Scalable Algorithms Enabled by Problem Structure
Karem A. Sakallah and Igor L. Markov, Dept. of EECS, University of Michigan

C. Project Description

C.1. Results from Prior NSF Support

C.1.1. NSF Award No. 9404632

Summary of results: This project extended the framework of timing analysis that we developed under a previous NSF award (MIP-9014058) by adding sufficient functional (logical) content in the models to permit more accurate (less pessimistic) performance estimation while maintaining, or not significantly reducing, the level of coverage and efficiency of the functionless analysis. The most significant result of this work has been the development of a symbolic functional waveform model that provides a consistent mathematical basis for timing analysis and simulation centered around a differential calculus that allows us to describe the transient behavior of logic waveforms in terms of appropriate time derivatives. This model highlights the intrinsically conditional nature of signal delay and can be shown to encompass most delay models proposed to date. It also, for the first time, allows precise functional and temporal abstractions to be made in order to generate compact “system-level” timing views with guaranteed accuracy bounds. An unanticipated but highly significant result has been the development of GRASP, a SAT solver that is central to the current proposal scalable algorithm development. GRASP was originally developed as part of an efficient search-based functional timing analysis methodology. We have since found it to be applicable to, and surprisingly effective in, many other problem areas. Two students who were supported on this project have graduated: João P. Marques Silva and V. Chandramouli.

Publications:

Summary of results: This is an ongoing project concerned with the application of Boolean methods to the geometric problems that arise in the design of integrated circuits. Specifically, we have focused on routing field-programmable gate arrays (FPGAs) by casting the routing problem as a large instance of Boolean satisfiability. We have developed a number of routers that were successful in generating detailed routing solutions for a variety of FPGA test cases. These routers can additionally report the fewest number of tracks needed to route a given circuit, as well as give an estimate of routability for a given number of tracks. We are currently working on refinements to increase the scale of problems that can be tackled, as well as on formulations that combine global and detailed routing to improve routability.

Publications:


C.2. Motivation and Background

Despite the fact that many of the computational tasks employed in designing, synthesizing, and verifying human-engineered objects are exponentially complex in the worst case, such objects continue to increase in complexity and are routinely made and deployed. The focus of our proposal is on scalable algorithms for solving hard problems. In particular, we conjecture that man-made artifacts have innate structures that make them tractable for design, synthesis, and verification regardless of their absolute size. We further suggest that uncovering such structure can further propel the development of efficient algorithms that scale well with increasing complexity.

Our interest in studying problem structure arose during ongoing research in combinatorial optimization, logic and physical synthesis, as well as Boolean satisfiability. We made several “discoveries” during this work including:
Humpty-Dumpty sat on a wall,

had a great fall.

Threescore men and four,

Cannot place as well as bef.

**Construction by symmetry**: starting from three elementary building blocks, we construct a hierarchical system, largely using “trivial” symmetry and composition operations. Vice versa, if presented with a construction problem, detecting symmetries can dramatically simplify construction algorithms [55].

**Sparsity, hierarchy, and self-similarity** in the Internet structure [17]

**Hierarchical structures**: the picture illustrates hierarchical self-similarity (fractal properties), common in plant organisms, electronic circuits, etc. [31]. In these cases, objects appear similar to their parts, scaled up. In particular, the numbers of wires connecting circuit parts at different levels of design are similar.

**Self-similarity**: A simple text compression example illustrates how the Lempel-Ziv algorithm finds repetitions (a type of “self-similarity”) and exploits them to losslessly represent the given text in more compact form. However, this algorithm is not able to fully exploit less obvious repetitions, common in C code (the example on the right is matrix-vector multiplication).

**Figure 1.** Examples of structure in human-engineered artifacts
• Accounting for functional symmetries in logical specifications always leads to better circuit implementations.

• Accounting for symmetries in a conjunctive normal form (CNF) formula leads, in most cases, to a dramatic speedup in the search for a satisfying assignment to the formula.

• Structural properties of a CNF formula can be visualized by ordering the variables with a sparsity-aware algorithm.

While we do not attempt to define the notion of “structure in general”, our experience suggests that considering such sample structural phenomena as (i) symmetry, (ii) sparsity, (iii) hierarchical properties, and (iv) self-similarity of very large systems would be useful. We view a system as a collection of components and interconnections between the components. In these terms, sparsity, for example, is distinguished by relatively few interconnections per system component, on average. Below we elaborate the implications of these structural phenomena on system analysis and synthesis, setting the stage for further study.

