An Approach to Extracting a Small Unsatisfiable Core

Research Thesis

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
QUALITY ASSURANCE AND RELIABILITY

Maya Koifman

Submitted to the Senate of
the Technion - Israel Institute of Technology

TISHREI, 5766 HAIFA OCTOBER, 2006

The Research Thesis Was Done Under The Supervision of
Dr. Ofer Strichman
In the Interdepartmental Program of
Quality Assurance and Reliability.

The Generous Financial Help Of the Technion Is Gratefully Acknowledged.

Table of Contents

List of Tables

List of Figures

Abstract

The thesis addresses the problem of finding a small unsatisfiable core of an unsatisfiable CNF formula. The proposed algorithm, *CoreTrimmer*, iterates over each internal node d in the resolution graph that 'consumes' a large number of clauses M (i.e. a large number of original clauses are present in the unsat core, whose sole purpose is proving d) and attempts to prove them without the M clauses. If this is possible, it transforms the resolution graph into a new graph that does not have the M clauses at its core. *CoreTrimmer* can be integrated into a fixpoint framework similarly to Malik and Zhang's fix-point algorithm (RUN_TILL_FIX). We call this option TRIM_TILL_FIX. Experimental evaluation on a large number of industrial CNF unsatisfiable formulas shows that TRIM_TILL_FIX doubles, on average, the number of reduced clauses in comparison to RUN_TILL_FIX. It is also better when used as a component in a bigger system that enforces short timeouts.

Chapter 1

Introduction

Given an unsatisfiable CNF formula, an unsatisfiable core (UC) is any subset of these clauses that is still unsatisfiable. The problem of finding a *minimum*, *minimal* or just a *small* UC has been addressed rather frequently in the last few years [3, 11, 20, 13, 8], partially due to its increasing importance in formal verification.

The decision problem corresponding to finding the *minimum* UC is a Σ_2-complete problem [7] and we are not aware of an algorithms for finding it that scales. Finding a *minimal* UC (any subset of clauses such that the removal of any one of them makes the formula satisfiable), according to Papadimitriou and Wolfe [14], is D^P-complete[1].

It is questionable whether finding a minimal UC has a practical value, however, since a non-minimal UC can be smaller than a minimal one, as long as it is not contained in it. Therefore heuristics that do not guarantee minimality, can be both faster and better than those that guarantee minimality. The latter are useful only when their result is compared to the core from which they started, and thus can be used, for example, after another, faster algorithm, has already extracted a small core and cannot find a smaller one.

Typically UCs are needed as part of a larger system (such as an abstraction/refinement loop as we will soon describe), and the influence of the size of the UC on the other

[1] D^P is the class containing all languages that can be considered as the difference between two languages in NP, or equivalently, the intersection of a language in NP with a language in co-NP.

parts of the system is only vaguely known. Hence, although more computation time can lead to finding smaller cores, it is not clear whether it is cost-effective in the overall system. This suggests once again that minimality per se is not so important in practice. Algorithms for extracting small cores should be measured instead by their *velocity*: how many clauses they remove from the initial formula per time unit, on average. They should also be measured by how small they can make the core within a time limit, in comparison with other algorithms, and whether they can contribute to a setting in which several of these algorithms are run sequentially or even in parallel. In Section 6 we measure our suggested technique, called *CoreTrimmer*, with these criteria.

Before we describe previous work on this problem, let us mention some of the typical usages of UCs. A small unsatisfiable core reflects a more precise and focused explanation of the unsatisfiability of a given formula. In verification, it is used in several contexts, some of which are the following. Amla and McMillan [2] suggest to use UCs for a proof-based abstraction-refinement model-checking process: the UC of an unsatisfiable BMC instance contains information on the state variables that are sufficient for proving that no bug can be found up to a given depth; based on these state variables they build a refined abstract model and continue to iterate. Kroening et al. [9] use unsatisfiable cores for an iterative process of solving Presburger formulas: the UC is used for checking whether certain under-approximating restrictions on the solution space were used in the proof of unsatisfiability. If the answer is yes, these restrictions should be relaxed. A similar usage of UCs is by Grumberg et al. [6], in a process of under-approximation and widening of BMC formulas corresponding to a multi-threaded process (the UC here again is used for detecting whether the proof of unsatisfiability relies on the underapproximating constraints). Outside verification, the identification of an inconsistent kernel can be important for solving the inconsistency in any constraints satisfaction problem. Furthermore, looking beyond the Propositional world, finding a small unsatisfiable set of constraints is important

for the efficiency of decision procedures like MathSat and CVC [18] that rely on explanations of the reason of unsatisfiability in order to prune the search space. The techniques we will discuss in this paper are equally relevant to such systems as they are for systems based on propositional reasoning.

1.1 Related work

Lynce and Silva [11] suggested an approach for finding a minimal UC, in which a new 'clause selector' variable cs_i, $1 \leq i \leq m$, is added to each of the m clauses of the formula (for example, the i^{th} clause $(l_1 \vee l_2)$ is replaced with $(cs_i \vee l_1 \vee l_2)$). The cs variable is set to TRUE if and only if the clause is not selected. They then use a SAT solver that decides first on the cs variables. When all the clauses become satisfied, it backtracks to the most recent cs variable set to true and changes its assignment to false. If the solver reaches a conflict and consequently backtracks to the cs variables, it means that an unsatisfiable core was found. In such a case it records the size of the core and continues to search for a smaller one, after adding a clause over the cs variables that blocks the solver from repeating the same core. A similar process was suggested also by Oh et al. [13] (the 'Amuse' algorithm), although they modify the backtracking mechanism so it performs a bottom-up search for a UC instead of searching for a satisfying assignment. Different decision heuristics result in different UCs, which are not necessarily minimal.

Huang suggests the 'MUP' (Minimal Unsatisfiability Prover) algorithm in [8]. Rather than using m clause selector variables, he suggests to augment the clauses with minterms over $\log(m+1)$ variables. The augmented formula, he proves, is minimally unsatisfiable iff there are exactly m models over the y variables (because in this case every clause that is removed makes the formula satisfiable). Hence, the problem of proving that an existing set is minimal is reduced to that of model-counting, which MUP performs with a variable elimination technique over BDDs. This technique can be taken one step further towards finding a minimal core, by running it not

more than m times. MUP shows better experimental results than RUN_TILL_FIX (see below), but only, apparently, on hand-made and relatively small formulas, like the pigeonhole problem. None of the benchmarks reported in [8] has more than several thousand clauses, and it is not clear how it scales to industrial problems.

A more practical approach is to find a small core without guaranteeing minimality, while attempting to be efficient and produce intermediate valuable results in case the external process does not wish to wait for the final result.

Zhang and Malik [20] were the first in the verification community, as far as we know, to address this problem from a practical point of view. They suggested a simple and effective iterative procedure for deriving a small unsatisfiable core: they extract an unsatisfiable core from an unsatisfiability proof of the formula provided by a SAT solver and then they run the SAT solver again starting from this core, which may result in an even smaller core. Their script RUN_TILL_FIX repeats this process until the core is equal to a core derived in the previous iteration, or, in other words, until it reaches a fixpoint. The solution and its implementation seem to be the most practical one available, and is indeed widely used. The experimental results that we present in Section 6 are compared against RUN_TILL_FIX.

1.2 Thesis Outline

We describe a new heuristic, called *CoreTrimmer*, for finding a small UC. *CoreTrimmer* takes the role of zVerify in RUN_TILL_FIX. It can be either applied once (and generate a core smaller or equal to that generated by zVerify) or as part of a fixpoint computation, in an algorithm we call TRIM_TILL_FIX. We will concentrate on *CoreTrimmer* from hereon and return to TRIM_TILL_FIX in the description of the experimental results.

The most common approach to solving SAT formulas (finding a satisfying assignment or declaring that the formula is unsatisfiable) is DPLL-solving. Such a solver

performs a search for a satisfying assignment using constraint propagation, intelligently backtracking and learning conflict clauses that prune the search space [12].

New conflict clauses are derived in a process called Conflict Analysis, by (conceptually) traversing backwards the conflict graph and locating the reason for the conflict. This process can be interpreted as a series of resolution steps [20]. The SAT solver can output a graph reflecting the resolution steps, known as the *resolution graph*. The nodes of a resolution graph represent clauses, and the single sink node of this graph represents the empty clause. Each internal node has two parents, which represent the clauses from which it was resolved. In practice this graph can represent *Hyper-resolution* (a result of several resolution steps) and hence each node can have more than two parents. The general idea of the *CoreTrimmer* algorithm, described in detail in Section 4, is the following. *CoreTrimmer* locates internal nodes in the resolution graph that *dominate* other nodes, called the *minions* (i.e., all the paths from a minion node to the sink node go through the dominator), and checks whether they can be proved without their minions. If the answer is yes, the minions can be removed, and consequently the size of the UC is decreased. In such a case the resolution graph has to be transformed so it reflects the new proof. This transformation is the subject of Section 4.1. *CoreTrimmer* repeats this process until no changes in the graph can be made. Experimental results show that integrating this procedure in a fixpoint script in the style of RUN_TILL_FIX, is better than RUN_TILL_FIX, at least with the relatively short timeouts we tried (30 and 60 minutes). *CoreTrimmer* has the advantage that it generates intermediate results rather fast. Hence, while in many cases RUN_TILL_FIX times out (i.e. it cannot finish the first iteration after the initial core within the time limit), *CoreTrimmer* almost always finishes several iterations by that time, even if in the long run RUN_TILL_FIX produces smaller cores.

Chapter 2

Preliminaries

This thesis relies on propositional logic and, to be more specific, CNF (Conjunctive Normal Form) formulas. Following are 6 definitions adapted from Gallier [4].

Definition 1
Boolean variable v_i - *or just* variable*, is a propositional symbol*
Literal l_i - *is either a propositional symbol* v_j *or the negation* $\neg v_j$ *of a propositional symbol*
Clause c_i - *a disjunction of literals, that can be represented as a set of literals*

$$c_i = (l_1 \vee \ldots \vee l_k) \qquad LS_{c_i} = \{l_1, \ldots, l_k\}$$

CNF formula φ - *a conjunction of clauses, that can be represented as a set of clauses*

$$\varphi = (c_1 \wedge \ldots \wedge c_n) \qquad CS_\varphi = \{c_1, \ldots, c_n\}$$

It can be shown that every propositional formula can be converted to an equally satisfiable Conjunctive Normal Form in polynomial time [19].

Definition 2 *The set of* truth values *is the set* $\{T, F\}$*. Each logical connective* X *is interpreted as a function* H_X *with range* $\{T, F\}$*. The logical connectives are*

interpreted as follows:

P	Q	$H_{\neg}(P)$	$H_{\wedge}(P,Q)$	$H_{\vee}(P,Q)$
T	T	F	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	F

Definition 3 *A* truth assignment *is a function* $v : PS \to \{F, T\}$ *assigning a truth value to all the propositional symbols. Every assignment* v *extends to a unique function* $\tilde{v} : PROP \to \{F, T\}$*, where* $PROP$ *is a set of propositional formulae (propositions), satisfying the following clauses for all* $A, B \in PROP$ *:*

$$\begin{aligned}
\tilde{v}(\bot) &= F \\
\tilde{v}(P) &= v(P), \forall P \in PS \\
\tilde{v}(\neg A) &= H_{\neg}(\tilde{v}(A)) \\
\tilde{v}((A \wedge B)) &= H_{\wedge}(\tilde{v}(A), \tilde{v}(B)) \\
\tilde{v}((A \vee B)) &= H_{\vee}(\tilde{v}(A), \tilde{v}(B))
\end{aligned}$$

Definition 4 *A proposition* A *is* satisfiable *if there is a truth assignment* v *such that* $\tilde{v}(A) = T$*. Such an assignment is called a* satisfying assignment*. A proposition is* unsatisfiable *if it is not satisfied by any assignment.*
The problem of determining whether any arbitrary proposition is satisfiable is called the satisfiability problem *or, in short, SAT.*

Definition 5 *Given a set of propositions* Γ*, we say that* A *is* logically implied by Γ*, denoted by* $\Gamma \models A$*, if for all assignments* v*,* $\tilde{v}(B) = T$ *for all* $B \in \Gamma$ *implies that* $\tilde{v}(A) = T$*.*

Definition 6 *A clause is* conflicting *with regard to an assignment* v
if $\forall l_j \in \{l_1, \ldots, l_k\}$ $\tilde{v}(l_j) = F$*.*

Definition 7 *An* empty clause *is a clause consisting of* $\bot$*. This clause is unsatisfiable - it is conflicting for any assignment.*

Definition 8 *An* unsatisfiable core *of an unsatisfiable CNF formula* φ, $CS_\varphi = \{c_1, \ldots, c_n\}$, *is also a CNF formula* ψ, $CS_\psi = \{c_i, \ldots, c_j\}$, *where* $1 \leq i < j \leq n$, *so that* $\{c_i, \ldots, c_j\} \subseteq \{c_1, \ldots, c_n\}$ *and* ψ *is also unsatisfiable.*

Resolution is a proof system for CNF formulas with one inference rule:

$$\frac{(A \vee x)\ (B \vee \neg x)}{(A \vee B)}$$

where A,B are disjunctions of literals (possibly with 0 disjuncts, i.e. the constant FALSE). The clause $(A \vee B)$ is the *resolvent*, and $(A \vee x)$ and $(B \vee \neg x)$ are the *resolving clauses*. The resolvent of the clauses (x) and $(\neg x)$ is the empty clause $(\perp)$. Each application of the resolution rule is called a *resolution step*.

Lemma 1 *A propositional CNF formula is unsatisfiable if and only if there exists a finite sequence of resolution steps ending with the empty clause.*

A sequence of resolution steps, each one uses the result of the previous step as one of the resolving clauses of the current step, is called *Hyper-resolution*. For example, from

$$(x_1 \vee x_2 \vee x_3)(\neg x_1 \vee x_4)(\neg x_2 \vee x_5)$$

we can derive $(x_3 \vee x_4 \vee x_5)$ by two resolution steps (first over x_1, then over x_2), or by one hyper-resolution step.

The hyper-resolution steps leading to the derivation of the empty clause can be depicted in a *Hyper-resolution graph* (or, simply, a resolution graph). From hereon, we use the terms *node* and *clause* interchangeably, since every node represents a clause.

Definition 9 *A* Hyper-resolution graph *corresponding to an unsatisfiability proof by resolution, is a Directed acyclic Graph* $G(V, E, s)$ *with a single sink node* $s \in V$, *in which the nodes represent CNF clauses: the leaf nodes (the sources) represent original clauses, the inner nodes represent clauses derived by resolution, and the sink represents the empty clause. Each node can be inferred from its parent nodes by some sequence of resolution steps.*

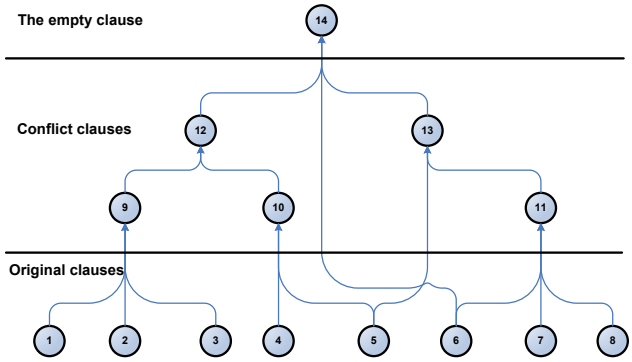

Figure 2.1: A resolution graph

Modern DPLL-based SAT solvers can output a Hyper-resolution proof of unsatisfiability. The intermediate clauses in this proof are the conflict clauses that were generated during the run, and that are on a path from the leafs to the empty clause.

