

---

# NO-GOOD MODEL EXISTENCE, A GLIMPSE OF ASP SOLVING

---

Programming on parallel architectures

Santi Enrico  
University of Udine  
Academic year 2022/23

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	ASP and the no-goods . . . . .	3
<b>2</b>	<b>Problem presentation</b>	<b>3</b>
2.1	Reduction to SAT . . . . .	3
2.2	The algorithm outline . . . . .	4
2.3	Outline of the next sections . . . . .	5
<b>3</b>	<b>A serial implementation</b>	<b>6</b>
3.1	The data structures . . . . .	6
3.2	The methods . . . . .	7
3.3	Input format . . . . .	8
3.4	Some test results . . . . .	9
<b>4</b>	<b>A CUDA implementation</b>	<b>11</b>
4.1	The kernels . . . . .	11
4.2	The (current) final version . . . . .	16
4.3	Some test results . . . . .	16
4.4	Some remarks . . . . .	19
<b>5</b>	<b>Test generation and testing</b>	<b>20</b>
<b>6</b>	<b>Conclusions</b>	<b>21</b>
6.1	Further developments . . . . .	21

# 1 Introduction

Logic programming is a programming paradigm in contrast with the declarative one. In logic programming we write a set of rules the output of our algorithm must satisfy, but we don't specify how to find such output, nor we know if such output exists. Indeed we aren't describing how the computation will proceed, since the computation is carried out by the declarative programming system (usually referred as *solver*) using the process of automated reasoning [1].

Under the umbrella of the declarative paradigm fall many different approaches, in this very brief introduction we focus on ASP in particular.

## 1.1 ASP and the no-goods

Answer set programming or ASP is a logic programming approach in which the semantics of the solution is based on the definition of stable model or answer set. The goal of a solver in this case is to find (whether they exists) the answer sets for a given program [2]. The solvers are thus a fundamental part of the logic paradigm and they usually employ sophisticated search algorithms which in the ASP case are improvements of DPLL (such as CDCL) that were originally used in SAT solvers [3]. In a nutshell CDCL utilizes two techniques that improve it over DPLL, backjumping and clause learning. This second technique is based on the concept of *no-goods*. CDCL indeed uses the no-goods, which are sets of literals that can't be all satisfied at the same time, to store the results of the inference done and uses them as constraints to derive new knowledge (i.e. more constraints restrict the search space).

## 2 Problem presentation

The problem we aim to solve (and thus the goal of the next sections) is to implement an algorithm that:

*Given a set of tuples (no-goods) find whether an assignment exists for the variables such that no no-good is satisfied (i.e. the assignment doesn't make true all literals in the no-good).*

Such algorithm could be used in ASP to detect if the inference done on a given assignment for a formula results unsatisfiable, allowing a faster exploration of new parts of the search space.

### 2.1 Reduction to SAT

The presented problem can be reduced to SAT since each no-good can be viewed as a conjunction which we don't want to satisfy. First we can put the no-goods in disjunction with each other since we want none of them to be satisfied. Up to this point we can see the new problem as finding an assignment for a DNF formula which falsifies it. By applying De Morgan laws such problem can be turned into a CNF formula we want to satisfy, thus a SAT instance.

For example consider the set of no-goods  $\phi = \{\{-a, b, c\}, \{c, a, \neg d\}\}$ , solving the problem above for such set means finding an assignment falsifying  $\phi' = (\neg a \wedge b \wedge c) \vee (c \wedge a \wedge \neg d)$ . Which in turn it means finding an assignment satisfying  $\phi'' = \neg\phi' = (a \vee \neg b \vee \neg c) \wedge (\neg c \vee \neg a \vee d)$ .

## 2.2 The algorithm outline

The general idea of the algorithm implemented in the next sections follows DPLL while leaving room for implementing future improvements such as backjumping, clause learning and other techniques used in CDCL. We now show a high level pseudocode for DPLL[4] and the abstract idea of the algorithm we are going to implement.

```
DPLL ( $\phi$ ):
  //unit propagation
  while  $\phi$  has a clause of length 1:
    assign to the literal T/F according to its sign1 and
    propagate;
  //pure literal elimination
  while  $\phi$  has a monotone/pure literal:
    assign to the literal T/F according to its sign;
  // stopping conditions:
  if  $\phi$  is empty:
    return SATISFIABLE;
  if  $\phi$  contains an empty clause:
    return UNSATISFIABLE;
  //otherwise we choose an assignment for a literal:
  l = chooseLiteral( $\phi$ );
  return DPLL( $\phi[l=T]$ ) or DPLL( $\phi[l=F]$ );
```

The code that will be implemented in the next sections follows a similar idea<sup>2</sup> to the code shown below.

```
//the main:
noGoodSAT ( $\phi=\{\phi_1, \phi_2 \dots\}$ ):
  partial assignment:  $\eta$ ;
  noGood Set:  $\phi'=\phi$ ;
  //pure literal elimination update ( $\eta$  and  $\phi'$ )
  pureLiteralCheck( $\phi', \eta$ );
  //unit propagation update ( $\eta$  and  $\phi'$ )
  if unitPropagation( $(\phi', \eta) == \text{CONFLICT}$ )
    return UNSATISFIABLE;
  l = chooseLiteral( $\phi'$ );
  return solve( $\phi', \eta, l, \text{TRUE}$ ) or solve( $\phi', \eta, l, \text{FALSE}$ );
solve ( $\phi'=\{\phi'_1, \phi'_2 \dots\}$ , partial assignment:  $\eta$ , var to assign: x,
value v):
  updateAssignmentAndNoGoods( $\phi', \eta, x, v$ );
  pureLiteralCheck( $\phi', \eta$ );
  learnClauses();
  if (unitPropagation( $(\phi', \eta) == \text{CONFLICT}$ ))
    RETURN UNSATISFIABLE;
  if (allNoGoodSatisfied( $(\phi', \eta)$ ))
    RETURN (SATISFIABLE,  $\eta$ );
  if (allvariablesAssigned( $\eta$ ))
    RETURN UNSATISFIABLE;
  l = chooseLiteral( $\phi'$ );
  return solve( $\phi', \eta, l, \text{TRUE}$ ) or solve( $\phi', \eta, l, \text{FALSE}$ );
```

While the code structure resembles a bit a CDCL approach the actual implementation would just provide the shell to future CDCL implementations since the actual content of the methods will simply be based on DPLL<sup>3</sup>: The main reference used for the CDCL structure of the algorithm above is [5], the algorithm has been slightly modified, among these modification it has been made recursive.

