MUStICCa (User Guide)

Johannes Dellert, Christian Zielke, Michael Kaufmann

University of Tübingen, Germany

1 Installation and Startup

1. Download and unpack the archive, e.g. `musticca-yyyy-mm-dd.tar.gz`.
2. Change into the `minisat-extended` directory, run `make`.
3. Put the `minisat-extended` directory into your path, so that the executable in this directory is the version of MiniSat that gets executed if the command `minisat` is run from the console. This version includes some custom extensions for proof output and unit assumptions, and MUStICCa will exhibit unpredictable behaviour on other versions of MiniSat.

MUStICCa is distributed as a JAR archive, and it is started by running the command `java -jar musticca.jar`. For now, start MUStICCa without any arguments.

2 First Steps

After starting MUStICCa, the first step is to load a DIMACS CNF file of your choice (at more than 10,000 clauses the interface responsiveness decreases too much to be enjoyable), or one of the examples in the `examples` directory, all of which exhibit very interesting search spaces. A new instance can be loaded at any time via the **Load Instance** item in the **Instance** menu.

The GUI will rebuild, and afterwards the reduction graph will contain a single node labeled with the number of clauses in the instance. The number in brackets says how many clauses in the respective US are of unkown status. At this moment, nothing is known about the search space yet, so that both numbers are equal. In the US view, all the clauses in the instance are displayed. Execute your first manual reduction attempts by double-clicking on some of the clauses. You will see that either the clause becomes red, indicating that it was determined to be critical, or that a new node appears in the reduction graph, representing the result of a successful reduction attempt.

Having become familiar with manual reduction, select the top node of the reduction graph and turn your attention towards the reduction agent view in the bottom part of the MUStICCa window. In the definition window on the left, select one of the predefined reduction heuristics, then click on the **Start** button. A field containing information about the agent you just started appears in the agent overview to the right, and automated reduction attempts begin to be executed. Wait and observe until the agent has found a MUS.

2.1 View Components

The **reduction graph view** forms the central component for navigating between different unsatisfiable subsets, and is an explicit representation of the explored parts of the search space. By default, each node just displays information about the size of the US it represents (as a number displayed at the left of each node label), and the number of clauses about which we have no reducibility information yet (in parentheses). Node colours are used according to the colour schema in Figure 1 to encode a few special properties. Navigation in the search space works by selecting nodes in the reduction graph, causing the selected node to be highlighted in yellow, and the contents of the respective US to be displayed in the US view.

Color	**Explanation**
red	this subset is a MUS, i.e. all its clauses are critical
dark green	all possible reductions have been tried in this US, and at least one attempt was successful
light green	there are no clauses of unkown status left, but not all possible reductions have been tried
white	unkown status, none of the other conditions apply

Fig. 1. The default colour schema for encoding US states.

The **US view** is not much more than a list of clauses whose font colour encodes the respective value in the reduction table for that clause. The default color schema for these clause states is given in Figure 2. For advanced interactions, one or more clauses can be selected, and are then highlighted by a yellow background colour.

Color	**Explanation**
dark red	the clause is critical in the US
dark green	the clause was a successful deletion candidate in the US
light green	the clause is known to be unnecessary in the US
black	unkown status, none of the other conditions apply

Fig. 2. The default colour schema for encoding clause states.

MUStICCa allows you to select interesting or relevant sets of clauses in an US via a **selection refinement** interface which is accessible as a hierarchy of submenus in the US view's context menu. Starting from some selected subset of the clauses in the current US, you have the options to subselect only the clauses of a given status, the clauses with a given number of literals, the clauses

containing a given literal, just the first or the last few clauses of the selection, or a random subset of a given size.

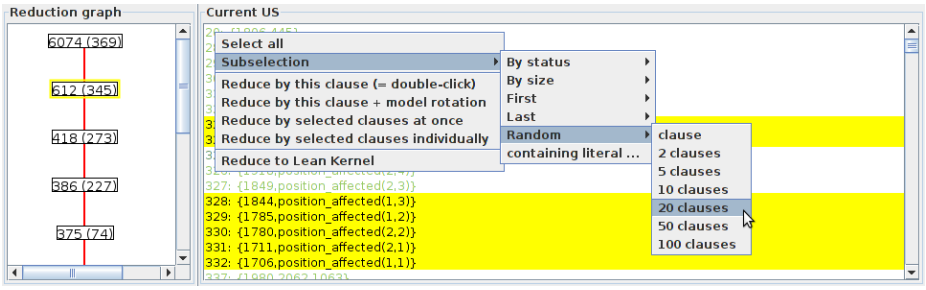

Fig. 3. Subselecting twenty random clauses.