We now generalize resolution graphs to *Clause Implication Graphs*:

Definition 10 (Clause Implication Graph) *A* Clause-Implication Graph *(CIG)* $G(V, E, s)$ *is a directed acyclic graph with a single sink node* $s \in V$*, in which the nodes represent CNF clauses, and each node is logically implied by the conjunction of clauses represented by its parents.*

A CIG is less restrictive than hyper-resolution graphs. They can have such edges as shown in Figure **2.2**, where Φ_1, Φ_2 are disjunctions of literals, and p, x are variables.

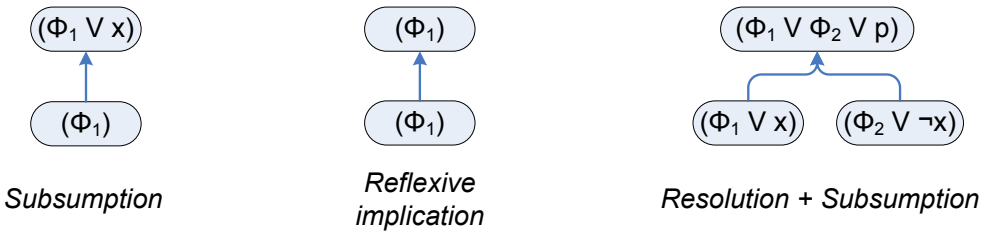

Figure 2.2: CIG Edges

Other implications forbidden by hyper-resolution are also possible. Figure **2.3** depicts an example of a Clause Implication Graph.

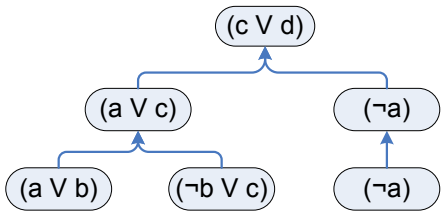

Figure 2.3: A Clause Implication Graph (CIG)

Let L denote the leaf nodes of a CIG, and assume that the sink node s represents the empty clause. By definition of CIG, the conjunction of the L clauses is unsatisfiable, and hence there exists a corresponding resolution proof of unsatisfiability starting from the same nodes. Therefore, for the purpose of finding small UCs, CIGs are sufficient for the analysis. Our construction will begin from the hyper-resolution graph, which can be derived from the resolution trace given to us by the SAT solver, but will transform it to a CIG as the algorithm progresses.

Chapter 3

Dominators

Prosser [16] introduced the notion of dominance in the context of Flowgraph analysis (originally a term related to code analysis and compilers). A Flowgraph $G = (V, E, r)$ is a directed graph such that every vertex is reachable from a distinguished root vertex $r \in V$. A vertex $d \in V$ *dominates* $v \in V, v \neq d$, if every path from r to v includes d. d *immediately dominates* v if it dominates v and there is no other node on the path between them that dominates v. We name v a *minion* of d. The set of minions of d is denoted by $M(d)$. Note that there is a total order of dominance within the set of dominators of every vertex. A node is called a *dominator* if it dominates at least one node.

In order to adapt the notion of dominators to CIGs, we conceptually reverse the edges of the CIG. Thus, the sink node now becomes the root. Note that CIGs, in contrast to Flowgraphs, are acyclic, although this does not change the definition of dominators. Figure **3.1** presents a *Dominator Tree*, which represents the immediate dominance relation, of a CIG.

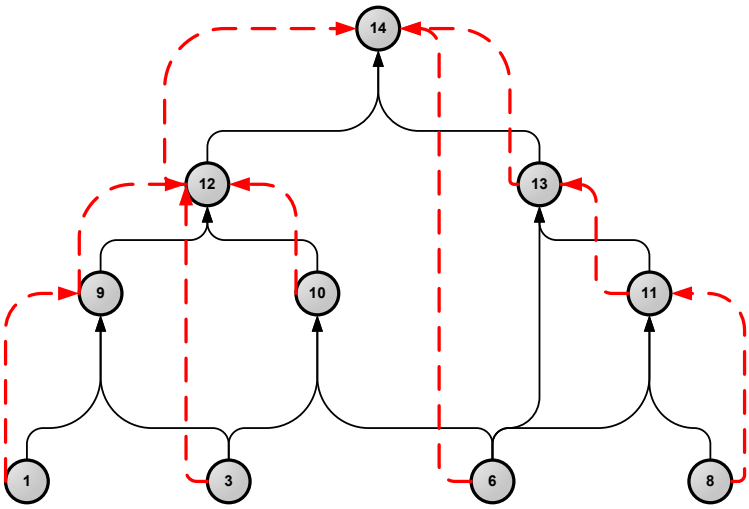

Figure 3.1: A Dominator Tree over a reversed CIG. Solid edges belong to the CIG, dashed edges belong to the Dominator Tree. There is a dashed arrow from clause c to c' in Dominator Tree if c is the immediate dominator of c'.

3.1 Finding Dominators

3.1.1 An Intuitive Algorithm

Aho and Ullman [1] and Purdom and Moore [15] describe a straightforward algorithm for finding dominators in $O(|V| \cdot |E|)$ time:

For each vertex $v \neq r$:

1. Determine, by means of a search from r, the set S of vertices reachable from r by paths which avoid v.

2. The vertices in $V - v - S$ are exactly those which v dominates.

3.1.2 The Simple Lengauer-Tarjan Algorithm

The algorithm is based on the concept of *semidominators*, which give an initial approximation to the immediate dominators:

Suppose that all the nodes in a flowgraph G are numbered from 1 to n during a DFS from the root node in the order, in which they are reached during the search, and

that the nodes are identified by these numbers. A path $P = (u = v_0, v_1, \ldots, v_{k_1}, v_k = v)$ in G is a *semidominator path* if, according to this preorder node numbering, $v_i > v$ for $1 \leq i \leq k-1$. The *semidominator* of v is defined as

$$sdom(v) = \min\{u | \text{there is a semidominator path from } u \text{ to } v\}$$

The algorithm consists of the following general four steps:

Step 1 Carry out a depth-first search of the problem graph. Number the vertices from 1 to n as they are reached during the search.

Step 2 Compute the semidominators of all vertices. Carry out the computation vertex by vertex in decreasing order by number.

Step 3 Implicitly define the immediate dominator of each vertex.

Step 4 Explicitly define the immediate dominator of each vertex, carrying out the computation vertex by vertex in increasing order by number.

The algorithm is described in detail by Lengauer and Tarjan [10] and its variants are discussed by Georgiadis [5].

3.2 Dominators in a Clause-Implication Graph

We will refer from hereon to a clause set and the formula obtained by conjoining the clauses in the set as the same thing, when the meaning is clear from the context.

Let $LM(d) \subseteq L$ denote the leaf minions of some dominator d. By definition of a CIG, $\bigwedge_{l \in L} l \models d$. The significance of a dominator $d \in V$ in a CIG is that if $L \setminus LM(d) \models d$, then $\bigwedge_{l \in (L \setminus LM(d))} l \models d$. In other words, if d is implied by the leafs which are not its minions, then $LM(d)$ are redundant in the Unsatisfiable Core. Yet removing $LM(d)$ from the CIG is not sufficient, if we want to repeat this process. The problem is that such a removal does not leave us with a valid CIG. The *CoreTrimmer* algorithm, presented in the next section, iterates over dominators in the CIG, and substitutes whenever possible (i.e. when $L \setminus LM(d) \models d$) the old proof of the dominator d with a proof of $L \setminus LM(d) \models d$.

Chapter 4

The *CoreTrimmer* algorithm

Our algorithm for decreasing the size of the UC is sketched in Figure **4.1**.

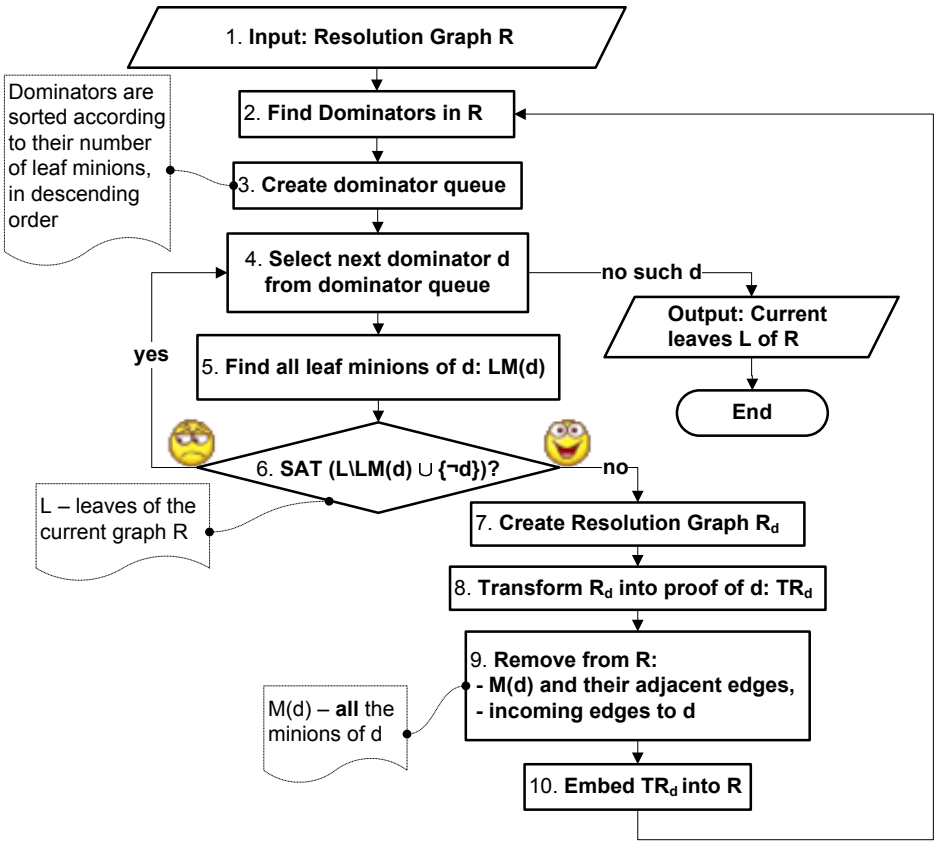

Figure 4.1: The *CoreTrimmer* algorithm

Until Step 5 *CoreTrimmer* is self explanatory. Step 6 Checks whether a dominator d has an alternative proof without $LM(d)$, which amounts to checking the satisfiability of $\varphi' : ((L \setminus LM(d)) \cup \{\neg d\})$, where $\{\neg d\}$ denotes the set of unit clauses corresponding to the negation of the clause d. For example, if $d = (z_1 \vee \ldots \vee z_n)$ is a dominator, then $\{\neg d\}$ are the clauses $(\neg z_1) \ldots (\neg z_n)$, which, for a reason that will soon be clear, we refer to as the *assumptions*. If φ' is satisfiable, the attempt failed and it proceeds to the next dominator in the queue. Otherwise, the solver creates a proof of unsatisfiability of φ' - the resolution graph R_d. Then, relying on the equivalence

$$((L \setminus LM(d)) \cup \{\neg d\}) \models \bot \qquad \Longleftrightarrow \qquad L \setminus LM(d) \models d,$$

in Step 8 *CoreTrimmer* transforms the resolution graph R_d into a proof of d, and builds a corresponding CIG TR_d. A transformation is needed because the proof of φ''s unsatisfiability, R_d, as generated by the SAT solver, is a proof of the empty clause that uses assumptions. We have to transform it into a proof of d without the assumptions, TR_d. We discuss two different methods for performing this transformation in Section 4.1. In step 9 *CoreTrimmer* removes from R the graph elements corresponding to the old proof of d and replaces it with the new one, TR_d, in step 10. That is, it removes all the minions of d together with their adjacent edges and incoming edges to d, and *embeds* TR_d into R instead.

Definition 11 (Graph embedding) *The embedding of a graph $G(V, E)$ in another graph $G'(V', E')$, is a graph $G''(V'', E'')$ such that $V'' = V \cup V'$ and $E'' = E \cup E'$.*

Figure **4.4** illustrates an example of the embedding operation.

After the old proof is replaced with the new one, the new graph is still a CIG, but has fewer leafs, and hence a smaller unsatisfiable core than the original graph.

To implement this idea *CoreTrimmer* uses three types of Conflict Implication Graphs:

- The *Main CIG* R, which initially is a resolution graph, and in subsequent steps holds the CIG with the smallest unsatisfiability core currently known of the original formula,

- The *Small CIG* R_d, which corresponds to a proof of unsatisfiability of $((L \setminus LM(d)) \cup \{\neg d\})$ for some dominator d,
- The *Transformed CIG* TR_d, which is the transformation of R_d into a CIG corresponding to a proof of $L \setminus LM(d) \models d$.

Note that in Step 3 of the algorithm the next dominator is selected according to a predefined order. We tried 3 types of ordering: beginning with the dominator with the largest number of leaf minions - HIGH, with the least number of leaf minions - LOW, and selecting dominators at random - RANDOM. The results can be seen in the Experimental Results section 6.3.2. LOW when compared to HIGH showed definite degradation in performance both with regard to velocity and core reduction. RANDOM definitely worsens core reduction too, without increasing velocity, in comparison with HIGH. Supported by these results, our algorithm uses HIGH ordering, which makes our algorithm greedy.

4.1 Transforming the Resolution Graph

Recall that in Step 8 *CoreTrimmer* is required to transform the resolution graph R_d, corresponding to a proof of $((L \setminus LM(d)) \cup \{\neg d\}) \models \bot$, into a CIG TR_d that corresponds to a proof of $L \setminus LM(d) \models d$. We present two possible ways to derive TR_d from R_d. Let $d = (z_1 \vee \ldots \vee z_n)$ be the dominator, and assume that no two literals in this clause are the same. As before we call the unit clauses in $\{\neg d\}$, *assumptions*.

