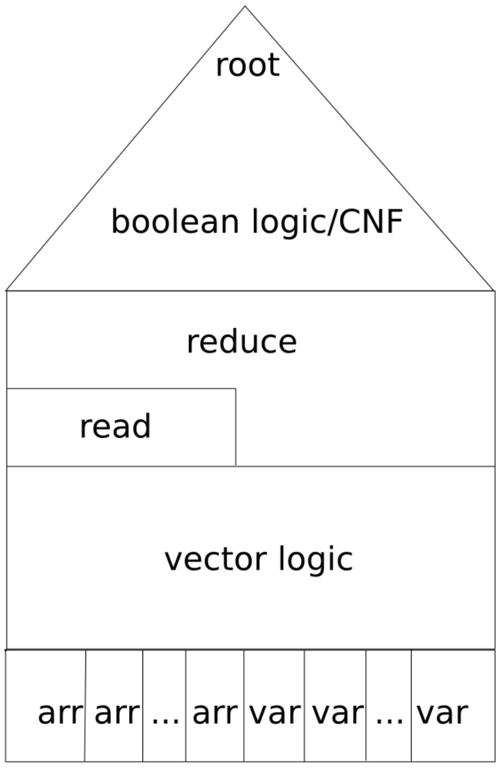


Figure 2.2: Schema of a randomly generated DAG

Each node has a flag which remembers if it already has a parent.

This flag is important for the last phase of the algorithm when all the generated sub-graphs are merged.

Before merging we have to get rid of array type nodes. Each array type node is provided with a new *read* node as its parent. The address of the read is chosen from the existing nodes. A new variable is introduced if there is no node with a matching bit-width.

The *root* node must have bit-width one. Because of this a reduction layer is generated above the read node layer. This is done by connecting reduction type nodes (operators prefixed “red”) to each fatherless node with bit-width larger than one.

Finally all sub-graphs are merged using a minimum number of logical operators. The root node is connected to the last remaining node without a parent. Figure 2.2 shows the parts of a DAG generated with our algorithm.

2.3.1 Algorithm parameters

Maximum graph depth, maximum node count These are only approximate values. They tell the generator when to start merging the generated sub-graphs. Merging starts when either the maximum node count or the maximum graph depth is reached. The default values are 9999 and 100.

Start variable count This value tells the generator how many variable nodes should be generated in the beginning. Note that this is not necessarily the maximum count of variables in the resulting graph. For the generation of the read layer we need address nodes and if there are no nodes with a matching bit-width the algorithm introduces additional variable nodes. The default value is 10.

DAG tendency This parameter indicates the probability in percent of a node having more than one parent node. For lower values more tree like structures tend to be generated. The default value is 50.

Node type distribution The exact distribution of nodes can not be defined by the user, because it contradicts the principle of random generation. However there is an option that forces the generator to use all node types in the graph. Furthermore the generation of array type nodes can be switched off. Default settings: array generation turned on, force usage of all node types turned off.

Bit-widths A minimum and a maximum bit-width can be set. There is also a value that limits the size of arrays. Actual bit-widths are chosen uniformly random in the given interval. The default values are minwidth=1, maxwidth=4, arrmaxwidth=4.

2.3.2 Influencing the satisfiability using CNFs

An alternative way of merging the remaining sub-graphs is to generate a CNF. In this way we can influence the satisfiability of the generated formula.

Using random logic operators in the last step of the algorithm the resulting formulas tend to be satisfiable, whereas generating a CNF with lots of clauses tends to produce unsatisfiable results.

The generator uses a parameter called clauses per variable ratio to determine how many clauses will be generated. To keep the algorithm simple only 3 variables per clause are used. The 3-SAT problem is NP-hard[11] and there is a heuristic connection between the number of variables and the number of clauses in a 3-CNF (see [10]).

The following algorithm is used by the generator to build a 3-CNF:

1. Clauses are generated so that each variable is involved at least once. If the number of variables cannot be divided by 3 without remainder, 1 or 2 already used variables are chosen randomly to complete the last clause of this step.
2. Additional clauses are generated until the desired ratio of variables to clauses is reached.
3. All remaining nodes without a parent are merged using *and* nodes.

In our algorithm one random number determines which variable is used in a clause and whether it is negated or not. If there are n variables, we calculate the next random number x with $0 \leq x \leq 2n - 1$. In our clause we use variable number $x \mod n$ and if $x > n$ the variable is negated. This method ensures a uniformly distributed selection from the set of all possible clauses.

3 Experiments with the generator

Considering randomly generated DAGs a few open questions need to be answered:

1. How do we find bugs more efficiently: Generate
   a) lots of small test cases?
   b) fewer but larger test cases?
2. How do the settings of the generator influence the satisfiability of generated BTOR files?
3. Is there a way to check the plausibility of Boolector's results for our generated files?

3.1 Setup of experiments

The experiments were performed using these bash scripts and tools:

generate.sh Uses the command line interface of the generator to produce input files. All parameters except one are fixed. This way we can create lots of different test cases easily, varying e.g. only the node size.

eval.sh Calls the Boolector executable using a given timeout. Gathers information about running time and return code.

awk Was used to transform the data files generated by our scripts so that gnuplot could work with them.

gnuplot Produced the diagrams.

3.1.1 Varying the graph size

For this experiment we generated a lot of BTOR files and let different Boolector versions evaluate them. The Boolector versions used here (named after their internal release number) are not to be confused with any of the officially released versions. Being early predecessors they contain code defects and were kept solely for the purpose of performing the experiments described in this text.

It should not matter if we regulate the graph size by graph depth or by maximum node count, so the latter was chosen. This experiment should give us an overview of how Boolector has improved over several revisions, however the results are not as comparable as one would wish because for each Boolector version different BTOR files had to be used due to incompatibility among the different (legacy) BTOR formats.

For experiment 1 approximate node sizes from 10 to 5000, with steps between of 100 nodes and 4 files generated for each size were used. Version 1 (newer) of Boolector was only tested with an upper bound of 2500, as larger problems produced only timeouts. The two oldest Boolector versions obviously could not cope with larger problems either, so their upper bound was set to 2000. Version 0 and the older version 1 did not yet support array conditionals nor array equalities, consequently there were a few parse errors. Version 0 could hardly solve any of the generated problems, so we decided to skip its diagram. The other generator options were chosen as

-clauses=4 -d=9999 -arrays=3 -writes=5 -vars=10 -minwidth=1
-maxwidth=5 -arrmaxwidth=4 -dag=50 -cnf=false -all=false
-noarrays=false -newformat=true -cnfonly=false

For the latest Boolector version under observation the switch **-newformat** had to be used. Boolector was given a maximum time of 10 seconds to evaluate each file and was run with default settings i.e. no command line arguments. The first file was generated with a random seed of 1, the random seed being incremented by 1 after each file.

Results are shown in figures 3.1 and 3.2. Many of the files generated caused our Boolector versions to crash with a segmentation fault. We can say very little about the performance of the oldest Boolector version. At least it could solve some of the problems in very little time. Version 1 (old) shows a big improvement over version 0. Version 1 (newer) could already parse all our generated DAGs.

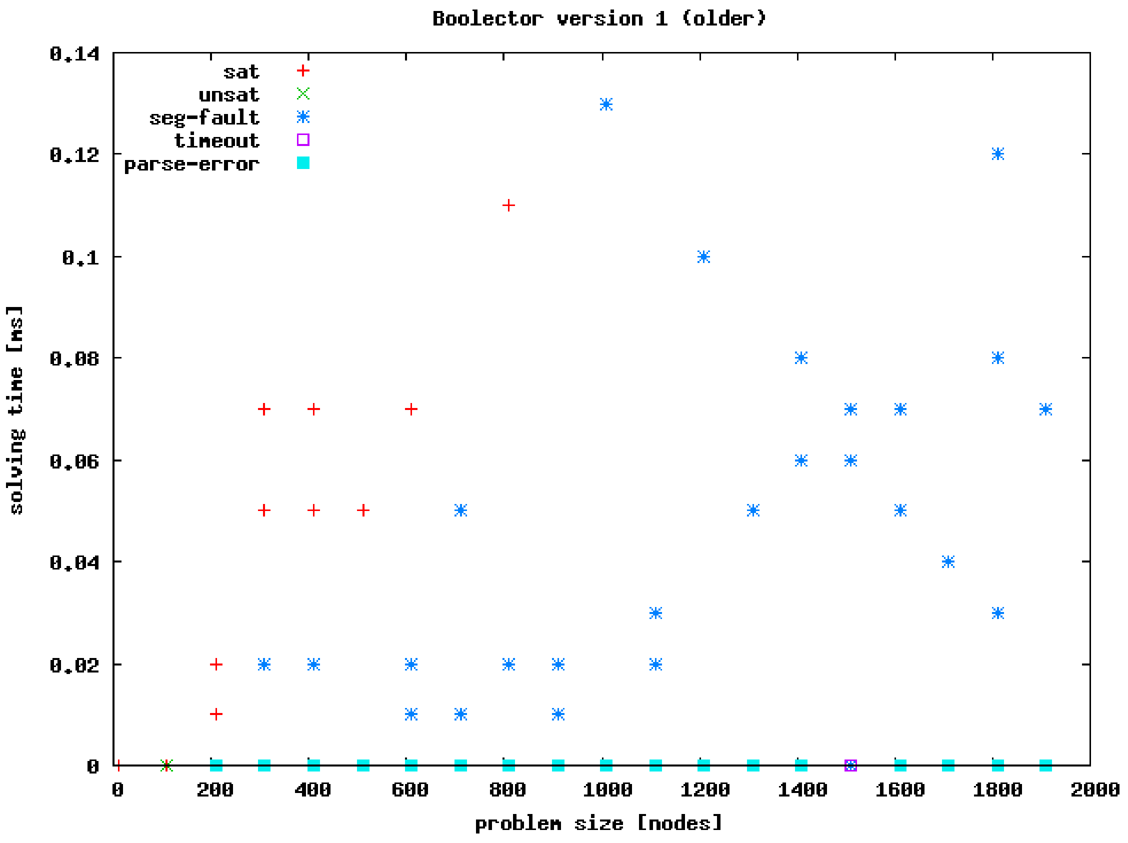
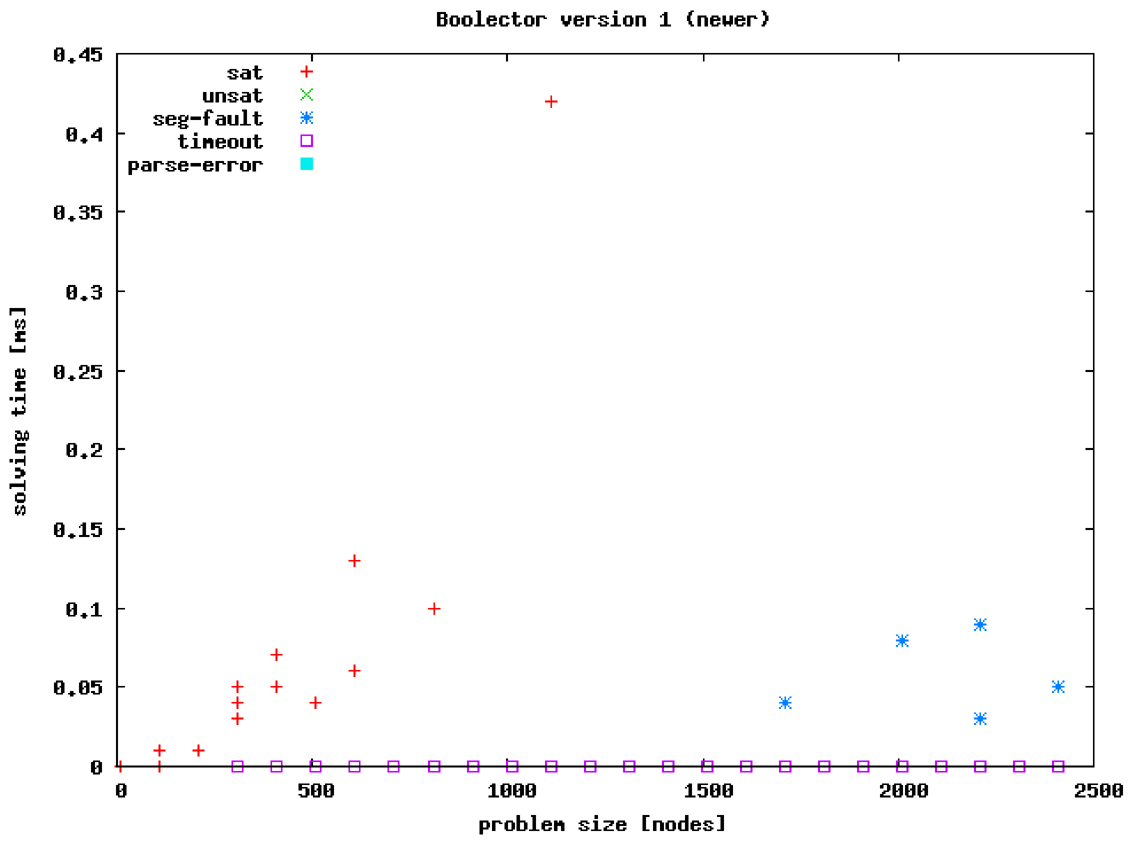