<sup>1</sup>Assign T if the literal appears without  $\neg$  in front, F otherwise.

<sup>2</sup>The name and the purpose of the methods resemble those of the methods actually implemented.

<sup>3</sup>e.g. there's no backjump, instead backtracking is used and the clause learning procedure is empty.

## 2.3 Outline of the next sections

The next sections are organized as follows:

- In Section 3 a serial implementation of the algorithm described above will be presented alongside some tests that measure its performance.
- In Section 4 a CUDA (parallel) version of the algorithm will be presented and some tests done will be shown.
- Section 5 will briefly discuss how the test files have been generated and how it's possible to use the provided script to do further testing.
- In Section 6 some comparisons between the two implementations will be presented and some conclusions will be drawn, leaving room for possible future improvements.

### 3 A serial implementation

Let's now see in detail how the algorithm presented in 2.2 has been implemented in C. First we introduce the data structures used then the methods and the general flow of the computation.

#### 3.1 The data structures

The data structures used in the program can be subdivided into two main categories, the *global* ones, which are remain the same<sup>4</sup> during the whole search process (i.e. we have one instance of the structure throughout the whole computation) and the *local* ones. This latter kind of data is present in multiple copies (in the activation record of the `solve` method) and allows an easier backtracking<sup>5</sup>.

The *global* structures are `int noVars`, which throughout the whole execution remains constant (after the initialization) to the number of variables in the model and `int noNoGoods` which keeps the number of clauses in the model. There are a few more *global* variables used to store whether a solution has been found and to check whether we have to stop after one solution is found or carry on with the computation.

The *local* structures are all enclosed in a struct of type `NoGoodData`.

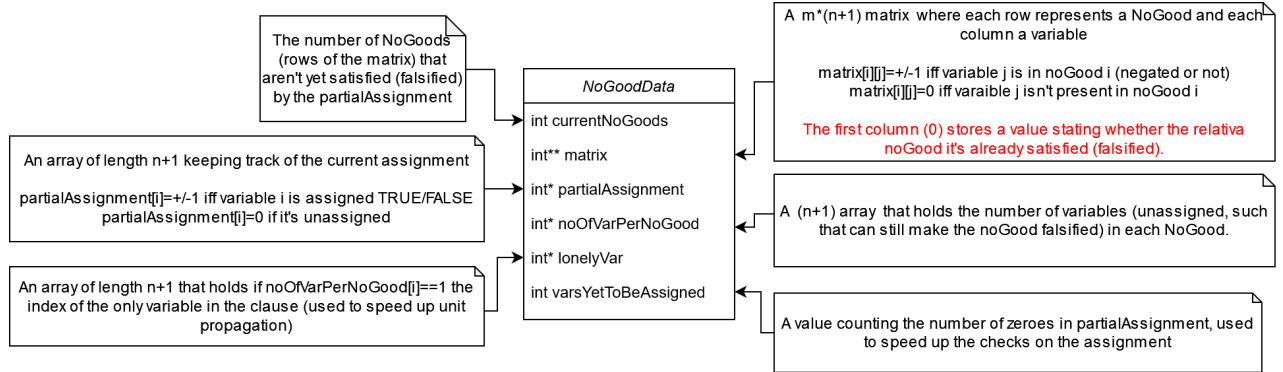


Figure 1: A schematization of the `NoGoodData` struct ( $n$  indicates the number of variables,  $m$  the no-goods).

In a later development stage an integer array `varsAppearingInRemainingNoGoodsPositiveNegative` has been added. Such variable is used to improve the selection of the variable to assign when a choice has to be made, while also improving the pure literal check procedure. For what concerns the former, `varsAppearingInRemainingNoGoodsPositiveNegative[i]` (for  $i \in [1, noVars]$ ) keeps track of how many cubes (no-goods yet unsatisfied) still contain a (negative) literal related to variable  $i$ . While `varsAppearingInRemainingNoGoodsPositiveNegative[i+noVars]` (for  $i \in [1, noVars]$ ) keeps track of the cubes (no-goods yet unsatisfied) containing a positive literal of the variable  $i$ . The array has size  $2*(noVars + 1)$  since we store separately the negative and the positive occurrences (first the negative and then the positive ones, to ease the access via an index without the need of an if statement). For what concerns the improvement over the variable choice, the idea is

<sup>4</sup>The structure is the same but their content can change.

<sup>5</sup>Still the dynamic data structures inside it must be duplicated explicitly each time.

that we want to assign variables (currently unassigned) **that are still present** in some no good not yet satisfied (falsified)<sup>6</sup>.

### 3.2 The methods

The main<sup>7</sup> methods used in the program are the following:

- `int readFile_allocateMatrix(const char *, struct NoGoodData*)` is the first method invoked by the main procedure, it's used to instantiate all the necessary dynamic (local) structures, read the model from the file specified by the first parameter and initialize the instantiated variables accordingly. It returns -1 if the file doesn't exist or it can't be opened, 0 otherwise.
- `bool solve(struct NoGoodData, int, int)` is the core of the solver, takes in input the struct containing the model description at the  $i - th$  computation step, a variable identifier (of an unassigned variable) and the truth value to associate to that variable. After performing the assignment, it updates the necessary data structures, calls `pureLiteralCheck`, `learnClause` and `unitPropagation`, it checks whether the assignment caused a conflict (if so we backtrack returning false) then checks also if there still exist no-goods yet to satisfy or we have variables left to assign. In the first case if all no goods are marked as "SATISFIED" then the current assignment satisfies the original problem (i.e. falsifies each no good) and thus we return true. For what concerns the latter check if no variables are left unassigned we can't hope to satisfy the remaining no goods (since we know that by not having returned true at the previous there are still unsatisfied cubes) thus we return false and we backtrack. If we didn't match any of the conditions above, it means we still have variables to be assigned and no-goods to satisfy thus we continue by choosing a variable to assign and we recursively call `solve` with both possible truth values.
- `int unitPropagation(struct NoGoodData*)` is the procedure responsible for carrying out the unit propagation algorithm, it returns a defined constant `CONFLICT` or `NO_CONFLICT` according to whether the propagation of the new value assigned in `solve` caused a no-good to be unsatisfiable<sup>8</sup> (this method uses `removeLiteralFromNoGoods`).
- `void pureLiteralCheck(struct NoGoodData*)` checks for each unassigned variable whether in the remaining model it only appears in the positive or negated form, if so it assigns the proper value in order to falsify (i.e. set to `SATISFIED`) the no-goods in which such variable is present. By remaining model we mean the set of no-goods currently unsatisfied, since a satisfied no-good even if it contained the variable we are looking at, won't be affected by its possible assignment.
- `void removeNoGoodSetsContaining(int***, int*, int**, int*, int, int)` it marks as `SATISFIED` (or "removes") from the matrix of the model (the first argument) the no-goods which contain the literal (fifth argument) with the sign specified as sixth