4.1.1 Simple Transformation

When $((L \setminus LM(d)) \cup \{\neg d\})$ is proven to be unsatisfiable, a subset $L' \subseteq L \setminus LM(d)$ has paths to the empty clause in the resolution graph. This implies that $L' \cup \{\neg d\}$ is unsatisfiable, or equivalently, that L' implies d. Thus, $TR_d(V, E)$ is defined by $V = L' \cup d$ and for all $l' \in L'$, $(l', d) \in E$. Embedding this graph into R corresponds

to adding edges from the L' clauses to d itself. The following drawing illustrates a simple transformation and embedding for dominator node 13:

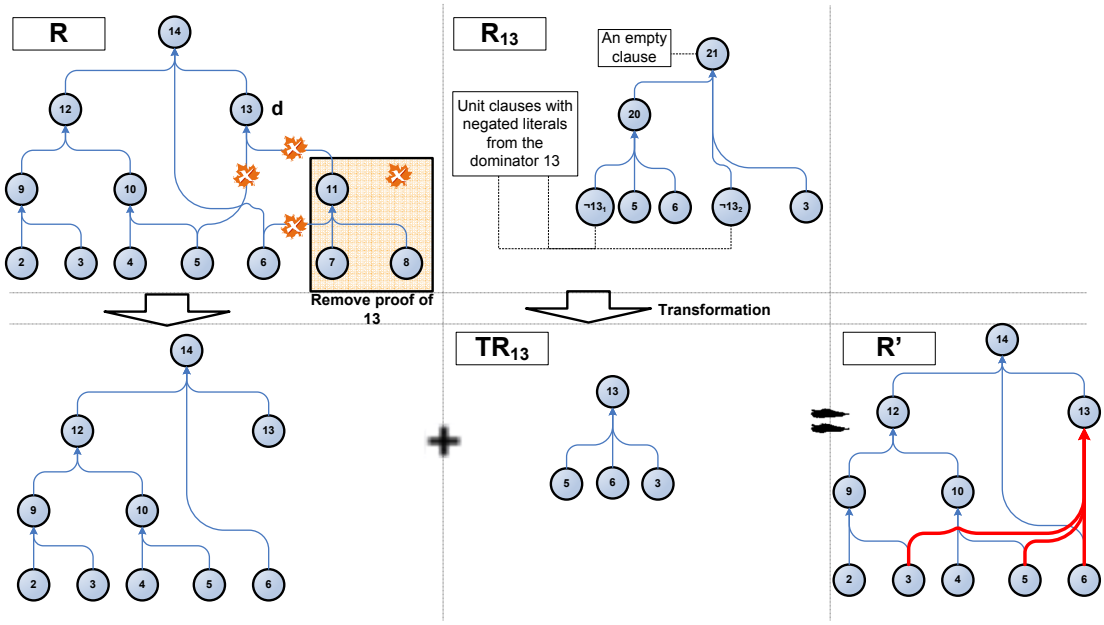

Figure 4.2: The Simple Transformation

The disadvantage of this method is that it is too coarse. See Section 6.3.1 for the relevant experimental results. Since it disregards the conflict clauses, it loses the information about the way these original clauses imply the dominator. Consequently it provides little opportunity for removing more dominators in the main resolution graph. On the other hand, we cannot simply add the conflict clauses, because some of them are derived from the assumptions. What we need is a method for deriving a resolution proof of d from L'. We suggest the *Bubble transformation* method for this derivation.

4.1.2 Bubble Transformation

For a given clause $d = \{z_1, ..., z_k\}$ and clauses $\{c_1, ..., c_n\}$ we build an assumption set $A = \{(\neg z_1), ..., (\neg z_k)\}$ and a new formula $F = \{c_1, ..., c_n\} \cup A$.

The *Convert* recursive transformation, which appears below, converts a resolution proof Π of the unsatisfiability of F provided by a SAT solver, to a new proof of d. It is initially called with the empty clause. Note that *Convert* is never called with an assumption leaf and that the assumption leaves do not participate in the transformed graph. The reason for that is that there can be no resolution between two assumptions, because they do not share any variables. Therefore, any node can have at most one assumption as a child and such cases are covered in lines 3 and 4. The *Resolve* step resolves between two transformed clauses on the same variable as the original resolution variable, if it still exists in both clauses in different polarity. In the end of this section we give an intuitive description of an implementation of this procedure, while for now we concentrate on correctness. The relevance of this general procedure to our case is clear: d is the dominator, A is $\{\neg d\}$ and $\{c_1, \ldots, c_n\}$ are the clauses of $L \setminus LM(d)$.

1: **procedure** CONVERT(Node: n)
2: **if** n is leaf **then return** NewNode(n)
3: **if** $left(n) = (\neg z_i)$ **then return** $Convert(right(n))$
4: **if** $right(n) = (\neg z_i)$ **then return** $Convert(left(n))$
5: **return** NewNode($Resolve(Convert(right(n), Convert(left(n))))$)

The following drawing demonstrates a bubble transformation with *Convert*, where $z \in d$:
The following drawing illustrates a bubble transformation and embedding for dominator node 13:

Proposition 1 *Let $\perp$ denote the empty clause of the proof Π (the proof of F's unsatisfiability). Then $Convert(\perp)$ returns a valid resolution proof Π' of $\{c_1, \ldots, c_n\} \models d'$, so that $literals(d') \subseteq literals(d)$.*

Proof We use the term *proof of unsatisfiability* in order to emphasize that our proof is based on a resolution graph, not a hyper-resolution graph. The information

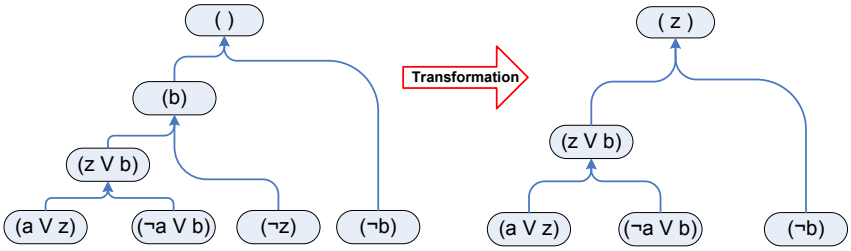

Figure 4.3: A bubble proof transformation, where $z \in d$

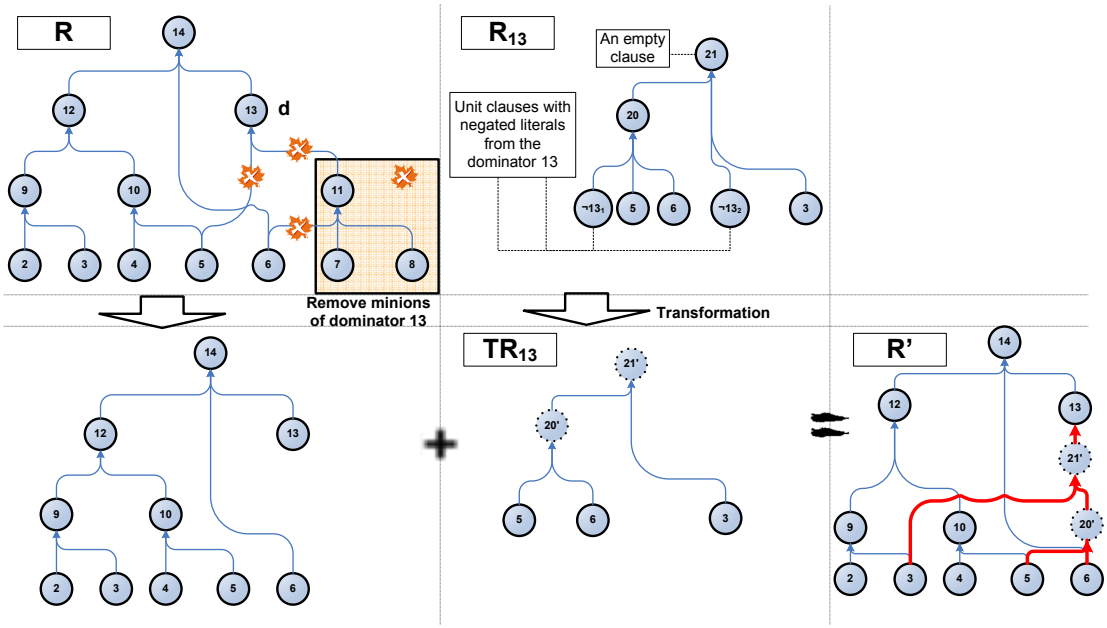

Figure 4.4: The Bubble Transformation

provided by the SAT solver is enough for reconstructing any of these graphs. In order to simplify presentation of the proof even more, we use set notation for clauses to represent their literal sets.

Let $n' = Convert(n)$. We will prove the proposition by induction on the resolution graph structure using the following invariant:

- n' is well-defined
- $n \subseteq n' \subseteq (n \cup d)$.

Base step: if n is a leaf then $n' = n$, which is well-defined and, trivially,

$$n \subseteq n' \subseteq (n \cup d)$$

Induction step: there are two different cases - one for lines 3 and 4, and the other - for line 5.

Lines 3 and 4: Suppose that n is an inner node that was resolved by the two clauses n_l and n_r using the resolution variable t. Let $n'_r = Convert(n_r)$ and $n'_l = Convert(n_l)$. If w.l.o.g. $n_l = (\neg z_i)$, then, according to the algorithm: (1) $n' = n'_r$. Since the proof is a DAG, n' is well-defined by the induction hypothesis. Also, by induction: (2) $n_r \subseteq n'_r \subseteq (n_r \cup d)$. It must hold that $t = z_i$, since this is the only variable that might be common to n_l and n_r. Therefore: (3) $n \cup \{z_i\} = n_r$. Combining these expressions we get

$$n \overset{(3)}{\subseteq} n_r \overset{(2)}{\subseteq} n'_r \overset{(2)}{\subseteq} (n_r \cup d) \overset{(3)}{=} (n \cup \{z_i\}) \cup d \overset{z_i \in d}{=} (n \cup d)$$

Therefore

$$n \subseteq n'_r \overset{(1)}{=} n' \subseteq (n \cup d)$$

Line 5: Assuming that the invariant holds for n'_r and n'_l, we need to prove that a resolution step is valid on clauses n'_r and n'_l, i.e. that, they have opposite literals of at least one variable. Now, since Π was a valid proof, it must hold that there exists a literal t so that w.l.o.g $t \in n_r$ and $\neg t \in n_l$. Since $n_r \subseteq n'_r$ and $n_l \subseteq n'_l$, it holds that

$t \in n'_r$ and $\neg t \in n'_l$. Therefore n' can be derived by resolution between n'_l and n'_r on the same t, and n' is well-defined.

We need to prove that $n \subseteq n' \subseteq (n \cup d)$. Indeed,

$$n \overset{Resolution}{=} ((n_r \cup n_l) \setminus \{t, \neg t\}) \overset{Induction}{\subseteq} ((n'_r \cup n'_l) \setminus \{t, \neg t\}) \overset{Resolution}{=} n'$$

$$\begin{aligned} n' &= ((n'_r \cup n'_l) \setminus \{t, \neg t\}) \overset{Induction}{\subseteq} (((n_r \cup d) \cup (n_l \cup d)) \setminus \{t, \neg t\}) \\ &= (((n_r \cup n_l) \setminus \{t, \neg t\}) \cup (d \setminus \{t, \neg t\})) = (n \cup (d \setminus \{t, \neg t\})) \subseteq (n \cup d) \end{aligned}$$

In particular, the invariant implies that for the empty clause $\perp$:

$$Convert(\perp) \subseteq (\perp \cup d) = d$$

□

Convert can also be implemented with the following, more intuitive procedure:

1: **for** each assumption $(\neg z_i)$, $1 \leq i \leq n$ in R_d **do**
2: Add z_i to all clauses on all the paths from $(\neg z_i)$ to the sink node.
3: Remove the assumption $(\neg z_i)$ from the graph.

Chapter 5

Variations and Optimizations

5.1 Successful Optimizations

The following optimizations proved useful in improving performance, as can be seen in the experimental result section 6.3. Our tool includes all of them.

5.1.1 Dominator Ordering

In step 3 of the algorithm (Figure **4.1**) dominators are ordered in a queue according to a given criterion and then every time the algorithm reaches step 4, a dominator is selected from that queue. The dominators are ordered according to their respective number of leaf minions, in decreasing order.

5.1.2 Incremental Solving

In step 6 of the algorithm (Figure **4.1**) rather than checking just $((L \setminus LM(d)) \cup \{\neg d\})$, *CoreTrimmer* conjoins with this formula all the conflict clauses in R that are not on any path from the minions to the sink node. This addition does not change the satisfiability of the formula, because these clauses are logically implied by $L \setminus LM(d)$. But they make the SAT solving stage incremental[17], and hence far more efficient.

5.1.3 CIG Renewal

In step 8, if none of the assumptions participate in the proof, *CoreTrimmer* takes a different route. In this case R_d, which is the proof of unsatisfiability of $((L \setminus LM(d)) \cup \{\neg d\})$, can also be seen as the proof of unsatisfiability of $L \setminus LM(d)$, which are a subset of the clauses in the original formula. Let $L' \subseteq L \setminus LM(d)$ be the leaves of R_d. L' is a UC of $L \setminus LM(d)$, but also of the original formula, and it is smaller than the smallest core known so far (because the core of the current R is L). So, *CoreTrimmer* assigns $R = R_d$ and returns to line 2.

5.1.4 Minion Caching

After every successful trial the main graph is changed and, therefore, the dominators are recomputed. On account of this, the set of minions of many dominators has not changed, or has even increased. Therefore, caching the leaf minions of dominators after unsuccessful trials helps prevent the abovementioned redundant trials. The next time a dominator is selected for trial, its current leaf minions are compared to the cached ones. If the current minions include cached minions, then there is no use trying this dominator, since the result will be "satisfiable" again.

Proposition 2 *Let L be the current leaves, $LM(d)$ - the current leaf minions, L' - the previous leaves and $LM'(d)$ - the cached leaf minions.*

$$LM'(d) \subseteq LM(d) \quad \textit{and} \quad \exists \alpha, \alpha \models ((L' \setminus LM'(d)) \cup \{\neg d\}) \quad \Longrightarrow$$

$$\exists \beta, \beta \models ((L \setminus LM(d)) \cup \{\neg d\})$$

Proof Since that trial was unsuccessful, there is an assignment α so that: $\alpha \models ((L' \setminus LM'(d)) \cup \{\neg d\})$. Since leaves are only removed, $L \subseteq L'$. Moreover, from the definition of leaf minions $LM(d) \subseteq L$. Hence, if $LM'(d) \subseteq LM(d)$ then $LM'(d) \subseteq LM(d) \subseteq L \subseteq L'$. From this follows $(L \setminus LM(d)) \subseteq (L' \setminus LM'(d))$. Therefore, $\alpha \models ((L \setminus LM(d)) \cup \{\neg d\})$.

□

5.2 Less Successful Variations

The following variations were tried, but have not yielded significant improvements:

5.2.1 Refinement

If the result of the trial at step 6 is "satisfiable", the trial can be refined by removing only part of the minions. This can be done in several ways:

- Repeat the trial, but add back those leaf minions that are not satisfied by the assignment that made the trial's result "satisfiable". This will ensure that the assignment does not satisfy the new formula, and will increase the chances that the formula is unsatisfiable. There is at least one such unsatisfied minion, as suggested by the following proposition:

  Proposition 3

  $$\exists \alpha, \alpha \models ((L \setminus LM(d)) \cup \{\neg d\}) \quad \Longrightarrow \quad \alpha \not\models LM(d)$$

  Proof Let α be the satisfying assignment $\alpha \models ((L \setminus LM(d)) \cup \{\neg d\})$. Assume by contradiction that this truth assignment satisfies all the leaf minions: $\alpha \models LM(d)$. Then it follows that
  $\alpha \models (((L \setminus LM(d)) \cup \{\neg d\}) \cup LM(d)) \quad \Longrightarrow \quad \alpha \models (L \cup \{\neg d\}) \quad \Longrightarrow \quad \alpha \models L$,
  in contradiction to the fact that L is an unsatisfiable core.

  □

- Retry with all the leaf minions added back - the formula to check is $L \cup \{\neg d\}$. Since L is unsatisfiable, this formula is also unsatisfiable. If there is a nonempty set of leaf minions $LM'(d) \subset LM(d)$ that do not participate in the resolution of the empty clause, then they can be removed from the core, and the new proof of d, with sources in $L \setminus LM'(d)$, can be added to the main graph. However,

if all the leaf minions of d are involved in the resolution, the trial is considered unsuccessful, since it provides no new information in addition to the already known fact that the dominator is implied by the leaves of the CIG.