Figure 3.1: Experiment 1 - Varying file sizes



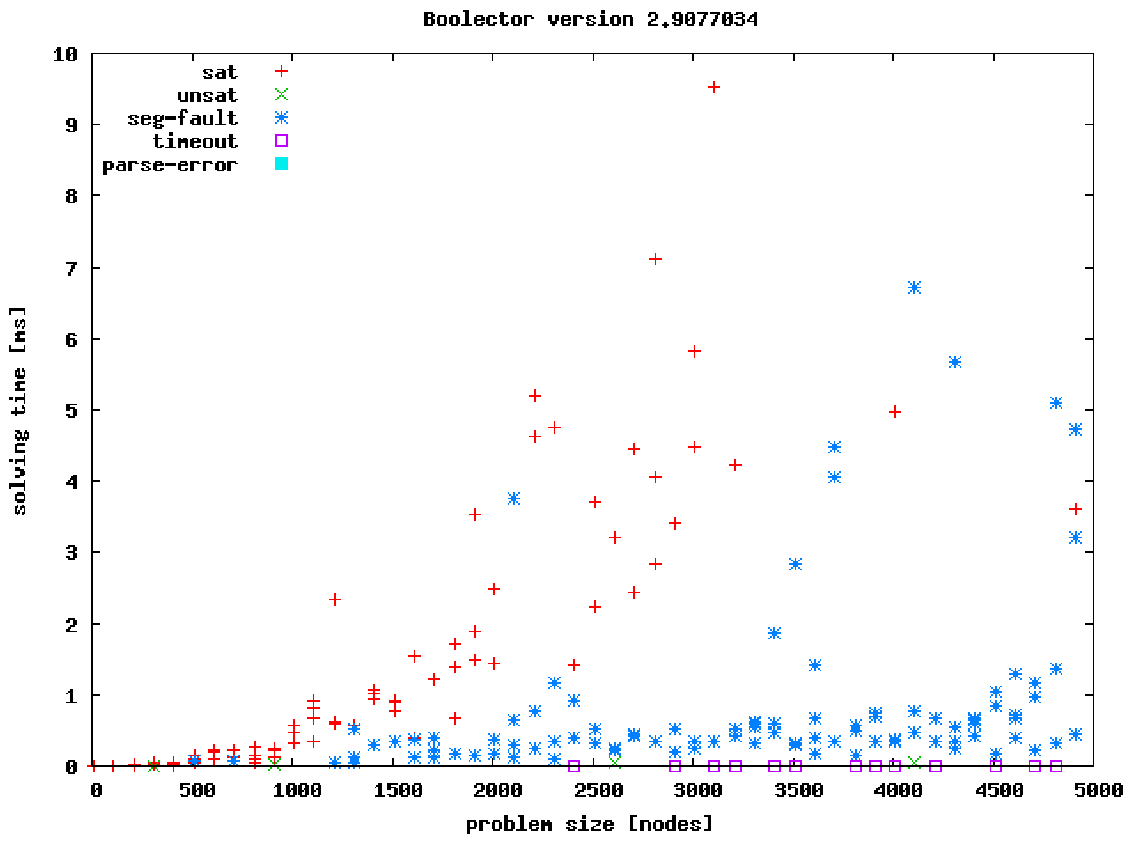


Figure 3.2: Experiment 1 - Varying file sizes

Obviously the solving time increases with the node count in all three images which is not surprising. Generally we can say that with these settings the major part of the problems is satisfiable. Only a handful of green crosses can be found in all of the diagrams. Even the latest Boolector version seems to crash regularly on inputs consisting of more than 3000 nodes, although it could solve some of the bigger problems. It could also solve some of the problems that suffered from timeout given 10 seconds more time.

Figures 3.1 and 3.2 do not answer the questions posed at the beginning of this chapter. To find out which maximum graph size setting finds the highest percentage of bugs using as little time as possible we created the plot in figure 3.3. For each node size we divided the number of failure inducing files by the sum of all files of that size. The result was divided by the average time it took Boolector to evaluate those files. Considering the definition of our function, it should indicate the optimum graph size for operating the generator with a peak. The graph in figure 3.3 has two almost equally high maxima, but it is not yet clear whether increasing the maximum graph size even more could improve results. As a consequence experiment 2 was repeated, with new BTOR files and an upper bound of 10000 nodes per DAG, see figure 3.4. Starting at a DAG size of around 2500 the Boolector version under observation almost constantly produced segmentation faults for larger graphs. All fluctuations of our function after this point are caused by minor differences in the running time before the crash.

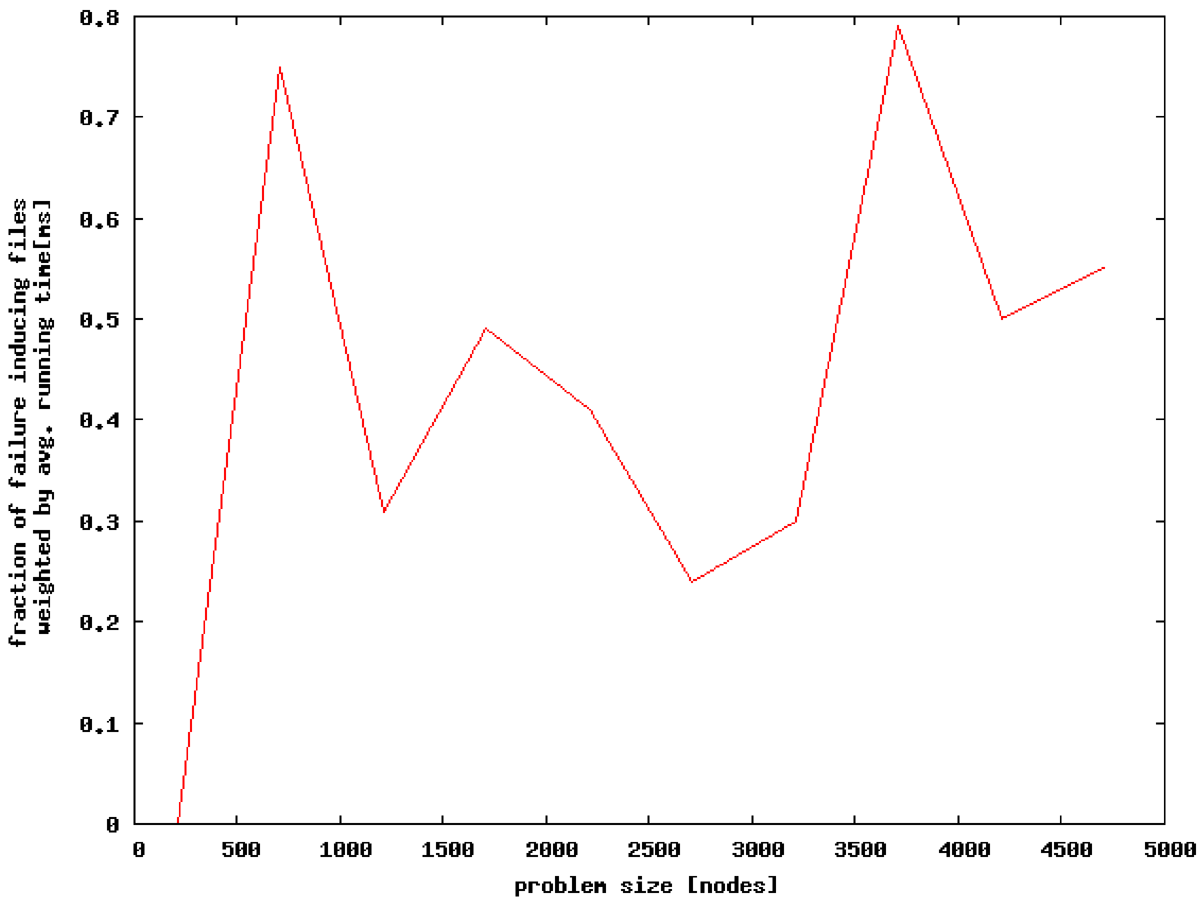


Figure 3.3: Generator experiment 2 - Shown is a different visualization of the data from experiment 1.