Symmetry: Different types of symmetry are present in everyday life as well as in complex man-made systems. Symmetries may reflect geometric, temporal or behavioral properties of a system, or even its composition. These types of symmetry can be easily exploited. For example, in order to design a system consisting of many symmetric components, it often suffices to design one component. Such design-by-symmetry (see Fig. 1-a) leads to cost and productivity improvements, e.g., by enabling self-assembly in nanoelectronics [55]. Not only design, but also analysis can be enabled by symmetries. In a classical example, symmetries of the Schrödinger equation of a physical system, such as the hydrogen atom, hold the key to solutions that constructively describe actual configurations that the system may assume. Large on-line databases can sometimes be considerably improved through the use of symmetries. For example, databases of chess end-games exploit symmetries of the chess-board to implicitly represent multiple positions by one database record [33].

Hierarchy and Fractal Properties: Large systems that are difficult to analyze as a whole, often admit coarse views that capture structure and filter out superfluous details. For example, city maps skip buildings of little interest, but capture malls, stadiums, bridges and the street structure. Coarsening can be applied many times to produce a hierarchy of views, that increasingly lose detail, in order to better address the system as a whole. Interestingly, views produced with different levels of coarsening may look alike, e.g., Lunar (or even geographic) maps at different scales may be difficult to distinguish for a computer program.

While hierarchical properties of our world have been on the agenda since the dawn of science, hierarchical properties of human-created systems, such as the Internet (Fig. 1-b) and integrated circuits, present a relatively new field of study. In particular, it has been shown that large integrated circuits of practical interest have similar structure at different levels of the coarsening hierarchy (the Rent’s rule, Fig. 1-c) [18], and this structure can be characterized by only one non-local parameter (Rent’s exponent) [31]. Rent’s rule is fundamental for wirelength estimation and prediction when new chip architectures are evaluated. Roughly speaking, it says that the wirelength of a circuit equals the sum of wire lengths inside its subcircuits plus a “hierarchical” term that describes the length of wires connecting the components.

Sparsity: Spatially-embedded and otherwise distributed systems are often sparse because it is impossible or inconvenient for many components, interconnections and large functional parts of the system to occupy the same spatial regions. Additionally, very large data sets become extremely difficult to process on a computer if there are too many relevant interconnections to store in physical memory. For example, transportation systems (subways, highways, railroads, etc.) are sparse because they connect remote stations/cities/etc. On average, there are at most 2-3 major highways passing through a city, at most 1-2 subway lines with stations in a given neighborhood, etc. Communication networks, such as the Internet (Fig. 1-b), phone
networks and mail are sparse because of cost considerations. Every connection costs money and the number of connections is minimized, subject to providing interrupt-free service. Most phone and Internet users have a single service provider, and each service provider may have 1-3 connections to higher-level networks, etc.

Such diverse applications as web search engines [8][9], circuit design [31], and plasma simulations entail solving large sparse eigenvalue problems and systems of linear equations, but are very sensitive to run time. The $O(n^3)$ time required by the classic Gaussian elimination procedure is often unacceptable, especially since no solution is available until the algorithm completes. Additionally, the precision of Gaussian elimination is unnecessary. Therefore, specialized algorithms and data structures were developed for sparse linear systems. Sparse-matrix storage supports linear-time matrix-vector multiplications, and iterative linear solvers require no other computationally non-trivial operations. In this approach, one starts with an arbitrary iterate and only in linear time a better iterate is produced. Overall, a relatively small number of iterations is required, and good iterates can be reused. In many cases, reproducing a sparse-matrix representation every time the original phenomenon changes is unacceptably expensive. This suggests more direct sparsity exploitation.

**Self-similarity:** Texts in natural languages and other human-produced data bear a large degree of redundancy. The most straightforward manifestations of such redundancy are in the possibility to reconstruct complete sentences after some letters, or even words, are erased. A related manifestation of redundancy is the frequent occurrence of identical or very similar words/phrases in natural-language texts. This phenomenon found extensive use in compression, indexing, search and data-mining applications dealing with natural text [42][47][52]. We illustrate the latter in Fig. 1-d. which shows how Lempel-Ziv compression finds and exploits repeated fragments of text to produce a compact representation of input. Using string matching, the Lempel-Ziv algorithm finds those pieces of text that occurred before and substitutes fixed-size back-references (each holding a location pointer and the number of characters copied from the pointed location). While such representation is implicit and does not allow constant-time random access to the text, it may result in significant storage savings. The figure also shows a type of structure that Lempel-Ziv is not able to capture completely—instances of a commonly-occurring for-loop construct differ only by the name of the counter variable. However, Lempel-Ziv will split the construct into four pieces and use a separate back-reference for each.