- Try to add back minions iteratively - add back one minion, if the formula is still satisfiable then add back another minion without stopping the solver, because all the learned conflict clauses are still valid. Continue these sub-trials until one of them succeeds or until all the clauses are added back. If the latter is the case, the trial is considered unsuccessful. However, due to the fact that a dominator may have many leaf minions, which implies many such iterative sub-trials, immediate minions can be used instead. Try adding back immediate minions of the dominator, which are not satisfied by the satisfying assignment, one by one, beginning, for instance, with immediate minions that in turn dominate the least number of leaf minions. The reason for adding back immediate minions is that they, in a way, represent their respective leaf minions. If an immediate minion is needed for proving the dominator, then its minions are needed too, and instead of trying unsuccessfully a group of leaf minions, they are added back all at once, when the appropriate immediate minion is added back. However, the fact that an immediate minion is not needed does *not* necessarily mean that its leaf minions are not needed. Therefore, there are cases where some immediate minions can be removed, but none of the leaf minions can.

5.2.2 Lazy Dominator Evaluation

In every successful trial the main CIG is modified (minions are removed and a sub-graph is added). Unfortunately, every such modification invalidates the dominator tree and, hence, the dominators are recomputed. In order to minimize the number of these costly computations, we tried "lazy evaluation" of dominators. After every successful trial, when the graph was changed following the embedding, we marked as irrelevant all the dominators in the dominator queue, which might have become

invalidated. In addition to the dominator nodes that have been actually removed, these are the nodes that used to dominate the new sources of the dominator, as can be seen in Figure **5.1**. So, for the following trial we select a dominator, which has not been invalidated, from the queue. Only when the queue is empty, the dominator queue is recomputed and the dominators are inserted into the dominator queue.

However, this technique modifies the dominator ordering strategy. Dominators are selected according to their historic number of minions, which does not necessarily reflect the current state.

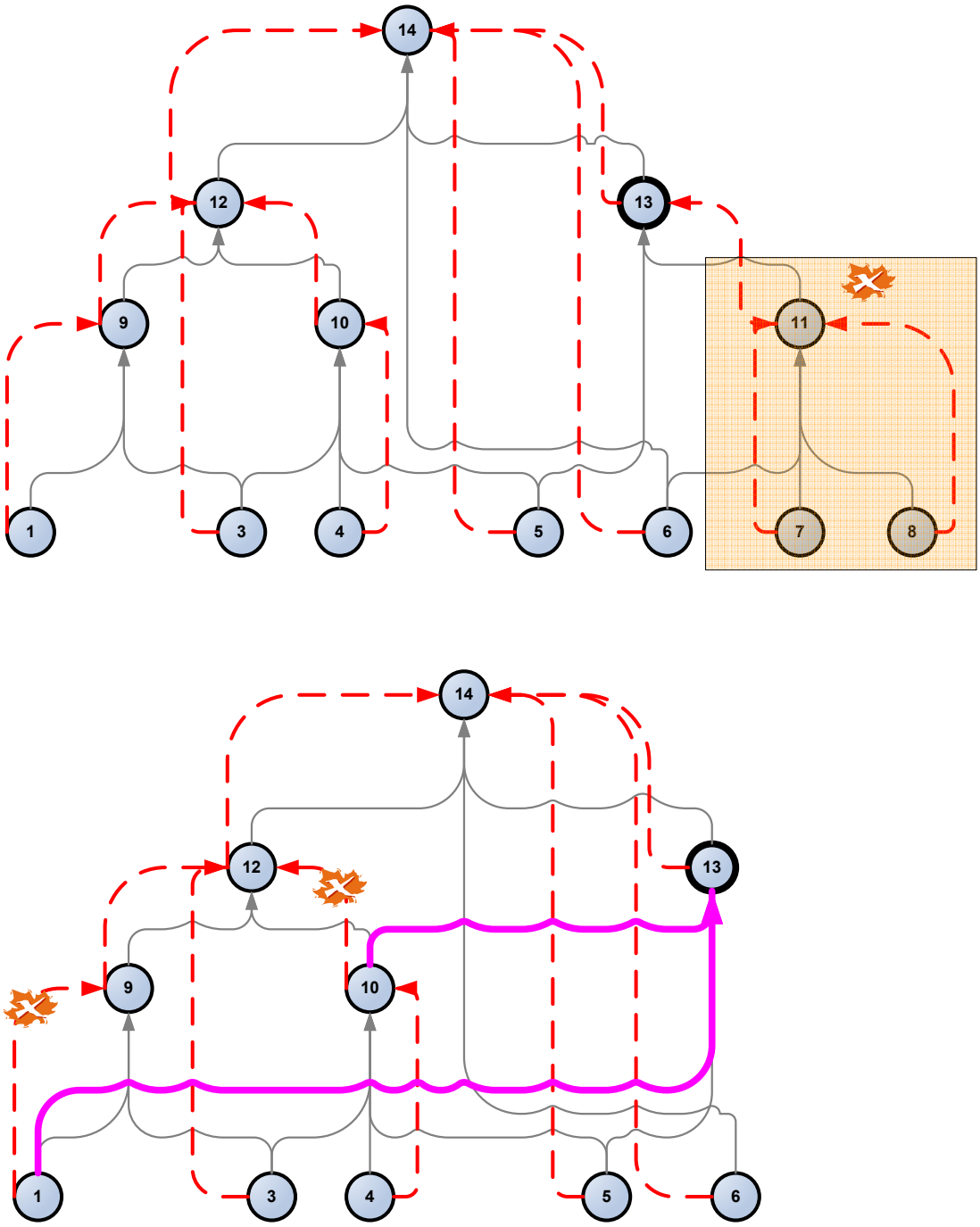

Figure 5.1: Lazy Dominator Evaluation - example. Node 13 - dominator, nodes 1 and 10 are its new parents. Node 1 is no longer dominated by node 9, and node 10 is no longer dominated by node 12.

Chapter 6

Experimental Results

6.1 Experimental Results

The implementation of the dominator algorithm in our tool TRIMMER is the SLT variant of the Lengauer-Tarjan algorithm[10] (which runs in $O(|E| \log |V|)$ time), as provided by the authors of [5] and published on their web site.

We used version 2004.11.15 of zChaff, zVerify and RUN_TILL_FIX for both the comparison and the extraction of the resolution traces.

The benchmark suite is composed of 120 unsatisfiable CNF instances from the industrial category of the SAT competitions, from IBM formal verification benchmarks, and BMC instances from the Sun's PicoJava benchmarks that were used in [2]. We did not include benchmarks that timed-out with both *CoreTrimmer* and RUN_TILL_FIX. The initial number of clauses ranges from $1,300$ to $800,000$, and the largest initial core size, which is our starting point, has around 160,000 clauses.

We measured two parameters: core reduction (the difference between the final and the initial number of clauses) and average velocity (core reduction divided by the time spent on the reduction). We used two different timeouts - $1,800$ seconds and $3,500$ seconds. Since UCs are typically used within a larger system in which they are extracted many times, relatively short timeouts reflect what is practically done for best overall tuning. For such systems velocity seems to be more relevant,

assuming the process of decreasing the size of the UC is interrupted after a while, without waiting for the smallest core possible. The timeouts do not include the time of the first run of the solver that extracts the first resolution trace, since this step is common to all tools.

The competing systems in our benchmark are:

(T) A single run of TRIMMER.

(Z) RUN_TILL_FIX.

(A) TRIM_TILL_FIX:

Running TRIMMER until it terminates, then running zChaff on the new core, then rerunning (T) starting from the new resolution graph, and so on until either a fixpoint or a timeout is reached.

(A‖Z) Running (A) and (Z) in parallel (on different machines) until the first one stops or a timeout is reached. The smallest core produced by the two programs so far is the resulting core of (A‖Z). This approach can be useful if (A) and (Z) are sufficiently different, and neither one dominates the other.

The following table summarizes our results with time-out of 3500 sec. *Core reduction* measures the number of clauses removed from the initial core, hence a larger number is better. An intriguing result is the superiority of (A) over (A‖Z) when it comes to clause reduction. This is because the number of clauses counted for (A‖Z) is due to the system that finishes first, which may remove fewer clauses than the other system.

The comparison between (Z) and (A) reveals that TRIM_TILL_FIX removes twice as many clauses on average as RUN_TILL_FIX but RUN_TILL_FIX is 50% faster. Note, however, the medians: the median of TRIM_TILL_FIX is 5 times larger on core reduction and 14 times larger on velocity, which is important in the realm of short timeouts. In other words, if we ran these benchmarks with a shorter timeout, the results would favor TRIM_TILL_FIX much stronger. This is also evident from Figure **6.6**: although

System	Velocity		Core Reduction	
	Median	Average	Median	Average
(Z)	1.1	200.8	729	3126.8
(A)	14.5	130.3	3404	6212.1
(A‖Z)	14.6	239.3	3310	5985.3
(T)	33.0	160.8	1464	3863.1

Table 6.1: Experimental results summary

(Z)'s velocity is typically better, it suffers from a large number of timeouts, which is counted as 0 velocity in our calculations.

Figure **6.1** presents the total number of clauses removed as a function of time, if all the benchmarks were run in parallel. We can clearly see that the total number of clauses removed from cores grows more rapidly with TRIM_TILL_FIX and continues to grow even after TRIM_TILL_FIX stabilizes.

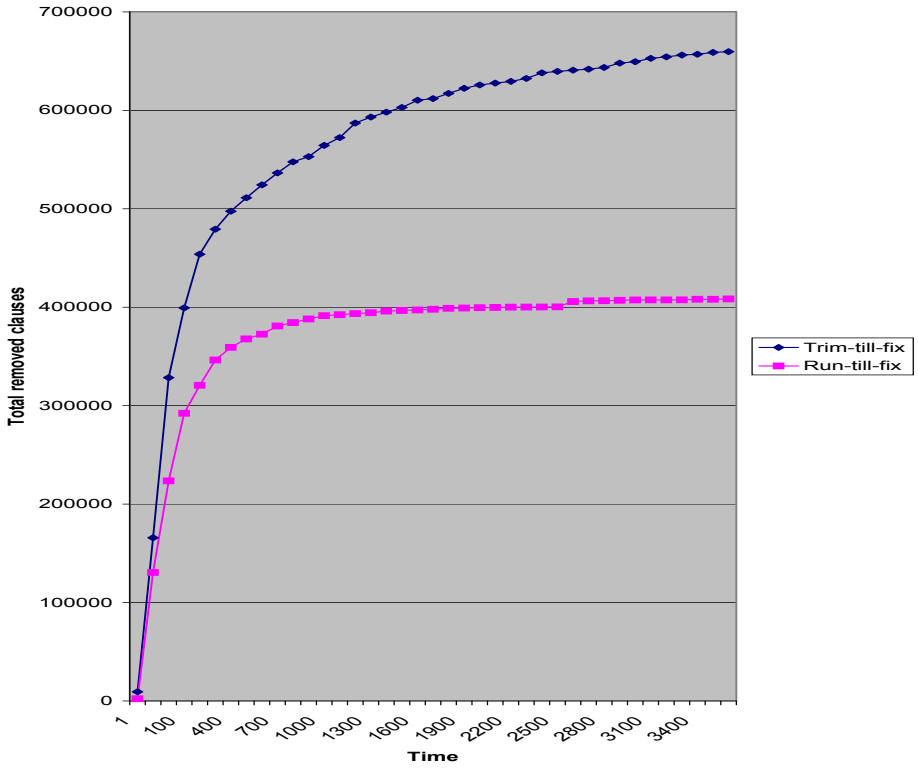

Figure 6.1: Total clauses removed

Following are some examples of the experimental results, where x-axis is time and y-axis is the number of clauses in the unsatisfiable core produced by the respective program, TRIM_TILL_FIX (A) or RUN_TILL_FIX (Z):
In diagrams of figure **6.2** we can see that the graduate core reduction by TRIM_TILL_FIX is a disadvantage at first, but becomes its advantage afterwards, when it is able to continue even after RUN_TILL_FIX quits.
Figure **6.3** shows an example run of TRIM_TILL_FIX versus RUN_TILL_FIX, where the first core produced by zVERIFY is very hard for the solver, so that the next reduction

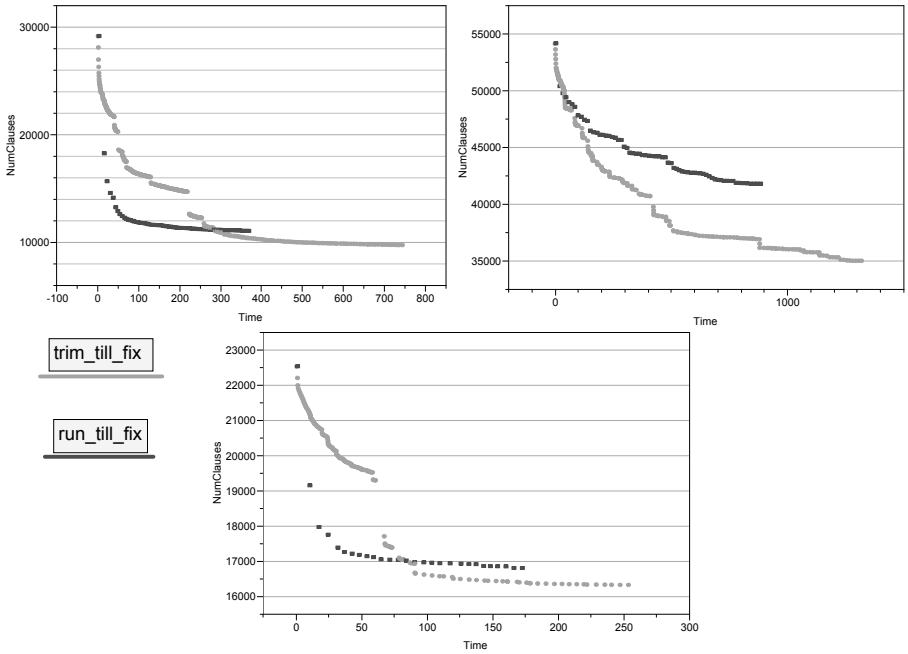

Figure 6.2: Example runs 1

occurs after almost 3000 seconds (the two points with RUN_TILL_FIX's results are encircled). However, our TRIMMER starts reducing the initial core almost immediately. By the first half-hour TRIMMER has managed to reduce about 5000 clauses, whereas RUN_TILL_FIX still has no new core.

Figure **6.4** shows two runs where there is no advantage of TRIM_TILL_FIX over RUN_TILL_FIX. The right diagram is an example of a run where the first step of RUN_TILL_FIX is so lucky, that TRIM_TILL_FIX can't catch up with it. The left diagram shows an example run where TRIMMER cannot do much and, except for a couple of initial iterations, TRIM_TILL_FIX is pretty much similar to RUN_TILL_FIX. It is evident from the large time gaps between the points, that the TRIMMER does not find appropriate dominators, but just extracts resolution graph leaves and continues with them, just as RUN_TILL_FIX does.