<sup>6</sup>There's a bit of dualism in the algorithm, we refer to *SATISFIED* no-goods as the ones we don't need to keep track anymore (thus a cube already falsified).

<sup>7</sup>We exclude secondary auxiliary submethods, methods used for output purposes or memory management.

<sup>8</sup>i.e. we have assigned all variables present in the no-good, and all their values match the literal sign, thus there's no way to assign a variable and falsify the cube.

argument. It also and decreases the number of currently “non satisfied no-goods” (second argument). To conclude it updates the third argument, namely `varsAppearingInRemainingNoGoodsPositiveNegative`.

- `int chooseVar(int *, int *)` returns the index of the variable to assign at the current `solve` step. It returns the index of the first unassigned variable present in at least one of the unsatisfied no-goods (this is done by using as second argument `varsAppearingInRemainingNoGoodsPositiveNegative`).
- `void learnClause()` is an empty function which in the future could be used (alongside `backjump`) to shift the algorithm towards a more CDCL approach.
- `int removeLiteralFromNoGoods(struct NoGoodData* data, int,int)` modifies the struct by removing the literal (second argument) from the currently unsatisfied no-goods when they contain the relative variable and it’s assigned a value which goes along<sup>9</sup> with its sign (third argument). Returns `CONFLICT` whether the removal causes a no-good to become empty. `NO_CONFLICT` is returned otherwise.

### 3.3 Input format

As described above, the main procedure for reading the model takes in input a file path from which load the instance (i.e. set of no-goods). Such file must satisfy a particular format (similar to DIMACS) in order to be properly read by the program.

The file can start with a series of lines of comments which start with the character `c`<sup>10</sup>. Then the following line must be present: `p nogood <n> <m>`. Such line indicates how many variables (labelled from 1 to `<n>`)<sup>11</sup> and how many no-goods (`<m>`) the model will contain. After such line the list of no-goods can start. Each no-good is represented by a sequence of numbers (literals) each belonging to `[-<n>, <n>]\{0}` where a negative number represents the negation of the variable represented by the modulo of the number. Every no-good must terminate with a 0 and a line termination character (i.e. we have one no-good per line). An example of instance is the following:

```
c this is a comment
c also a comment
p nogood 4 6
1 -2 3 0
-3 4 0
-1 -3 2 0
-4 1 0
1 0
3 4 0
```

<sup>9</sup>i.e. the truth value true and a positive literal or vice versa (we remove literals that don’t falsify the cube).

<sup>10</sup>Note that the comments can only be at the beginning of the file, they can’t be interleaved with the actual model.

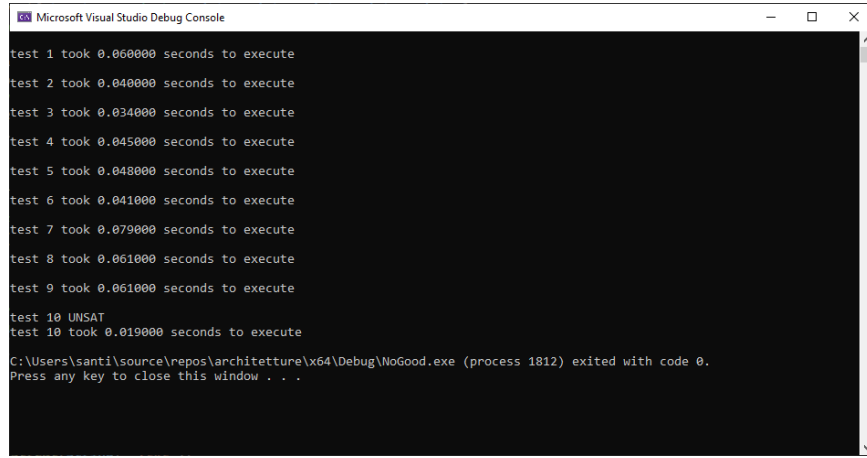
<sup>11</sup>Note that `p nogood` must be written at the beginning of the line and without more than one space character in between. This condition must be followed since the program doesn’t implement an actual parser but just skips the first eight characters of the first line after the comments.



### 3.4 Some test results

The original test carried out can be subdivided into four categories according to the size of the model: small, medium, big and huge (in later stages another set of bigger tests was introduced). Each test category has its own folder<sup>12</sup> and in Section 5 we will see how these have been generated. For reference the small folder contains 30 models with the number of variables ranging from 20 to 40 and up to 80 no-goods. Meanwhile the huge folder contains 5 models with 500 to 1000 variables and with 1500 to 3000 no-goods.

