Rules of the 2011 SAT Competition

Matti JÄRVISALO, Daniel LE BERRE and Olivier ROUSSEL

organizers@satcompetition.org

Version of this document

The version number and the date of this document (as recorded by the versioning system) are given below. They let you identify quickly if you have the most recent version of this document.

```
Version: $Rev: 137 $
Last modification: $Date: 2011-02-21 15:02:50 +0100 (Mon, 21 Feb 2011) $
```

1 Introduction

This document provides detailed rules of the 2011 SAT competition. Section 2 details the different tracks of the competition. Section 3 explains how solvers are ranked and what happens in case a solver is found to report incorrect answers. Section 4 describes the the input file format to which the competing solvers have to conform to. Section 5 details how a solver must report its results. Finally, Section 6 details the requirements for submitting a solver.

2 Competition Tracks

The following tracks are organized.

- Main Track
- Minimally Unsatisfiable Subset (MUS) Special Track
- Minisat Hack Track
- Certified Unsat Track

2.1 Main track

In the main track, the goal is to determine whether a given SAT instance in conjunctive normal form (CNF) is satisfiable or not *as quickly as possible*. For satisfiable formulas, solvers are required to output a model of the formula. Solver do not have to output proofs of unsatisfiability for unsatisfiable formulas; this is the focus of the Certified Unsat special track, see section 2.4).

This year there are no special tracks dedicated to sequential or parallel solvers. Sequential and parallel solvers will both compete in the main track. However, two different rankings will be used, one based on wall clock and another based on CPU time. It is expected that parallel solvers will perform better in the first ranking while sequential solvers will perform better in the second ranking. In any case, all solvers will appear in both rankings.

The first ranking is based on wall clock time and will promote solvers which use all available resources to answer as quickly as possible. The second ranking is based on CPU time and will promote solvers which use resources as efficiently as possible. This latter ranking was the one used in the previous competitions. In the wall clock based

ranking, timeout will be imposed on the wall clock time. In the CPU based ranking, timeout will be imposed on CPU time.

Parallel solvers will be allocated NBCORE processing units. The exact value of NBCORE will depend on the available computing resources, but will be at least 4. All parallel solver will be allocated the same number of processing units.

At least 7 gigabytes of main memory will be allocated to each solver (possibly more given that resources permit). Solvers that want to use all this memory must be compiled in 64-bit mode.

2.2 Minimally Unsatisfiable Subset (MUS) Special Track

This special track assesses the efficiency of algorithms and solvers for computing minimally unsatisfiable subsets of given CNF formulas. João Marques-Silva is in charge of collecting a set of relevant benchmarks for this track.

2.2.1 Plain MUS track

In this track, the goal is to find a (plain) minimally unsatisfiable subset S' of a given set of clauses S, i.e., a subset S' of S such that each subset of S' is satisfiable.

Definition 1 *Given an unsatisfiable set of clauses S (a CNF formula), a set $S' \subseteq S$ is a* minimally unsatisfiable subset *of S if S' is unsatisfiable and each $S'' \subset S'$ is satisfiable.*

As a solution to a given input CNF S, solvers have to provide one such MUS S'. However, instead of reporting the MUS in full, solvers are expected to provide as a list the indices in the input CNF S of the clauses in the found MUS S'.

2.2.2 Group oriented MUS track

The computation of *group oriented MUSes* (or *set MUSes*) corresponds to the *high-level MUSes* [**?**].

In the *group oriented MUS* problem, the input is an unsatisfiable set of clauses (a CNF formula) $\mathcal{C} = \mathcal{D} \cup \mathcal{G}_1 \cup \ldots \cup \mathcal{G}_k$ that is *explicitly partitioned* into the groups $\mathcal{D}, \mathcal{G}_1, \ldots, \mathcal{G}_k$ of clauses so that $\mathcal{D} \cap \mathcal{G}_i = \emptyset$ and $\mathcal{G}_i \cap \mathcal{G}_j = \emptyset$ hold for each $i, j \in \{1, \ldots, k\}$ with $i \neq j$.