We believe that a thorough understanding of structural attributes, such as those highlighted above, is key to the development of computational algorithms that can scale to the problem sizes encountered in today’s systems. To be sure, these observations are not novel. The use of structure lies at the core of efficient algorithm development and in many ways fueled recent breakthroughs in IT such as wide-spread use of parallel computers, fast networking and large databases as well as prominent applications of IT, such as mapping of the human genome, online public trading and distributed processing of radio-telescope data for SETI projects. However, our research experience in very large scale integrated circuit computer-aided design (VLSI CAD) and design of leading-edge computer hardware gives us a unique perspective on structural phenomena in large systems. We intend to leverage this knowledge, skills, and ideas in the wider Information Technology domain.

### C.3. Preliminary Research

We encountered the utility of structure in attacking large-scale problems while attempting to solve a) propositional satisfiability, and b) logic synthesis, a fundamental problem in hardware design. In this section we present a synopsis of what we learned in this process.
C.3.1. SAT and Symmetry

Many authors have recognized that generic search can be accelerated by exploiting symmetries in the problem instance [4][5][10][19][22][23][28][32][39]. The presence of symmetries naturally partitions the solution space into equivalence classes. Accounting for only a single representative from each class can drastically reduce the search space without affecting the completeness of the search algorithm. Much of the prior work on symmetries in search, however, fails to link the identification of symmetries with effective ways of using them to prune the search space. Typically, new approaches are validated on well-known problems where the authors can manually describe all symmetries, e.g., the N-queens problem. Moreover, new approaches proposed to handle symmetries in search often rely upon a particular search technique and do not easily apply to other techniques. In our preliminary research, we proposed and implemented a generic symmetry extraction engine for satisfiability instances and were able to connect it to a generic SAT solver. In particular, we were able to show that a state-of-the-art SAT solver can be improved through the explicit use of symmetries in SAT instances.

Given a Boolean function, we distinguish syntactic and semantic permutational symmetries. Semantic symmetries are permutations of variables that never (i.e., for no assignment of values) change the value of the function. Syntactic (aka structural) symmetries are those permutations that do not change a given representation of the function. It follows that any syntactic symmetry is also a semantic symmetry. As an example, consider the Boolean function \( f(a, b) = a \land a' \). Since this function is identically 0 (false), the permutation \( a \leftrightarrow b \) is a semantic, but not a syntactic, symmetry. We developed a computational method to provably identify all symmetries of a given Boolean formula in the conjunctive normal form (CNF). The output of this method is a list of all possible syntactic symmetries. At the core of the method is a reduction to Graph Automorphism for which excellent open-source implementations exist (e.g., the nauty package [40] shipped with the GAP system for symbolic computations [29]).

In order to use this information with a generic SAT solver, we pre-process the CNF formula based on its symmetries. Each of the permutation generators is represented by its cycles; e.g., \((123)(45)(78)\) is a permutation on (at least) eight elements that maps \(1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 1, 4 \rightarrow 5, 5 \rightarrow 4, 7 \rightarrow 8, 8 \rightarrow 7\) and leaves the remaining elements, e.g., 6, fixed. For each such generator, we rank cycles by the smallest element, and choose the lowest-ranked cycle of length 2 or 3; in this case, \((123)\). Assume that the index of variable \(a\) is less than that of variable \(b\), and the index of variable \(b\) is less than that of variable \(c\). For a cycle of the form \((ab)\), we augment the formula with the clause \((a' \lor b)\), and for a cycle of the form \((abc)\), we add clauses \((a' \lor b)\) and \((b' \lor c)\). Adding these clauses—one for each generator—completes the pre-processing step. A key theorem asserts that the satisfiability of the augmented CNF formula is equivalent to the satisfiability of the original formula.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Result</th>
<th>Search time</th>
<th>Result</th>
<th>Search time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>hole6.cnf</td>
<td>Unsat</td>
<td>0.75</td>
<td></td>
<td>7.72</td>
<td>7.76</td>
</tr>
<tr>
<td>hole7.cnf</td>
<td>Unsat</td>
<td>9.68</td>
<td></td>
<td>8.06</td>
<td>8.19</td>
</tr>
<tr>
<td>hole8.cnf</td>
<td>Unsat</td>
<td>133.58</td>
<td></td>
<td>8.7</td>
<td>9.57</td>
</tr>
<tr>
<td>hole9.cnf</td>
<td>Unsat</td>
<td>1655</td>
<td></td>
<td>10.21</td>
<td>20.99</td>
</tr>
<tr>
<td>hole10.cnf</td>
<td>Abort</td>
<td>10000</td>
<td></td>
<td>13.66</td>
<td>121.17</td>
</tr>
</tbody>
</table>

Table 1: SAT and symmetries. Given are run times of the SAT solver GRASP [51] on the original SAT instances, time to generate syntactic symmetries, GRASP run time on pre-processed instances that explicitly account for symmetries, and total run time taken by the symmetry-aware approach.
The newly added *symmetry-breaking* [23] clauses act by allowing only one of many equivalent variable assignments to be a potential solution to the formula. If the original formula is satisfiable, the number of solutions may considerably decrease after pre-processing, clearly indicating that the search space was reduced. However, even if the original instance was not satisfiable, “the number of equivalent roads leading nowhere” would be reduced, and a generic SAT solver (complete or not) is likely to conclude much faster that no solutions exists. As an illustration, Table 1 presents run time results for some of the difficult and unsatisfiable members of the benchmark family “hole” [26] (that relates to the Pigeonhole principle). Our technique clearly shows a considerable speed-up when used with the leading-edge SAT solver GRASP [51].