All the test cases provided in the small folder run in almost no time (less than tens milliseconds) on the machine used<sup>13</sup>. Also the other test ran quite fast, the runtimes do go up a bit (still less than 100ms) for the tests in the big folder, while for the huge folder the time goes up to a couple hundred of ms. All the models tested (except the *huge* ones) run in



```
Microsoft Visual Studio Debug Console

test 1 took 0.060000 seconds to execute
test 2 took 0.040000 seconds to execute
test 3 took 0.034000 seconds to execute
test 4 took 0.045000 seconds to execute
test 5 took 0.048000 seconds to execute
test 6 took 0.041000 seconds to execute
test 7 took 0.079000 seconds to execute
test 8 took 0.061000 seconds to execute
test 9 took 0.061000 seconds to execute
test 10 UNSAT
test 10 took 0.019000 seconds to execute

C:\Users\santi\source\repos\architettura\x64\Debug\NoGood.exe (process 1812) exited with code 0.
Press any key to close this window . . .
```

Figure 2: The times of the tests of the *big* folder carried out on the Windows machine specified.

almost no time (i.e. from 0ms to 100ms) on MiniSat<sup>14</sup>. The comparison between the results of program developed and MiniSat has been used to discover bugs and improve a bit the reliability of the program. How the DIMACS models (counterparts of the no-good models) for MiniSat have been generated will be discussed in Section 5 .

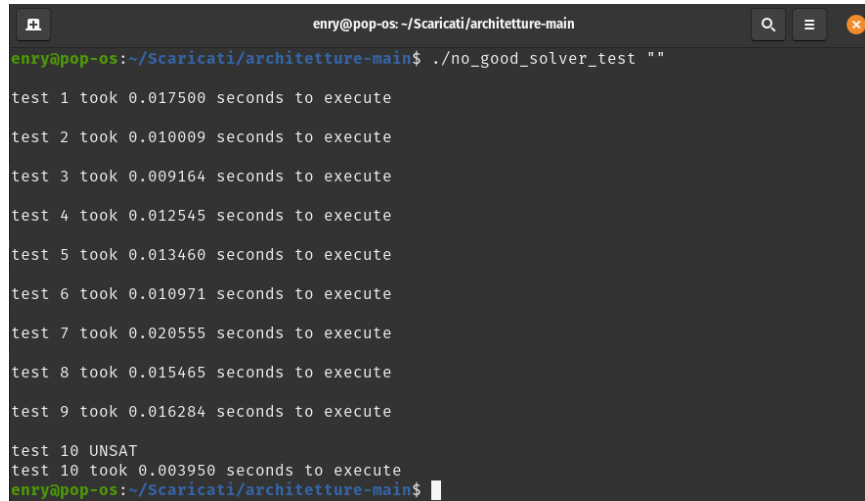
During testing an interesting phenomena showed up, on some instances the program ran better on a Linux machine with a GT920m and 4GB of DDR3 (1600MHz) RAM rather than on the more powerful desktop configuration (mentioned above) using Windows<sup>15</sup>. The same phenomena will be noticed on some tests for what concerns the CUDA implementation of the algorithm compiled via NVCC. The differences in times was less significative (but still present) for the *huge* tests. One possible explanation is that on Visual Studio the program was running under “debug mode” (still without any breakpoint), thus the time increment may be due to the debugger configuration and launch.

<sup>12</sup><https://github.com/EnrSanti/architettura/>

<sup>13</sup>A Desktop computer with a Ryzen3 1200, 8GB of DDR4 (2400MHz) ram and a GTX1050).

<sup>14</sup>Which uses far more advanced techniques such as restart and counterexample learning [6]

<sup>15</sup>On Linux the code was compiled via GCC, while on Windows Visual studio 2020 had been used, which at least for standard C uses a different compiler than GCC.

A terminal window titled 'enry@pop-os: ~/Scaricati/architettura-main'. The prompt is 'enry@pop-os:~/Scaricati/architettura-main\$'. The command entered is './no\_good\_solver\_test ""'. The output shows the execution time for 10 tests. Tests 1 through 9 are successful, and test 10 is 'UNSAT'.

```
enry@pop-os:~/Scaricati/architettura-main$ ./no_good_solver_test ""
test 1 took 0.017500 seconds to execute
test 2 took 0.010009 seconds to execute
test 3 took 0.009164 seconds to execute
test 4 took 0.012545 seconds to execute
test 5 took 0.013460 seconds to execute
test 6 took 0.010971 seconds to execute
test 7 took 0.020555 seconds to execute
test 8 took 0.015465 seconds to execute
test 9 took 0.016284 seconds to execute
test 10 UNSAT
test 10 took 0.003950 seconds to execute
enry@pop-os:~/Scaricati/architettura-main$
```

Figure 3: The times of the tests of the *big* folder carried out on the Linux machine specified.

## 4 A CUDA implementation

Let's now see how the parallel version of the algorithm has been implemented.

While the overall idea remains the same many methods changed and are now intended to run on the device. Unfortunately the algorithm itself isn't much parallelizable in the classical sense of the term and for this reason the device capabilities have been used in a *serial way*, i.e. first computing the whole counterpart of the pure literal check and then the whole unit propagation.

On the other hand, while such methods are still been executed one after the other (the same is true for the subprocedures `removeNoGoodSetsContaining` and `removeLiteralFromNoGoods`) they are now executed as kernels on the device, meaning that many cores execute them on a different part of the data. One of the reasons for which such *serialization* of the parallel parts is needed it's the data itself, i.e. `removeNoGoodSetsContaining`, `pureLiteralCheck` and `removeLiteralFromNoGoods` need the data to be coherent and updated, thus they can't be easily executed in different orders or in parallel with one another.

We will now present the structure of the code (and the subsequent execution flow) which was used as an (advanced) starting point for some of the optimizations presented in subparagraph 4.2 which describes the current state of the parallel algorithm. The code structure and kernel description is presented in Subparagraph 4.1 while the execution flow can be seen in Image 4. Already from the image we can see that the computation is heavily based on the device, while leaving just few and simple tasks to run on the host. The reason for this is that during the early stages of the development it has been seen that wasn't worth it to make the host execute (in less time) some of the serial parts because that would have required two copies of the data (one on the device and one on the host) to remain synchronized at each iteration step (i.e at call of the solve method). Indeed not only the little time saved by exploiting the compute capabilities of the host on the serial part of the code was lost during the copies from host to device and vice versa, but such copies required more time than the serial part being executed by only one kernel on the device.

By understanding this, the idea was then to keep all the updated data on the device (also storing the copies of the data used for backtracking in global memory) and copy at each solve call just the few bytes needed on the host (which still maintains a data structure even though it's not synchronized with the one on the device<sup>16</sup>).

