VLSI DESIGN
1995, Vol. 3, Nos. 3-4, pp. 359-371
Reprints available directly from the publisher
Photocopying permitted by license only

Decomposition and Reduction: General Problem-Solving Paradigms[1]

MICHAL SERVÍT AND JAN ZAMAZAL
Department of Computers, Faculty of Electrical Engineering, Czech Technical University, Karlovo nám. 13, 121 35 Prague 2, Czech Republic

In this article we will be discussing the utilization of decomposition and reduction for development of algorithms. We will assume that a given problem instance can be somehow broken up into two smaller instances that can be solved separately. As a special case of decomposition we will define a reduction, i.e. such a decomposition that one of the resulting instances is trivial. We will define several versions of decomposition and reduction in a hierarchical way. Different kinds will be distinguished by their ability to preserve an optimal solution of the original instance. General schema of an algorithm utilizing the proposed notions will be introduced and a case-study demonstrating the adaptation of this schema for the covering problem will be provided.

Key Words: *Algorithms, Complexity, Decomposition, Reduction, Heuristics, Empirical evaluation*

1. INTRODUCTION

The notion of *decomposition* is known to have several different interpretations:

- The problem itself represents a decomposition of an entity like a graph, network, set, FSM, etc. into two or more entities of the same kind. Example: The problem of partitioning a hypergraph into the specified number of disconnected parts by removing hyperedges.
- Decomposition is a solution technique which is based on the transformation of the (original) problem to be solved into several subproblems of different kinds that are then solved sequentially. Example: The problem of VLSI layout is usually decomposed into partitioning, floorplanning, and routing.
- Decomposition is a solution technique which is based on the transformation of an instance of a particular problem into several instances of the same kind that are then solved independently. Example: Dynamic programming approach to the minimum Steiner tree problem [9].

Throughout this article we will deal with the last of the above-mentioned interpretations, particularly with respect to the following class of problems: *A finite set and a system of constraints are given, and a minimal subset satisfying constraints is to be found.*

Many theoretical problems can be formulated in this manner, e.g. the shortest path problem, Steiner tree problem, covering problem, knapsack problem and others. These problems have a large number of applications in various areas, particularly in VLSI design.

Let us assume the shortest path problem as an example: Given a weighted graph G and two nodes v_s and v_t, construct a shortest path connecting v_s and v_t. This problem can be formulated in a different manner: Given a set of weighted edges, construct a subset of edges so that a connected graph containing nodes v_s and v_t is created and the total weight of edges in the subset is minimized (Fig. 1).

Throughout this article this class of problems will be denoted *Set Minimization Problems*. The common feature of all the set minimization problems is their discrete and finite nature. The finiteness of these problems guarantees that an optimal solution can be found by a *brute-force method*: generate consecutively all subsets, check whether the constraints are satisfied, and always store the best solution found so far. Such methods, unfortunately, can succeed in a reasonable time for small instances only. This is why in this article we will focus our attention on the design and evaluation of efficient

[1]This work has been supported by the Czech Technical University under grant No. 8095 and by the Czech Grant Agency under grant No. 102/93/0916.

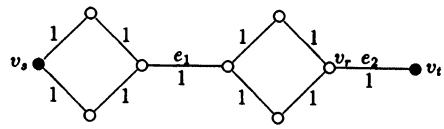

FIGURE 1 An example of the shortest path problem.

algorithms for solution of large set minimization problems based on a decomposition and reduction of an instance of the problem.

Informally, to *decompose* a problem instance means to change it to two (or more) instances that can be solved independently. The solution of the original instance is then obtained from solutions of the new instances. In cases in which one of the resulting instances is trivial, the decomposition is called *reduction*.

Let us demonstrate these notions with the above-mentioned shortest path problem (see Fig. 1). On the assumption that the edge e_1 is known to be a part of a shortest path, the problem can be *decomposed* to finding a shortest path from v_s to this edge, and to finding a shortest path from v_t to this edge. Similarly, on the assumption that the edge e_2 is a part of a shortest path, the original problem can be *reduced* to the problem of finding a shortest path from v_s to v_r.

Apparently, decomposition makes the size of the instance to be solved smaller and in this way it may reduce computational requirements of solution algorithms. Moreover, recursive decomposition may lead directly to a minimal solution. Generally, however, there is no guarantee that such an approach will be successful. This is why the process of (exact) decomposition is often combined with exhaustive or heuristic search techniques.

The notions of decomposition and reduction seem to be quite simple and straightforward. The main problem, however, is to find a way *how* to decompose and/or reduce an instance so that the solution of the subinstances generated quickly leads to finding a solution of the original instance. The theory of *dynamic programming* introduces the notion of *decomposable problem* [9] and conditions that must be fulfilled so that a certain type of decomposition can be used. In this article we will extend this approach by introducing a hierarchy of types of decomposition and reduction and propose a general schema of an algorithm utilizing the proposed notions.

The rest of this article is organized as follows: In the first part we give the formal framework. Several types of decomposition and reduction of an instance of a set minimization problem are then introduced and a general algorithm utilizing the hierarchy is proposed. The second part of this article provides a case-study demonstrating the general notions with the covering problem. First, the formal definition of the covering problem is given and its applications in the field of VLSI design are mentioned. Several rules of decomposition and reduction are then derived, and the general algorithm proposed in the first part of this article is adopted for the covering problem. A complexity analysis and an empirical testing of this algorithm is described. The last part of this article is devoted to conclusions.

2. SET MINIMIZATION PROBLEMS

In this section we will formally define the class of problems that will be considered further. We will be dealing with a special type of optimization problems denoted *set minimization problems*.

Definition 2.1—Set Minimization Problem [9]

A **Set Minimization Problem** is a minimization problem corresponding to the following description:

Instance: a triple $\Pi = (A, T, W)$ where $A = \{a_1, a_2, \ldots, a_n\}$, is a finite set, T is a system of constraints, and W is a mapping $W: A \rightarrow R^+$,

Configurations: all subsets of the set A,

Solutions: all subsets $S \subseteq A$ so that all constraints in T are satisfied,

Minimize:

$$W(S) = \sum_{a_i \in S} W(a_i) \bullet$$

The following notions will be found useful for the proposed definitions and descriptions:

Definition 2.2—Infeasible/Feasible Instance

An instance $\Pi = (A, T, W)$ is called **Infeasible** iff there exists no solution S of Π. Otherwise Π is called **Feasible**. •

Definition 2.3—Size of an Instance

Let $\Pi = (A, T, W)$ be an instance of the set minimization problem. A (non-negative) number $SIZE(\Pi)$ calculated as follows, is called a **Size** of the instance Π:

$$SIZE(\Pi) = 2^{card(A)} * C(T)$$

where $card(A)$ is the number of elements in A and $C(T)$ is a measure of the complexity of the system of constraints T. •

Size of an instance corresponds to the time complexity of a brute-force method: System of constraints T is evaluated $2^{card(A)}$ times by this method and $C(T)$ corresponds to the time requirements of a single evaluation.

Definition 2.4—Trivial Instance

An instance $\Pi = (A, T, W)$ is called **Trivial** iff one of the following conditions holds:

1. T is empty,
2. A contains at most one element. •

The importance of this notion is obvious: If T is empty, $S = \emptyset$ is the only minimal solution. If T is not empty and A is empty, the instance is infeasible. If A contains one element, then either $S = \emptyset$ is the only minimal solution, or $S = A$ is the only minimal solution, or the instance is infeasible. Thus, trivial instances can be solved in polynomial time.