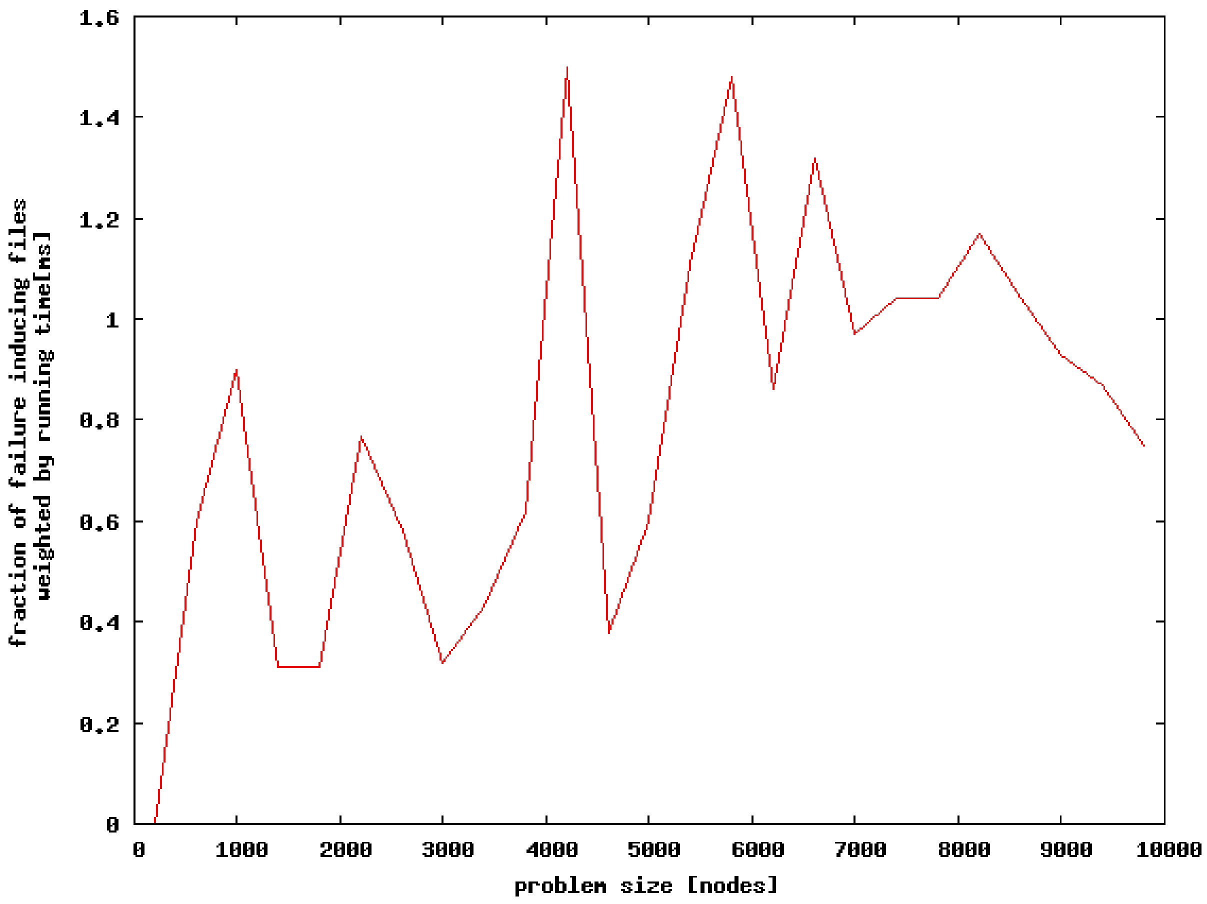


Figure 3.4: Generator experiment 2 - New BTOR files were generated for this diagram. The maximum size was increased to 10000.

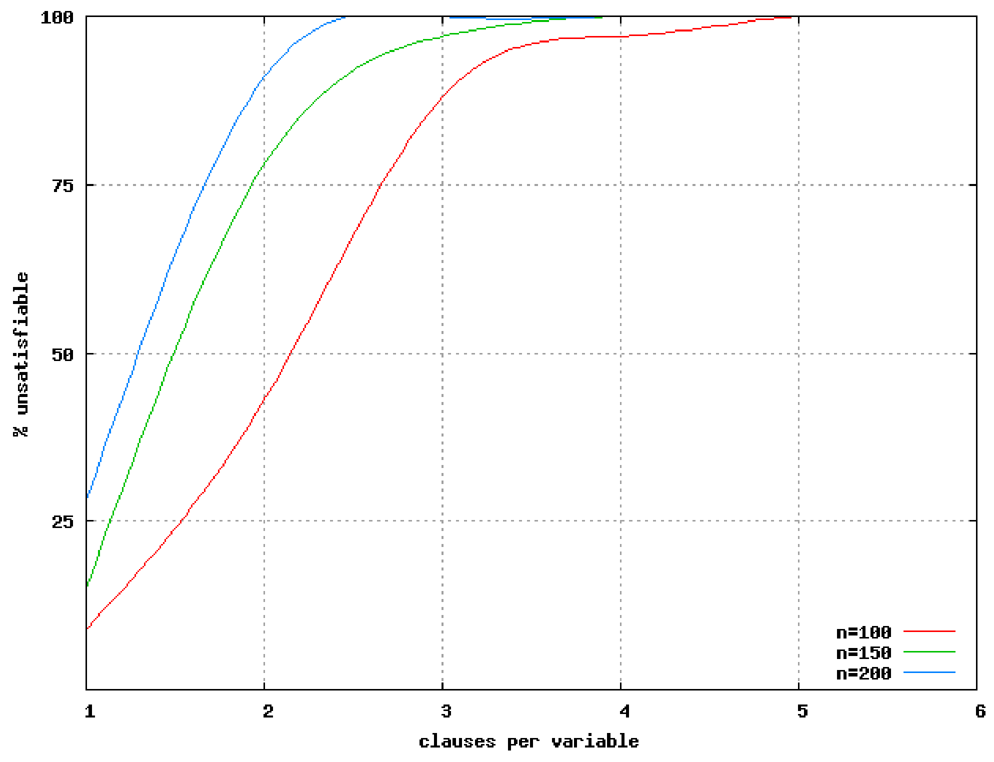


Figure 3.5: Experiment 3 - DAGs with CNF: Varying the ratio of variables to clauses in the CNF layer

3.1.2 Toggling CNF generation

In the course of experiments 1 and 2 unsatisfiable formulas were rarely generated. In this section we want to test our CNF generation, and observe how it changes the ratio of unsatisfiable to satisfiable test cases. This time we let our generator produce files with a varying node size of 100, 150 and 200. At the same time we kept the number of variable and array nodes on which the DAGs were built fixed at 10 and 3. We varied the setting of clauses per variable from 1 to 10, generating 100 test cases for each value.

For randomly generated 3-CNFs a value of about 4.25 clauses per variable is the point where the ratio of satisfiable to unsatisfiable is about equal, given a large number of variables. These problems are the hardest to solve[10]. With bit-vector logic connected to the leafs of a CNF we would expect this number to be lower as the bit-vector logic imposes additional constraints. As the experimental results in figure 3.5 show, the 50% mark moves towards lower values, as we increase the ratio DAG size to initial variable count. In other words the more bit-vector logic we build upon a fixed number of variable and arrays, the less likely the whole structure is satisfiable. The functions in the diagram were smoothed using B-splines.

3.2 Conclusions from generator experiments

Unfortunately we cannot make a clear statement concerning the optimum parameters for the generator. As our experiments with the varying graph size showed, the optimum parameter settings depend largely on the properties of the particular bug. Concerning the maximum DAG size we conclude that any value of around 1000 or above is probably good enough for practical reasons as further increasing the value doesn't significantly improve the results regarding the efficiency of the search.

Concerning the satisfiability of the generated files we have definitely found a means of shifting the results in the direction we want: By enabling CNF generation on top of the DAG we can on the one hand increase the numbers of clauses per CNF variable and the results will tend to unsatisfiable. On the other hand we can achieve the same by fixing the number of clauses per variable and increasing the amount of bit-vector logic in relation to the number of leaf nodes in the graph. Satisfiable examples can be deliberately generated simply by disabling CNF generation.

Implicitly we have answered the third question from the beginning of the chapter: Boolector's results can be probed for plausibility by generating test cases using options that have a high tendency towards one result. If Boolector gives an unexpected answer we should then examine this case diligently as it could very likely point us to a bug.

4 Delta Debugging

Delta Debugging is a technique for minimizing failure inducing program input. It preserves the input's failure inducing property while minimizing the input according to a desired criterion. This criterion will be the input size in most cases but could also reflect the program's running time, or a different property.
In its most general form a usual Delta Debugging technique has 3 steps:

1. Prune the current program input.
2. Run the program on the pruned input.
3. If input is minimal then stop.
   If the failure persists then continue with 1.
   Else revoke the last pruning step, choose a different part to prune and continue with 1.

The performance of the algorithm can be increased by pruning with respect to the input language. The selection method of the part to prune depends on the particular algorithm. In order to be efficient the algorithm has to take the specific structure of the input into account.

4.1 Existing algorithms

In this part we will discuss Delta Debugging algorithms from the literature and how they were modified and implemented to better fit the needs of the BTOR format.