C.3.2. Linking Functional and Topological Structure in Multi-Level Logic Synthesis

The automatic synthesis of logic circuits from Boolean function specifications is a central concern to the electronics industry (computers, communications, consumer electronics, etc.) and has been actively studied since the nineteen fifties. This problem presents a major computational challenge that is further compounded by the scale of circuits (hundreds of thousands to millions of gates) that need to be synthesized. To deal with such complexity, the dominant logic synthesis paradigm in use today divides logic synthesis into two consecutive stages: 1) technology-independent logic restructuring; and 2) technology mapping, or cell-library binding [50]. The primary decomposition technique in the first stage is *algebraic division* which relies on the distributive law of Boolean algebra to derive appropriate factored forms from the logic specifications. Other laws of Boolean algebra, such as idempotence \( (x \cdot x = x) \) and annihilation \( (x \cdot x' = 0) \), are deliberately excluded in order to yield unique “quotients” that can be computed efficiently. The technology mapping stage is typically based on a dynamic programming tree-mapping algorithm that structurally matches subtrees in the factored logic network with pattern trees for cells from a specified gate library. For this approach to work, the logic network is transformed into a leaf-DAG [24][35] by pushing all fanout back to the primary inputs (duplicating logic nodes where necessary), and both logic network nodes and library cells are represented by trees of NAND2 gates. The resulting matching algorithm is quite efficient and can be tailored to reduce area and/or delay based on more accurate technology data than was available in the first stage. Despite its efficiency, this synthesis paradigm has been known to perform poorly on “regular” logic such as arithmetic and other datapath functions, and its primary use has been in generating implementations of control logic (also known as “random” logic) whose specifications are assumed to lack any regularity.

As an alternative to this paradigm, we undertook an exploration of other approaches to logic synthesis that recognize structure in the functional specifications and attempt to reflect such structure in the synthesized implementation. The premise guiding our exploration was that (human-created) logic specifications are not random, even for so-called random control logic. We validated this conjecture by developing a pro-
totype automatic synthesis tool, called M31 [36], that constructively builds an implementation by simultaneously decomposing, mapping, and re-expressing the logic specifications. Starting at the primary inputs and working towards the primary outputs, M31 repeatedly selects a suitable decomposition of the yet-to-be-implemented logic in terms of carefully-chosen library primitives. This step is illustrated in Fig. 2. The $n$-input un-implemented logic function $f(x): \{0, 1\}^n \rightarrow \{0, 1\}$ is re-expressed as $h(g(x_g), x_h)$ where $x_g$ is a set of $s$ decomposition variables, $x_h$ represents the rest of the variables, $g: \{0, 1\}^s \rightarrow \{0, 1\}$ is a $t$-input primitive decomposition functions, and $h: \{0, 1\}^t \times \{0, 1\}^{n-s} \rightarrow \{0, 1\}$ is an $n-(s-t)$-variable composition function. This decomposition template is characterized by two parameters: $s$, the number of decomposition variables and $t$ the number of decomposition functions. Parameter $s$ is chosen such that the $g$ functions are small enough to be primitive gates in a given technology library; in current CMOS technology, $s = 4$ is typical. Parameter $t$ controls the width of the implementation and, if possible, is chosen to be smaller than $s$ yielding a composition function $h$ that depends on fewer variables than the original function $f$. The existence of such a support-reducing decomposition requires that the number of distinct cofactors of the function $f$ with respect to the decomposition variables $x_g$ be less than $2^{s-1}$, a condition that is satisfied by the class of functions that are symmetric in $s$ or more variables. Thus, a functional property of the specification, namely symmetry, is directly linked to a topological property of the implementation, namely network width.