3. DECOMPOSITION AND REDUCTION

In this section we will introduce a hierarchy of types of decomposition and reduction. Apparently, a particular instance of the set minimization problem can have more than one minimal solution. Thus, various types of decomposition can be classified by their ability to keep none, at least one, or all minimal solutions.

The following definition corresponds to the conditions of the decomposable problem introduced in [9]:

Definition 3.1—Approximate Decomposition of an Instance

Let $\Pi = (A, T, W)$ be an instance of the set minimization problem. A pair (Π', Π'') where $\Pi' = (A', T', W')$ and $\Pi'' = (A'', T'', W'')$ are instances is called an **Approximate Decomposition** of Π iff the following condition holds:

- if S' and S'' are minimal solutions of Π' and Π'' respectively, $S = S' \cup S''$ *is a solution of* Π. •

As a result of an approximate decomposition we obtain two instances Π' and Π'' that can be solved separately. If minimal solutions of the new instances are found, a solution of the original instance can be constructed as their union. However, there is no guarantee that this solution will be optimal. Moreover, infeasibility of one of the new instances does not necessarily imply the infeasibility of the original instance. This is why we introduce a stronger condition preserving at least one minimal solution.

Definition 3.2—Pure Decomposition of an Instance

Let $\Pi = (A, T, W)$ be an instance of the set minimization problem. A pair (Π', Π'') where $\Pi' = (A', T', W')$ and $\Pi'' = (A'', T'', W'')$ are instances is called a **Pure Decomposition** of Π iff one of the following conditions holds:

1. Π, Π' and Π'' are feasible instances and if S' and S'' are minimal solutions of Π' and Π'' respectively, $S = S' \cup S''$ is a minimal solution of Π,
2. Π is an infeasible instance and at least one of Π', Π'' is an infeasible instance. •

As a result of a pure decomposition we obtain two instances Π' and Π'' that can be solved separately. *At least one* minimal solution of the original instance can be obtained from minimal solutions of the new instances. In addition, infeasibility of one of the new instances implies the infeasibility of the original instance. However, there is no guarantee that *all* minimal solutions of the original instance can be derived from minimal solutions of the new instances. This is why we introduce a stronger condition preserving all minimal solutions.

Definition 3.3—Proper Decomposition of an Instance

Let $\Pi = (A, T, W)$ be an instance of the set minimization problem. A pair (Π', Π'') where $\Pi' = (A', T', W')$ and $\Pi'' = (A'', T'', W'')$ are a pure decomposition of Π is called a **Proper Decomposition** of Π iff the following condition holds:

- for each minimal solution S of Π such minimal solutions S' and S'' of Π' and Π'' respectively exist, that $S = S' \cup S'$. •

As a result of a proper decomposition we obtain two instances Π' and Π'' that can be solved separately. *All* minimal solutions of the original instance can be obtained from minimal solutions of the new instances.

Naturally, all these types of decomposition can be defined for more than two new instances. Doing so, however, brings nothing substantially new because a decomposition to more than two instances can be obtained by repeated decomposition to two instances.

The size of an instance (see Definition 2.3) is a measure of the time required to solve the instance by the brute-force method. The total time to solve the new instances is proportional to $2^{card(A')} + 2^{card(A'')}$, while the original instance is solved in time proportional to $2^{card(A)}$. Assuming that $card(A) = card(A') + card(A'')$, the "best" decomposition is such that $card(A') = card(A'') = \frac{1}{2} card(A)$.

Generally, finding a "true" decomposition consisting of two non-trivial instances is a complicated task. Quite often, however, a trivial part of an instance can be found and extracted. As a result of such a "special" decomposition we obtain, in fact, just one instance of smaller size. This is why it is called a *reduction*. As in the case of decomposition, we will define a hierarchy of different types of reduction.

Definition 3.4—Approximate Reduction of an Instance

Let $\Pi = (A, T, W)$ be an instance of the set minimization problem and let (Π', Π'') be an approximate decomposition of Π so that $\Pi'' = (A'', T'', W'')$ is a trivial instance the minimal solution of which is a set R. Then the pair (Π', R) is called an **Approximate Reduction** of Π. •

Definition 3.5—Pure Reduction of an Instance
Let $\Pi = (A, T, W)$ be an instance of the set minimization problem and let (Π', Π'') be a pure decomposition of Π so that $\Pi'' = (A'', T'', W'')$ is a trivial instance the minimal solution of which is a set R. Then the pair (Π', R) is called a **Pure Reduction** of Π. •

Definition 3.6—Proper Reduction of an Instance
Let $\Pi = (A, T, W)$ be an instance of the set minimization problem and let (Π', Π'') be a proper decomposition of Π so that $\Pi'' = (A'', T'', W'')$ is a trivial instance the minimal solution ofwhich is a set R. Then the pair (Π', R) is called a **Proper Reduction** of Π. •

Naturally, if a decomposition of an instance has been found, the resulting instances can be recursively decomposed again until trivial and/or infeasible instances are encountered. Results of such recursive decompositions are then to be treated according to the type of decompositions used in the recursion.

Thus, the above-mentioned hierarchy of decompositions and reductions can be used for classification of algorithms:

1. Algorithms based on utilization of all three types of decomposition and/or reduction. Generally, if a solution is found by such an algorithm, it is a correct solution of the original instance. However, this class of algorithms does not guarantee that a minimal solution will be found. Moreover, if a solution is not found by such an algorithm, no conclusion can be made about the feasibility of the original instance.
2. Algorithms based on utilization of pure and proper decomposition and/or reduction. This class of algorithms guarantees finding *at least one* minimal solution of the instance to be solved. If an infeasible (sub)instance is encountered, the original instance is infeasible.
3. Algorithms based on utilization of only proper decomposition and/or proper reduction. This class of algorithms guarantees finding *all* existing minimal solutions of the instance to be solved. If an infeasible (sub)instance is encountered, the original instance is infeasible.

4. UTILIZATION OF DECOMPOSITION AND REDUCTION

In this section we will present a strategy of application of the above-mentioned notions. First, we will present a skeleton of an algorithm and a discussion of various possibilities of search strategies. Thereafter, we will discuss the evaluation of effectiveness of such an algorithm.

4.1 The Algorithm

In the previous section we have introduced a hierarchy of types of decomposition and reduction according to their ability to keep none, at least one, or all minimal solutions. Based on this hierarchy we have introduced a hierarchy of algorithms. However, only algorithms of the first type represent the most general approach applicable in all cases. The reason is simple: the recursive utilization of pure and proper decomposition does not need to lead to a set of trivial subinstances. In other words, such a non-trivial subinstance can be encountered that is not further decomposable using only proper and/or pure decomposition and reduction rules. To solve such a subinstance, an approximate decomposition or reduction must be used. This is why the proposed algorithm utilizes all three types of decomposition.

The question now is, in which order particular types of decomposition should be applied. The following strategy seems to be appropriate: The possibilities of proper decomposition and/or reduction are to be checked first. If the instance has not been solved, we can admit that some minimal solutions will be lost using pure decomposition and/or reduction. If the instance still has not been solved, approximate decomposition and/or reduction can be applied.