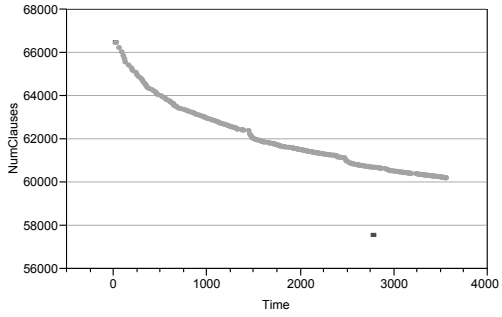

Figure 6.3: Example runs 2

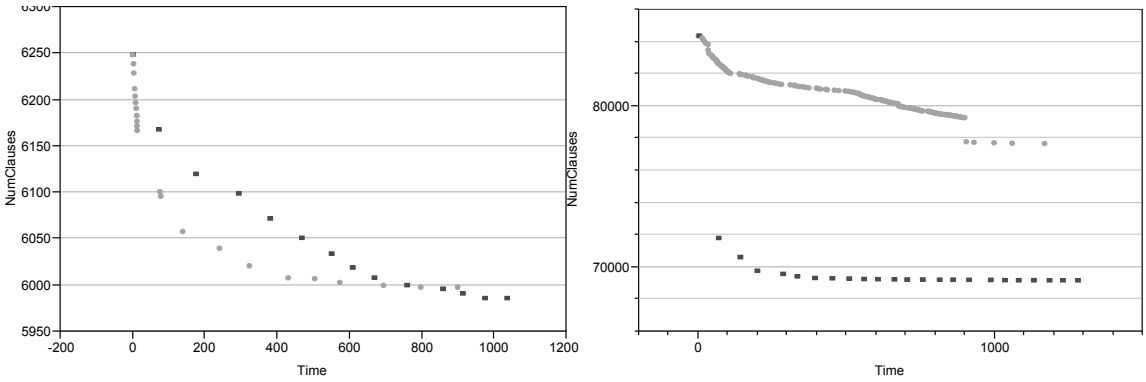

Figure 6.4: Example runs 3

6.2 Statistical Analysis of the Experimental Results

6.2.1 Ordinary Sign Test

The *ordinary sign test* is a nonparametric method for hypothesis testing, which does not rely on assumptions about the population from which the samples are drawn. We tested hypotheses about median differences in core reduction and in velocity. For each benchmark, we measured the differences in core reduction and velocity between every pair of systems (s_1, s_2). These differences are then classified into one of two categories - "success" $(+)$ means that (s_1) has higher velocity, or achieved larger core reduction, while "failure" $(-)$ means that (s_1) has lower velocity, or achieved smaller

core reduction. Samples with zero differences are discarded. The parameters are the probability of "success", denoted by p_+, and the probability of "failure", p_-. The occurrences of "success" or "failure" are assumed to follow a Bernoulli process.

We use the following one-sided test:

The null hypothesis is	$H_0 : \ p_+ = p_-$
The alternative hypothesis is	$H_1 : \ p_+ > p_-$
$S_+ =$ the number of plus signs observed	

The null hypothesis is rejected in favor of the alternative hypothesis if the *p-value*, which is the probability of obtaining a sample result as large or larger than S_+ under the null hypothesis H_0, is less than $\alpha = 0.01$.

If the test statistic S_+ is significantly smaller than its expected value $0.5 * n$, where n is the number of samples, then we use the opposite one-sided test:

The null hypothesis is	$H_0 : \ p_+ = p_-$
The alternative hypothesis is	$H_1 : \ p_+ < p_-$
$S_- =$ the number of minus signs observed	

We ran a detailed statistical analysis on the results, with the *ordinary sign test*. The results, referring to the differences in the medians of velocity and core reduction, are summarized in Figure **6.5**. We see that there is a statistically significant difference between the competing programs both in velocity and in core reduction, with (A) and (A‖Z) being the winners. Note that this result is consistent with our previous conclusions. However, analyzing Figure **6.6** shows us that most of the difference stems from the fact that (Z) reaches a timeout before producing even one trimmed core on many more cases than the other programs. In these cases both the velocity and core reduction is 0, which is much lower than the results of the programs that create at least one trimmed core.

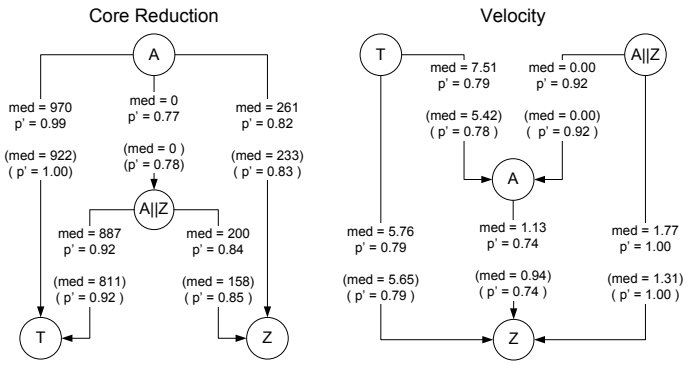

Figure 6.5: Results summary of the statistical analysis of the difference in median values of velocity and core reduction. The nodes represent the competing systems, and an edge from a to b represents 99% confidence (i.e. $\alpha = 0.01$) in a's superiority over b. *med* is the median of the difference of values between the parent and its child. p' is the estimated probability of the parent's success (which is equal to the ratio of its success). The results without parentheses correspond to a timeout of $3,500$ sec., and within parentheses to $1,800$ sec. (A) is the ultimate leader in core reduction, and (T) and A‖Z are the fastest.

6.3 Variations and Optimizations

6.3.1 Bubble Transformation vs. Simple Transformation

T_bubble_T_simple_CoreSign - one run: "success" is defined as "bubble transformation run resulted in higher core reduction".

$$H_0 : \ p_+ = p_-$$
$$H_1 : \ p_+ > p_-$$
$$S_+ = \text{the number of plus signs observed}$$

The result: p-value $< \alpha = 0.01$, therefore, H_0 is rejected in favor of H_1. We can conclude that there is a difference in core reduction in favor of bubble transformation run.

T_bubble_T_simple_VelSign - one run: "success" is defined as "bubble transformation run resulted in higher velocity".

$$H_0 : p_+ = p_-$$

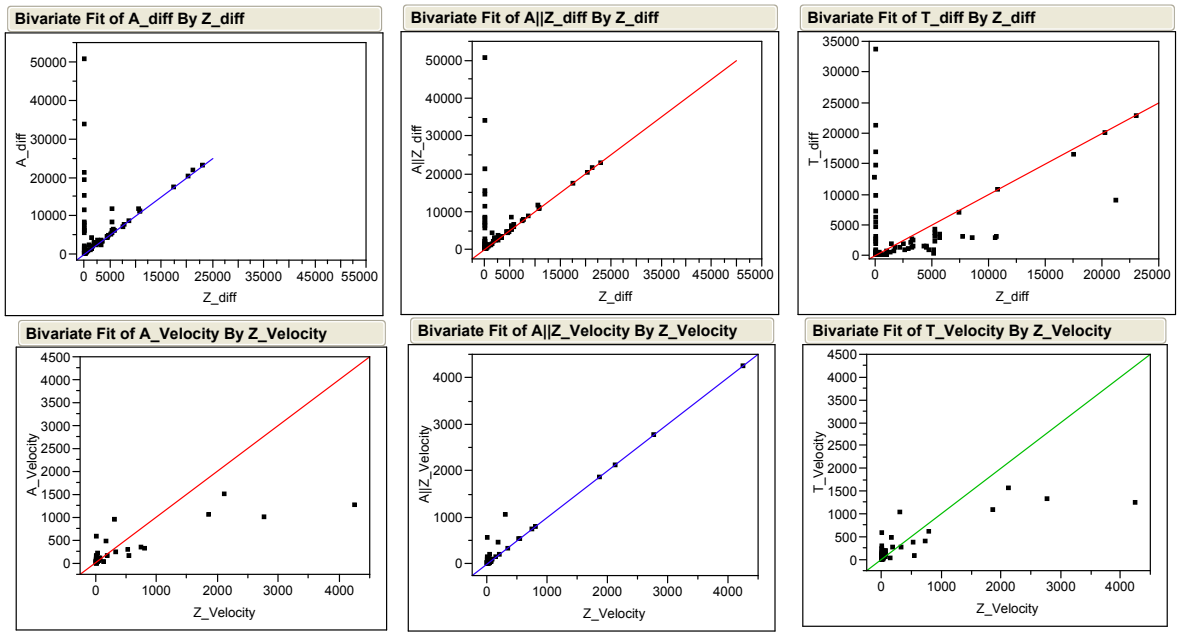

Figure 6.6: Core Reduction (top) and Velocity (bottom) of A, A||Z and T compared to Z

$$H_1 : p_+ < p_-$$
$$\text{Test statistic } S_- = \text{number of minus signs observed}$$

The result: p-value $= 0.1734 > \alpha = 0.01$, therefore, H_0 cannot be rejected in favor of H_1. We cannot conclude that there is a difference in velocities.

A_bubble_A_simple_CoreSign - trim_till_fix: "success" is defined as "bubble transformation TRIM_TILL_FIX run resulted in higher core reduction".

$$H_0 : p_+ = p_-$$
$$H_1 : p_+ > p_-$$
$$\text{Test statistic } S_+ = \text{number of plus signs observed}$$

The result: p-value $< \alpha = 0.01$, therefore, H_0 is rejected in favor of H_1. We can conclude that there is a difference in core reduction in favor of bubble transformation TRIM_TILL_FIX run.

Timeout			Level	Count	Estimated Probability	Difference Median		Level tested	Hypothesized Probability	p-value	Meaning
1800	T vs. Z	Velocity [Cls/sec]	+	56	0.79		Prob <= p	+	0.5	**0.00**	H0 is rejected in favor of H1. **T** velocity is mostly higher than that of **Z**.
			-	15	0.21						
			Total	71	1.00	5.65					
		Core [Cls]	+	34	0.48		Prob <= p	+	0.5	**0.68**	H0 cannot be rejected.
			-	37	0.52		Prob <= p	-	0.5	**0.41**	There is not enough evidence for either core reduction being larger.
			Total	71	1.00	0					
	A vs. Z	Velocity [Cls/sec]	+	53	0.74		Prob <= p	+	0.5	**0.00**	H0 is rejected in favor of H1. **A** velocity is mostly higher than that of **Z**.
			-	19	0.26						
			Total	72	1.00	0.94					
		Core [Cls]	+	59	0.83		Prob <= p	+	0.5	**0.00**	H0 is rejected in favor of H1. **A** core reduction is mostly larger than that of **Z**.
			-	12	0.17						
			Total	71	1.00	233					
	A||Z vs. Z	Velocity [Cls/sec]	+	56	1.00		Prob <= p	+	0.5	**0.00**	H0 is rejected in favor of H1. **A||Z** velocity is mostly higher than that of **Z**.
			-	0	0.00						
			Total	56	1.00	1.31					
		Core [Cls]	+	47	0.85		Prob <= p	+	0.5	**0.00**	H0 is rejected in favor of H1. **A||Z** core reduction is mostly larger than that of **Z**.
			-	8	0.15						
			Total	55	1.00	158					
	A vs. T	Velocity [Cls/sec]	+	14	0.22						H0 is rejected in favor of H1(-).
			-	50	0.78		Prob <= p	-	0.5	**0.00**	**A** velocity is mostly lower than that of **T**.
			Total	64	1.00	-5.42					
		Core [Cls]	+	64	1.00		Prob <= p	+	0.5	**0.00**	H0 is rejected in favor of H1. **A** core reduction is mostly larger than that of **T**.
			-	0	0.00						
			Total	64	1.00	922					
	A||Z vs. T	Velocity [Cls/sec]	+	25	0.38						H0 cannot be rejected.
			-	40	0.62		Prob <= p	-	0.5	**0.04**	There is not enough evidence for either velocity being higher.
			Total	65	1.00	-0.40					
		Core [Cls]	+	59	0.92		Prob <= p	+	0.5	**0.00**	H0 is rejected in favor of H1. **A||Z** core reduction is mostly larger than that of **T**.
			-	5	0.08						
			Total	64	1.00	811					
	A||Z vs. A	Velocity [Cls/sec]	+	33	0.92		Prob <= p	+	0.5	**0.00**	H0 is rejected in favor of H1. **A||Z** velocity is mostly higher than that of **A**.
			-	3	0.08						
			Total	36	1.00	0.00					
		Core [Cls]	+	6	0.22						H0 is rejected in favor of H1(-).
			-	21	0.78		Prob <= p	-	0.5	**0.00**	**A||Z** core reduction is mostly smaller than that of **A**.
			Total	27	1.00	0					

Figure 6.7: Statistical Analysis of the Experimental Results with Timeout 1800 sec.

A_bubble_A_simple_VelSign - trim_till_fix: "success" is defined as "bubble transformation TRIM_TILL_FIX run resulted in higher velocity".

$$H_0 : p_+ = p_-$$
$$H_1 : p_+ > p_-$$
$$\text{Test statistic } S_+ = \text{number of plus signs observed}$$

The result: p-value $< \alpha = 0.01$, therefore, H_0 is rejected in favor of H_1. We can conclude that there is a difference in velocity in favor of bubble transformation TRIM_TILL_FIX run.

Conclusions: Experimental results confirm our expectations - core reduction is much larger using Bubble Transformation, and whereas there is no definite difference in velocity with one run, TRIM_TILL_FIX runs yield better velocity too.

Timeout			Level	Count	Estimated Probability	Difference Median		Level tested	Hypothesized Probability	p-value	Meaning
3500	T vs. Z	Velocity [Cls/sec]	+	57	0.79		Prob <= p	+	0.5	0.00	H0 is rejected in favor of H1. **T** velocity is mostly higher than that of **Z**.
			-	15	0.21						
			Total	72	1.00	5.76					
		Core [Cls]	+	31	0.43		Prob <= p	-	0.5	0.14	H0 cannot be rejected. There is not enough evidence for either core reduction being larger.
			-	41	0.57						
			Total	72	1.00	-53					
	A vs. Z	Velocity [Cls/sec]	+	53	0.74		Prob <= p	+	0.5	0.00	H0 is rejected in favor of H1. **A** velocity is mostly higher than that of **Z**.
			-	19	0.26						
			Total	72	1.00	1.13					
		Core [Cls]	+	58	0.82		Prob <= p	+	0.5	0.00	H0 is rejected in favor of H1. **A** core reduction is mostly larger than that of **Z**.
			-	13	0.18						
			Total	71	1.00	261					
	A||Z vs. Z	Velocity [Cls/sec]	+	56	1.00		Prob <= p	+	0.5	0.00	H0 is rejected in favor of H1. **A||Z** velocity is mostly higher than that of **Z**.
				0	0.00						
			Total	56	1.00	1.77					
		Core [Cls]	+	46	0.84		Prob <= p	+	0.5	0.00	H0 is rejected in favor of H1. **A||Z** core reduction is mostly larger than that of **Z**.
			-	9	0.16						
			Total	55	1.00	200					
	A vs. T	Velocity [Cls/sec]	+	15	0.21		Prob <= p	-	0.5	0.00	H0 is rejected in favor of H1(-). **A** velocity is mostly lower than that of **T**.
			-	58	0.79						
			Total	73	1.00	-7.51					
		Core [Cls]	+	67	0.99		Prob <= p	+	0.5	0.00	H0 is rejected in favor of H1. **A** core reduction is mostly larger than that of **T**.
			-	1	0.01						
			Total	68	1.00	970					
	A||Z vs. T	Velocity [Cls/sec]	+	26	0.36		Prob <= p	-	0.5	0.01	H0 cannot be rejected. There is not enough evidence for either velocity being higher.
			-	46	0.64						
			Total	72	1.00	-1.64					
		Core [Cls]	+	60	0.92		Prob <= p	+	0.5	0.00	H0 is rejected in favor of H1. **A||Z** core reduction is mostly larger than that of **T**.
			-	5	0.08						
			Total	65	1.00	887					
	A||Z vs. A	Velocity [Cls/sec]	+	33	0.92		Prob <= p	+	0.5	0.00	H0 is rejected in favor of H1. **A||Z** velocity is mostly higher than that of **A**.
			-	3	0.08						
			Total	36	1.00	0.00					
		Core [Cls]	+	6	0.23		Prob <= p	-	0.5	0.00	H0 is rejected in favor of H1. **A||Z** core reduction is mostly smaller than that of **A**.
			-	20	0.77						
			Total	26	1.00	0					

Figure 6.8: Statistical Analysis of the Experimental Results with Timeout 3500 sec.