### 4.1 The kernels

Some of the methods found in 3.2 are similar to their counterparts in the CUDA implementation<sup>17</sup>. On the other hand new functions and kernels have been introduced and also previous host functions in 3.2 have now become kernels themselves. We now describe all the kernels implemented:

- `pureLiteralCheck(int*, int*, int*, int*, int *)`: It's responsible for checking whether an unassigned variable is present in the unsatisfied no-goods appearing with only one sign. To such variables we assign the proper value to falsify (i.e. set to `SATISFIED`) the no-goods in which they are found. We also set to 0 the corresponding

---

<sup>16</sup>Such structure is just used to initially copy the model on the device.

<sup>17</sup>i.e. minor changes can be found when allocating memory for the data, since now also the memory for the device is allocated from the host.

positions of the counterpart of `varsAppearingInRemainingNoGoodsPositiveNegative` on the device for the positions of such variables (i.e. we assert that such variables aren't showing up unassigned in any no-good). To conclude, the kernel puts in a variable passed as fourth parameter (`SM_dev_varsYetToBeAssigned`) the number of newly assigned variables (this number will be later subtracted from the counter keeping track of how many variables at the  $i$ -th step aren't assigned). In this kernel each thread ideally takes care of a single variable, when the number of variables are more than the number of cores in the device, each thread will do more (i.e.  $\lceil \frac{noVars}{cores} \rceil$ ) work.

- `removeNoGoodSetsContaining(int*, int*, int*, int*, int*,int *)`: It's responsible for the removal of no-goods (i.e. sets them to `SATISFIED`) when a literal makes them false<sup>18</sup>. This method alongside marking no-goods as `SATISFIED` keeps also track and returns (via the second parameter) an array (`returningNGchanged`) of length `noNoGoods` containing for each position 1 if the no-good of the corresponding index has been set to `SATISFIED` during this execution of the kernel or 0 if it hasn't been modified. Such data structure (never transferred on the host, but kept only on the device) will be passed as an argument to the kernel `decreaseVarsAppearingInNGsatisfied`. In `removeNoGoodSetsContaining` we also store in a variable passed as sixth parameter (`SM_dev_currentNoGoods`) the number of newly no-goods set to `SATISFIED` by the method (this number will be later subtracted to the counter keeping track of how many no-goods at the  $i$ -th step aren't assigned). In this kernel each thread takes care of a single no-good, it loops throughout all its literals and performs the above mentioned operations. If there are more no-goods than cores in the device again each thread will do more work (similar as the previous kernel).
- `decreaseVarsAppearingInNGsatisfied(int*, int*, int* , int*)`: It's in charge of decreasing the counter of each unassigned variable present in a no-good newly set to `SATISFIED` (i.e. we get the `returningNGchanged` as second parameter) in order to make the array `dev_varsAppearingInRemainingNoGoodsPositiveNegative` coherent with the state of the other variables. In this kernel each thread takes care of a single no-good, checks whether it has been "recently" modified (i.e. checking if the element at the considered position of second parameter is 1) and in such case decreases the value of the cell of `dev_varsAppearingInRemainingNoGoodsPositiveNegative` corresponding to the literal (positive or negative) found in the considered no-good. Again if there are more no-goods than cores in the device each thread will do more work.
- `removeLiteralFromNoGoods(int*, int*,int*, int*, int*,int*)`: It's responsible for the removal of literals (related to assigned variables) from a (not yet falsified) no-good when these don't make it `SATISFIED`. This kernel seems the most complex one, but when profiled with nvvp it's not as influential as other kernels (see the execution times of `decreaseVarsAppearingInNGsatisfied`). In this kernel each thread takes care of a single (or a constant number of, as the some kernels above) no-good.

<sup>18</sup>It may be confusing but when a literal make a no-good false, we don't need to keep track of such no-good anymore, thus we see it as already `SATISFIED`.

Compute
34,0% removeNoGoodSetsContaining(int*, int*, int*, int*, int*, int*)
33,8% decreaseVarsAppearingInNGsatisfied(int*, int*, int*, int*)
27,0% removeLiteralFromNoGoods(int*, int*, int*, int*, int*, int*)
5,0% unitPropagation2(int*, int*, int*, int*, int*, int*, int*, int*)
0,1% parallelSum(int*, int*)
0,0% chooseVar(int*, int*)
0,0% pureLiteralCheck(int*, int*, int*, int*, int*)
0,0% assingValue(int*, int*, int*)

Figure 5: The percentages of execution time taken by each kernel (representative of “big” and “huge” models).

- `unitPropagation2(int*,int*, int*,int* ,int *,int *, int *, int*)`: It’s responsible for scanning the no-goods currently unsatisfied and check if only a single literal remains unassigned in them (i.e. check if `dev_noOfVarPerNoGood[i] == 1`). If such condition is met, then the variable indicated by `dev_lonelyVar[i]` is assigned to the proper value according to the literal sign, in order to falsify the no-good. An important thing to notice in this program version (“the advanced starting point”) is that this kernel is launched with just one block and one thread, so it doesn’t exploit a whole warp (which in general is a bad CUDA practice). Still the kernel is quite short and looking at nvvp at least up to the *big* tests it doesn’t slow the computation. **The choice for moving this function on the device rather than keeping it on the host side comes from the observation that even with just one thread the overall time of execution is less than the time needed to copy the necessary data on the host and executing the function host side.**
- `chooseVar(int*, int*)`: It’s the counterpart of the analogous method of the serial version of the algorithm. Given `dev_partialAssignment` and `dev_varsAppearingInRemainingNoGoodsPositiveNegative` returns the index of the first variable unassigned and still showing up in some unsatisfied no-good. As in the previous case also this kernel is launched with one block and a thread, the reasons for having this function on the device are those presented in `UnitPropagation2`.
- `assingValue(int*,int *,int *)`: It assigns the specified value (specified by the global variable `dev_valueToAssing`) to the indicated variable (indicated by `dev_varToAssign`), then it decreases the counter which keeps track of the yet unassigned variables, and sets to 0 the two positions relative to `dev_varToAssign` in the array `dev_varsAppearingInRemainingNoGoodsPositiveNegative` (i.e. we specify that such variable doesn’t show up unassigned in any no-good). There are no loops involved in this kernel and it’s executed by just one thread, the reasons are the same as for the last two kernels.