If at least one approximate decomposition and/or reduction has been applied, there is no guarantee that a minimal solution will be found. Moreover, an infeasible subinstance can be encountered even though the original instance is feasible. This is why some techniques should be used to increase the probability of finding a "good" solution. We propose to use a combination of two well-known techniques: *branch and bound* and *heuristic search* [12].

A schema of an algorithm based on this idea is shown in Fig. 2. Each decomposition can result in one of three ways:

- at least one of the resulting instances is non-trivial and all the remaining are trivial and feasible,
- at least one of the resulting instances is trivial and infeasible,
- all the resulting instances are trivial and feasible.

In the first case the process of decomposition is repeated. Otherwise an *analysis* is to be performed and according to its result either the process is interrupted (and the best solution found so far claimed to be the result), or the status before the last approximate decomposition is restored and another possibility of decomposition is attempted.

As a result, the proposed algorithm is able to perform a systematic exhaustive search, i.e. to examine *all* relevant possibilities of approximate decomposition. On

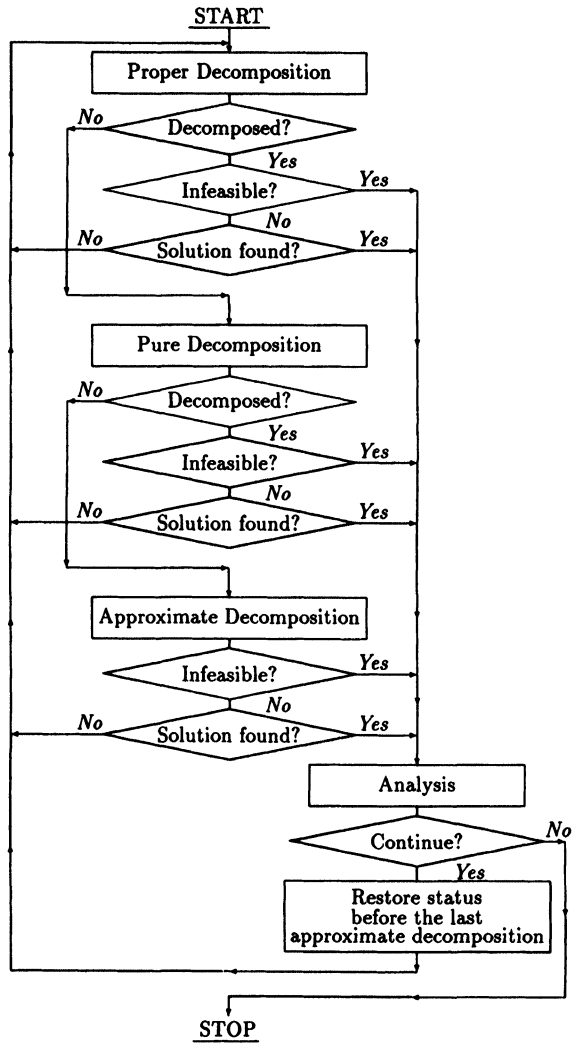

FIGURE 2 General schema of an algorithm utilizing various types of decomposition.

the other hand, the algorithm may perform a heuristic search, i.e. to examine only several possibilities of approximate decomposition. In addition, *bounding* (not shown in Fig. 2) can be added in both cases. The *analysis* step of the algorithm controls in fact the degree of branching, or, in other words, the number of returns.

Possibilities of proper and pure decompositions seem to be very promising, at least at first sight, because they keep all or at least one minimal solution. On the other hand, to evaluate whether such a decomposition can be applied, takes some time. Moreover, in some cases no such decomposition exists. Thus, pure and proper decompositions are to be treated with care and must be well tested before being incorporated into an algorithm.

Since pure and/or proper decompositions do not need be always found, the approximate decomposition (together with the analysis) is the keypoint of this algorithm. Basically, there are two extreme strategies:

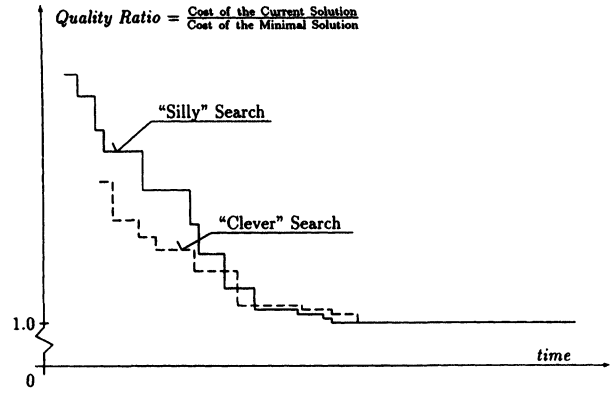

FIGURE 3 Convergence of solution towards the optimum.

-a "clever" search for a good approximate decomposition guided by a heuristic function (the evaluation of which, however, will take some time) and a limited number of returns, or

-a "silly" (but fast) approximate decomposition with a higher number of returns.

Fig. 3 shows an example of the convergence curves of the both strategies. Notice that the analysis step should take into consideration the shape of the convergence curve as well as the number of approximate decompositions when deciding whether to stop the search. Commonly known search strategies, such as *greedy search*, *simulated annealing*, *tabu search* etc. (for overview see [15]) are in fact special cases of the general analysis step.

4.2 Evaluation of the Algorithm

In the previous subsection we have discussed the utilization of various possibilities of decomposition in the proposed algorithm. The question now is, *how* to find out, whether a particular decomposition rule is useful, or not. In addition, usefullness of a decomposition rule may be affected by the search strategy (the analysis step) employed in the algorithm.

From the practical point of view, the following two values are the most important: the *run time* of the algorithm, and the *quality of result* in comparison with the minimal solution.

Since the behaviour of the algorithm depends heavily on the particular circumstances, there is no general answer to the above-mentioned question. In some cases the behaviour of a particular decomposition rule can be derived analytically. Similarly, an analysis of the structure of an instance may give the probability that a decomposition rule will be applied successfuly. Obviously, only decomposition rules with good run time complexity and high enough efficiency should be incorporated into the algorithm.

However, in most cases we are forced to resort to *empirical* testing on a set of benchmarks [15]. Sometimes these test instances will have arisen in a real situation, but real data are scarce and may represent only a small fraction of the possible population of instances. Thus, benchmarks are often generated by randomly sampling from what is assumed to be the population of instances of problem of the appropriate type.

5. COVERING PROBLEM

In this section we will demonstrate the utilization of the notions introduced in the first part of this article with the covering problem. First, the covering problem and several related notions will be formally defined. Thereafter, several rules will be proposed to find a decomposition and/or reduction of the covering problem, and finally, the adaptation of the general algorithm with respect to these rules and its evaluation will be described.

5.1 Definition and Complexity

In this subsection we will introduce a formal definition of the covering problem and discuss its complexity.