4.1.1 The ddmin algorithm

This algorithm was introduced by Zeller in 2002. We discuss the version printed in [4]. It regards the input file as a set which can be partitioned as needed. Partitions in terms of the algorithm are considered having roughly the same size. The empty set (no input) is considered as passing the test by default. The algorithm performs a kind of binary search for the smallest possible failure inducing subset of the input. Zeller defines the smallest possible failure inducing input as being 1-minimal: removing a single character from it would cause the failure to disappear. For each test the program under observation is called with the current subset of the original failing test case.

The Minimizing Delta Debugging Algorithm ddmin:

1. Let $count(partitions) = 2$, current test case fails, empty test case passes.
2. Test each partition of current test case. If test of $partition_i$ fails then replace test case by $partition_i$ and continue with 2. Test complement of each partition. If test of $complement_i$ fails then replace test case by $complement_i$ and goto 2.
3. If number of partitions equals element count in test case then goto 4.
   Else double the number of partitions. If there are more partitions than elements in the test case use element count of test case instead. Goto 2.
4. Done. Current test case is 1-minimal.

The algorithm was written in prose for easier readability, consult [4] for the original, more formal definition. In ideal cases, for example when a single element causes the failure, the running time of ddmin is that of binary search. However in the worst case ddmin needs up to $n^2 + 3n$ tests for an input file consisting of n elements. This occurs when the failure inducing parts form a pattern at the finest (element) granularity. The example in figure 4.1 contains only 4 elements. Element 1, 2 and 3 make the failure appear, but only when they are all part of the test case. We suppose that the test result is unresolved (indicated by a "?") when the test case only contains a subset of the 3. We assume a simple implementation which does not cache results, only in the case of 2 partitions the implementation skips the unnecessary test of the complement. With result caching some tests would only need to be performed once.

Issues with the ddmin implementation

In implementing ddmin for the BTOR format, we came across several problems. Fortunately their solution could also be applied in the algorithms that are presented next:

- BTOR files cannot be arbitrarily cut into pieces. Single characters cannot be used as our elements. This is only a minor problem, which we solve by defining an element to be one node of a DAG.

Test no.	1	2	3	4	result	comment
1	1	2			?	
2			3	4	?	increase granularity: 4
3	1				?	
4		2			?	
5			3		?	
6				4	pass	
7		2	3	4	?	
8	1		3	4	?	
9	1	2		4	?	
10	1	2	3		fail	decrease granularity: 3
11	1				?	
12		2			?	
13			3		?	
14		2	3		?	
15	1		3		?	
16	1	2			?	
Result	1	2	3			needed tests: 16

Figure 4.1: ddmin: example with a quadratic number of tests

- We cannot simply remove a partition because then the BTOR file would become syntactically incorrect. When a partition consisting of several nodes is removed, we have to somehow adjust affected nodes which reference the removed nodes.

- We also want to apply Delta Debugging to input files which force Boolector into an infinite loop. As a consequence our initial test case has an indeterminate result according to Zeller's algorithm. This conflict can be resolved by allowing only two test case results: passed and failed. Passed means to us that the return code of the Boolector process instance didn't change compared to the initial return code. Additionally our implementation uses a timeout value for handling infinite running times. When Boolector times out on the initial test case, then the goal during Delta Debugging is to keep this initial property in smaller test cases, i.e. the minimized test case should also time out.

By defining an element to be one node of the DAG, we lose some granularity. We must assume that the failing test case is not a simple parsing problem for example caused by a misspelled keyword. This is nothing to worry about since we can assume that the BTOR format is simple enough, that problems with the parser can all be well covered by unit-tests. Removable partitions in Boolector are all sub-graphs of the DAG. When

and	x	**1**	**0**
y	x and y	y	**0**
1	x	**1**	**0**
0	**0**	**0**	**0**

or	x	**1**	**0**
y	x or y	**1**	y
1	**1**	**1**	**1**
0	x	**1**	**0**

nand	x	**1**	**0**
y	x nand y	-y	**1**
1	-x	**0**	**1**
0	**1**	**1**	**1**

nor	x	**1**	**0**
y	x nor y	**0**	-y
1	**0**	**0**	**0**
0	-x	**0**	**1**

add	x	**1**	**0**
y	add(x,y)	add(y, **1**)	y
1	add(**1**,x)	add(**1**, **1**)	**1**
0	x	**1**	**0**

implies	x	1	0
y	y implies x	1	-y
1	x	1	0
0	1	1	1

xor	x	**1**	**0**
y	y xor x	-y	y
1	-x	**0**	**1**
0	x	**1**	**0**

eq	x	**1**	**0**
y	eq(y, x)	eq(y,**1**)	eq(y, **0**)
1	eq(**1**, x)	1	0
0	eq(**0**, x)	0	1

not	x	**1**	**0**
	-x	**0**	**1**

cond(x, y, z)	x = 1	x = 0
	y	z

Figure 4.2: Operator simplification rules - A lot more rules could have been implemented but the goal was not make it too simple for Boolector. Note that **1** + **1** = -2 in two's complement. This was not used, as it does not help simplification.

these sub-graphs are removed, they are replaced by single *var* or *zero* nodes, the latter having an optional negation sign. So for each removal of a sub-graph we test up to 3 cases, depending on which one works. *zero* nodes represent bit-vectors with every bit being set to logical 0. When writing -*zero* we mean a negated *zero* node i.e. a bit-vector consisting of logical ones only.

When a sub-graph consists of one node only, the before mentioned introduction of additional nodes doesn't cause a size reduction of the DAG. However we can reduce the DAG size anyway because the newly introduced *zero* nodes increase the probability that the DAG can be simplified using a predefined rule set. In our implementation the simplification step is optional for each algorithm. We will compare results with and without simplification in chapter 5. Figure 4.2 offers an overview on the implemented simplification rules for bit-vectors. The vector containing only ones is denoted as **1**, the vector with only zeros as **0**.

Algorithm: ddmin for BTOR

1. Parse BTOR input file. The result is a node array.
2. Traverse the DAG using DFS thereby numbering nodes ascendingly. Sort the node array using those numbers.
3. Call Boolector on BTOR DAG to get the initial return value (which also encodes a possible timeout)
4. Set number of partitions to 1, create a zero node for each bit-width.
5. If partitions == current DAG size then write out the DAG and exit.
   partitions:= max(current DAG size, 2 ∗ partitions), calculate partitions using current DAG size.
6. For each partition:
   a) Back up the DAG

   b) Replace partition nodes by *zero* nodes while keeping the DAG syntactically correct. Simplify the DAG (optional). Call Boolector.
   c) If return value doesn't equal the initial one then restore DAG.
      else partitions:= 1, goto 5.

   d) Replace partition nodes by -*zero* nodes. Simplify the DAG (optional). Call Boolector.
   e) If return value doesn't equal the initial one then restore DAG.
      else partitions:=1, goto 5.

   f) Replace partition nodes by new *var* nodes. Simplify the DAG (optional). Call Boolector.
   g) If return value doesn't equal the initial one then restore DAG.
      else partitions:= 1, goto 5.

7. For each partition complement: perform steps a to g.
   Goto 5.

Allowing only pass and fail as test results in our implementation one could conjecture that the running time of the worst case is reduced. Let our input consist of n elements: $1 \dots n$. Let the failure inducing part be caused by all even numbered elements together, with n being even. In the first phase of the algorithm all test cases pass until granularity n is reached. The test of the last complement on that granularity fails. Up to that point $2(2+4+8+\dots+n) = 2(n+\frac{n}{2}+\frac{n}{4}+\dots) = 4n$ tests were performed in total. Granularity is decreased to $n-1$. Then again $n-1$ tests pass and the first complement fails. The remaining subset is reduced by 1 element until it is minimal. The number of tests performed in phase 2 are in total:

$$\underbrace{(n-1+1)+(n-2+1)+(n-3+1)+\dots}_{\frac{n}{2}} =$$

$$\underbrace{n+(n-1)+(n-2)+\dots}_{\frac{n}{2}} =$$

$$\frac{n(n+1)}{2} - \frac{(\frac{n}{2}-1)(\frac{n}{2}-1+1)}{2} = \frac{3}{8}n^2 + \frac{3}{4}n$$

This proves that our implementation of ddmin still needs $O(n^2)$ tests [1] in the worst case.

4.1.2 Hierarchical Delta Debugging

Hierarchical Delta Debugging algorithms suit the structure of the BTOR format better than the plain ddmin algorithm. They were specifically designed for input formats with a tree structure and thus can be extended to work with DAGs easily. The hierarchical algorithms and their analysis was taken from [3].

The basic HDD algorithm:

```
procedure HDD(input_tree)
  level := 0
  nodes := TAGNODES(input_tree, level)
  while nodes <> 0 do
    minconfig = DDMIN(nodes)
    PRUNE(input_tree, level, minconfig)
```

[1]Actually we perform each test three times (replace with *zero*, *-zero* and *var*). This is not shown in the calculation, as it does not affect the asymptotic running time.

```
    level := level + 1
    nodes := TAGNODES(input_tree, level)
  end while
end procedure
```

As we can see HDD applies ddmin to each tree level. It starts at the root level. The nodes of the current level are tagged. Then a minimum configuration of these nodes is determined. The tree is pruned and the algorithm increases the level by one. The performance of the algorithm depends on the shape of the tree, more balanced trees take less running time. Asymptotically HDD never performs more tests than ddmin. In the worst case $O(n^2)$ tests are performed, n being the number of nodes in the tree.

The implementation of HDD for the BTOR format was straightforward: Using the already implemented ddmin algorithm for BTOR we only had to replace sub-trees by sub-graphs in the algorithm. Every time we prune the DAG, we prune an entire sub-graph instead of only one reference to it.

4.1.3 The HDD+ algorithm

HDD+ is a simple extension of the HDD algorithm. First it performs HDD on the input, then BFS (see 4.3.1) is used to remove each node exactly once.

```
procedure HDDPLUS(input_tree)
  HDD(input_tree)
  old_size := 1
  new_size := 0
  while new_size < old_size
    old_size := NODECOUNT(input_tree)
    BFS(input_tree)
    new_size := NODECOUNT(input_tree)
  end while
end procedure
```

4.1.4 The HDD* algorithm

This algorithm performs HDD repeatedly on the input until no more size reduction is achieved.

```
procedure HDDSTAR(input_tree)
  old_size := NODECOUNT(input_tree)
  new_size := 0
  while new_size < old_size
    old_size := NODECOUNT(input_tree)
    HDD(input_tree)
    new_size := NODECOUNT(input_tree)
  end while
end procedure
```

4.2 deltabtor's algorithm

deltabtor is a C program by Armin Biere, written before the start of this project. It is a simple yet effective Delta Debugging application and the first that could handle BTOR files. The algorithm involved has an asymptotic running time complexity of $O(n)$. deltabtor is quite similar to ddmin. The main difference is that deltabtor does not care about complements which makes it faster than ddmin. Complements are handled implicitly.