Definition 2 *Given an explicitly partitioned unsatisfiable CNF formula $\mathcal{C} = \mathcal{D} \cup \bigcup_{G \in \mathcal{G}} G$, where $\mathcal{G} = \{\mathcal{G}_1, \ldots, \mathcal{G}_k\}$, and $\mathcal{D}$ and each $\mathcal{G}_i$ are disjoint sets of clauses, a* group oriented MUS *of $\mathcal{C}$ is a subset $\mathcal{G}'$ of $\mathcal{G}$ such that $\mathcal{D} \cup \bigcup_{G \in \mathcal{G}'} G$ is unsatisfiable and, for every $\mathcal{G}'' \subset \mathcal{G}'$, we have that $\mathcal{D} \cup \bigcup_{G \in \mathcal{G}''} G$ is satisfiable.*

In other words, a group oriented MUS is a subset $\mathcal{G}'$ of $\{\mathcal{G}_1, \ldots, \mathcal{G}_k\}$ which, in conjunction with $\mathcal{D}$, is minimally unsatisfiable, that is, every proper subset of $\mathcal{G}'$ in conjunction with $\mathcal{D}$ is satisfiable. Notice that $\mathcal{D}$ and the clauses in $\mathcal{D}$ do not contribute to the size of a group oriented MUS, and can hence be viewed as *don't care* or irrelevant clauses w.r.t. the sizes of the group oriented MUSes of $\mathcal{C}$.

As a solution to a given explicitly partitioned CNF $\mathcal{C}$, solvers have to provide one group oriented MUS of $\mathcal{C}$. However, instead of reporting the found group oriented MUS in full, solvers are expected to provide as a list the indices of the groups in the group oriented MUS.

2.3 Minisat Hack Track

This track was originally introduced for the SAT 2009 competition and is organized again this year. This year the latest version of Minisat (2.2.0) is used as the base solver. Its source code can be found at

`http://www.minisat.se/downloads/minisat-2.2.0.tar.gz`.

The motivation of this track is twofold. On one hand, we want to lower the threshold for Master's and PhD students to enter the SAT competition by bringing new and innovative ideas to the original Minisat solver. On the other hand, the aim is to see how far the performance of Minisat can be improved by making only minor changes to the source

code. We strongly encourage usual participants to the main track (including non-students) with "hacked" Minisat solver to participate in this track.

Our motivation behind using Minisat as the base solver is as follows. Minisat is a well-known and widely adopted solver that can be seen as a canonical CDCL solver, offering an efficient (and necessary) CDCL basis and easily modifiable code. Indeed, Minisat has been often improved by introducing only minor changes to its code ("hacking it"). Furthermore, looking back at recent progresses in CDCL solvers, most of them are based on minor changes (< 10 lines each): phase caching, blocked literals, and Luby series for rapid restarts.

The motivation behind competing with minor hacks is that it can help the community to detect which types of modifications pay off: in order to isolate the actual causes of improvements when modifying solvers, it is necessary to perform scientifically valid experiments on the effects of individual changes made to a solver (e.g., individual new techniques introduced to the base solver).

Depending on the number of submissions in this track, the organizers may decide to use a shorter time limit or a smaller set of benchmarks than in the main competition track. Actual award will not be given in this track, but we hope that the whole community will benefit from analyzing the results and the proposed hacks. However, the best solvers identified in this track will automatically enter the main competition to be compared on the same basis as other solvers, with the possibility of being awarded in the main track.

Many thanks to Niklas Eén and Niklas Sörensson for allowing us to conduct this special track.

2.4 Certified Unsat Track

This track is organized by Allen Van Gelder.

In the certified unsat track of the 2011 SAT Competition, SAT and QBF solvers produce some verification output and are only tested on unsatisfiable (for SAT) or invalid (for QBF) benchmarks. We are also looking for verification tools to be submitted.

Because of the experimental nature of this track, there are no prizes and no official ranking system. See the poster shown at SAT 2007 for that year's track. We expect 2011 to be conducted similarly. There are several accepted proof formats, each with precise specifications. These formats are designed to be neutral to the SAT solver's methodology as far as possible.

For QBF, there is a new format, called QIR, which is an encoding for a Q-resolution proof that is easier to verify than existing formats, and might be more useful for debugging because it is easier for humans to analyze it. QIR is also suitable for SAT by considering all variables to be existential.

Submitted verification tools do not need to accept an "official" format natively, if we can set up a simple translation script. Similarly, submitted solvers might produce output in a different form, which can be post-processed into an "official" format.