The quality of synthesized implementations using this constructive approach naturally depends on the amount of structure in the logic specifications. We performed an extensive analysis of a large set of public benchmarks to determine how much symmetry exists. A summary of this study, confirming the presence of a great deal of structure in these circuits, is shown in Table 2. In addition, synthesized implementations of many of these benchmarks had consistently superior quality than implementations using the traditional structure-oblivious synthesis paradigm yielding, on average, 30% less area, 62% less depth (i.e. delay), and 23% fewer and 53% shorter wires. An unanticipated by-product of this structure-aware synthesis method was the discovery, by the tool, of a new architecture for carry-look-ahead (CLA) adders in which the carry acceleration circuit is exclusively built out of 3-input majority gates. Unlike traditional generate/propagate

<table>
<thead>
<tr>
<th>Benchmark suite</th>
<th>Total # of output functions</th>
<th># of output functions with no symmetries</th>
<th># of output functions with high-order symmetries</th>
<th>Max order of symmetry</th>
<th>Max symmetry group size</th>
<th>High-to-first order permutation ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-level MCNC</td>
<td>1560</td>
<td>76</td>
<td>41</td>
<td>4</td>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>Multi-level ISCAS85</td>
<td>703</td>
<td>85</td>
<td>16</td>
<td>3</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>Two-level MCNC</td>
<td>549</td>
<td>68</td>
<td>10</td>
<td>4</td>
<td>64</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2: Summary of symmetry characterization of benchmark circuits [37]. This data show the symmetry of each primary output function in terms of the primary input variables in its transitive fan-in. Symmetry was defined as invariance under the simultaneous swap of equal-sized groups of (possibly complemented) variables. The order of symmetry refers to the group size: first-order symmetry involves swaps of single variables; invariance under swaps involving groups of two or more variables is referred to as high-order symmetry. These symmetries were efficiently identified by hierarchical partitioning of the variables and by checking the equivalence of corresponding functional cofactors using binary decision diagrams (BDDs) [11]. The data clearly show the existence of a large amount of symmetry in these circuits.
CLA adders, this new adder architecture is rich with symmetries and has numerous variants that can be optimized to achieve any desired trade-offs between area, delay, and fan-out. Fig. 3 shows the schematic of a 4-bit “M&M” adder that was synthesized by M31.

**C.4. Proposed Research**

Our research seeks a new fundamental understanding of what makes problems and their instances easy or hard for particular solution techniques, and how to use such knowledge to custom-tailor solution methods that achieve scalability. We will further extend our preliminary work in the area, leverage the research infrastructure that we have built so far, and consolidate current knowledge. Our long-term goal is to significantly extend the performance envelope of practical algorithms in order to handle very large hard problem instances through intelligent utilization of problem structure. We seek generic and fundamental results with applicability beyond currently-popular worst-case bounds that appear irrelevant to empirically-observed performance. We therefore plan to focus on the satisfiability problem because a number of NP-complete problems have been successfully approached in recent years via reductions to SAT, e.g., exact wire routing [46], planning [34], microprocessor verification [53], etc. A closely related problem is max SAT which enables us to diagnose large infeasible constraint satisfaction problems, and explain reasons for unsatisfiability—a feature lacking in all leading-edge SAT solvers. Finally, we propose to extend our work on structure-aware logic synthesis to tackle a richer variety of circuit styles. The following sections describe the approaches we plan to take in studying each of these topics.

**C.4.1. Scalable Approaches to Satisfiability**

Our primary goal here is to greatly expand the scale of satisfiability problems that can be attacked in practice. We are not tying ourselves to specific numerical targets because absolute problem size is not always a good indicator of difficulty. For problems of a given type, we strive to attack larger and more complex instances. SAT instances that arise in typical applications of interest to us have thousands to tens to thousands of variables and millions to tens of millions of clauses. We will explore a variety of approaches that exploit structure in these large instances to make them tractable.
Symmetry and Sparsity in CNF Formulas: We noticed that SAT instances are typically sparse, in the sense that the average number of clauses per variable is bounded by a small constant (e.g., 10 or 100). It is well-known that the specific value of the C/V ratio is correlated with the complexity of randomly constructed 3-SAT instances [44], but this has been shown to be false for structured instances coming from many application domains [1][2]. Therefore, we are only concerned with the fact that there is a small upper bound for the C/V ratio. Moreover, new theoretical connections have recently been discovered between sparsity and symmetry of hypergraphs and satisfiability [38].

From our past experience with hypergraph partitioning, we know that such sparse structures often have very small cuts [12]. In the case of CNF formulae, this suggests the possibility of partitioning the variables into small groups such that very few clauses include variables from more than one group. We use the term “cut” to denote the number of such clauses. In the ideal case, when the cut is zero, the original SAT instance is decomposed into independent instances—the original instance is satisfiable if and only if each of the newly created instances is. When the cut is non-zero, we define a sub-instance by including only those clauses that contain variables from a given group. In this case, the satisfiability of each sub-instance is still a necessary, but not sufficient, condition of the satisfiability of the original instance. Intuitively, the size of the cut measures the discrepancy between the necessity and sufficiency (only one-way: large cut does not necessarily mean that the discrepancy is large). Our earlier experiments with naive partitioning heuristics were not successful [1][2], primarily because we were not able to find small cuts. However, using leading-edge partitioners developed as part of another project [12][13], we were able to produce successful decompositions of SAT instances from the DIMACS benchmark suite [26].