6.3.2 Dominator Ordering

High_low_core: "success" is defined as "dominator ordering from the highest to the lowest number of minions resulted in higher core reduction than dominator ordering from the lowest to the highest number of minions".

$$H_0 : p_+ = p_-$$
$$H_1 : p_+ > p_-$$
$$\text{Test statistic } S_+ = \text{number of plus signs observed}$$

The result: p-value $= 0.0003 < \alpha = 0.01$, therefore, H_0 is rejected in favor of H_1. We can conclude that there is a difference in core reduction in favor of dominator ordering from the highest to the lowest number of minions.

High_low_vel: "success" is defined as "dominator ordering from the highest to the lowest number of minions resulted in higher velocity than dominator ordering from the lowest to the highest number of minions".

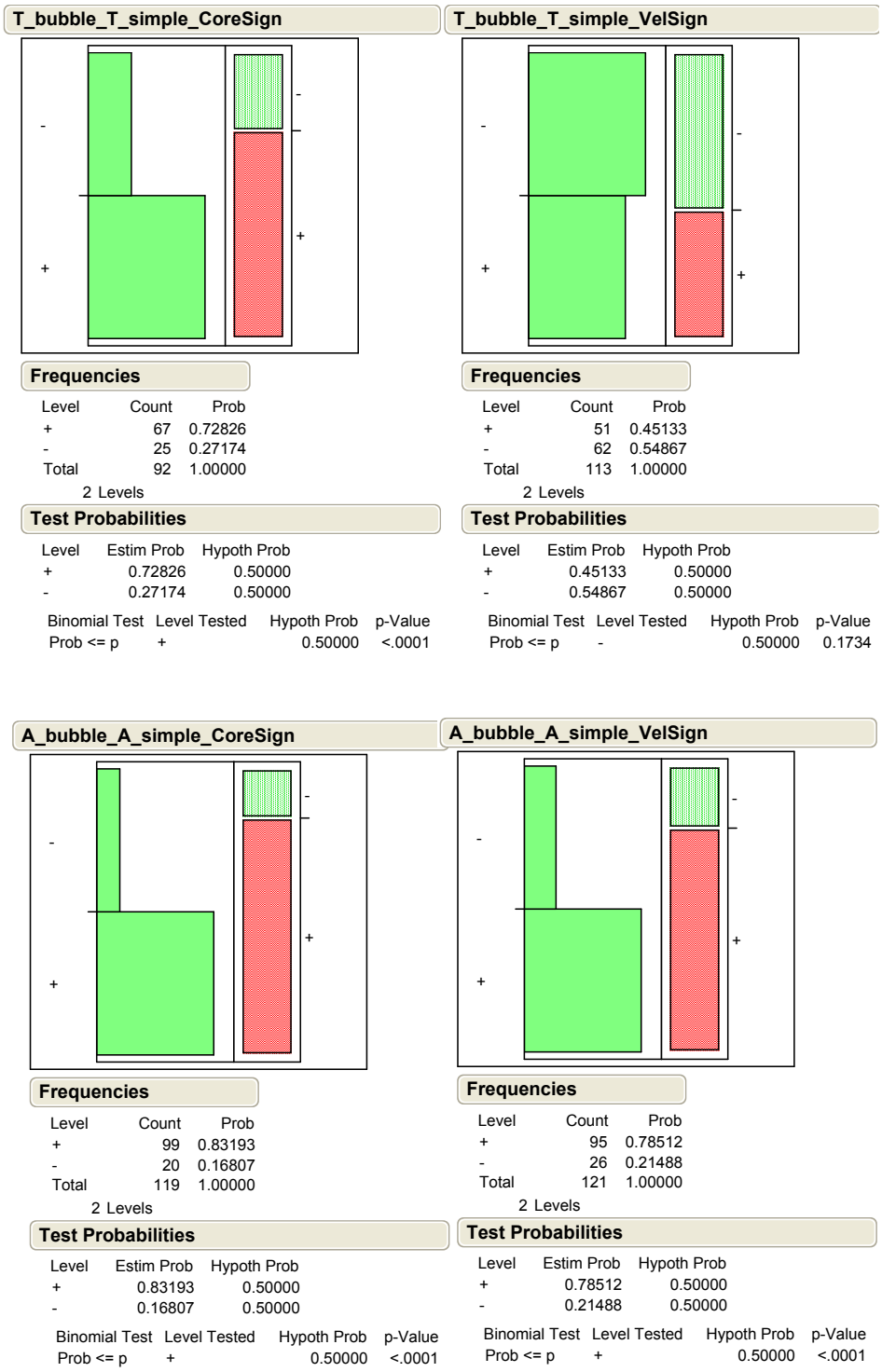

Figure 6.9: Bubble Transformation versus Simple Transformation

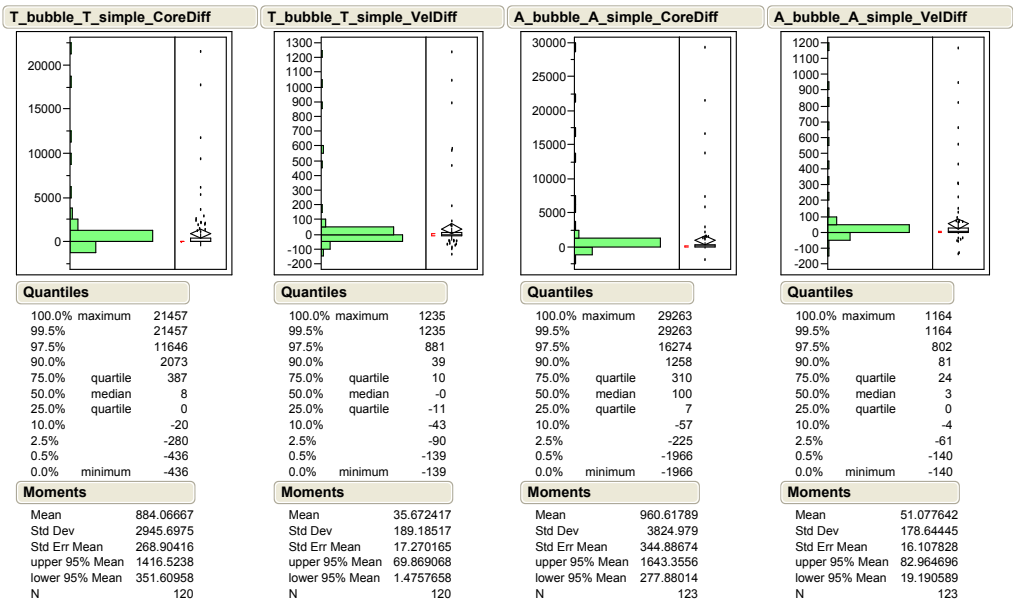

Figure 6.10: Bubble Transformation versus Simple Transformation difference distribution

$$H_0 : p_+ = p_-$$
$$H_1 : p_+ > p_-$$
$$\text{Test statistic } S_+ = \text{number of plus signs observed}$$

The result: p-value $= 0.0023 < \alpha = 0.01$, therefore, H_0 is rejected in favor of H_1. We can conclude that there is a difference in velocity in favor of dominator ordering from the highest to the lowest number of minions.

High_rand_core: "success" is defined as "dominator ordering from the highest to the lowest number of minions resulted in higher core reduction than random dominator ordering".

$$H_0 : p_+ = p_-$$
$$H_1 : p_+ > p_-$$
$$\text{Test statistic } S_+ = \text{number of plus signs observed}$$

The result: p-value $= 0.0019 < \alpha = 0.01$, therefore, H_0 is rejected in favor of H_1. We can conclude that there is a difference in core reduction in favor of dominator

ordering from the highest to the lowest number of minions.

High_rand_vel: "success" is defined as "dominator ordering from the highest to the lowest number of minions resulted in higher velocity than random dominator ordering".

$$H_0 : p_+ = p_-$$
$$H_1 : p_+ > p_-$$
$$\text{Test statistic } S_+ = \text{number of plus signs observed}$$

The result: p-value $= 0.1481 > \alpha = 0.01$, therefore, H_0 cannot be rejected in favor of H_1. We cannot conclude that there is a difference in velocities.

Conclusions: The ordinary sign test analysis of the experimental results shows that the greedy strategy - first trying to remove the largest number of minions at once - is definitely better than the opposite strategy, both in core reduction and in velocity. The greedy strategy is also better in core reduction than randomly selecting dominators for trials. However, there is no statistically significant difference in velocity between these two strategies. Therefore, we can conclude that the greedy strategy is the best of the three dominator ordering strategies.

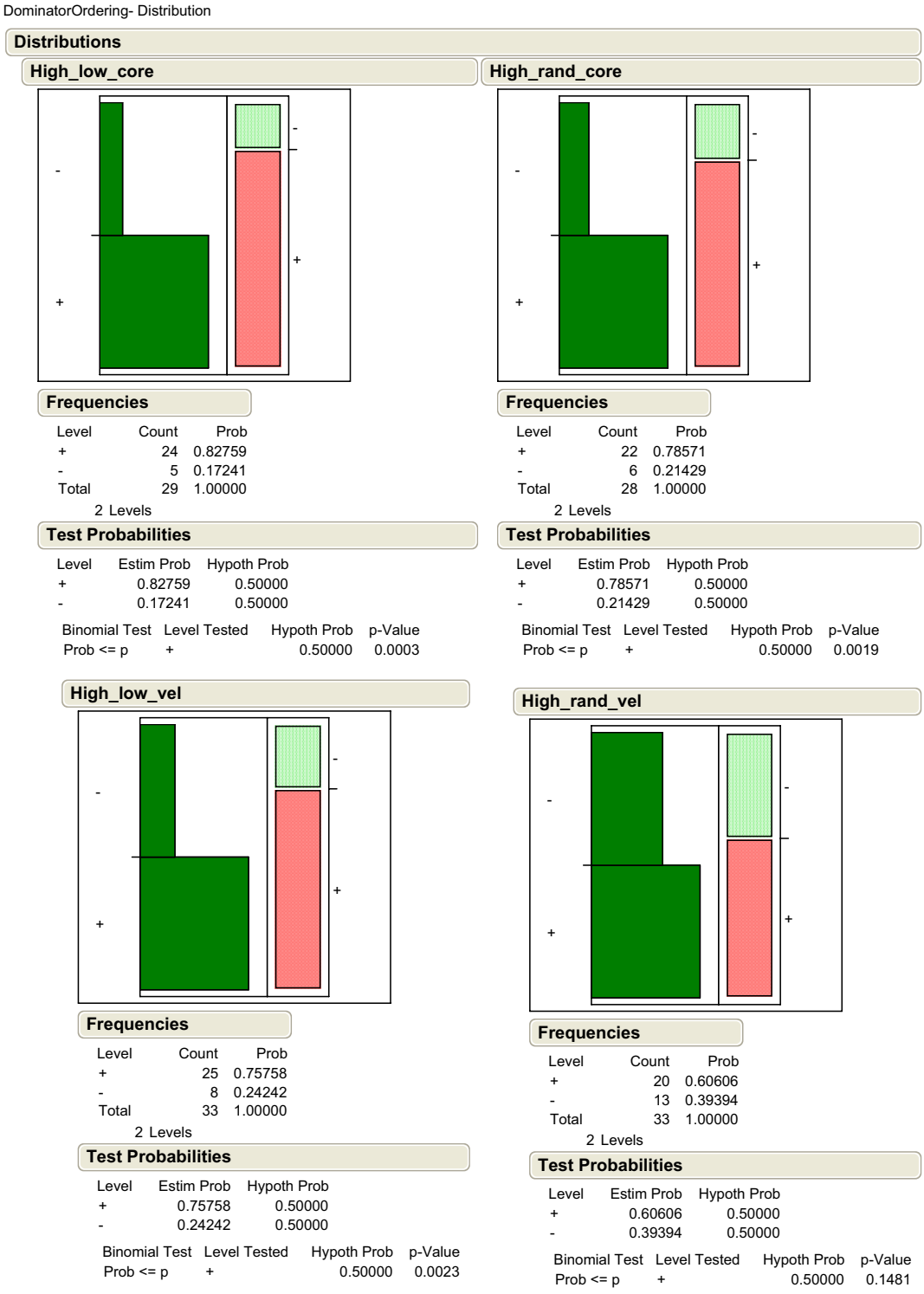

Figure 6.11: Dominator ordering

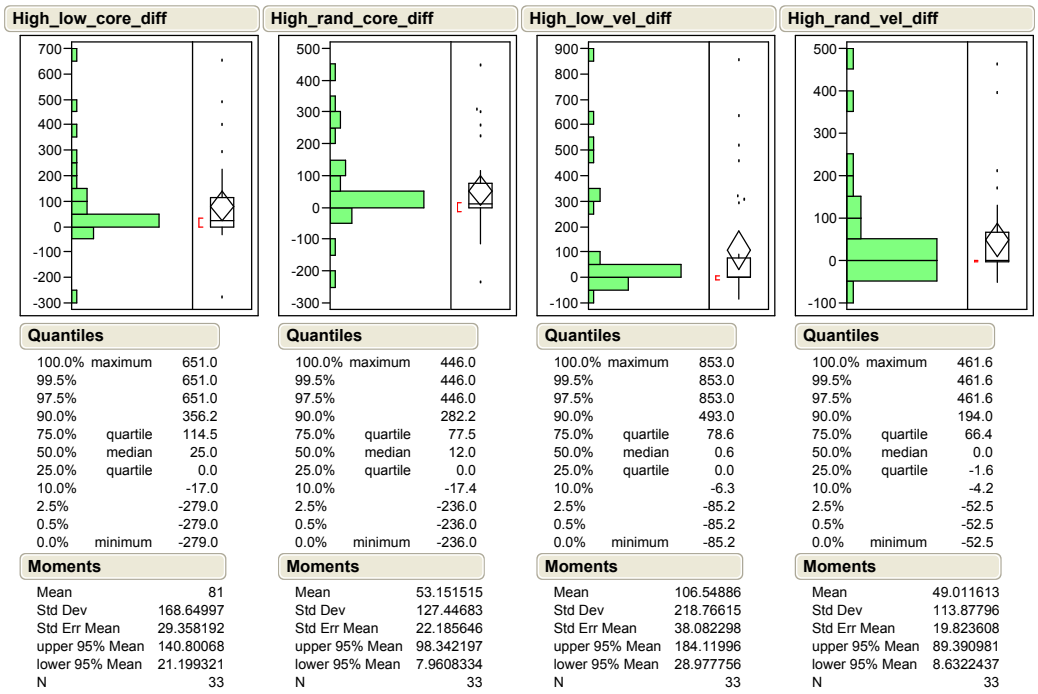

Figure 6.12: Dominator ordering - differences distribution

6.3.3 Minion Caching

We compared the minion caching technique described in Section 5 item 5.1.4 with no caching at all.