Definition 5.1—Covering Problem (CP)

The **Covering Problem** is a set minimization problem the instance of which is as follows:

Instance: a triple $\Pi = (A, T, W)$ where $A = \{a_1, a_2, \ldots, a_n\}$ is a set of n boolean variables, T is a system of m boolean disjunctions:

$$T_1 = (\hat{a}_{11} \vee \hat{a}_{12} \vee \ldots \vee \hat{a}_{1n})$$
$$T_2 = (\hat{a}_{21} \vee \hat{a}_{22} \vee \ldots \vee \hat{a}_{2n})$$
$$\ldots$$
$$T_m = (\hat{a}_{m1} \vee \hat{a}_{m2} \vee \ldots \vee \hat{a}_{mn})$$
$$\text{where } \hat{a}_{ji} = a_i \text{ or } \hat{a}_{ji} = \bar{a}_i \text{ or } \hat{a}_{ji} = 0,$$

and W is a mapping $W: A \rightarrow R^+$

Solutions: all such subsets $S \subseteq A$ that the following holds:

$$\forall T_j \; \exists a_i: [(a_i \in S) \wedge (\hat{a}_{ji} = a_i)] \vee [(a_i \notin S) \wedge (\hat{a}_{ji} = \bar{a}_i)] \bullet$$

Let us note that a solution of an instance of the covering problem is traditionally referred to as a *cover*. A minimal solution is called a *minimal cover*. Similarly, a boolean variable in a constraint is called a *literal*. A literal is called an *inverted literal* if $\hat{a}_{ji} = \bar{a}_i$. If $\hat{a}_{ji} = a_i$, it is called a *direct literal*.

We say that $\hat{a}_{ji}$ is *true* if $a_i \in S$ and $\hat{a}_{ji} = a_i$, or if $a_i \notin S$ and $\hat{a}_{ji} = \bar{a}_i$. If there is any $\hat{a}_{ji}$ *true* in a constraint T_j, we say that T_j is *covered* by variable a_i; otherwise we say that T_j is an *uncovered constraint*.

Example 5.1

An instance of the covering problem may look as follows:

$$\Pi: A = \{a_1, a_2, a_3, a_4\}$$
$$T: T_1 = (a_1 \vee a_2 \vee 0 \vee 0),$$
$$T_2 = (0 \vee a_2 \vee a_3 \vee \bar{a}_4),$$
$$T_3 = (0 \vee \bar{a}_2 \vee a_3 \vee a_4),$$
$$T_4 = (0 \vee 0 \vee a_3 \vee a_4),$$
$$W: W(a_i) = 1, W(a_2) = 2, W(a_3) = 3, W(a_4) = 4$$

$S = \{a_1, a_2, a_3, a_4\}$ is a cover of this instance; other covers are $\{a_1, a_2, a_3\}$, $\{a_1, a_3\}$, $\{a_2, a_4\}$, etc. $S = \{a_1, a_3\}$ is the only minimal cover. The cost of this cover $W(S) = W(a_1) + W(a_3) = 4$.

In the following text we will use a shortened notation; 0's are obviously not necessary in T and can be omitted:

$T_1 = (a_1 \vee a_2)$, $T_2 = (a_2 \vee a_3 \vee \bar{a}_4)$, $T_3 = (\bar{a}_2 \vee a_3 \vee a_4)$, $T_4 = (a_3 \vee a_4)$

5.2 History and Applications

The covering problem has a wide range of applications in VLSI design. Probably the best known example is the Quine-McCluskey's method of minimization of boolean functions [10]: given a set of prime implicants, a subset of minimum total cost is sought such that all 1-minterms are covered. This condition can be expressed as a boolean formula in a conjunctive normal form [13], that maps directly to the system of constraints T. In this case T contains *no* inverted literal. This is why this type of covering problem has been called *Unate Covering Problem*. Other applications of the unate covering problem in the field of circuit design are the following (for overview see [19]):

- design of a minimum-cost two-level combinational network,
- design of a minimum-cost TANT network,
- finding a minimum test set, and
- minimizing the bit dimension of the control memory.

The *Binate Covering Problem* is a generalization of the unate covering problem where T is not restricted to be unate and therefore can express more complex constraints.

Probably the best known application of a binate covering problem is the problem of state reduction in incompletely specified finite state machines [6], [7]: given a set of maximal compatibles, a cover of minimal total cost is sought such that closure constraints are satisfied. In the field of electronic circuit design the problem has more applications:

- minimization of boolean relations [2],
- synthesis of sequential logic circuits [4],

- library-based technology mapping [17], and
- minimum test pattern generation [8].

Let us note that there exists a decision problem closely related to the covering problem. This problem is called a *Satisfiability Problem*. The satisfiability problem has been proven to be an NP-complete problem if the system of constraints T contains at least three variables in each constraint [3]. A polynomial time algorithm has been proposed in [1] for such instances of the satisfiability problem that contain at most two variables in each constraint T_j. The covering problem, however, is NP-hard if the system of constraints T contains at least two variables in each constraint.

5.3 Decomposition and Reduction of a CP

5.3.1 Decomposition Rules

In this subsection we will deal with a "true" decomposition, i.e. such a decomposition where an instance of the covering problem is broken up into two non-trivial smaller instances that are solved separately. The solution of the original instance is obtained as a union of solutions of the new instances.

We will propose three theorems exemplifying different types of decomposition. All the theorems will be considering three instances of the covering problem denoted $\Pi = (A, T, W)$, $\Pi' = (A', T', W')$, and $\Pi'' = (A'', T'', W'')$ satisfying the following set of conditions:

$$\left.\begin{array}{lcl} A & = & A' \cup A'' \\ T & = & T'_1, T'_2 \ldots T'_{m'}, T''_1, T''_2, \ldots T''_{m''} \\ W'(a_i) & = & W(a_i) \text{ for each } a_i \in A' \\ W''(a_i) & = & W(a_i) \text{ for each } a_i \in A'' \end{array}\right\} \quad (1)$$

First, we will propose a simple observation that will be found useful for proving the subsequent theorems:

Theorem 5.1 Let $\Pi = (A, T, W)$, $\Pi' = (A', T', W')$, and $\Pi'' = (A'', T'', W'')$ be feasible instances of the covering problem and let S' and S'' be minimal covers of Π' and Π'' respectively. The set $S = S' \cup S''$ is a cover of Π if the set of conditions (1) is fulfilled and if $S' \cap A'' = S'' \cap A'$. ◇

Proof: Apparently, set A is divided into three disjoint parts: $A_1 = A \backslash A''$, $A_2 = A \backslash A'$, and $A_3 = A' \cap A''$. Variables in A_1 have no effect on constraints in T'' and variables in A_2 have no effect on constraints in T'. Variables in A_3 are contained in both T' and T''; this is why each variable from A_3 either must be contained in both S' and S'', or must not be contained in any of them. Then the set $S = S' \cup S''$ is a cover of Π. □

Example 5.2

Let us consider the following instances:

$\Pi : A = \{a_1, a_2, a_3, a_4\}$
$T : T_1 = (a_1 \vee a_2), T_2 = (a_2 \vee a_3), T_3 = (a_1 \vee a_3),$
$T_4 = (a_3 \vee a_4), T_5 = (a_2 \vee a_4)$
$W : W(a_1) = 1, W(a_2) = 1, W(a_3) = 1, W(a_4) = 1$
$\Pi' : A' = \{a_1, a_2, a_3\}$
$T' : T'_1 = (a_1 \vee a_2), T'_2 = (a_2 \vee a_3), T'_3 = (a_1 \vee a_3)$
$W' : W'(a_1) = 1, W'(a_2) = 1, W'(a_3) = 1$
$\Pi'' : A'' = \{a_2, a_3, a_4\}$
$T'' : T''_1 = (a_2 \vee a_3), T''_2 = (a_3 \vee a_4), T''_3 = (a_2 \vee a_4)$
$W'' : W''(a_2) = 1, W''(a_3) = 1, W''(a_4) = 1$

Apparently, $S' = \{a_1, a_2\}$ is a minimal cover of Π' and $S'' = \{a_2, a_4\}$ is a minimal cover of Π''. Since it holds that $S' \cap A'' = S'' \cap A' = \{a_2\}$, the set $S = S' \cup S'' = \{a_1, a_2, a_4\}$ is a cover of Π. However, the minimal cover of Π is the set $S_1 = \{a_2, a_3\}$.