The workings of deltabtor are illustrated in figure 4.3. It tries to remove nodes by setting partitions to *zero*, *-zero* or *var* nodes. Then it simplifies the resulting DAG and calls a given executable to find out if the new DAG still produces a failure. deltabtor implements less simplification rules than Boolector Debugger. In the image we see how the partitions are chosen. First deltabtor uses the whole file except the *root* node (1), then continues with (2), (3) and so on. In the process smaller and smaller parts are cut off the DAG. Every time the DAG size is reduced, the intermediate result is written to a temporary file. In this way deltabtor can be terminated any time and continue to work on the same file when told to. For n nodes in the input files it performs a maximum of $3(n + \frac{n}{2} + \frac{n}{4} + \ldots + 1) = 6n$ tests.

deltabtor did not participate in the evaluation in chapter 5. As the C implementation was already working satisfactorily when the Boolector Debugger was written there was no need for a reimplementation.

4.3 New proposed algorithms

This section describes the BFS and LW algorithms which can both be applied to generic hierarchical data structures as well as BTOR files. Following up are the changeref and

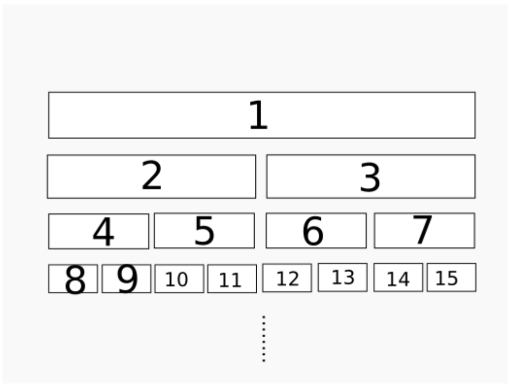


Figure 4.3: deltabtor partitioning scheme

changeref+ binary search algorithms, which were both especially developed for BTOR files with better performance in mind.

4.3.1 BFS algorithm

The BFS (breadth first search) algorithm traverses the given DAG in BFS style. For each node we try our three replacements (zeros, ones and vars), simplify the DAG and call Boolector. This algorithm was implemented for completeness and because it is used by HDD+. In the worst case it calls Boolector $3n$ times (with n nodes in the DAG). The recursive BFS procedure is initially called with the root node as argument 1:

```
procedure BFS(node, btor_dag)
  children = GETCHILDREN(node)
  for each child c in children
    backup := BACKUP(btor_dag)
    SETTOZERO(c)
    SIMPLIFY(btor_dag)
    return_code = CALLBOOLECTOR(btor_dag)
    if return_code == initial_return_code
      continue
    else
      btor_dag := backup
    end if
    SETTOONE(nodelist)
    SIMPLIFY(btor_dag)
```

```
      return_code = CALLBOOLECTOR(btor_dag)
      if return_code == initial_return_code
        continue
      else
        btor_dag := backup
      end if
      SETTOVAR(nodelist)
      SIMPLIFY(btor_dag)
      return_code = CALLBOOLECTOR(btor_dag)
      if return_code == initial_return_code
        continue
      else
        btor_dag := backup
      end if
    end for
    for each child c in children
      BFS(c, btor_dag)
    end for
  end procedure
```

4.3.2 LW algorithm

LW (layerwise) is an algorithm which tries to remove complete layers of the BTOR file.

```
procedure LW(btor_dag)
  depth := 0;
  maxdepth := 1;
  while depth < maxdepth
    nodelist := GETNODES(btor_dag, depth)
    if COUNT(nodelist) == 0
      return
    end if
    maxdepth := MAXDEPTH(btor_dag)
    backup := BACKUP(btor_dag)
    SETTOZERO(nodelist)
    SIMPLIFY(btor_dag)
    return_code = CALLBOOLECTOR(btor_dag)
    if return_code <> initial_return_code
     btor_dag := backup
    else
      depth := depth + 1
      continue
    end if
    SETTOONE(nodelist)
    SIMPLIFY(btor_dag)
    return_code = CALLBOOLECTOR(btor_dag)
    if return_code <> initial_return_code
     btor_dag := backup
    else
      depth := depth + 1
      continue
    end if
    SETTOVAR(nodelist)
    SIMPLIFY(btor_dag)
    return_code = CALLBOOLECTOR(btor_dag)
    if return_code <> initial_return_code
     btor_dag := backup
    else
      depth := depth + 1
    end if
  end while
end procedure
```

We define the depth of a node n in a DAG as the longest possible path from the *root* node to n. The algorithm starts at depth 0 (*root* node) and increases the working depth in the DAG with each iteration. This way we hope to remove as many nodes as early as possible. At each depth we try to set the current node layer to all-zeros, all-ones vectors or new *var* nodes of matching bit-width.

4.3.3 changeref algorithm

The idea of this algorithm is to short-circuit middle parts of the DAG, which possibly contain information irrelevant to the failure. For each node, starting with the root, we change references to its children, so that they point to other nodes deeper in the graph. We cannot arbitrarily connect nodes as the bit-widths must match. With array type references also the array size has to fit. The candidate nodes are sorted according to their graph depth. We start trying the nodes in greater depths first, so we can cut off as many nodes as possible if the failure is located near the leaf nodes. This strategy can also be thought of as putting a root node over each sub-graph and then probing it for failure. An advantage of this algorithm is that it does not have to introduce new nodes artificially, so successive simplification is not needed, although we implemented it to be able to compare if the achieved results are different. Suppose our algorithm is working on the DAG shown in figure 4.4 and the current node is the *and* node. Our options for edge x are indicated by dotted lines. The number next to the dotted line indicates in which order these options are tried.

The changeref algorithm:

```
procedure changeref(node, btor_dag, depth)
  if VISITED(node)
    return
  end if
  for each child c in GETCHILDREN(node)
    nodelist = GETMATCHINGNODES(c, depth+1)
    backup := BACKUP(btor_dag)
    for node n in nodelist
      c := n
      return := CALLBOOLECTOR(btor_dag)
      if return <> initial_return
        btor_dag := backup
      else
        break
      end if
    end for
  end for
```

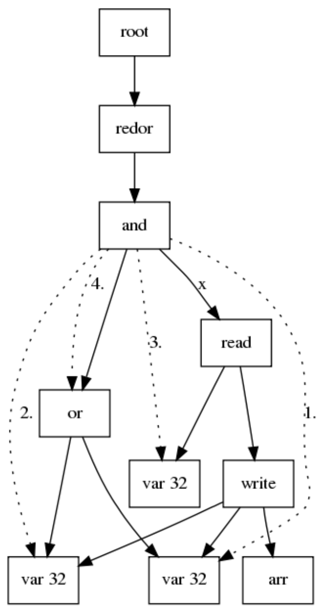


Figure 4.4: changeref algorithm: alternative positions of edge x

```
  SETVISITED(node)
  for each child c in GETCHILDREN(node)
    changeref(c, btor_dag, depth+1)
  end for
end procedure
```

4.3.4 Improving the changeref algorithm

The changeref algorithm was initially developed to reduce the size of BTOR files which could not be made smaller by other already existing algorithms. This algorithm is rather expensive if it is used on large DAGs. Consider the case in figure 4.5. We are currently processing the *read* node. The irrelevant children of the *write* nodes and the rest of the DAG are not shown for clarity. Suppose the failure occurs when at least one *write* is present in this part of the DAG. Our algorithm tries to set reference x to *write* 1, *write* 2 and so on. This way it only removes one node per call to Boolector. We have $n + (n-1) + (n-2) + \ldots + 2 + 1$ calls, in sum $\frac{n(n+1)}{2}$ calls. In other words in this worst case changeref performs $O(n^2)$ calls to the executable (Boolector in our case), with n nodes in the DAG. This degenerate case is not unlikely to appear in practice, as long *write* node chains are commonplace in BTOR files. Consequently it made sense to invest time in the improvement of the changeref algorithm.

Our worst case scenario shows us that it is not always necessary to try all matching nodes deeper in the graph. The idea is to leave out certain nodes in the search systematically. We decided to use a kind of binary search approach, hence the name changeref+ binary search:

1. The candidate nodes are sorted by graph depth ascendingly.
2. Try the node near the middle of the depth interval first.
3. If the last node did not work then divide the interval closer to the root into halves. Again try the node between those halves.
   ...

Using this approach our possible choices shrink from n to $\log n$. So in the worst case the number of performed tests is:

$$\begin{aligned} \log_2(n) + \log_2(n-1) + \log_2(n-2) + \ldots + 2 &= \\ \log_2(n) + \log_2(n-1) + \log_2(n-2) + \ldots + \log_2(1) &= \\ \log_2(n(n-1)(n-2)\ldots(n-n+1)) &= \log_2(n!) \end{aligned}$$

From theory we know that $\log_2(n!)$ is also the upper bound of comparisons for the best sorting algorithms [12], so $\log_2(n!)$ is of $O(n \log n)$. We can conclude that the changeref+ algorithm performs much better in the worst case than the original changeref algorithm. The drawback is that not all possible combinations are tried. One could combine the two changeref algorithms into one by first performing the binary search version and then using changeref on the presumably smaller file.

4.4 Combining algorithms

In reality when there is a failure inducing input file we need not settle on one particular algorithm to reduce the size of the file. We simply choose the algorithm which works best on the given file and call it repeatedly if necessary. This is where the Boolector Debugger (see chapter A) comes in handy. If one algorithm doesn't succeed in reducing the size sufficiently, we can try another algorithm on the already reduced file. If one algorithm can cut off only a little piece maybe another algorithm can again make progress and so on.