Figure 4 shows the dialog for creating and starting new **reduction agents**. The most important option is the possibility to select one of the predefined heuristics from the drop-down menu. The signal colour for each new agent is initialized with a random point in RGB space, but it can be redefined via the Change button which allows you to define a different colour using Swing's standard colour chooser dialog. Finally, you can independently activate or deactivate model rotation and autarky pruning for the reduction agent.

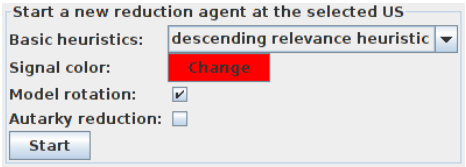

Fig. 4. The dialog for starting a new reduction agent.

A handful of different simple deletion heuristics come pre-defined with the prototype. The heuristics are based either directly on the clause IDs or on a relevance ordering of the selection variables based on their frequency in the last resolution proof. Both of these lists can be traversed either from top to bottom, from the bottom up, or in a spiraling movement starting in the middle of the list. Taken together, we therefore arrive at six different predefined deletion heuristics, all of which are listed in Figure 5. Each of these heuristics can be motivated by structural considerations, but none of them can be expected to perform better

than all the others across instances.

ascending index heuristic	goes through the US clauses by ascending ID
descending index heuristic	goes through the US clauses by descending ID
centered index heuristic	spirals through the US clauses starting in the middle
ascending relevance heuristic	goes through the US clauses in order of relevance, starting with the least relevant clause
descending relevance heuristic	goes through the US clauses in order of relevance, starting with the most relevant clause
centered relevance heuristic	starts with the US clauses of medium relevance, then spiralling out to ever more and less relevant ones

Fig. 5. Table of predefined deletion heuristics.

To provide information about the running agents, an overview of all the executed reduction agents is displayed on the right side of the reduction agent view. This view is a scrollable list of frames which compactly represent relevant information about the currently running reduction agents, each displaying information on the US the respective agent is currently operating on, as well as statistics about the number of successful and unsuccessful SAT calls. The first set of statistics is not based on the SAT calls which were actually executed, but on the number of SAT calls the reduction agent would have had to make if there had been no information about the reduction graph beforehand. Internally, a largely unified treatment of both types of SAT calls is achieved by means of simulated reduction attempts which do not differ from genuine reduction steps except that being mere step data retrievals, they are executed much faster. The second sets of statistics counting the reduction attempts which actually had to be executed are given in brackets. While a reduction agent is running, it can be stopped at any time by clicking the **Stop** button in the corresponding field of the reduction agent overview.

To visualize the actions of reduction agents in the reduction graph, one can visualize what we will call **agent traces**, i.e. the downward paths through the powerset lattice by which the different agents explore the search space. The trace of each readuction agent in the reduction graph is highlighted in that agent's signal colour. If several different agents happen to share a common path segment, this segment will be drawn using parallel lines with multiple colours.

In Figure 7, we see an example of a reduction graph with two different agent traces. There is a red trace on the left side of the graph which ends in a MUS, and another trace which has not yet arrived at a MUS. The thin black lines to the other nodes are the result of manual reductions which were performed without the help of the reduction agent system. Using the **Hide** button in the

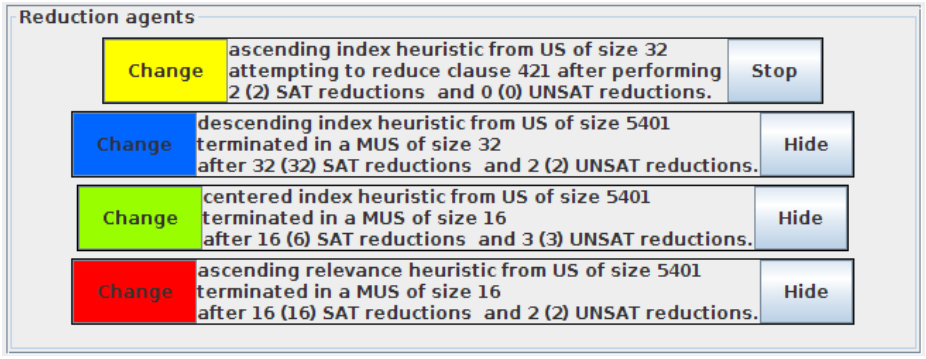

Fig. 6. Example of the reduction agent overview.

information panel of an agent, the coloured trace of each agent can be removed from the reduction graph, leaving behind only a thin black line which looks as if the reductions had been executed manually. This hide functionality is a valuable tool for decluttering the reduction graph of optically dominant details that have become irrelevant.