Theorem 5.2 Let $\Pi = (A, T, W)$, $\Pi' = (A', T', W')$, and $\Pi'' = (A'', T'', W'')$ be instances of the covering problem and let S' and S'' be the only minimal covers of Π' and Π'' respectively. The pair (Π', Π'') is an approximate decomposition of Π if the set of conditions (1) is fulfilled and if $S' \cap A'' = S'' \cap A'$.

Proof: Proof is this theorem follows directly from Theorem 5.1 and from properties of approximate decomposition (see Definition 3.1). □

Example 5.3

Let us consider the following instances:

$\Pi : A = \{a_1, a_2, a_3, a_4\}$
$T : T_1 = (a_1 \vee a_2), T_2 = (a_2 \vee a_3), T_3 = (a_1 \vee a_3),$
$T_4 = (a_3 \vee a_4), T_5 = (a_2 \vee a_4)$
$W : W(a_1) = 2, W(a_2) = 2, W(a_3) = 3, W(a_4) = 2$
$\Pi' : A' = \{a_1, a_2, a_3\}$
$T' : T'_1 = (a_1 \vee a_2), T'_2 = (a_2 \vee a_3), T'_3 = (a_1 \vee a_3)$
$W' : W'(a_1) = 2, W'(a_2) = 2, W'(a_3) = 3$
$\Pi'' : A'' = \{a_2, a_3, a_4\}$
$T'' : T''_1 = (a_2 \vee a_3), T''_2 = (a_3 \vee a_4), T''_3 = (a_2 \vee a_4)$
$W'' : W''(a_2) = 2, W''(a_3) = 3, W''(a_4) = 2$

Apparently, $S' = \{a_1, a_2\}$ is the only minimal cover of Π' and $S'' = \{a_2, a_4\}$ is the only minimal cover of Π''. Since it holds that $S' \cap A'' = S'' \cap A' = \{a_2\}$, the set $S = S' \cup S'' = \{a_1, a_2, a_4\}$ is a cover of Π. However, the only minimal cover of Π is the set $S_1 = \{a_2, a_3\}$.

The following theorem introduces a condition similar to that proposed by Pipponzi and Somenzi in [14]:

Theorem 5.3 Let $\Pi = (A, T, W)$, $\Pi' = (A', T', W')$, and $\Pi'' = (A'', T'', W'')$ be instances of the covering problem; let S' and S'' be the only minimal covers of Π' and Π'' respectively. The pair (Π', Π'') is a proper decomposition of Π if the set of conditions (1) is fulfilled and if $S' \cap S'' = A' \cap A''$.

Proof: First, let us consider that Π is a feasible instance. Then it is sufficient to prove that $S = S' \cup S''$ is the only minimal cover of Π. Apparently, $S = S' \cup S''$ is a cover of Π, because conditions of Theorem 5.1 are fulfilled. Cost of this cover is:

$$W(S) = \sum_{a_i \in S' \cup S''} W(a_i) = \sum_{a_i \in S'} W(a_i) + \sum_{a_i \in S''} W(a_i) - \sum_{a_i \in S' \cap S''} W(a_i)$$

$W(S)$ increases with the cost of S' and S'' and decreases with the cost of $S' \cap S''$. Costs of S' and S'' are minimal; $W(S)$ thus can be minimized only with the increase of the cost of $S' \cap S''$. Obviously, the cost of this set is maximal in cases such that $S' \cap S'' = A' \cap A''$. Since S' and S'' are the only minimal covers of Π' and Π'', $S = S' \cup S''$ is the only minimal cover of Π.

Now, let us consider that Π is an infeasible instance. By contradiction: Let S', S'' be minimal covers of Π', Π'' respectively so that $S' \cap S'' = A' \cap A''$. Then, obviously, $S' \cap A'' = S'' \cap A'$, and the set $S = S' \cup S''$is a cover of Π (Theorem 5.1). This is why no such decomposition exists for an infeasible instance. □

Example 5.4

Let us consider the following instances:

$\Pi : A = \{a_1, a_2, a_3, a_4, a_5\}$
$T : T_1 = (a_1 \vee a_3), T_2 = (a_2 \vee a_4), T_3 = (a_4 \vee a_5)$
$W : W(a_1) = 1, W(a_2) = 2, W(a_3) = 2,$
$W(a_4) = 1, W(a_5) = 2$
$\Pi' : A' = \{a_1, a_2, a_3, a_4\}$
$T' : T'_1 = (a_1 \vee a_3), T'_2 = (a_2 \vee a_4)$
$W' : W'(a_1) = 1, W'(a_2) = 2, W'(a_3) = 2,$
$W'(a_4) = 1$
$\Pi'' : A'' = \{a_4, a_5\}$
$T'' : T''_1 = (a_4 \vee a_5)$
$W'' : W''(a_4) = 1, W''(a_5) = 2$

Apparently, $S' = \{a_1, a_4\}$ is the only minimal cover of Π' and $S'' = \{a_4\}$ is the only minimal cover of Π''. $S = S' \cup S'' = \{a_1, a_4\}$ is the only minimal cover of Π because it holds that $A' \cap A'' = \{a_1, a_2, a_3, a_4\} \cap \{a_4, a_5\} = \{a_4\} = S' \cap S''$.

The following theorem presents a special case of Theorem 5.3:

Theorem 5.4 Let $\Pi = (A, T, W)$, $\Pi' = (A', T', W')$, and $\Pi'' = (A'', T'', W'')$ be instances of the covering problem. The pair (Π', Π'') is a proper decomposition of Π if the set of conditions (1) is fulfilled and it holds that $A' \cap A'' = \emptyset$.

Proof: Apparently, if Π is a feasible instance, the set $S = S' \cup S''$ is a cover of Π, because conditions of Theorem 5.1 are fulfilled for all covers of Π' and Π''. The cost of this cover is:

$$W(S) = \sum_{a_i \in S' \cup S''} W(a_i) = \sum_{a_i \in S'} W(a_i) +$$

$$\sum_{a_i \in S''} W(a_i) - \sum_{a_i \in S' \cap S''} W(a_i)$$

Since $A' \cap A'' = \emptyset$ it must hold that $S' \cap S'' = \emptyset$. $W(S)$ is minimum because S' and S'' are minimal covers of Π' and Π'' respectively.