Table 3 shows our preliminary results with a recursive bisection implementation created during an earlier research project [13][14]. In these experiments, we applied recursive min-cut bisection to each of the SAT instances and used that to induce an ordering on the instance variables. This ordering was given to the GRASP SAT solver as additional information. We recorded GRASP results (success status and run time) achieved with this “fixed ordering” and also in the default mode with a “dynamic” ordering that chooses the next decision variable based on the results of previous decisions. According to Table 3, GRASP run time is almost always reduced when the recursive bisection ordering is used. However, for particularly easy

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DLCS Order Result</th>
<th>Search Time</th>
<th>CAPO-Generated (Fixed) Order Result</th>
<th>CAPO time</th>
<th>Search Time</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ii16b1.cnf</td>
<td>Abort</td>
<td>10000</td>
<td>Sat</td>
<td>98.92</td>
<td>6.35</td>
<td>105.27</td>
</tr>
<tr>
<td>ii16a2.cnf</td>
<td>Abort</td>
<td>10000</td>
<td>Sat</td>
<td>107.33</td>
<td>5.49</td>
<td>112.82</td>
</tr>
<tr>
<td>par16-1-c.cnf</td>
<td>Sat</td>
<td>50.04</td>
<td>Sat</td>
<td>4.41</td>
<td>22.03</td>
<td>26.44</td>
</tr>
<tr>
<td>par16-1.cnf</td>
<td>Sat</td>
<td>52.68</td>
<td>Sat</td>
<td>14.99</td>
<td>29.8</td>
<td>44.79</td>
</tr>
<tr>
<td>hole10.cnf</td>
<td>Abort</td>
<td>10000</td>
<td>Unsat</td>
<td>1.47</td>
<td>595.05</td>
<td>596.52</td>
</tr>
<tr>
<td>hole9.cnf</td>
<td>Unsat</td>
<td>1655</td>
<td>Unsat</td>
<td>0.94</td>
<td>69.84</td>
<td>70.78</td>
</tr>
<tr>
<td>ssa2670-130.cnf</td>
<td>Unsat</td>
<td>1.55</td>
<td>Unsat</td>
<td>11.82</td>
<td>2.7</td>
<td>14.52</td>
</tr>
<tr>
<td>ssa2670-141.cnf</td>
<td>Unsat</td>
<td>3.05</td>
<td>Unsat</td>
<td>10.32</td>
<td>1.13</td>
<td>11.45</td>
</tr>
<tr>
<td>hanoi5.cnf</td>
<td>Abort</td>
<td>10000</td>
<td>Sat</td>
<td>48.91</td>
<td>79.91</td>
<td>128.82</td>
</tr>
<tr>
<td>barrel3.dimacs.cnf</td>
<td>Unsat</td>
<td>0.48</td>
<td>Unsat</td>
<td>4.39</td>
<td>0.48</td>
<td>4.87</td>
</tr>
</tbody>
</table>

Table 3: Improvements in search time due to the ordering of Boolean variables by recursive min-cut bisection.

1. Given a difficult SAT instance, one can add disconnected variables and clauses to achieve any given C/V ratio without making the instance any easier.
SAT instances recursive bisection itself required more time than GRASP with either “fixed” or “dynamic” ordering. Clearly more research is needed to study the effects of partitioning and reordering on SAT instances and formulate practical recommendations that guarantee improvement, e.g., in the worst-case or average sense.

An intriguing piece of evidence that recursive-bisection ordering of variables exposes the structure of CNF instances is given in Fig. 4. In this picture, variables are represented as points on the x-axis and clauses are represented by red stars that connect those points. The center-point of each star is elevated proportionally to the span of the clause with the given ordering of variables (i.e., the distance between the rightmost and the left-most variables in the clause).

Our results regarding the use of symmetries in partitioning (Table 2) suggest that it is the extraction of symmetries, not their use in search, that deserves most attention. This is in contrast with most works on symmetry-based search that assume that symmetry information is magically available. Symmetry extraction algorithms can be generically classified as syntactic and semantic, depending on whether they can directly address a given Boolean function or remain sensitive to artifacts of particular representations, such as CNF. To this end, we note that no scalable semantic symmetry extraction algorithms are available yet, and even the best syntactic algorithms have exponential worst-case time (while often running much faster in practice) due to the complexity of the Graph Automorphism problem. Clearly, finding all symmetries is not necessary, and fast opportunistic symmetry detection algorithms may be of great practical value.