Minion caching versus no caching Figure **6.13**:

T_satM_T_noPrev_CoreSign - one run: "success" is defined as "minion caching resulted in higher core reduction".

$$H_0 : p_+ = p_-$$
$$H_1 : p_+ > p_-$$
$$\text{Test statistic } S_+ = \text{number of plus signs observed}$$

The result: p-value $= 0.1662 > \alpha = 0.01$, therefore, H_0 cannot be rejected in favor of H_1. We cannot conclude that there is a difference in core reduction.

T_satM_T_noPrev_VelSign - one run: Velocity with minion caching was higher

than that without caching on all instances.

A_satM_A_noPrev_CoreSign - trim_till_fix: "success" is defined as "minion caching TRIM_TILL_FIX run resulted in higher core reduction".

$$H_0 : p_+ = p_-$$
$$H_1 : p_+ > p_-$$
$$\text{Test statistic } S_+ = \text{number of plus signs observed}$$

The result: p-value $< \alpha = 0.01$, therefore, H_0 is rejected in favor of H_1. We can conclude that there is a difference in core reduction in favor of minion caching TRIM_TILL_FIX run.

A_satM_A_noPrev_VelSign - trim_till_fix: "success" is defined as "minion caching TRIM_TILL_FIX run resulted in higher velocity".

$$H_0 : p_+ = p_-$$
$$H_1 : p_+ > p_-$$
$$\text{Test statistic } S_+ = \text{number of plus signs observed}$$

The result: p-value $= 0.0207 > \alpha = 0.01$, therefore, H_0 cannot be rejected. We cannot conclude that there is a difference in velocity.

Conclusions: We can infer from the statistical analysis that the minion caching strategy improves velocity and, at least, doesn't hurt core reduction.

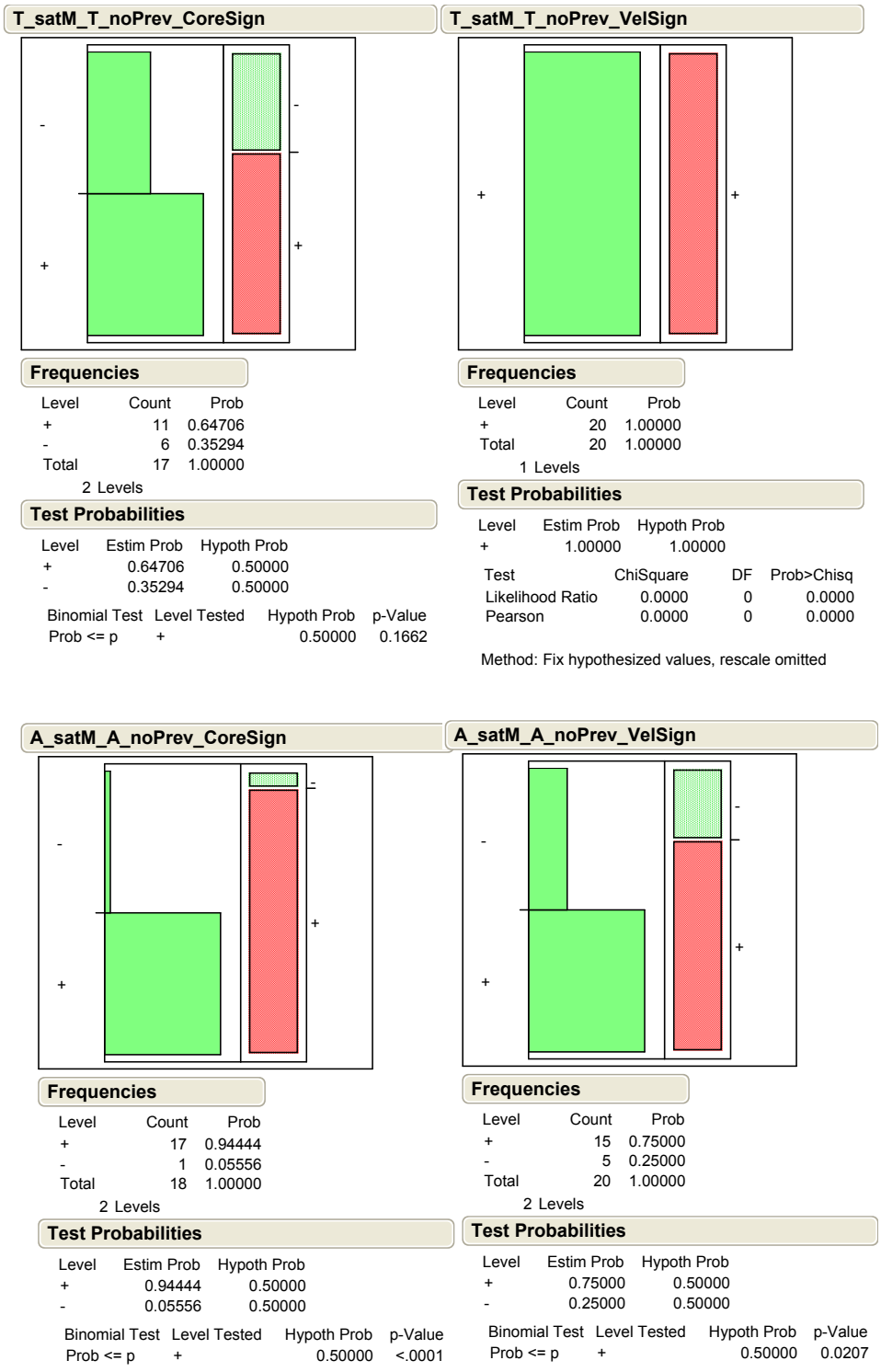

Figure 6.13: Minion caching versus no caching

6.3.4 Refinement

T_satM_T_ref_CoreSign - one run: "success" is defined as "no refinement run resulted in higher core reduction".

$$H_0 : p_+ = p_-$$
$$H_1 : p_+ < p_-$$
$$\text{Test statistic } S_- = \text{number of minus signs observed}$$

The result: p-value $< \alpha = 0.01$, therefore, H_0 is rejected in favor of H_1. As expected, we can conclude that there is a difference in core reduction in favor of refinement.

T_satM_T_ref_VelSign - one run: "success" is defined as "no refinement run resulted in higher velocity".

$$H_0 : p_+ = p_-$$
$$H_1 : p_+ > p_-$$
$$\text{Test statistic } S_+ = \text{number of plus signs observed}$$

The result: p-value $< \alpha = 0.01$, therefore, H_0 is rejected in favor of H_1. We can conclude that there is a difference in velocities in favor of no refinement.

A_satM_A_ref_CoreSign - trim_till_fix: "success" is defined as "no refinement TRIM_TILL_FIX run resulted in higher core reduction".

$$H_0 : p_+ = p_-$$
$$H_1 : p_+ > p_-$$
$$\text{Test statistic } S_+ = \text{number of plus signs observed}$$

The result: p-value $= 0.0851 > \alpha = 0.01$, therefore, H_0 cannot be rejected in favor of H_1. We cannot conclude that there is a difference in core reduction.

A_satM_A_ref_VelSign - trim_till_fix: "success" is defined as "no refinement TRIM_TILL_FIX run resulted in higher velocity".

$$H_0 : p_+ = p_-$$
$$H_1 : p_+ > p_-$$
Test statistic S_+ = number of plus signs observed

The result: p-value $< \alpha = 0.01$, therefore, H_0 is rejected in favor of H_1. We can conclude that there is a difference in velocities in favor of no refinement TRIM_TILL_FIX run.

Conclusions: The tests confirm our expectations, that refinement improves core reduction in a single run, while decreasing velocity. On the other hand, in TRIM_TILL_FIX the benefit of core reduction is lost, and the decreased velocity remains. Therefore, we can conclude that there is no much use in refinement, but for short single runs, where core size is much more important than velocity.

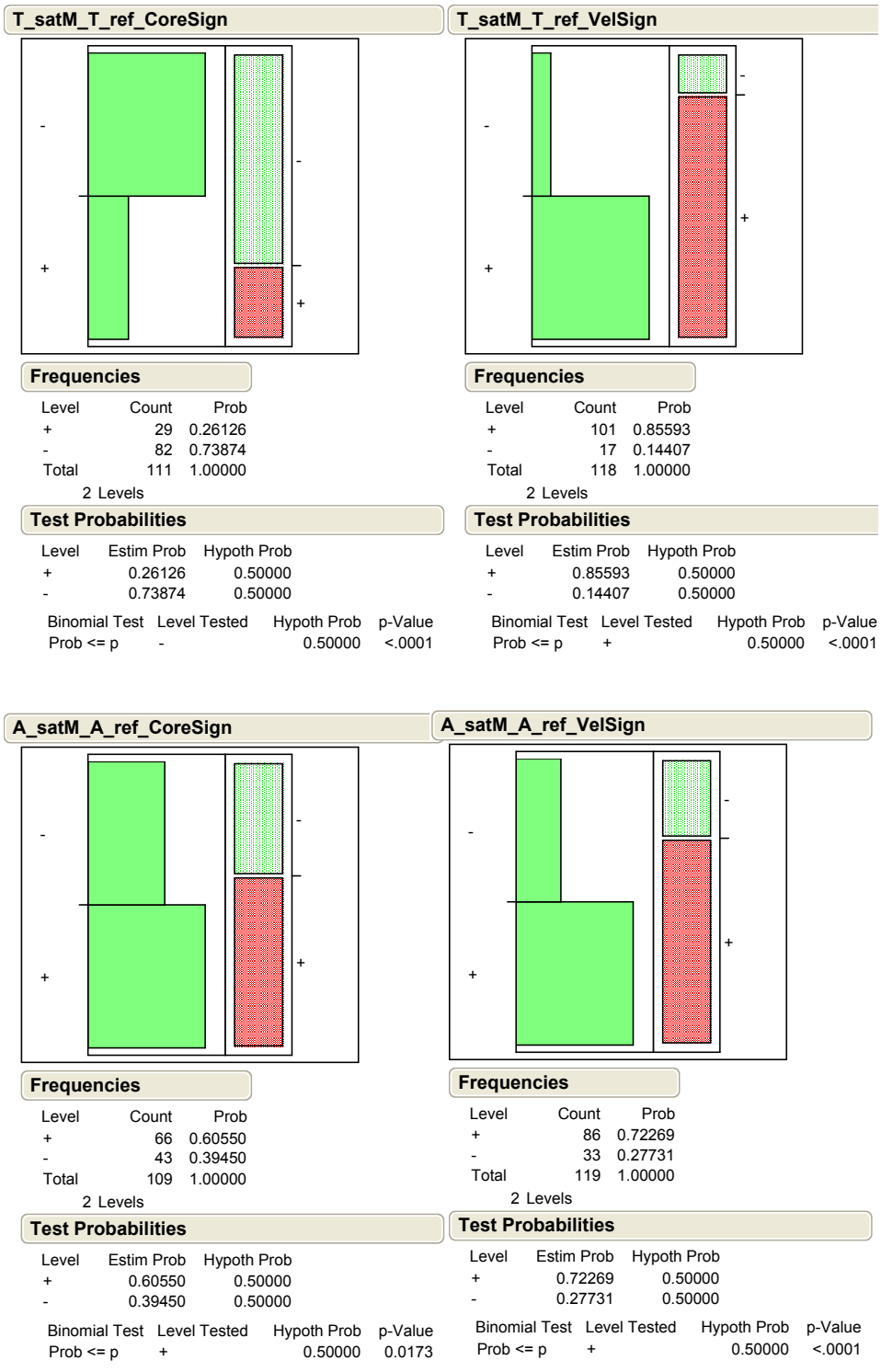

Figure 6.14: No Refinement versus Refinement Sign Test

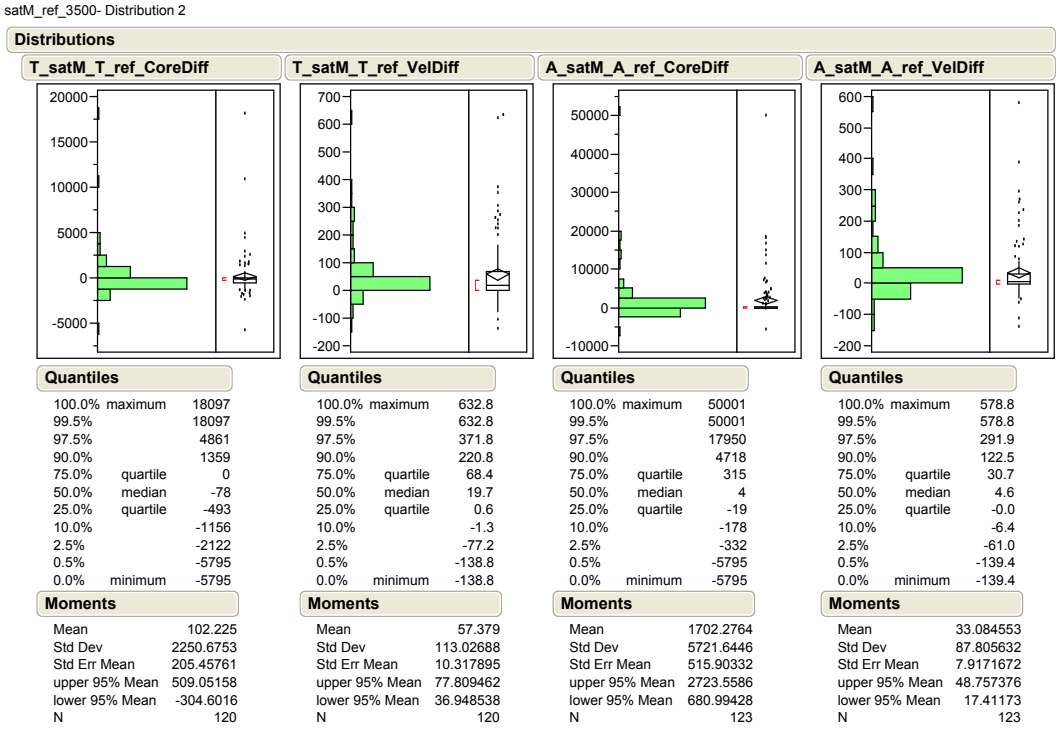

Figure 6.15: No Refinement versus Refinement Distribution

6.3.5 Lazy Evaluation

T_satM_T_lazySatM_CoreSign - one run: "success" is defined as "not lazy resulted in higher core reduction".

$$H_0 : p_+ = p_-$$
$$H_1 : p_+ > p_-$$
$$\text{Test statistic } S_+ = \text{number of plus signs observed}$$

The result: p-value $< \alpha = 0.01$, therefore, H_0 is rejected in favor of H_1. We can conclude that there is a difference in core reduction in favor of not lazy dominator evaluation.