There is an additional kernel, which has been excluded from the list above since it’s not fundamental in order to understand the flow of computation (though necessary in order to have a correct result). Such kernel is `parallelSum(int* ,int*)` which is used to compute

a parallel reduction (sum) of the values of the array specified as first parameter and store the resulting value in the second argument. This kernel is often used in the code to collect the results of methods such as `removeNoGoodSetsContaining` or `pureLiteralCheck`. Indeed these kernels can be launched on multiple blocks and they yield one partial result for each block (stored in the arrays starting with “SM\_”). What `parallelSum` does is just take the values of these arrays and produce a scalar value which will then be used to decrease a variable. The reason for the introduction of this method is that the `AtomicAdd` operations performed in the two kernel mentioned aren’t atomic with respect to the whole system but just within blocks, thus we needed another way to combine the results of different blocks. `ParallelSum` can be launched on multiple threads (which could not be multiple of 32) but only on one block (for simplicity), thus we have the limitation/assumption that the code must run on devices which need to have more cores per SM than SMs<sup>19</sup>.

In addition to such kernels, a host method has been added, `int getSMcores(struct cudaDeviceProp)`, which queries the device and returns the number of cores per streaming multiprocessor. This has been used to calculate the number of blocks to launch in order to fully use the capabilities of the GPU and make the program more scalable.

---

<sup>19</sup>Such assumption is not too unrealistic since also new GPUs have usually less than 100 SM while having at least 128 core per SM, e.g. a GTX3090 has over 10000 cores while “only” 84 SM (with 128 cores each).

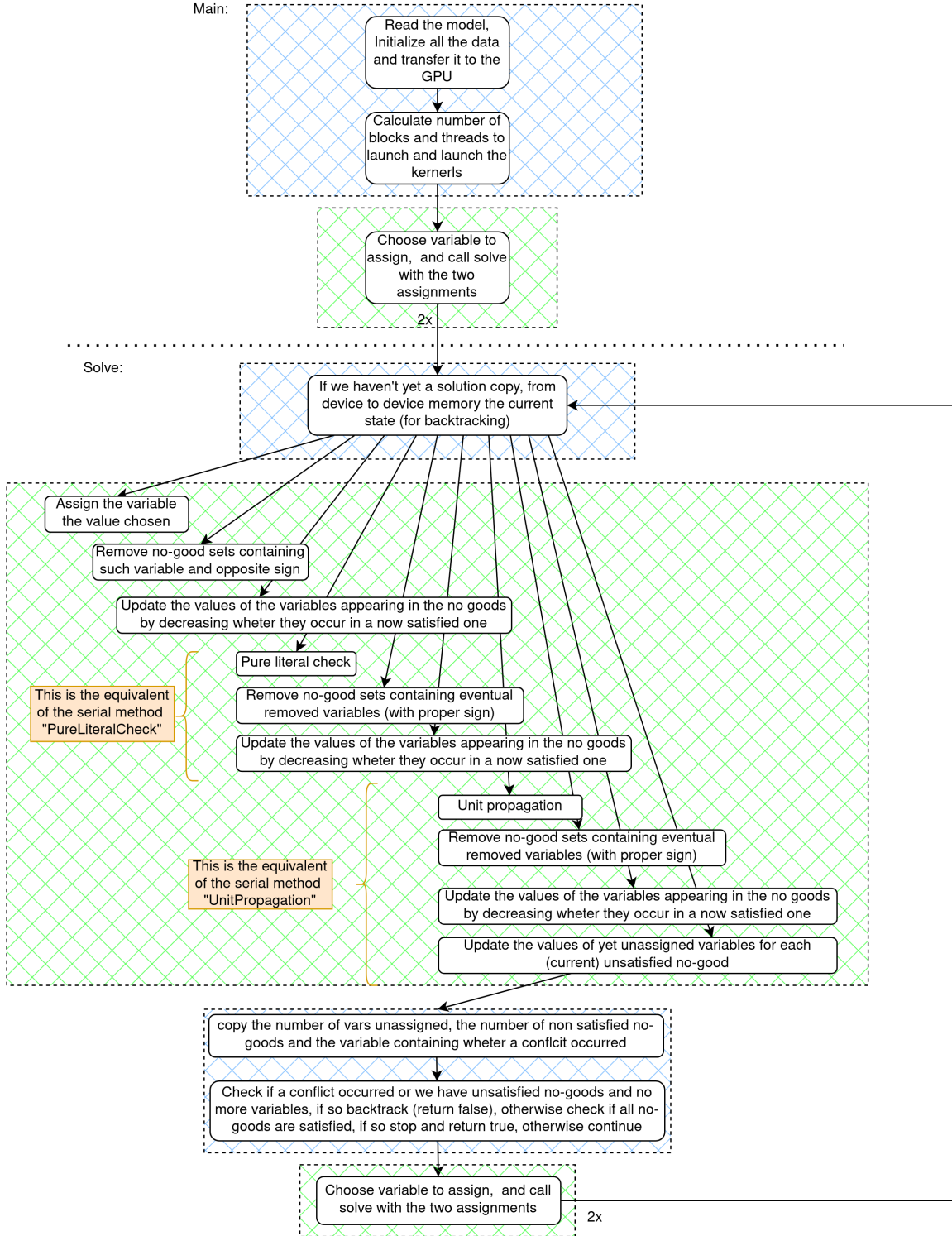


Figure 4: The execution flow of the parallel algorithm (the procedures on a green background are kernels, while those on a blue background are host code).

## 4.2 The (current) final version

We start by stating that also in this version the constraint for which the device running the algorithm needs to have more cores per SM than SMs is still present.

The main changes with respect to the advanced starting point are the following:

- `pureLiteralCheck` and `removeNoGoodSetsContaining` don't use anymore the instruction `AtomicAdd`, instead they return via an additional parameter an array on which a parallel reduction (sum) will be performed. This change itself at least up the “huge” instances didn't make much difference in the execution times.
- More shared memory has been used in some kernels to reduce time accesses to the data.