We also note that randomly constructed SAT instances typically do not have symmetries, but instances coming from particular application domains often do. At the same time, even the simplest symmetry detection algorithm, exponential in the worst-case, takes linear time (in fact, only three passes) on all but exponentially few graphs of a given size [3]. Graph Automorphism algorithms have considerably improved over the last several years, and we expect that further improvements are available. Such improvements will automatically make our techniques for symmetry detection more competitive.
Implicit Representation of Large Clause Sets: Recent advances in search techniques for SAT have enabled the solution of very large instances, mostly arising in the design, analysis, or synthesis of hardware [46][6]. Most of these techniques employ some form of “learning”: clauses that were not part of the original set are dynamically identified during the search and added. To prevent memory requirements from growing exponentially, limits are typically placed on the size of the added clauses, and occasional “garbage collection” is invoked to remove “useless” clauses.

Recently, Simon reported on an implementation of the original resolution-based Davis-Putnam procedure that succeeded in solving several hard SAT instances that had defied search-based methods [15]. Resolution algorithms for satisfiability have long been abandoned in favor of search-based techniques because of their well-known exponential space complexity. The new implementation of resolution avoided this problem by storing the CNF formula implicitly using a special form of binary decision diagram (BDD) called a zero-suppressed BDD, or ZBDD. The dramatic compression possible with this representation is illustrated in Fig. 5: for Hole-40, for example, the number of clauses during the resolution procedure peaks at over $10^{60}$, whereas the size of the ZBDD used to implicitly represent these clauses peaks at under 25,000 nodes.

ZBDDs were introduced by Minato [43] as a data structure that is particularly well suited for set manipulation of very large sets. Examples of their use include the implicit computation of the sets of prime implicants during two-level logic minimization [20] and solving graph optimization problems [21]. ZBDDs can be viewed as a generalization of “tries”, or digital search trees, that are commonly used to implement symbols tables and that provide quick insertion and retrieval [49][42][47][52]. In both tries and ZBDDs, stored objects correspond to paths in the corresponding graph data structure. Unlike tries, however, being DAGs enables ZBDDs to store a number of objects that is potentially exponential in the size of the ZBDD (see Fig. 6). The key to harnessing this compressive power is to re-cast algorithms that operate on individual set elements (in this case single clauses) so that they can be applied to sets of elements (sets of clauses). Simon developed just such an algorithm for resolution-based satisfiability.

Viewing ZBDDs as an alternative data structure to the arrays and linked lists typically used to store CNF formulas, we propose to develop set-based search algorithms for satisfiability. Specifically, we will explore the ramifications of performing backtracking search using an implicit representation of the clause database. We will study the effectiveness of such an implicit set-based representation in performing decision assignments, implication assignments, and conflict analysis. We conjecture that an approach that appropriately
Figure 6. Compression power of ZBDDs [43]. Twenty-three variations of the news headline “Americans divided over Ashcroft nomination” are shown in table (a) where each column represents an allowable sentence, and the words in the sentence are indicated by 1s (the first column encodes the original headline.) Such an explicit listing requires 240 symbols, but can clearly be compressed if structural self-similarities are exploited. One such compact representation is the trie in part (b) which requires only 58 symbols. A much more compact representation is the ZBDD in part (c) with only 12 symbols. A valid sentence in the ZBDD encoding is a path from the top node to the terminal “1” node, with symbols traversed along solid edges included and symbols traversed along dashed edges excluded.
combines resolution and search techniques, built on such an implicit representation, holds promise for solving extremely large SAT instances. We plan to empirically validate this conjecture.

**Practical Complexity Metrics:** The well-known phase transition of random 3-SAT instances at a clause-to-variable ratio of about 4.3 [44] has been shown to be inapplicable to large structured SAT instances arising from engineering applications. The C/V ratio is too coarse a measure for predicting instance hardness. Instead, we believe that metrics derived from the constraint structure of a SAT instance are more likely to correlate with the computational effort required to solve the instance. In earlier work [1][2] we examined three such metrics: (i) constraint graph width, (ii) number of “shared” variables after simple partitioning, and (iii) size of the largest strongly-connected component (SCC) in the constraint graph. While none of these metrics was perfect, the size of the largest SCC gave the best correlation with run time. In addition, as we have demonstrated above, more effective partitioning can reveal the innate structure of a SAT instance and may yet hold some promise of yielding a predictive measure of complexity. In addition, we will look for sparsity, symmetries, hierarchical structure, etc. as possible components in such a measure.

We plan to pursue this quest for structure-based complexity measures and to validate it on a wide variety of SAT instances from many application domains. This study, besides its theoretical value [7][27][54], has great practical significance [16]. In particular, having accurate and easy-to-compute complexity metrics allows us to know, in advance, if a particular formulation is suitable for a given solution scheme before expending unnecessary computational resources.