T_satM_T_lazySatM_VelSign - one run: "success" is defined as "not lazy run resulted in higher velocity".

$$H_0 : p_+ = p_-$$

$$H_1 : p_+ > p_-$$
$$\text{Test statistic } S_+ = \text{number of plus signs observed}$$

The result: p-value $= 0.4637 > \alpha = 0.01$, therefore, H_0 cannot be rejected in favor of H_1. We cannot conclude that there is a difference in velocities.

A_satM_A_lazySatM_CoreSign - trim_till_fix: "success" is defined as "not lazy TRIM_TILL_FIX run resulted in higher core reduction".

$$H_0 : p_+ = p_-$$
$$H_1 : p_+ > p_-$$
$$\text{Test statistic } S_+ = \text{number of plus signs observed}$$

The result: p-value $< \alpha = 0.01$, therefore, H_0 is rejected in favor of H_1. We can conclude that there is a difference in core reduction in favor of not lazy dominator evaluation.

A_satM_A_lazySatM_VelSign - trim_till_fix: "success" is defined as "not lazy TRIM_TILL_FIX run resulted in higher velocity".

$$H_0 : p_+ = p_-$$
$$H_1 : p_+ > p_-$$
$$\text{Test statistic } S_+ = \text{number of plus signs observed}$$

The result: p-value $= 0.2037 > \alpha = 0.01$, therefore, H_0 cannot be rejected in favor of H_1. We cannot conclude that there is a difference in velocities.

Conclusions: The ordinary sign test analysis of the experimental results shows that lazy evaluation worsens core reduction without significantly improving velocity.

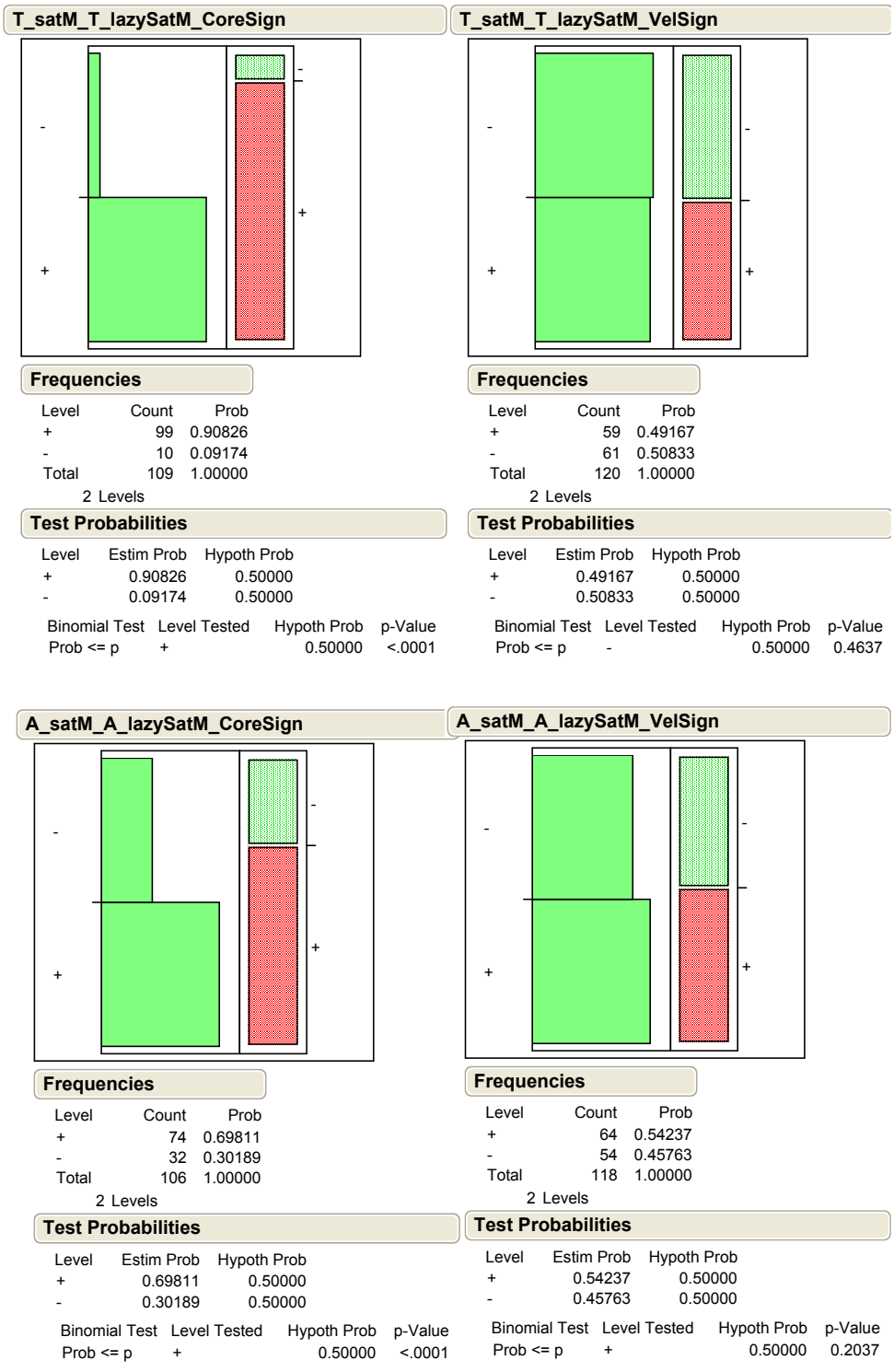

Figure 6.16: Normal versus Lazy dominator evaluation

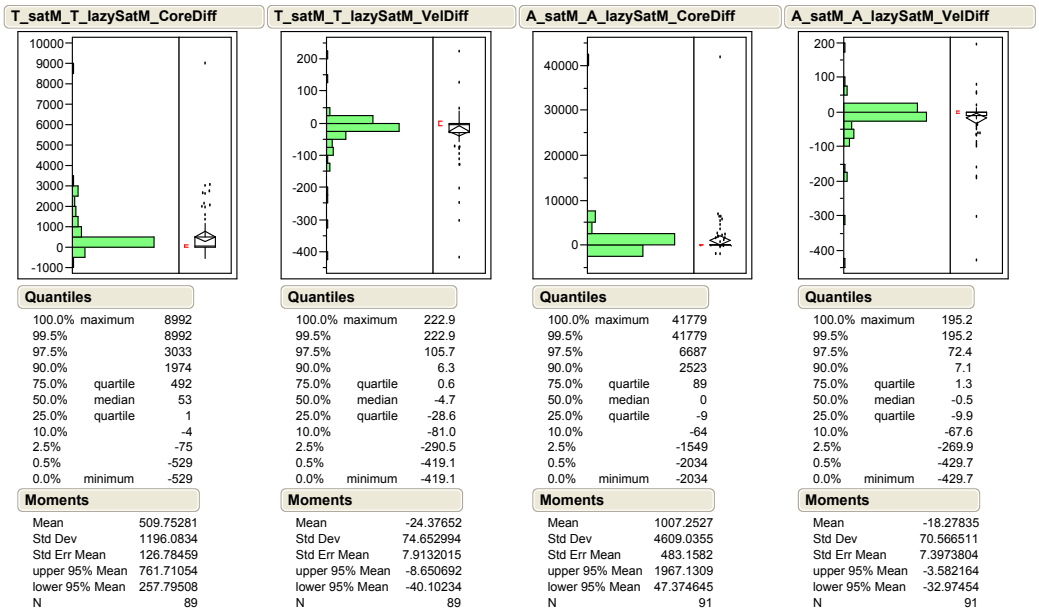

Figure 6.17: Normal versus Lazy dominator evaluation - differences distribution

Chapter 7

Conclusions

The thesis presented a family of techniques for reducing the unsatisfiable core of an unsatisfiable CNF formula, with the goal of reducing the core further and faster than currently possible. The basic idea is to replace subgraphs in the resolution proof with alternative proofs that use less original clauses. This technique has two main advantages over RUN_TILL_FIX (the only competing technique in a practical setting): it removes more clauses on average, and is faster on average in the realm of short time-outs. This result is important in the context of the typical usage of unsatisfiable core algorithms, in which the effect of reducing the core on the overall system is only vaguely known, and hence beyond a certain (short) time-out it is typically not cost-effective to continue reducing the core. Our statistical analysis of the results (based on the simple-sign-test) show the significance of various alternatives of this algorithm.

Although the problem of minimizing the core has gained significant attention in the last few years, for many applications in formal verification it is more important to minimize the number of variables, rather than clauses, in the core. A possible extension of the current thesis is thus to adapt these ideas to the problem of minimizing the number of variables in the core.

Appendix A

Implementation

The code of Trimmer is in the attached CDROM. A flowgraph with some implementation details is given in Figure **1.1**.

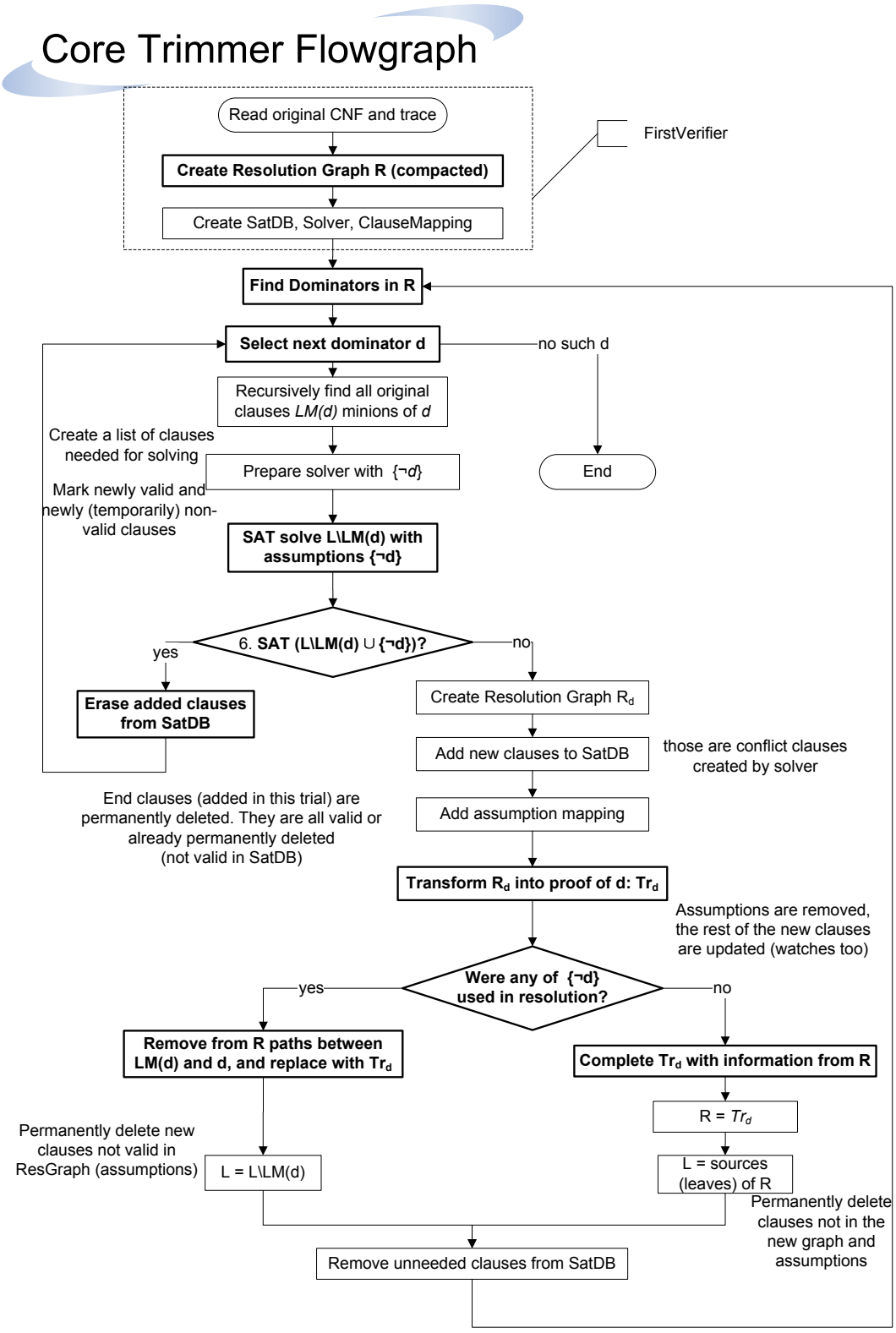

Figure 1.1: Program Flowgraph

Bibliography

[1] U. J. Aho A.V. *The Theory of Parsing, Translation, and Compiling, Vol. II: Compiling*. Prentice-Hall, Englewood Cliffs, N.J., 1972.

[2] N. Amla and K. McMillan. Automatic abstraction without counterexamples. In H. Garavel and J. Hatcliff, editors, *TACAS'03*, volume 2619 of *Lect. Notes in Comp. Sci.*, 2003.

[3] R. Bruni. Approximating minimal unsatisfiable subformulae by means of adaptive core search. *Discrete Appl. Math.*, 130(2):85–100, 2003.

[4] J. H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row, 1986.

[5] L. Georgiadis, R. F. Werneck, R. E. Tarjan, S. Triantafyllis, and D. I. August. Finding dominators in practice. In *12th Annual European Symposium on Algorithms (ESA 2004)*, volume 3221 of *LNCS*, pages 677–688, 2004.

[6] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages*, pages 122–131. ACM Press, 2005.

[7] A. Gupta. *Learning Abstractions for Model Checking*. PhD thesis, Carnegie Mellon University, 2006.

[8] J. Huang. Mup: A minimal unsatisfiability prover. In *Proc. of the 10th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 432–437, 2005.

[9] D. Kroening, J. Ouaknine, S. Seshia, and O. Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In R. Alur and D. Peled, editors, *Proc. 16th Intl. Conference on Computer Aided Verification (CAV'04)*, number 3114 in Lect. Notes in Comp. Sci., pages 308–320, Boston, MA, July 2004. Springer-Verlag.

[10] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[11] I. Lynce and J. Marques-Silva. On computing minimum unsatisfiable cores. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, pages 305–310, 2004.

[12] J. P. Marques-Silva and K. A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[13] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. Amuse: a minimally-unsatisfiable subformula extractor. In *DAC '04*, pages 518–523, 2004.

[14] C. H. Papadimitriou and D. Wolfe. The complexity of facets resolved. *J. Comput. Syst. Sci.*, 37(1):2–13, 1988.

[15] J. Paul W. Purdom and E. F. Moore. Immediate predominators in a directed graph [h]. *Commun. ACM*, 15(8):777–778, 1972.

[16] R. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Proceedings of the Eastern Joint Computer Conference*, pages 133–138, 1959.

[17] O. Shtrichman. Prunning techniques for the SAT-based bounded model checking problem. In *proc. of the 11th Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, Edinburgh, Sept. 2001.

[18] A. Stump, C. Barrett, and D. Dill. CVC: a cooperating validity checker. In *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, 2002.

[19] G. Tseitin. On the complexity of proofs in propositional logics. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning: Classical Papers in Computational Logic 1967-1970*, volume 2. Springer-Verlag, 1983. Originally published 1970.

[20] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *Theory and Applications of Satisfiability Testing*, 2003.