All formats may be found at the ProofChecker web page

`http://users.soe.ucsc.edu/~avg/ProofChecker/`,

but we expect the most popular for SAT to be the RUP format.

The purpose of the RUP proof format is to allow a solver to "retro-fit" some verification output without too much trouble, and to allow for some possibility of derivation rules other than resolution. The short story is that solvers in the vein of grasp, chaff, berkmin, minisat, and descendants can simply output their conflict clauses (the final one being the empty clause).

Also, we believe that look-ahead solvers will be able to output a RUP proof without great difficulty. Notice that it is not necessary to actually use all the clauses you output, but if the redundancy is too great, you might hit file size limits or time limits. Practical experience is needed to see what can be achieved.

3 Ranking the solvers

3.1 Main track

Only solvers submitted in source and for which the author agrees to make it available in source to the community for research purposes after the competition will be ranked, i.e. have a chance to be awarded.

The main track will be run in two phases. In the first phase, every solver will compete with a timeout set to 1200 seconds. The best solvers of the first phase (selected by the jury) will enter the second phase and will be allocated a longer timeout (5000 seconds).

The solvers will be awarded according to the number of benchmarks solved during the second stage, using the cumulated time required to solve those benchmarks to break ties. In the main track, there will be one ranking based on CPU time and another one based on wall clock time.

A solver that answers incorrectly will be disqualified and will not appear in the rankings. A solver answers incorrectly if it reports SATISFIABLE but reports an assignments that does not satisfy the input CNF, or reports UNSATISFIABLE on a formula that is already known to be satisfiable.

The organizers may present other scorings based on the results of the solvers under the competition conditions, for providing different views of the competitions.

3.2 MUS tracks

The only restriction to be awarded in the MUS track is to make the solver available in binary form to the community for research purposes after the competition.

Solvers will be ranked by the number of instances for which a minimally unsatisfiable subset is identified by the solver. In case two solvers produce MUSes for the same number of instances, ties are be broken based on the total computation time used by the solvers for finding these MUSes.

A solver that answers incorrectly will be disqualified and will not appear in the rankings. A solver answers incorrectly if the MUS it identifies is either satisfiable, or if it the MUS is not minimally unsatisfiable (i.e., there is a proper subset of the reported subset that is unsatisfiable).

4 Input Format

4.1 CNF Input Format

The benchmark file format is in a simplified version of the DIMACS format for conjunctive normal form (CNF) propositional formulas. An example is provided in Figure 4.1.

```
c
c start with comments
c
c
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

Figure 1: Example of a CNF in the DIMACS format

SPECIFIC RULES FOR THE SAT COMPETITION

- The file can start with comments, that is lines beginning with the character `c`.
- Right after the comments, there is the line `p cnf <nbvar> <nbclauses>` indicating that the instance is in CNF format; `<nbvar>` is the exact number of variables appearing in the file; `<nbclauses>` is the exact number of clauses contained in the file. It is guaranteed that each variable between 1 and $nbvar$ appears at least once in a clause.
- Then the clauses follow. Each clause is a sequence of distinct non-null numbers between $-nbvar$ and $nbvar$ ending with 0 on the same line. Positive numbers denote the corresponding variables. Negative numbers denote the negations of the corresponding variables. A clause is not allowed to contain the opposite literals i and $-i$ simultaneously.

4.2 Group oriented CNF Input Format

The group oriented CNF file format is an extension of the Dimacs format that adds a group index to each clause. The group index is indicated inside curly braces at the beginning of the clause. An example is provided in Figure 4.2.