A little experimenting is always needed because none of our algorithms can give us a guarantee to succeed in making the given input smaller. Modifying the algorithms to

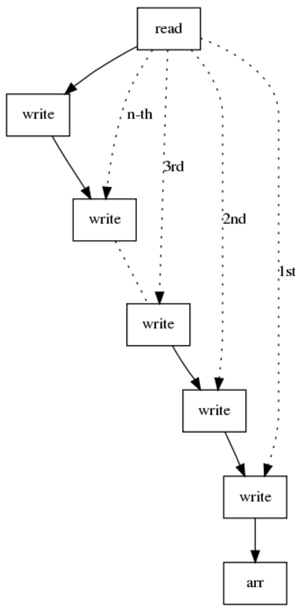


Figure 4.5: changeref algorithm - worst case example

work with the BTOR format involves a trade-off: We lost the deterministic running time of the original algorithms by making them work with the BTOR format. The running time of our algorithms is depending heavily on the specific DAG structure. Additionally the effect of simplification on the running time is hard to calculate.

In our last experiment we tried to reduce some of our input files as much as possible using all of the algorithms. The strategy was to apply one algorithm as long as it can reduce the size of the file, then move on to the next algorithm. HDD+ and HDD* were left out as HDD+ is just HDD* followed by BFS, and HDD* is just repeated use of HDD. We used 3 seconds as timeout and simplification was turned on.

For our files ex1 to ex5 the node size developed as follows:

$$
\begin{aligned}
ex_1 : 732 &\xrightarrow{DDMIN} 72 \xrightarrow{HDD} 30 \xrightarrow{BFS} 30 \xrightarrow{LW} 30 \xrightarrow{CHANGEREF+} 22 \xrightarrow{CHANGEREF} 17\\
ex_2 : 992 &\xrightarrow{DDMIN} 85 \xrightarrow{HDD} 57 \xrightarrow{BFS} 54 \xrightarrow{LW} 54 \xrightarrow{CHANGEREF+} 29 \xrightarrow{CHANGEREF} 28\\
ex_3 : 1687 &\xrightarrow{DDMIN} 44 \xrightarrow{HDD} 25 \xrightarrow{BFS} 20 \xrightarrow{LW} 20 \xrightarrow{CHANGEREF+} 17 \xrightarrow{CHANGEREF} 17\\
ex_4 : 1850 &\xrightarrow{DDMIN} 53 \xrightarrow{HDD} 27 \xrightarrow{BFS} 27 \xrightarrow{LW} 27 \xrightarrow{CHANGEREF+} 22 \xrightarrow{CHANGEREF} 21\\
ex_5 : 1853 &\xrightarrow{DDMIN} 72 \xrightarrow{HDD} 38 \xrightarrow{BFS} 38 \xrightarrow{LW} 38 \xrightarrow{CHANGEREF+} 19 \xrightarrow{CHANGEREF} 19
\end{aligned}
$$

The order in which we used the algorithms was deliberately chosen: From our experiments we knew that ddmin is the most effective algorithm so we started with it. Then we tried our hierarchical alternative HDD. We finished with changeref+ and changeref. We also tried to continue the loop over these algorithms but starting again with ddmin did not improve any of the results. The content of these 5 minimized BTOR files can be found in chapter A.3.

5 Implemented Delta Debugging algorithms in comparison

In this section we compare the performance of the implemented Delta Debugging algorithms on practical examples. From the experiments with the generator we arbitrarily chose 10 example BTOR files in the most recent BTOR format which caused Boolector (internal release version 2.907) to fail. The Delta Debugger was instructed to kill Boolector if it didn't succeed within 8 seconds. Further it was told to check every 100ms if the Boolector process is still running. Boolector was run with its default settings, no command line arguments specified. All experiments were done twice with simplification turned on and off. The comparison doesn't contain the actual running times. We rather show how often the Boolector executable was called while Delta Debugging a certain file. This measure is independent of the hardware.

The first two tables (5.1, 5.2) summarize the executable calls performed by each algorithm. Here changeref and HDD* performed the most calls with and without simplification. On average changeref+ performed less calls than changeref and HDD* needed more calls than HDD+. LW and ddmin were best with respect to number of calls. Turning off simplification reduced the number of calls for every algorithm, especially for HDD and HDD+, whereas changeref+ was not affected much by this.

Tables 5.3 and 5.4 show the initial versus final sizes of the files in nodes. The table rows are sorted ascendingly by average final node count. The best three algorithms in this category were HDD*, changeref and HDD+. With simplification turned on BFS produced smaller results than changeref+, whereas without simplification their order was reversed. The order of the other algorithms was not affected much by simplification, although with simplification the best algorithm was changeref rather than HDD*. Also with simplification on HDD*, BFS, changeref+ and HDD found smaller input configurations on average.

Another possibility of comparing the different algorithms is shown in tables 5.5 and 5.6. For each example file we calculated $100 \times (1 - \frac{size_{final}}{size_{initial}})$ and divided the result by

Algorithm	ex_1	ex_2	ex_3	ex_4	ex_5	$\frac{1}{10}\sum_{i=1}^{10} ex_i$
LW	59	65	57	51	54	57.9
DDMIN	394	138	64	65	166	128.2
BFS	267	319	247	359	640	399
CHANGEREF+	243	489	261	358	690	537
HDD	2972	214	172	493	4292	946
HDD+	3226	339	346	669	4553	1171
HDD*	4038	499	758	1184	4742	1487.4
CHANGEREF	2610	2331	1942	2767	2626	3397.3

	ex_6	ex_7	ex_8	ex_9	ex_{10}
LW	55	68	53	58	59
DDMIN	116	92	96	57	94
BFS	210	468	358	437	685
CHANGEREF+	470	1199	641	339	680
HDD	178	188	280	331	340
HDD+	317	468	421	481	890
HDD*	482	556	842	948	825
CHANGEREF	2149	6181	1563	6170	5634

Table 5.1: Algorithms compared by number of calls to executable. Simplification on.

Algorithm	ex_1	ex_2	ex_3	ex_4	ex_5	$\frac{1}{10}\sum_{i=1}^{10} ex_i$
LW	59	65	57	51	54	57.9
DDMIN	74	149	80	72	148	90.2
HDD	199	399	196	431	488	291.2
BFS	192	488	256	668	379	379.3
HDD+	335	630	335	542	671	476.3
CHANGEREF+	438	357	418	558	629	521.9
HDD*	892	2224	1010	686	1566	1275.3
CHANGEREF	2116	1696	1951	3724	2869	2900.4

	ex_6	ex_7	ex_8	ex_9	ex_{10}
LW	55	68	53	58	59
DDMIN	81	63	88	52	95
HDD	221	218	137	314	309
BFS	193	389	504	320	404
HDD+	382	459	261	637	511
CHANGEREF+	199	1566	535	286	233
HDD*	802	1338	765	1593	1877
CHANGEREF	2163	5487	2070	2183	4745

Table 5.2: Algorithms compared by number of calls to executable. Simplification off.

Algorithm	ex_1	ex_2	ex_3	ex_4	ex_5	ex_6	ex_7	ex_8	ex_9	ex_{10}	$\frac{1}{10}\sum_{i=1}^{10} ex_i$
	732	992	1687	1850	1853	1881	1981	2118	2219	2278	
HDD*	57	34	31	30	26	29	39	33	27	30	33.6
CHANGEREF	61	37	28	32	41	25	52	33	64	46	41.9
HDD+	60	33	37	37	62	30	42	32	28	121	48.2
BFS	51	66	49	69	125	33	78	77	73	101	72.2
CHANGEREF+	47	39	44	65	88	78	166	111	51	98	78.7
HDD	240	78	184	229	950	147	108	260	154	213	256.3
DDMIN	593	519	982	671	1414	1049	1496	634	819	1343	952
LW	731	977	1576	1685	1761	1648	1975	1801	2111	2058	1632.3

Table 5.3: Algorithms compared by final node count. Row 2 contains the initial sizes of the examples before Delta Debugging. Simplification on.

Algorithm	ex_1	ex_2	ex_3	ex_4	ex_5	$\frac{1}{10}\sum_{i=1}^{10} ex_i$
	732	992	1687	1850	1853	
CHANGEREF	38	44	28	33	41	38.2
HDD*	25	61	41	23	43	38.8
HDD+	32	52	35	26	50	43.1
CHANGEREF+	77	63	64	85	95	80.3
BFS	49	107	67	162	92	90
HDD	163	332	234	384	391	266.3
DDMIN	531	784	1175	691	1113	944.1
LW	731	977	1576	1685	1761	1632.3

	ex_6	ex_7	ex_8	ex_9	ex_{10}	
	1881	1981	2118	2219	2278	
CHANGEREF	26	61	28	54	29	
HDD*	20	42	33	54	46	
HDD+	35	44	32	77	48	
CHANGEREF+	33	196	91	58	41	
BFS	46	93	110	73	101	
HDD	201	201	207	307	243	
DDMIN	1137	1040	692	925	1353	
LW	1648	1975	1801	2111	2058	

Table 5.4: Algorithms compared by final node count. Row 2 contains the initial sizes of the examples before Delta Debugging. Simplification off.

Algorithm	ex_1	ex_2	ex_3	ex_4	ex_5	ex_6	ex_7	ex_8	ex_9	ex_{10}	$100 - \sqrt[10]{\prod_{i=1}^{10}(100 - ex_i)}$
DDMIN	0.048	0.346	0.653	0.980	0.143	0.363	0.266	0.730	1.107	0.437	0.508
HDD	0.023	0.431	0.518	0.178	0.011	0.516	0.503	0.313	0.281	0.267	0.304
BFS	0.348	0.293	0.393	0.268	0.146	0.468	0.205	0.269	0.221	0.140	0.275
CHANGEREF+	0.385	0.196	0.373	0.270	0.138	0.204	0.076	0.148	0.288	0.141	0.222
HDD+	0.028	0.285	0.283	0.146	0.021	0.310	0.209	0.234	0.205	0.106	0.183
HDD*	0.023	0.194	0.130	0.083	0.021	0.204	0.176	0.117	0.104	0.120	0.117
LW	0.002	0.023	0.115	0.175	0.092	0.164	0.004	0.282	0.084	0.164	0.111
CHANGEREF	0.035	0.041	0.051	0.036	0.037	0.046	0.016	0.063	0.016	0.017	0.036

Table 5.5: Algorithms compared using the quality measure $\frac{reduction[\%]}{call}$. Simplify on.