If Π is an infeasible instance, the proof is identical to that of Theorem 5.3. □

Example 5.5

Let us consider the following instances:

$\Pi : A = \{a_1, a_2, a_3, a_4, a_5\}$
$T : T_1 = (a_1 \vee a_2), T_2 = (a_1 \vee a_3), T_3 = (a_4 \vee a_5)$
$W : W(a_1) = 1, W(a_2) = 1, W(a_3) = 1,$
$W(a_4) = 1, W(a_5) = 1$
$\Pi' : A' = \{a_1, a_2, a_3\}$
$T' : T'_1 = (a_1 \vee a_2), T'_2 = (a_1 \vee a_3)$
$W' : W'(a_1) = 1, W'(a_2) = 1, W'(a_3) = 1$
$\Pi'' : A'' = \{a_4, a_5\}$
$T'' : T''_1 = (a_4 \vee a_5)$
$W'' : W''(a_4) = 1, W''(a_5) = 1$

Obviously, Π' and Π'' are a proper decomposition of Π. There is one minimal cover $S' = \{a_1\}$ of Π'. Π'' has two minimal covers $S''_1 = \{a_4\}$ and $S''_2 = \{a_5\}$. Thus, all minimal covers of Π are the following:

$$S_1 = S' \cup S''_1 = \{a_1, a_4\}$$

$$S_2 = S' \cup S''_2 = \{a_1, a_5\}$$

The application of Theorem 5.4 is quite straightforward. A system of m (i.e. number of constraints in T) subsets of A is created so that each subset A_j contains elements related to literals in constraint T_j. All subsets that are not disjoint are then united. As a result we obtain either a system of disjoint subsets of A, or the set A. Each subset corresponds to an "independent" part of the original instance that can be solved separately from the remaining ones. Apparently, such an evaluation of an

intersection of a system of subsets can be done in polynomial time.

On the other hand, we have not found any polynomial-time algorithm that would find whether there exists a decomposition of an instance according to Theorem 5.2 and/or 5.3.

An algorithm based on a theorem similar to Theorem 5.3 has been published by Pipponzi and Somenzi in [13]. However, this algorithm does not run in polynomial time and the results presented do not show substantial difference in comparison with a branch and bound algorithm.

5.3.2 Reduction Rules

Unlike "true" decompositions, possibilities of reduction of an instance of the covering problem have been investigated by other authors (see [2], [13]). Other reduction rules are known for the closely related satisfiability problem [5], [16] and can be easily adopted for the covering problem. This is why we will limit ourselves to an overview of the most important reduction rules. Those, who are interested in a complete survey are referred to [18].

Generally, the reduction rules are based on forms of occurrences of a variable in the system of constraints, and/or on relations between two variables, or two constraints. The following observation presents a simple rule of approximate reduction.

Observation 5.1

Let $\Pi = (A, T, W)$ be an instance of the covering problem and $a_x \in A$. Then an approximate reduction (Π', R) can be derived as follows:

1. $A' = A \backslash \{a_x\}$
2. Create T' from T in the following way:
   (a) Copy from T into T' all constraints T_j so that $\hat{a}_{jx} \neq a_x$
   (b) Remove all literals $\bar{a}_x$ from T'
3. $R = \{a_x\}$ $\diamond$

Let us note that a complementary rule can be derived by exchanging literals a_x and $\bar{a}_x$ and setting $R = \emptyset$.

A rule of pure reduction based on occurrences of a pair of variables (denoted *Dominant Variables*) is described in [18]. A similar notion is presented in [13] and is referred to as *column dominance*.

All the subsequent reduction rules present possibilities of proper reduction.

Definition 5.2—Essentially Accepted Variable

Let $\Pi = (A, T, W)$ be an instance and $a_x \in A$. Variable a_x is **Essentially Accepted** iff there exists such a constraint T_j in T that $T_j = (a_x)$. •

Definition 5.3—Essentially Rejected Variable

Let $\Pi = (A, T, W)$ be an instance and $a_x \in A$. Variable a_x is **Essentially Rejected** iff there exists such a constraint T_j in T that $T_j = (\bar{a}_x)$. •

Observation 5.2

Let $\Pi = (A, T, W)$ be an instance and $a_x \in A$ an essentially accepted variable. Then a proper reduction (Π', R) can be derived as follows:

1. $A' = A \backslash \{a_x\}$
2. Create T' from T in the following way: (a) Copy from T into T' all constraints T_j so that $\hat{a}_{jx} \neq a_x$ (b) Remove all literals $\bar{a}_x$ from T'
3. $R = \{a_x\}$ $\diamond$

A complementary observation can be derived for an essentially rejected variable. Let us note that the notions of essentially accepted and essentially rejected variables are known for both covering and satisfiability problems and are referred to as *essential rows* [13], or *unit clauses* [16]. In the following text the both types will be dealt together and referred to as *Essential Variables*.

Definition 5.4—Reducible Variable

Let $\Pi = (A, T, W)$ be an instance and $a_x \in A$. Variable a_x is Reducible iff for each constraint T_j in T it holds that $\hat{a}_{jx} = \bar{a}_x$ or $\hat{a}_{jx} = 0$. •

Observation 5.3

Let $\Pi = (A, T, W)$ be an instance and $a_x \in A$ a reducible variable. Then a proper reduction (Π', R) can be derived as follows:

1. $A' = A \backslash \{a_x\}$
2. Copy from T into T' all constraints T_j so that $\hat{a}_{jx} = 0$
3. $R = \emptyset$ $\diamond$

Let us note that a notion analogical to reducible variable is known for the satisfiability problem and is called a *pure literal* [16]. The adaptation of this notion for the covering problem is simple: only pure *inverted* literals can be applied.

Definition 5.5—Dominant Constraints

Let $\Pi = (A, T, W)$ be an instance and T_p and T_q any two distinct constraints in T. Constraint T_p is **dominated** by T_q iff for each two literals $\hat{a}_{pi}$ and $\hat{a}_{qi}$ it holds that $(\hat{a}_{pi} = \hat{a}_{pi}) \vee (\hat{a}_{pi} = 0)$. •

Observation 5.4

Let $\Pi = (A, T, W)$ be an instance and T_p, T_q any two distinct constraints in T so that T_p is dominated by T_q. Then a proper reduction (Π', R) can be derived as follows:

1. $A' = A$
2. Copy from T into T' all constraints except T_q
3. $R = \emptyset$ $\diamond$

A notion similar to this one is known and is referred to as *dominant rows* [13].

Definition 5.6—Resolvable Constraints

Let $\Pi = (A, T, W)$ be an instance and T_p and T_q any two distinct constraints in T. Constraints T_p and T_q are

resolvable iff there exists exactly one x so that $\hat{a}_{px} = a_x$ and $\hat{a}_{qx} = \bar{a}_x$, and for all the remaining values of $i = 1, 2, \ldots x-1, x+1, \ldots n$ it holds that $\hat{a}_{pi} = \hat{a}_{qi}$. Variable a_x is a resolving variable of T_p and T_q. •

Observation 5.5

Let $\Pi = (A, T, W)$ be an instance, and let T_p, T_q be any two resolvable constraints in T with a resolving variable a_x. Then a proper reduction (Π', R) can be derived as follows:

1. $A' = A$
2. Create T' from T in the following way:
   (a) Remove literal a_x from T_p
   (b) Copy from T into T' all constraints except T_q
3. $R = \emptyset$ ◇

Let us note that this observation is in fact a special case of the well-known *resolution principle* which is used for solving the satisfiability problem. However, this principle is used in the opposite direction: particular constraints are expanded using this rule until either a tautology, or a contradiction is found.