**C.4.2. Scalable Analysis of Infeasible Constraint Satisfaction Problems**

Many constraint satisfaction applications leverage recent efficient SAT solvers by reducing combinatorial problems to large-scale satisfiability instances. The main two drawbacks of this approach are (i) unpredictable run time, and (ii) lack of problem-specific feedback for unsatisfiable instances.

To illustrate these problems by example and explain our proposed solutions, we consider exact routing applications that stimulated interest since the late 1980s [25] through modern times [46][48]. In such applications one needs to lay out connections (routes) between given multiple groups of locations, subject to spatial constraints. No two connections can occupy the same spatial regions. Additional constraints may be technology- or methodology- specific. For example, global (hierarchical, coarse) routing solutions may be thought of as constraints for detailed (actual) routing. Routing instances can become difficult due to congestion, i.e., conflicts between connections that attempt to simultaneously use the same spatial regions. Reformulating routing instances into satisfiability instances allows one to apply sophisticated conflict-resolution techniques recently developed for satisfiability [51] and not known in the routing literature.

Reduction to satisfaction becomes successful when a SAT solver returns a solution, which can be translated into the language of the original routing instance. A SAT solver may also return a negative solution, which would imply that the original routing instance is not solvable. Such authoritativeness is often considered a virtue in comparison to typically heuristic routing algorithms. However, in this case, no explanation is available, and it is unclear what can be done in terms of the original routing formulation. On the other hand, classic constructive approaches like rip-up and re-route [45] and negotiation-based routing [41], have a limited ability to diagnose the unsatisfiability by identifying congested areas. This eventually enables back-annotation to previous optimizations, e.g., global routing and placement. In other words, the unsatisfiable problem instances are minimally modified in order to achieve satisfiability. We also point out that routing instances may be difficult to solve in a given small amount of time, in which case, one would
be interested in a diagnosis of such difficulties. Such reliable diagnoses are not available with known approaches.

In order to address the above concerns we propose to depart from the classical all-or-nothing constraint satisfaction mentality. When an instance is unsatisfied, or even unresolved after a given amount of time, we attempt to find a maximum set (i.e., one with the greatest possible cardinality) of constraints that can be satisfied simultaneously. In the general case, this problem does not lie in NP, and even in special cases when it does, it may be considerably harder (in worst-case terms) than the original problem. However, heuristic near-optimal solutions to this problem can provide valuable information for applications, and various approximative formulations may be easier to solve in practice than the original problem.

In the case of SAT-based detailed routing, identifying a small set of clauses that cannot be satisfied, would produce a reliable congestion map and would suggest modifications of the original global routing solutions to enable satisfiability. Instead of the maximum set of simultaneously-satisfiable constraints, we may look for maximal sets. In other words, such sets that would not allow additional constraints to be satisfied at the same time. Any maximum set is also a maximal set, but not necessarily the other way around. In practice, we expect most maximal sets to be fairly close to maximum sets by size. We will develop these and related approximative formulations and propose practical methods of solving them. The utility of our methods will be empirically validated by experiments in various application domains.

C.4.3. Scalable Logic Synthesis

We believe that our preliminary work on logic synthesis has just scratched the surface on making useful linkages between specification and implementation structures. Viewed as a proof-of-concept, we believe that it opens many opportunities for further study of such linkages. In this proposal we would like to expand this work along the following directions:

- While extracting and incorporating symmetries in the synthesis scheme, the current approach treats each of the output functions of a multi-output specification separately. Additional symmetries might be revealed, and profitably used, by a simultaneous consideration of all outputs.
- While it may be advantageous to use functional symmetry to reduce the width of the implementation network during each decomposition step, there may be instances where it is better to increase the implementation width before ultimately reducing it. This is strongly suggested by the structure of integer multipliers. We would like to examine a richer variety of implementation templates, besides support-reduction, including support-increasing and support-preserving decomposition. Another variation would be to construct the implementation from the primary outputs towards the primary inputs, which might work particularly well if all output functions are considered simultaneously.
- Besides symmetry, we would like to examine other structural properties of functional specifications and how they might be linked to implementation structures. For instance, it may be argued that control logic lacks any symmetries, and would not benefit from the symmetry-centric approach outlined so far. A synthesis approach that takes advantage of unateness in the specifications might be more appropriate in such cases.

Finally, we believe that the scalability of synthesis requires a departure from the use of binary decision diagrams as the underlying Boolean reasoning engine. We conjecture that a marriage between highly-scalable SAT solvers (e.g. based on ZBDDs) and efficient structure-finding techniques would permit the synthesis (and re-synthesis) of very large (hundreds of thousands of gates) high-quality logic circuits. Such an achievement, whose odds are long at the moment, would at the very least validate our thesis that the study and utilization of structure is key to achieving scalability.