Algorithm	ex_1	ex_2	ex_3	ex_4	ex_5	ex_6	ex_7	ex_8	ex_9	ex_{10}	$100 - \sqrt[10]{\prod_{i=1}^{10}(100 - ex_i)}$
DDMIN	0.371	0.141	0.379	0.870	0.270	0.459	0.754	0.765	1.121	0.427	0.556
HDD	0.391	0.167	0.439	0.184	0.162	0.402	0.412	0.659	0.274	0.289	0.338
BFS	0.486	0.183	0.375	0.137	0.251	0.505	0.245	0.188	0.302	0.237	0.291
CHANGEREF+	0.204	0.262	0.230	0.171	0.151	0.493	0.058	0.179	0.341	0.421	0.251
HDD+	0.285	0.150	0.292	0.182	0.145	0.257	0.213	0.377	0.152	0.192	0.225
LW	0.002	0.023	0.115	0.175	0.092	0.164	0.004	0.282	0.084	0.164	0.111
HDD*	0.108	0.042	0.097	0.144	0.062	0.123	0.073	0.129	0.061	0.052	0.089
CHANGEREF	0.045	0.056	0.050	0.026	0.034	0.046	0.018	0.048	0.045	0.021	0.039

Table 5.6: Algorithms compared using the quality measure $\frac{reduction[\%]}{call}$. Simplify off.

	with simplification		without simplification	
Algorithm	avg(reduced nodes)	avg(calls)	avg(reduced nodes)	avg(calls)
LW	119.8	57.9	119.8	57.9
BFS	1679.9	399	1662.1	379.3
DDMIN	800.1	128.2	808	90.2
HDD	1495.8	945	1485.8	291.2
HDD+	1703.9	1171	1709	476.3
HDD*	1718.5	1487.4	1713.3	1275.3
CHANGEREF	1710.2	3397.9	1713.9	2900.4
CHANGEREF+	1673.4	537	1671.8	521.9

Table 5.7: DD-algorithms: overall performance

the number of calls performed by the algorithm. Obviously, the bigger the reduction an algorithm achieves, the better. By weighting the reduction percentage with the number of calls, we arrive at a single number which expresses the effectiveness of the algorithm. When calculating the average of the 10 examples, we have to bear in mind that the geometric mean needs to be applied, as our reduction rate is expressed in percent. The averages are shown in the last column of the tables. According to this measure, ddmin was the clear winner, followed by HDD and BFS.

Eventually table 5.7 sums up the overall performance of our algorithms. To choose a unique winner we have to define our most important criterion: If it is reduction then HDD* with simplification and changeref can be recommended. If we seek efficiency as defined before then ddmin should be our first choice. changeref+ is a good choice somewhere in between with a much higher reduction than ddmin, consuming only a little more time on average.

In general we can conclude from our experiments that simplification does not significantly change the results. Depending on the particular algorithm it may slightly improve results.

6 Future work

Concerning random generation of bit-vector logic there are some aspects of our generator that should be extended in future versions. These extensions could then be used as a basis for additional experiments.

- Include better DAG shaping support in the generator. The current implementation of the generator supports only limited ways of DAG shaping. For example imposing a limit of nodes per layer we could produce arbitrarily shaped DAGs.
- Investigate code coverage / branching behavior of Boolector on generated files in detail.
- Include support for model checking extensions in the generator (*next* and *anext*)
- Measure quantitatively how satisfiability is influenced by bit-vector logic, taking into account also the shape of the DAG.

Possible improvements of the Boolector Debugger:

- Extended the Boolector Debugger so that results of tests are cached and further calls with the same configuration can be avoided. This could be realized by storing the already tested files in the file system, of course this would mean to accept a slight performance penalty. The penalty could be kept small by hashing the already tested files with SHA1.
- Find a way to decrease bit-widths in a reduced BTOR file. So far our Delta Debugging efforts only focused on decreasing the size of the file by reducing the number of nodes. At the end of this process we could also try to decrease the bit-widths and array-lengths involved in the file to find a truly minimal configuration according to the BTOR grammar. During this process we could also attempt to remove superfluous negations. A sub-optimal procedure performing this task could easily become very complicated as it has to preserve syntactical correctness. If we allow syntactically incorrect inputs one would need to compensate this by performing a significantly higher number of calls to the executable.

7 Summary

In this work we presented a way to generate random bit-vector logic with arrays. Our approach is to group node types with similar constraints (bit-width and array-length in our case) to reduce complexity. We start by generating the leaf nodes and build the DAG on top of this layer.

We found a way to influence the satisfiability of generated files. Our solution is based on generating a CNF on top of the bit-vector logic. By performing experiments with the generator in connection with Boolector we could show that the probability of an unsatisfiable case increases when more bit-vector logic is created for a fixed number of leaf nodes.

In the second part of the text we discussed Delta Debugging, which is essentially a search problem, which can be solved using several existing methods.

We described algorithms from the literature and proposed some new algorithms specifically designed for the BTOR format, which in theory could also be applied to similar input formats with a little effort.

Our main contribution are the changeref and changref+ algorithm, the latter having an asymptotic running time of $O(n \log n)$ on a DAG with n nodes. In our comparison of algorithms changeref was shown to be able to reduce bit-vector logic in size even further than other algorithms.

Regarding Delta Debugging as a search problem, the algorithms involved have already undergone a thorough evaluation in the past and thus there is not much left to be improved in the most general case (on character level). However given a concrete input format, BTOR in our case, there are still problems to be solved in order to apply the existing general Delta Debugging solutions to the particular input format effectively.

From a practical point of view we conclude that the development of our random generator and Delta Debugging tools for the BTOR format has paid off in the long run, as many bugs in Boolector could be found long before the first official release.

Bibliography

[1] Robert Brummayer and Armin Biere *Lemmas on Demand for the Extensional Theory of Arrays* SMT 2008 6th International Workshop on Satisfiability Modulo Theories, Princeton, New Jersey, USA, 2008.

[2] Robert Brummayer, Armin Biere and Florian Lonsing *BTOR: Bit-Precise Modeling of Word-Level Problems for Model Checking* BPR 2008 1st International Workshop on Bit-Precise Reasoning, Princeton, New Jersey, USA, 2008.

[3] Ghassan Misherghi, Zhendong Su *HDD: Hierarchical Delta Debugging* In Proceedings of the 28th International Conference on Software Engineering ICSE 2006, Shanghai, China, 2006.

[4] Andreas Zeller and Ralf Hildebrandt *Simplifying and Isolating Failure-Inducing Input* IEEE Transactions on Software Engineering, Vol. 28(2), 2002.

[5] Andreas Zeller *Isolating Cause-Effect Chains from Computer Programs* In Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, pages 1-10, Charleston, South Carolina, USA, 2002.

[6] Makoto Matsushita, Masayoshi Teraguchi, Katsuro Inoue *Effective Testing Debugging Methods and Its Supporting System with Program Deltas* In Proceedings of the International Symposium on Principles of Software Evolution ISPSE 2000, Kanazawa, Japan, 2000.

[7] Oliver Dagenais, Dwaight Deugo *TODD: Test-Oriented Development and Debugging* In Proceedings of the Fifth International Conference on Software Engineering Research, Management and Applications SERA 2007, Busan, Korea, 2007.

[8] Andreas Zeller *Automated Debugging: Are We Close?* Computer Vol. 34 Issue 11, pages 26-31, 2001.

[9] Andreas Zeller *Why Programs Fail: A Guide to Systematic Debugging* Morgan Kaufmann, ISBN 1558608664, 2005.

[10] David Mitchell, Bart Selman, Hector Levesque *Hard and Easy Distributions of SAT Problems* In Proceedings of the Tenth National Conference on Artificial Intelligence AAAI-92, San Jose, California, USA, 1992.

[11] Stephen A. Cook *The Complexity of Theorem-Proving Procedures* In Proceedings of the 3rd Annual ACM Symposium on Theory of Computing STOC 1971, pages 151-158, Ohio, USA, 1971.

[12] Donald E. Knuth *The Art of Computer Programming 3. Sorting and Searching* Vol. 3, first edition, ISBN 0-201-03803-X, 1973

[13] Z3, An efficient SMT solver, Microsoft Research, `http://research.microsoft.com/projects/Z3/`

[14] Wine, Wine Is Not an Emulator, `http://www.winehq.org/`

[15] Graphviz, Graph Visualization Software, `http://www.graphviz.org/`

All mentioned URLs were checked on October 5th, 2008.

A The Boolector Debugger application

The Boolector Debugger application is the graphical front-end of the Delta Debugger and generator. It is written in Java and can be used to visually inspect BTOR files. Visualization is based on dot [15].

A.1 Installation

1. Make sure your build environment works (GCC, make...).

   If necessary install graphviz and Java6

2. Compile Boolector, Picosat

   Suppose you have boolector.tgz and picosat.tgz files in your current directory:

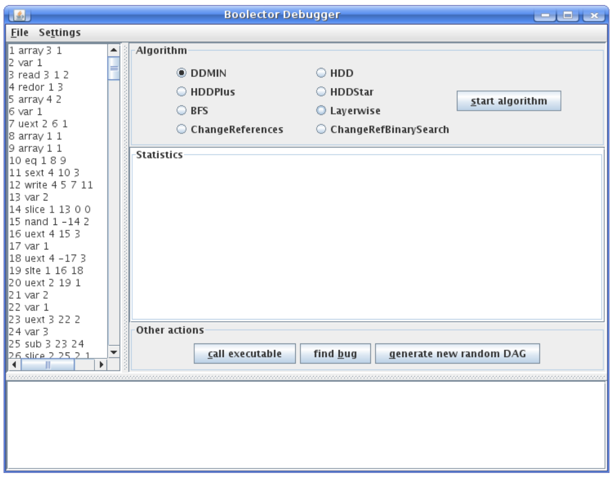
   ```
   $ tar -xzf boolector.tgz
   $ tar -xzf picosat.tgz
   $ cd picosat
   $ ./configure
   $ ./make
   $ cd ../boolector
   $ ./configure
   $ ./make
   ```

3. Copy bd.jar file to where you want it.

4. Start bd.jar

   ```
   $ java -jar bd.jar
   ```

   After this call the main window should appear on the screen (see figure A.1).



```
1 array 3 1
2 var 1
3 read 3 1 2
4 redor 1 3
5 array 4 2
6 var 1
7 uext 2 6 1
8 array 1 1
9 array 1 1
10 eq 1 8 9
11 sext 4 10 3
12 write 4 5 7 11
13 var 2
14 slice 1 13 0 0
15 nand 1 -14 2
16 uext 4 15 3
17 var 1
18 uext 4 -17 3
19 slte 1 16 18
20 uext 2 19 1
21 var 2
22 var 1
23 uext 3 22 2
24 var 3
25 sub 3 23 24
26 slice 2 25 2 1
```

Figure A.1: The main window.