In [18] we present one more rule of proper decomposition based on occurrences of a pair of variables denoted *Mutually Reducible Variables*.

5.3.3 Complexity Analysis

In this subsection we will estimate the upper bound time complexity of evaluation the above-mentioned reduction rules. We will consider an instance containing m constraints and n variables.

The reduction rules based on the evaluation of a single variable in all constraints require $O(m)$ operations for evaluation of each variable. For for n variables this means time $O(mn)$. By a more sophisticated implementation, however, the time complexity of evaluation of the reducible variable can be reduced down to $O(n)$ operations. For each variable we must just hold the number of its direct and inverted occurrences in the system of constraints T.

The evaluation of all pairs of variables in all constraints requires time $O(mn^2)$, and the evaluation of all pairs of constraints time $O(m^2n)$.

We have not found any possibility to analytically evaluate the probability of successful application of the reduction rules from characteristics of particular instance.

5.4 Empirical Testing

The empirical testing has been performed in two steps. In the first step our aim was to identify those decomposition and reduction rules which can be successfully used for construction of an effective algorithm solving the covering problem.

In the second step we have removed those rules that in the first step showed to be inefficient and/or too time-consuming. Simultaneously, we have added an approximate reduction step and thus implemented the complete algorithm from Fig. 2. In this series of experiments we wanted to examine the influence of various ways of the approximate reduction. At the same time we wanted to know which rules are worth evaluating with respect to the execution time and quality of result.

5.4.1 Experimental Environment

We have used the random instance generator described in [18] for testing of the algorithm: we have generated square instances (number of constraints m equal to number of variables n) of the following sizes: 10, 20, 40, and 80. To get an idea of the influence of instance characteristics, we have generated instances with 0% (unate instances) and 10% of inverted literals.

The both types of instances were further divided according to costs of variables and according to quantity of literals in particular constraints. Costs of all variables were either equal to 1 (unweighted instances), or randomly generated with equal probability in interval 1 to 5 (weighted instances).

All the experiments were repeated for various numbers of literals in each constraint. These numbers were randomly generated with equal probability as follows:

- sparse instances (number of literals in each constraint 10 to 25% of n),
- medium instances (25 to 75% of n), and
- dense instances (75 to 90% of n).

Thus, for each size 12 sets of examples were generated, each of them consisting of 20 experiments. The total number of experiments was 960.

All the programs were written in C and the tests run on Sun SparcStation 2.

5.4.2 First Experiment

In this step our aim was to identify those pure and proper decomposition/reduction rules which can be successfully used for construction of an effective algorithm.

We have developed a program depicted in Fig. 4 for this purpose. Notice that this program represents in fact the first two steps of the algorithm shown in Fig. 2.

Various combinations of reduction rules presented earlier and the decomposition according to Theorem 5.4 have been incorporated into the program and tested. For each combination of rules the relative decrease in the

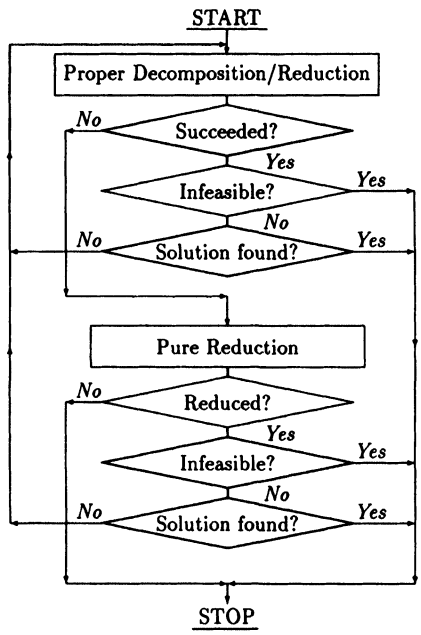

FIGURE 4 The algorithm for testing proper and pure decomposition and reduction rules of the covering problem.

instance size (see Definition 2.3) and the computational time have been evaluated. The complete set of results can be found in [18].

According to the results, only the following rules are worth further testing:

- Essential Variables (Observation 5.2),
- Reducible Variables (Observation 5.3), and
- Dominant and Resolvable Constraints (Observation 5.4 and 5.5).

The execution time of evaluation of the remaining reduction rules tested was considerably high with respect to the average decrease of the size of an instance (for detailed results see [18]).

On the other hand, the computational time of evaluation of the decomposition rule tested was short, but only four instances (out of the set of 960 examples) were successfully decomposed [18].

Let us note that the usefullness of application of Essential Variables is in accordance with results reported by other authors (see e.g. [13], [16]).

5.4.3 Second Experiment

In this step the algorithm as shown in Fig. 5 has been implemeted to determine:

- the effectivity of the remaining reduction rules, and
- the influence of approximate reduction (Observation 5.1) with respect to the search strategy.

Notice the similarity of Fig. 5 to the general scheme in Fig. 2.

To evaluate the effectivity of the tested proper reduction rules we have compared results of the following modifications of the algorithm (see Table 1):

- R: Proper reductions with only Essential Variables, approximate reductions according to randomly selected variables.
- R & RV: As R; added evaluation of Reducible Variables.
- R & CR: As R; added evaluation of Dominant and Resolvable Constraints.
- R & RV & CR: Combination of R & RV and R & CR.
- H: Approximate reductions guided by a heuristic function only.
- H & CR: As R; added evaluation of Dominant and Resolvable Constraints.

The modifications tested are not orthogonal, because the heuristic function selected (described with full details in [19] and [18]) detects the essential and reducible variables automatically. Since the usefullness of application of Essential Variables has been reported by other authors, we have not evaluated any algorithm without this rule.

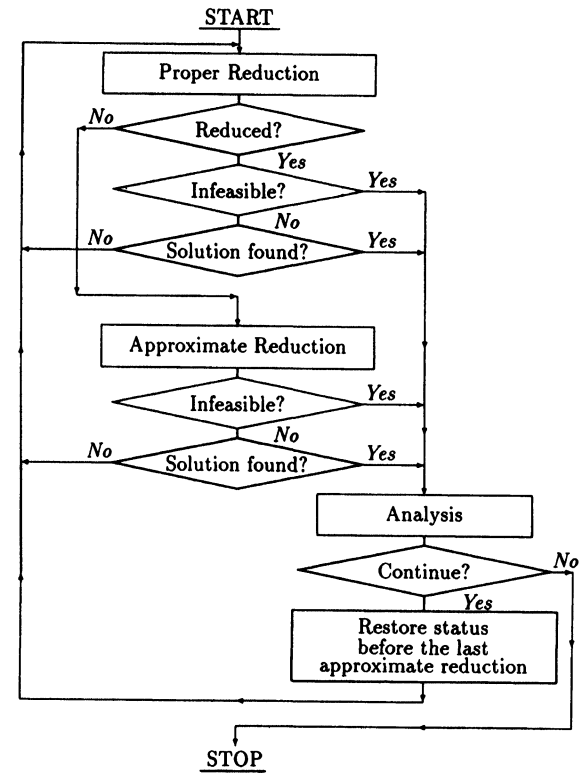

FIGURE 5 The optimized algorithm for the covering problem.