We are now going to compare this final version to the previous one and to the serial counterpart.

## 4.3 Some test results

We now present the results of some tests done on the Linux machine mentioned before (this machine has been used instead of the more powerful desktop for simplicity). The three versions (the serial one, the “advanced starting point” and the current/main parallel version) after the testing phase described in Section 5 have been tested and the results collected for the *medium*, *big* and *huge* test folders. In addition, after seeing the test results another folder *huge+* with few but bigger instances than those in *huge* has been created.

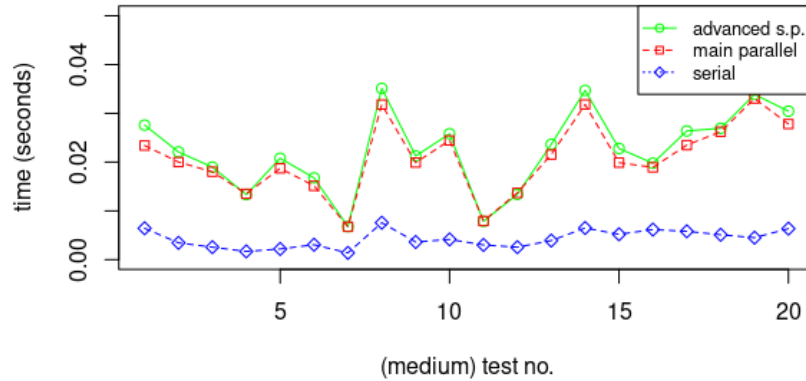


Figure 6: The times of the tests of the *medium* folder carried out on the Linux machine specified.

As it can be seen from the plots regarding the bigger instances we can conclude that the “current parallel version” improves the runtime with respect to the “advanced starting point version”. This difference is almost unnoticeable with smaller test, but it's assumed instead to increase if the instances were even bigger than those in *huge+* (which have roughly 6000



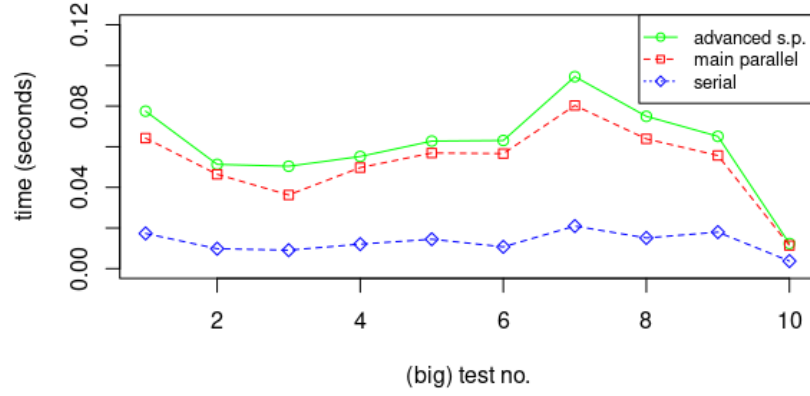


Figure 7: The times of the tests of the *big* folder carried out on the Linux machine specified.

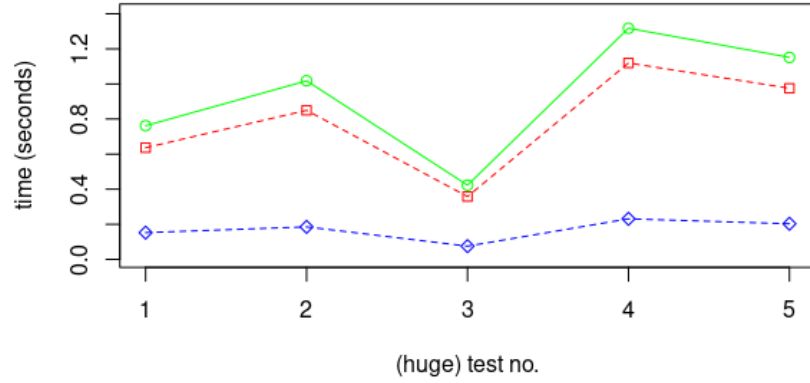


Figure 8: The times of the tests of the *huge* folder carried out on the Linux machine specified.

no-goods). The performances of the current parallel version are rather close to those of MiniSat for what concerns *huge* and *huge+* instances.

On the other hand, while being quite efficient the parallel implementations aren't comparable with the serial one. Such version if proven correct<sup>20</sup> would be no-match for its parallel counterparts.

<sup>20</sup>The results seem way too optimistic with respect even to MiniSat, which implies that there should be some kind of bug. Still more than 250 tests were run and checked.

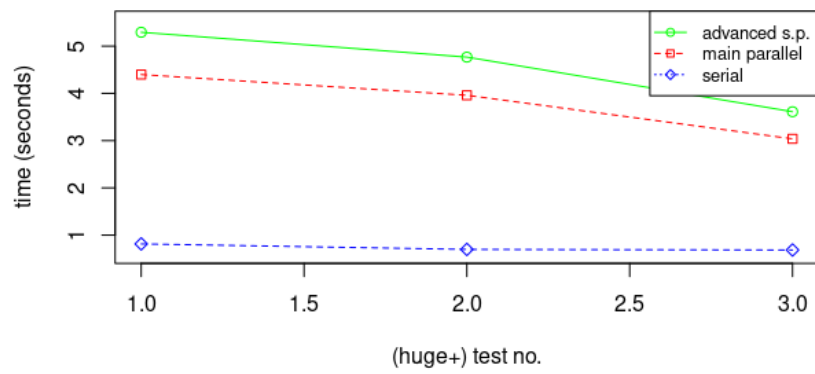


Figure 9: The times of the tests of the *huge+* folder carried out on the Linux machine specified.