3 Advanced Features

3.1 Semi-Automatization and Simultaneous Reduction

The US view provides more functionality than just the execution of single reduction attempts. Some more powerful operations which act not on a single clause, but on subsets of the current US, are accessible via the US view's context menu. The first of these options offers what could be called methods of **semi-automatization**. In essence, it allows the user to initiate a batch processing of reduction attempts, where all the clauses which are currently selected in the US view are reduced in turn. The main application of this is to quickly open up several new branches in the reduction graph at once, or to speed up criticality checks. This option is called semi-automatization to contrast it with full automatization based on reduction agents.

The second option is called **simultaneous reduction**, an obvious generalization of the default deletion-based method where multiple clauses can be deactivated at once by setting their selection variables. The corresponding context menu option causes the system to attempt the deletion of all the currently selected entries in the clause list at once. If the attempt was successful, a new node (or link) will appear in the reduction graph, just like in the case of deleting a single clause. The major difference is that after a simultaneous reduction, none of the clauses in the old US will receive the status of an explicitly reduced clause, again to avoid the unaccessibility of possible intermediate USes in the reduction graph.

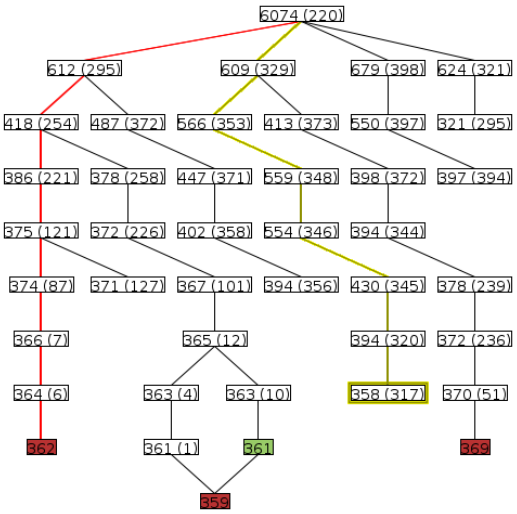

Fig. 7. Example of a reduction graph with reduction agent traces.

Simultaneous reduction should be used sparingly, since if such an attempt fails, no criticality information can be derived, because it is possible that some of the clauses could have been deleted as long as other would have stayed in.

3.2 Model rotation and Lean Kernel Extraction

Model rotation is an additional very useful technique for speeding up deletion-based or extraction, which uses the model returned by the SAT solver to get a lot more information out of unsuccessful reduction attempts. MUStICCa uses a variant called **recursive model rotation** because it continues to flip assignments in the model and then checks whether each model variant satisfies all but a single isolated clause, which is thereby found to be critical. With the model rotation flag, reduction agents can be set to apply model rotation after each unsuccessful reduction attempt. For manual reduction operations, model rotation is available via an item in the US view's context menu.

A very important characterisation of the lean kernel, i.e. the set of all clauses which can be used in some refutation proof, is that it is the complement of the unique largest autark subset. The set of all unusable clauses is identical to the largest subset that can be pruned away via autarky reduction, i.e. the largest autark subset. This result means that we can reduce any unsatisfiable clause set to its lean kernel by finding a maximal autarky and simply applying it. If clause set refinement is used, applying autarky reduction after each successful

reduction step is generally not worthwhile. Still, for purposes of experimentation, MUStICCa supports an autarky reduction flag for reduction agents, and manual lean kernel extraction via the US view's context menu.

3.3 Plug-in Interface for Custom Deletion Heuristics

In addition to providing a set of pre-defined deletion heuristics, MUStICCa allows the user to define and import their own deletion heuristics via its plug-in interface. The basic procedure is to inherit from the abstract Java class `org.kahina.logic.sat.muc.heuristics.ReductionHeuristic` and implementing or overloading its methods as needed. The interface is documented in Kahina's source code documentation at `http://kahina.org/javadoc`.