```
c
c Example of group oriented CNF
c
c Represents the following formula
c D  = {x1 or x2 or x3}
c G1 = {x1 -> x2, x2 -> x3}
c G2 = {x3}
c G3 = {x3 -> x2, -x2 or -x3}
c G4 = {x2 -> x3}
c
p gcnf 5 7 4
{0} 1 2 3 0
{1} -1 2 0
{1} -2 3 0
{2} -3 0
{3} 2 -3 0
{3} -2 -3 0
{4} -2 3 0
```

Figure 2: Example of a group oriented CNF

SPECIFIC RULES FOR THE SAT COMPETITION

- The file can start with comments, that is lines beginning with the character `c`.
- Right after the comments, there is the line `p gcnf <nbvar> <nbclauses> <lastgroupindex>` indicating that the instance is in group oriented CNF format; `<nbvar>` is the exact number of variables appearing in the file. It is guaranteed that each variable between 1 and $nbvar$ appears at least once in a clause. `<nbclauses>` is the exact number of clauses contained in the file. `<lastgroupindex>` is the last index of a group in the file number of components contained in the file. It is guaranteed that all groups index between 0 and `<lastgroupindex>` inclusive appear in the file.
- Then the clauses follow. Each clause starts with a component number `{x}` with x between 0 and $lastgroupindex$. The specific case of 0 means that the clause belong to the set of don't care clauses (D). The rest of the clause is a sequence of distinct non-null numbers between $-nbvar$ and $nbvar$ ending with 0 on the same line. Positive numbers denote the corresponding variables. Negative numbers denote the negations of the corresponding variables. A clause is not allowed to contain the opposite literals i and $-i$ simultaneously.

5 Output Format

For reporting output, solvers must print messages to the standard output. These messages will be used to check the results.

5.1 Messages

No specific order is imposed on the solvers' output lines. However, all lines, according to its first char, must belong to one of the categories described below. Lines that do not start with one of the patterns below will be considered a comment and hence ignored.

Comment ("c " lines) A comment line begin with a lower case `c` followed by a space (ASCII code 32).

Comment lines are optional and may appear anywhere in the solver output.

They contain any information that authors want to emphasize, such as #backtracks, #flips, etc, or internal CPU-time. They are recorded by the execution environment for later viewing but are otherwise ignored. At most one megabyte of solver output will be recorded. In is important to notice that if a solver is very verbose, some output lines may be lost.

Submitters are advised to avoid printing comment lines which may be useful in an interactive environment but otherwise useless in a batch environment. For example, outputting comment lines with the number of constraints read so far only increases the size of the logs with no benefits as regards the competition.

If a solver is really too verbose, the organizers will ask the submitter to remove some comment lines.

Solution ("s " lines) A solution line begins with a lower case `s` followed by a space (ASCII code 32).

Only one such line is allowed.

It is mandatory.

This line is used to provide the answer of the solver. It must be one of the following answers:

- `s SATISFIABLE`
  This line indicates that the solver has found a model of the formula. In such a case, a "v " line is mandatory.
  For decision problems, this line must be printed when the solver has found a solution.
- `s UNSATISFIABLE`
  This line must be output when the solver can prove that the formula has no solution.

- `s UNKNOWN`
  This line must be output in any other case, i.e., when the solver is not able to produce an answer for the input formula.

It is of uttermost importance to respect the exact spelling of these answers. Any typing errors on these lines will cause the answer to be disregarded.

Solvers are not required to provide any specific exit code corresponding to their answer.

If the solver does not output a *solution* line, or if the solution line is misspelled, then UNKNOWN will be assumed.

Values ("v " lines) A values line begins with a lower case `v` followed by a space (ASCII code 32).

More than one "v " line is allowed. If multiple "v" lines are output, the execution environment merges their content internally.

One values line is mandatory when the instance is satisfiable, or in the MUS tracks when the instance is unsatisfiable.

- For the satisfiability problem, a solver that outputs `"s SATISFIABLE"` must provide a solution. A solution is a model (i.e., a satisfying assignment) of the instance that will be used to check the correctness of the answer. For reporting a model, solvers must provide a list of non-contradictory literals which, when interpreted as being true, satisfies each clause of the input formula. The negation of a literal is denoted by a minus sign immediately followed by the identifier of the variable. The order of literals does not matter. Arbitrary white space characters, including ordinary white spaces, newline and tabulation characters are allowed between the numbers as long as each line containing these literals (or clause identifiers) is a *values* line, i.e. it begins with the two characters "v ". In any case, the last values line must be terminated by a "0" (zero) followed by a Line Feed character (the usual Unix line terminator '\n'. If the last "v " line does not end with this sequence, it will be assumed that the solver was interrupted before it could print out a complete solution, and this incomplete answer will be ignored.

  Note that we do not require a proof for unsatisfiability, which is the focus of a special track (see section 2.4).
- In the plain MUS track, the solution is a list of clause identifiers which form an MUS. The first clause of the input formula is numbered 1, so each identifier must be in the range [1..nbClauses]. Printing this line is a commitment that the given list of clauses is unsatisfiable and that any subset of the set consisting of these clauses is satisfiable.

  The last values line must be terminated by a "0" (zero) followed by a Line Feed character (the usual Unix line terminator '\n'. If the last "v " line does not end with this sequence, it will be assumed that the solver was interrupted before it could print out a complete solution, and this incomplete answer will be ignored.
- In the group oriented MUS track, the solution is a list of group identifiers (indices) which form a group oriented MUS. Printing this line is a commitment that the given list of groups, in conjunction with $\mathcal{D}$, is unsatisfiable and that any subset of this groups in conjunction with $\mathcal{D}$ is satisfiable.

  The last values line must be terminated by a "0" (zero) followed by a Line Feed character (the usual Unix line terminator '\n'. If the last "v " line does not end with this sequence, it will be assumed that the solver was interrupted before it could print out a complete solution, and this incomplete answer will be ignored.

  Notice that the zero that ends the values line is not related to the index 0 that is used to denote the don't care group $\mathcal{D}$.

5.2 Main Track Output

When the formula is satisfiable, the solver must output a 's SATISFIABLE' line and a 'v ' line containing a model of the formula.

When the formula is unsatisfiable, the solver must output a 's UNSATISFIABLE' line.

For instance, the following outputs are valid for the instance in Figure 4.1.