## 4.4 Some remarks

During the development of the parallel version some conclusions on the `unitPropagation` and `solve` method have been drawn. What has been noticed is that neither of these methods (as they were originally thought) were easily parallelizable (in single kernels) due to read/write conflicts on some variables. The cause of this is that the serial version allows *to change point of view* quite easily, e.g. in the serial `unitPropagation` we first start looping through each no-good, and then if some conditions are met we call a function looping on the variables. This scenario can't be easily reproduced on the device without a significant divergence and drop in performance (since each kernel operating on a no-good would launch the function looping through the variables). Indeed as it can be seen from Section 4.1 each kernel now adopts a more "static" point of view (i.e. either threads deal with no-goods or variables) and maintains it. For this reason, as it can be seen from Image 4 what once was done by a single procedure (e.g. `pureLiteralCheck` or `UnitPropagation` which called subprocedures) now it's unrolled in the `solve` method which launches multiple kernels in order to do these operations. For example the call made to `pureLiteralCheck` in the serial version now corresponds to a launch in sequence of the kernels: `pureLiteralCheck`, `parallelSum`, `removeNoGoodSetsContaining`, `parallelSum` and `decreaseVarsAppearingInNGsatisfied` (in the code these sequences are well delimited by comments).

To support the conclusion stated above we now report as a (wrong) example a possible version of the `unitPropagation` kernel initially written which presented a conflict:

```
--global__ void unitPropagation(int* dev_matrix,int *
dev_partialAssignment,int * dev_varBothNegatedAndNot,int *
dev_noOfVarPerNoGood,int* dev_lonelyVar, int*
dev_unitPropValuestoRemove,int * dev_varsYetToBeAssigned) {
    int thPos = blockIdx.x * blockDim.x + threadIdx.x;
    int decrease=0;
    int i;
    //for each no good (th deals with ng)
    for (int c = 0; c < dev_noNoGoodsperThread; c++) {
        i=thPos+c*dev_noNoGoodsperThread;
        if(i<dev_noNoGoods && *(dev_matrix+i*(dev_noVars+1)) ==
            UNSATISFIED && dev_noOfVarPerNoGood[i] == 1){
            //printf("UNIT PROP: at no good: %d, seing: %d (-2
            = UNSATISFIED)\n",i,*(dev_matrix+i*(dev_noVars
            +1)));
            //lonelyVar[i] is a column index
            dev_partialAssignment[dev_lonelyVar[i]] = *(
                dev_matrix+i*(dev_noVars+1)+dev_lonelyVar[i]) >
                0 ? FALSE : TRUE;
            decrease--;
            dev_unitPropValuestoRemove[dev_lonelyVar[i]]=
                dev_partialAssignment[dev_lonelyVar[i]] == TRUE
                ? NEGATED_LIT : POSITIVE_LIT;
        }
        __syncthreads();
    }
    if(decrease!=0)
        atomicAdd((dev_varsYetToBeAssigned), decrease);
}
```

It may not be visible at a first glance but while the accesses done via the *i* index are indeed on different elements, the accesses obtained by using `dev_lonelyVar[i]` as indexes may coincide on a same element, thus resulting in a read/write conflict.

## 5 Test generation and testing

The test instances used to obtain the results discussed have been generated via a Python script. Such script randomly generates a no-good file instance as well as the corresponding SAT instance (in DIMACS format). The number of tests to generate, the number of variables and no-goods can be easily modified in the script.

Several tests have been done to empirically check the correctness of the implementations. Of these tests some were monitored in order to get an idea of the performances of the program. The testing phase was done via slightly modified versions of the programs which can be found at under the testing branch<sup>21</sup>. These versions allow to call the main procedure (both the serial or parallel version) on all the files of a specified folder<sup>22</sup>. Such programs also check at the end of each test (for the satisfiable instances) the correctness of the partial assignment found<sup>23</sup>, this is done by calling the function in the header file `common.h`. In the case of unsatisfiable models, the unsatisfiability has been checked by running MiniSat on the correlative SAT instance of that model.

---

<sup>21</sup><https://github.com/EnrSanti/No-good-solver/tree/testing>

<sup>22</sup>Which is hard coded and can be easily changed.

<sup>23</sup>i.e. we check if the assignment actually falsifies each no-good.

## 6 Conclusions

This project has let me really comprehend how much the linear structure of an algorithm developed to be executed by a serial machine can change and become much more convoluted in order to exploit the full capabilities of a parallel device. It has also made me understand the importance of checking the result of each instruction (whether it's a memory copy or a kernel launch) when dealing with CUDA, since the errors in the framework aren't explicitly brought up to the developer (for example making the program crash). Such errors indeed can go unnoticed and lead to a complex debugging process in later development stages which could result quite frustrating.

### 6.1 Further developments

Beside some optimizations such as reading first the whole file and then using `sscanf` to populate the data structures in order to improve a little the input reading performances or the usage of some linked lists instead of arrays (to improve unit propagation complexity), one of the possible future developments could be the introduction of different policies for the method `chooseVar`<sup>24</sup> and the implementation of methods such as `learnClause` and `backjump` in order to shift more towards a CDCL approach. Another perhaps easier improvement could be the usage of vectorized accesses by using instead of an integer matrix and arrays, arrays of chars or int4 (8 bits would be more than enough to contain such data), or even better store inside a 32 bit integer eight 4-bit values, thus reducing by a factor of eight the memory used. By doing this also the dynamic allocation of memory and the copy of the data would sensibly improve. For what concerns CUDA the different memories (e.g. shared) could be exploited more to store some additional data while also simpler (and faster) instructions could be used.

---

<sup>24</sup>e.g. *choose the most constrained literal or choose one in the longest no-good.*

## References

- [1] V. Lifschitz, “Answer set programming (draft),” 2019.
- [2] —, *Answer set programming*. Springer Heidelberg, 2019.
- [3] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, *Answer Set Solving in Practice*. Morgan Claypool Publishers, 2012.
- [4] M. Ouyang, *Implementations of the DPLL algorithm*. Rutgers The State University of New Jersey, School of Graduate Studies, 1999.
- [5] J. Marques-Silva, I. Lynce, and S. Malik, *Conflict-driven clause learning SAT solvers*, 1st ed., ser. Frontiers in Artificial Intelligence and Applications. Netherlands: IOS Press, 2009, no. 1, pp. 131–153.
- [6] N. Een, A. Mishchenko, and N. Amla, “A single-instance incremental sat formulation of proof- and counterexample-based abstraction,” 2010.