For purposes of illustration (and as a simple example to build upon), we give the code for a very simple heuristic here.

The source file `RandomHeuristic.java` as displayed in Figure 8 implements random selection of deletion candidates. The decisive points in the code are how the methods `setUC()` (the callback when we have arrived at a new US because the last reduction was successful) and `getNextCandidate()` (which MUStICCa uses to poll deletion candidate IDs) are implemented. The clause IDs representing the current US are randomly permuted and stored. The heuristic keeps track of the candidates that were already tried to determine when it has arrived at a MUS, in which case the `getNextCandidate()` will return a 0.

We can compile this heuristic from the command line if `musticca.jar` is on the classpath: `javac -cp musticca.jar RandomHeuristic.java`. The directory where the resulting class file results must then be added to the `CLASSPATH` environment variable.

To specify a set of deletion heuristics different from the default selection for MUStICCa to use, we create a simple text file which contains the fully qualified class names of all the desired heuristics, one per line (see Figure 9). We can tell MUStiCCa to use this modified set of heuristics instead of the default set by means of the `-hf` command-line option: `java -jar musticca.jar -hf heuristics.txt`. This will cause the specified heuristics to appear in the reduction heuristics drop-down menu in the reduction agent manager (see Figure 10).

```
import java.util.*;
import org.kahina.logic.sat.muc.MUCStep;
import org.kahina.logic.sat.muc.heuristics.ReductionHeuristic;

public class RandomHeuristic extends ReductionHeuristic {
    Set<Integer> alreadyProcessed;
    ArrayList<Integer> candidateList;
    int nextCandidateIndex = 0;

    public RandomHeuristic(){
        alreadyProcessed = new HashSet<Integer>();
        candidateList = new ArrayList<Integer>();
    }

    public void setNewUC(MUCStep uc){
        this.uc = uc;
        candidateList = new ArrayList<Integer>();
        candidateList.addAll(uc.getUc());
        Collections.shuffle(candidateList);
        nextCandidateIndex = 0;
    }

    public int getNextCandidate(){
        for (int i = nextCandidateIndex; i < candidateList.size(); i++){
            int candidate = candidateList.get(i);
            if (!alreadyProcessed.contains(candidate)){
                alreadyProcessed.add(candidate);
                nextCandidateIndex = i+1;
                return candidate;
            }
        }
        return -1;
    }

    public void deliverCriticalClauses(Set<Integer> criticalClauses){
        alreadyProcessed.addAll(criticalClauses);
    }

    public String getName(){
        return "random heuristic";
    }
}
```

Fig. 8. Implementation of the example heuristic `RandomHeuristic.java`

```
RandomHeuristic
org.kahina.logic.sat.muc.heuristics.AscendingIndexHeuristic
org.kahina.logic.sat.muc.heuristics.CenteredIndexHeuristic
org.kahina.logic.sat.muc.heuristics.DescendingIndexHeuristic
```

Fig. 9. Example `heuristics.txt` file for specifying a custom set of reduction heuristics.

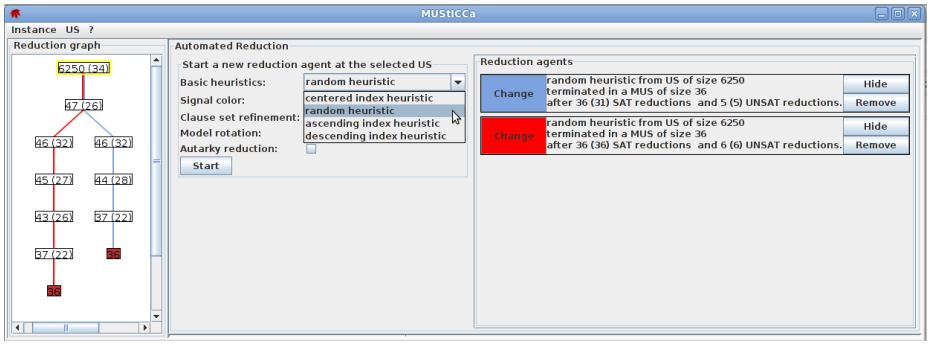

Fig. 10. MUStICCa with a user-defined set of reduction heuristics.