5. Set up the options

   Select `settings->options...` from the main menu. The options dialog window pops up as shown in figure A.2.

   a) Fill in path to the Boolector executable file
   b) Fill in path to the dot executable

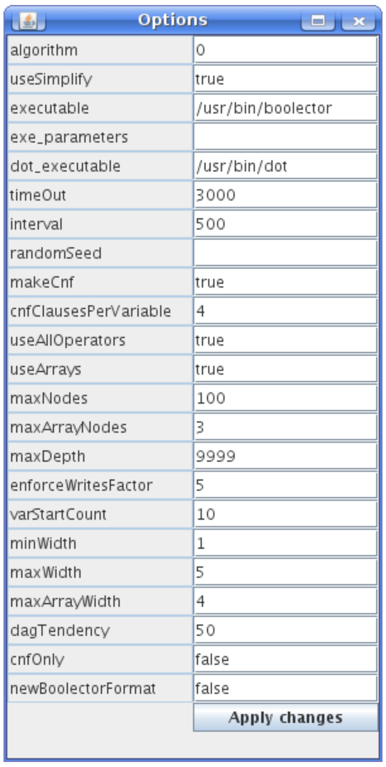
   The settings are saved to a file called `options.ini`. The file is a human readable property file. Each line contains a *key* = *value* pair. It has to be located in the directory from which java is invoked (pwd).

A.2 User interface of Boolector Debugger

BTOR files can be opened and saved using the `File->Open` and `File->Save` menu items. To run the selected executable on the currently opened BTOR file simply click on the `call executable` button. The executable's standard output is being written to the console (as shown in figure A.3).

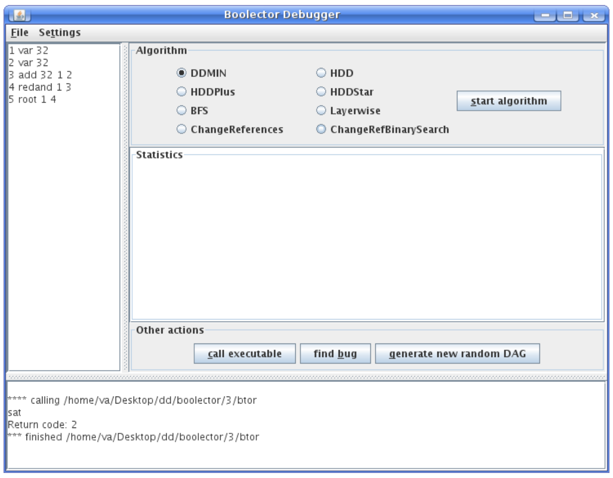
A.2.1 Graphical versus plain text view

The currently opened BTOR file is shown in the upper left corner of the main window. It can be shown as text or as a graphic (see figure A.4). The view can be toggled by selecting `Settings->View text / view graph` from the main menu.



algorithm	0
useSimplify	true
executable	/usr/bin/boolector
exe_parameters	
dot_executable	/usr/bin/dot
timeOut	3000
interval	500
randomSeed	
makeCnf	true
cnfClausesPerVariable	4
useAllOperators	true
useArrays	true
maxNodes	100
maxArrayNodes	3
maxDepth	9999
enforceWritesFactor	5
varStartCount	10
minWidth	1
maxWidth	5
maxArrayWidth	4
dagTendency	50
cnfOnly	false
newBoolectorFormat	false

Figure A.2: The options dialog window.



```
1 var 32
2 var 32
3 add 32 1 2
4 redand 1 3
5 root 1 4
```

Algorithm: [✓ DDMIN] HDD HDDPlus HDDStar BFS Layerwise ChangeReferences ChangeRefBinarySearch

```
**** calling /home/va/Desktop/dd/boolector/3/btor
sat
Return code: 2
*** finished /home/va/Desktop/dd/boolector/3/btor
```

Figure A.3: After the “call executable” button has been clicked the console shows the standard output of the process (Boolector in this case).

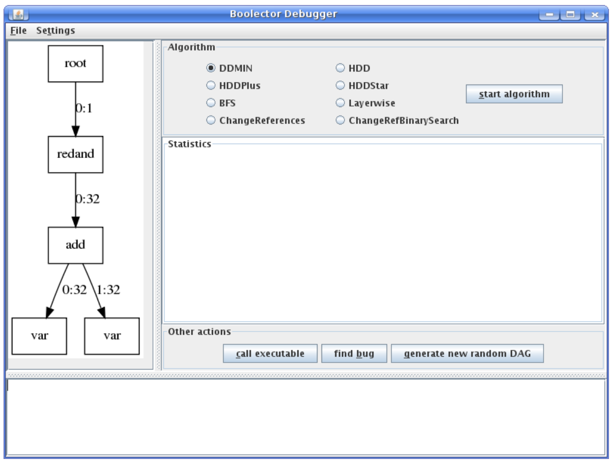


Figure A.4: Main window: graphical view of a BTOR file

A.2.2 Generating BTOR files

Check if your generation settings are correct in the options dialog window. Click the `generate new random DAG` button. The new graph will appear shortly afterwards in the text/graph view.

To generate BTOR files using the command line only, open a terminal and type:

```
$ java -cp bd.jar fmv.main.Generator -?
numeric parameters: -s -arrays -n -d -clauses -writes -vars
-minwidth -maxwidth -arrmaxwidth -arrays -dag
boolean parameters: -cnf -all -noarrays -newformat
$
```

The names of the possible generation options are listed. Numeric parameters need an additional integer argument (e.g. “-n 3”), whereas boolean parameters can stand on their own. To generate a BTOR file using the standard options simply omit the “`-?`”.

A.2.3 Running a Delta Debugging algorithm

Open a BTOR file. Select an algorithm by clicking on one of the radio buttons in the Algorithm section, then click the `start algorithm` button.

You can stop the Delta Debugging process at any time by clicking the `stop...` button. During Delta Debugging all other menu items are disabled.

Alternatively you can use the command line:

```
$ java -cp bd.jar fmv.main.DeltaDebugger -?
usage: DeltaDebugger [-timeout <nr>] [-interval <nr>]
       [<algorithm>] [-exe <boolector>]
       [-newformat] inputfile outputfile
       <algorithm> can be one of: -lw -bfs -ddmin
                   -hdd -hddplus -hddstar -cref -cbref
       <boolector> path of boolector executable and parameters
$
```

The self-explanatory text on the screen describes the usage of the Delta Debugger.

A.2.4 Bug search mode

This mode can be used to start the generator repeatedly using the current settings. The starting random seed is taken from the options and is incremented by one after each run of the generator. After a file has been generated, the Boolector executable is run on it and its output is parsed for indication of a failed assertion or segmentation fault. If Boolector fails the random seed is printed to the console. Bug search mode can be stopped any time by simply clicking the `stop` button. At this point no automatic Delta Debugging is triggered when bugs are found. This behavior should be improved in future versions, but the same task can already be performed by simple scripts.

A.3 Minimized BTOR files

Listing A.1: ex1-allmin.btor

```
1 array 3 2
2 zero 2
3 zero 3
4 write 3 2 1 2 3
5 udiv 3 3 3
6 write 3 2 1 2 5
7 eq 1 4 6
8 zero 1
9 implies 1 7 8
10 var 1
11 acond 3 2 -10 4 1
```

```
12 read 3 11 2
13 redxor 1 12
14 eq 1 6 1
15 xnor 1 13 14
16 nand 1 9 15
17 root 1 16
```

Listing A.2: ex2-allmin.btor

```
1 var 3
2 var 3
3 uaddo 1 -1 2
4 array 2 3
5 var 3
6 var 2
7 write 2 3 4 -5 6
8 read 2 7 1
9 redxor 1 8
10 write 2 3 4 1 6
11 read 2 10 5
12 redor 1 11
13 xor 1 9 12
14 write 2 3 7 -1 6
15 write 2 3 4 5 6
16 eq 1 14 15
17 xnor 1 13 16
18 xnor 1 3 17
19 var 1
20 write 2 3 7 2 6
21 acond 2 3 19 15 20
22 read 2 21 1
23 redxor 1 22
24 xor 1 18 23
25 read 2 4 1
26 redor 1 25
27 or 1 24 26
28 root 1 27
```

Listing A.3: ex3-allmin.btor

```
1 var 1
2 array 2 1
3 zero 1
4 zero 2
5 write 2 1 2 3 4
6 acond 2 1 1 2 5
7 read 2 6 3
8 redor 1 7
9 var 2
10 write 2 1 2 3 -9
11 var 1
12 write 2 1 10 -11 4
13 var 1
14 write 2 1 2 -13 4
15 eq 1 12 14
16 xor 1 8 15
17 root 1 16
```

Listing A.4: ex4-allmin.btor

```
1 var 1
2 array 4 1
3 zero 1
4 zero 4
5 write 4 1 2 3 4
6 write 4 1 5 1 4
7 acond 4 1 -1 5 6
8 var 4
9 write 4 1 2 3 -8
10 eq 1 9 2
11 read 4 7 10
12 redxor 1 11
13 var 1
14 write 4 1 2 -13 4
15 read 4 14 3
16 redxor 1 15
17 and 1 12 16
18 eq 1 2 5
19 xnor 1 18 3
```

```
20 nand 1 17 19
21 root 1 20
```

Listing A.5: ex5-allmin.btor

```
1 var 1
2 array 1 2
3 var 2
4 var 1
5 write 1 2 2 3 4
6 var 1
7 write 1 2 2 3 6
8 acond 1 2 1 5 7
9 read 1 8 3
10 eq 1 7 2
11 xnor 1 4 10
12 eq 1 5 2
13 zero 2
14 write 1 2 2 13 4
15 read 1 14 3
16 xnor 1 12 15
17 xor 1 11 16
18 xor 1 9 17
19 root 1 18
```

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am 13. Oktober 2008

Andreas Vida