```
mycomputer: $ ./mysolver myinstance-sat
c mysolver 6.55957 starting with TIMEOUT fixed to 1000s
c Trying to guess a solution...
s SATISFIABLE
v 1 2 -5
v 4 -3 0
c Done (mycputime is 234s).

mycomputer: $ ./mysolver myinstance-unsat
c mysolver 6.55957 starting with TIMEOUT fixed to 1000s
c Trying to guess a solution...
c Contradiction found!
s UNSATISFIABLE
c Done (mycputime is 2s).
```

5.3 MUS Track Output

In that context, the benchmarks will be known to be unsatisfiable. Thus the solver is expected to output a 's UNSATISFIABLE' line.

In the plain MUS track, the solver must output a 'v ' line containing the index of the clauses that form a MUS .

In the group oriented MUS track, the solver must output a 'v ' line containing the index of the groups that form a MUS in conjunction with the don't care group D.

```
mycomputer: $ ./mysolver myinstance-unsat
c mysolver 6.55957 starting with TIMEOUT fixed to 1000s
c Trying to guess a solution...
c Contradiction found!
s UNSATISFIABLE
c computing a minimally unsatisfiable subformula
v 1 2
v 27 75 0
c Done (mycputime is 3s).
```

6 Requirements for submitting a solver

6.1 Categories

The SAT competition used to run all the solvers on all the benchmarks. This year, submitters are allowed to choose in which competition category to participate.

In the main track, there are three categories, *application*, *crafted*, and *random*. Each category is defined through the type of instances it contains.

Application Instances encoding various "application" problems in CNF.

These instances are typically large (containing up to tens of million of variables and clauses). The motivation behind this category is to provide hints about the kind of application SAT solvers may be useful for.

Crafted Instances that are often designed to give a hard time to SAT solvers, or represent otherwise problems which are challenging to typical SAT solvers (including, e.g., instances arising from difficult puzzle games).

These benchmarks are typically small. The motivation behind this category is to highlight current challenge problem domains that reveal the limits of current SAT solver technology.

Random Randomly generated uniform k-SAT formulas.

This category is motivated by the fact that the instances can be fully characterized, and furthermore, its connections especially to statistical physics.

6.2 Execution environment

Solvers will run on a cluster of computers using the Linux operating system. Each solver will be run under the control of another program (runsolver) that enforces some limits on the memory and the total CPU and wall clock time used by the solver.

Solvers will be run inside a sandbox that will prevent unauthorized use of the system (network connections, file creation outside the allowed directory, among others).

Solvers can be run as either as 32-bit or 64-bit applications. In case an executable is submitted, the author is required to provide us with an ELF executable (preferably statically linked). Authors submitting solvers in source form will have to specify whether it should be compiled in 32-bit or in 64-bit mode.

Two executions of a solver with the same parameters and system resources should output the same result in approximately the same time (so that the experiments can be repeated).

During the submission process, the submitter will be asked to provide the organizers with a suggested command line that should be used to run the solver. On this command line, the submitter is to use the following placeholders that will be replaced with the execution environment with actual values.

- BENCHNAME will be replaced with the name (including path and extension) of the file containing the instance to solve. The solver must use this parameter, or alternatively one of the following variants: BENCHNAMENOEXT (name of the file with path but without extension), BENCHNAMENOPATH (name of the file without path but with extension), BENCHNAMENOPATHNOEXT (name of the file without path nor extension).
- RANDOMSEED will be replaced with a random seed (a number between 1 and 4294967295). This parameter MUST be used to initialize any random number generator used by the solver. The actual seed is recorded by the execution environment. This will allow the organizers to run the solver on a given instance again under the same conditions if necessary.
- TIMELIMIT (or TIMEOUT) represents the total CPU time (in seconds) that the solver may use before being killed. May be used to adapt the solver strategy.
- MEMLIMIT represents the total amount of memory (in MiB) that the solver may use before being killed. May be used to adapt the solver strategy.
- NBCORE will be replaced by the number of processing units that have been allocated to the solver. Note that, depending on the available hardware, a processing unit may be either a processor, a core of a processor or a "logical processor" (in hyper-threading).
- TMPDIR is the name of the only directory where the solver is allowed to read/write temporary files. TMPDIR may be equal to DIR.
- DIR is the name of the directory where the solver files will be stored, as well as the instance file (BENCHNAME). It is also the working directory when the solver is run.

Examples of command lines:

```
DIR/mysolver BENCHNAME RANDOMSEED
DIR/mysolver --mem-limit=MEMLIMIT --time-limit=TIMELIMIT \
             --tmpdir=TMPDIR BENCHNAME
java -jar DIR/mysolver.jar -c DIR/mysolver.conf BENCHNAME
```

As an example, these command lines could be expanded by the execution environment with the following result:

```
/solver10/mysolver /tmp/file.cnf 1720968
/solver10/mysolver --mem-limit=900 --time-limit=1200 \
                   --tmpdir=/tmp/job12345 /tmp/file.cnf
java -jar /solver10/mysolver.jar -c /solver10/mysolver.conf /tmp/file.cnf
```

The command line provided by the submitter is only a suggested command line. Organizers may have to modify this command line (e.g., memory limits of the Java Virtual Machine (JVM) may have to be modified in order to cope with the actual memory limits).

The solver may also (optionally) use the values of the following environment variables:

- TIMELIMIT (or TIMEOUT) — the number of seconds it will be allowed to run.
- MEMLIMIT — the amount of RAM in MiB available to the solver.
- TMPDIR — the absolute pathname of the only directory where the solver is allowed to create temporary files.

After TIMEOUT seconds have elapsed, the solver will first receive a SIGTERM to give it a chance to output the best solution it found so far (in the case of an optimization problem). One second later, the program will receive a SIGKILL signal from the controlling program to terminate the solver.

The solver cannot write to any file except standard output, standard error and files in the TMPDIR directory. Solvers may use several processes or threads. Children of a solver process are allowed to communicate through any convenient means (Pipes, Unix or Internet sockets, IPC, ...). Any other communication is strictly forbidden. Solvers are not allowed to perform actions that are not directly related to the resolution of the problem.

6.3 Special considerations for parallel solvers

The execution environment will bind the solvers to a subset of all available processing units. The environment variable NBCORE will indicate how many processing units have been granted to the solver. The solver will not have access to more processing units than NBCORE. This implies that if the solver uses x threads or processes (with $x >$NBCORE), $x-$NBCORE threads or processes will necessarily sleep at one time.

As an example, if the competition is run on hosts with 2 quad-core processors (8 cores in total), several scenarios are possible:

- one single solver is run on the host, it is allowed to use all 8 cores (NBCORE=8).
- two solvers are run simultaneously, each one being assigned to a given processor (which means that a solver is assigned 4 cores, hence NBCORE=4).
- 4 solvers are run simultaneously, each one being assigned to a fixed set of 2 cores (belonging to the same CPU), hence NBCORE=2.
- more generally, a single solver may be assigned any number x of cores (from 1 to 8 in this example) to simulate the availability of x processing units.

The solver might use the NBCORE environment variable to adapt itself to the number of available processing units.

A solver must not modify its processor affinity (calls to `sched_setaffinity(2)` or `taskset(1)`) to get access to a processing unit that was not initially allocated to the solver. It may however modify its processor affinity to use a subset of the initially allocated processing units.