Results in Table 1 show the average quality ratio (AQR) and the runtime (Time) of the first solution found. The application of neither Reducible Variables, nor Dominant and Resolvable Constraints leads to improvement of the average quality ratio, but increases the computational time.

Thus, we cannot recommend the application of these rules. In further experiments we will deal with the R and H versions only.

The best average quality ratio is provided by the H algorithm, however, the computational time is higher than that of the R algorithm. This is why we have evaluated the convergence curves of each of these algorithms. The smoothed complete convergence curves for examples 40 * 40 are shown in Fig. 6. Table 2 shows one important point of the convergence curve: average quality of the result delivered by the R algorithm at the time when the H algorithm delivers the first solution.

We may conclude that the H algorithm provides better average quality ratio than the R algorithm even if the computational time is taken into consideration.

6. CONCLUSIONS

In this article decomposition and reduction have been discussed as general paradigms for solving set minimization problems. A hierarchy of types of decomposition and reduction has been proposed according to the ability to keep none, at least one, or all minimal solutions of the original instance. A general algorithm schema utilizing all kinds of decomposition and reduction has been presented and discussed.

The second part of this article contains a case-study demonstrating the above-mentioned general approach with the covering problem. Several reduction/decomposition rules and search strategies have been

TABLE 1
Comparison of various proper and pure reduction rules 0% Inverted Literals, Unweighted Instances

Size	R		R & RV		R & CR		R & RV & CR		H		H & CR	
	AQR	Time	AQR	Time	AQR	Time	AQR	Time	AQR	Time	AQR	Time
10 * 10	1.484	0.000	1.484	0.001	1.484	0.001	1.484	0.001	1.018	0.001	1.018	0.002
20 * 20	1.515	0.001	1.518	0.003	1.518	0.010	1.518	0.012	1.028	0.004	1.028	0.006
40 * 40	1.597	0.007	1.597	0.015	1.588	0.078	1.588	0.085	1.040	0.022	1.040	0.090
80 * 80	1.677	0.069	1.677	0.125	1.677	0.403	1.677	0.457	1.006	0.141	1.006	0.341
	0% Inverted Literals, Weighted Instances											
Size	R		R & RV		R & CR		R & RV & CR		H		H & CR	
	AQR	Time	AQR	Time	AQR	Time	AQR	Time	AQR	Time	AQR	Time
10 * 10	1.526	0.000	1.526	0.001	1.530	0.001	1.530	0.001	1.056	0.001	1.056	0.002
20 * 20	1.994	0.001	1.994	0.004	1.978	0.009	1.978	0.010	1.066	0.005	1.066	0.005
40 * 40	2.342	0.008	2.342	0.017	2.342	0.072	2.342	0.078	1.051	0.023	1.053	0.081
80 * 80	2.994	0.069	2.994	0.108	2.994	0.319	2.994	0.346	1.065	0.148	1.065	0.345
	10% Inverted Literals, Unweighted Instances											
Size	R		R & RV		R & CR		R & RV & CR		H		H & CR	
	AQR	Time	AQR	Time	AQR	Time	AQR	Time	AQR	Time	AQR	Time
10 * 10	1.456	0.000	1.456	0.000	1.456	0.001	1.456	0.001	1.089	0.001	1.089	0.001
20 * 20	1.540	0.001	1.540	0.002	1.540	0.005	1.540	0.007	1.229	0.003	1.229	0.003
40 * 40	1.733	0.005	1.733	0.009	1.733	0.033	1.733	0.037	1.390	0.021	1.390	0.040
80 * 80	1.523	0.037	1.523	0.050	1.523	0.219	1.523	0.230	1.604	0.086	1.604	0.164
	10% Inverted Literals, Unweighted Instances											
Size	R		R & RV		R & CR		R & RV & CR		H		H & CR	
	AQR	Time	AQR	Time	AQR	Time	AQR	Time	AQR	Time	AQR	Time
10 * 10	1.846	0.001	1.846	0.000	1.808	0.001	1.808	0.001	1.168	0.001	1.168	0.000
20 * 20	2.254	0.002	2.254	0.002	2.196	0.005	2.196	0.006	1.220	0.003	1.220	0.002
40 * 40	2.155	0.006	2.155	0.008	2.155	0.031	2.155	0.032	1.407	0.023	1.407	0.045
80 * 80	2.216	0.034	2.216	0.048	2.216	0.232	2.216	0.243	1.659	0.087	1.659	0.170

R ... Random approximate reductions
H ... Heuristically guided approximate reductions
RV ... Reducible Variables evaluated
CR ... Constraint Reductions (Dominant and Resolvable Constraints) evaluated
AQR ... Average Quality Ratio of the first solution
Time ... Run time necessary to find the first solution (sec CPU time, Sun SparcStation 2)

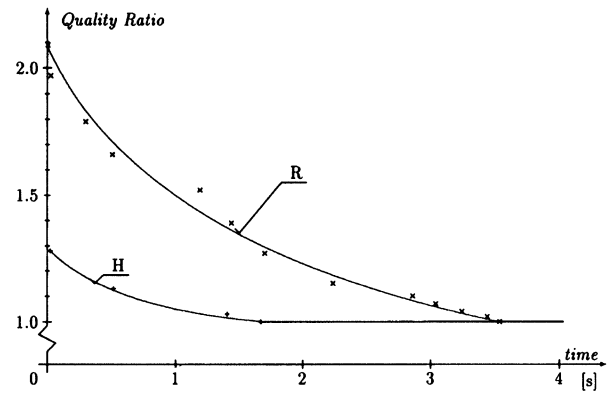

FIGURE 6 Evaluation of convergence of the tested algorithms for the covering problem.

presented and tested. The empirical testing of the efficiency of particular rules and strategies allows us to propose an optimized version of the general algorithm for the covering problem [19], [18].

TABLE 2
Average quality ratio of the R algorithm running the same time as the H algorithm

0% Inverted Literals, Unweighted Instances		
Size	R	H
10 * 10	1.484	1.018
20 * 20	1.515	1.028
40 * 40	1.539	1.040
80 * 80	1.647	1.006
0% Inverted Literals, Weighted Instances		
Size	R	H
10 * 10	1.526	1.056
20 * 20	1.994	1.066
40 * 40	2.126	1.051
80 * 80	2.560	1.065
10% Inverted Literals, Unweighted Instances		
Size	R	H
10 * 10	1.456	1.089
20 * 20	1.540	1.229
40 * 40	1.585	1.390
80 * 80	1.501	1.604
10% Inverted Literals, Weighted Instances		
Size	R	H
10 * 10	1.846	1.168
20 * 20	2.254	1.220
40 * 40	1.861	1.407
80 * 80	1.808	1.659

References

[1] Aspvall, B., Plass, M., Tarjan, R.: A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Inform. Process. Lett.*, vol. 8, 1979, pp. 121–123.

[2] Brayton, R.K., Somenzi, F.: An Exact Minimizer for Boolean Relations. *Proc. ICCAD-89*, Santa Clara, 1989, pp. 316–319.

[3] Cook, S.A.: The Complexity of Theorem Proving Procedures. *3rd STOC*, 1971, pp. 151–158.

[4] Damiani, M., DeMicheli, G.: Synthesis and Optimization of Logic Circuits from Recurrence Equations. *Proc. EDAC*, Brussels, 1992, pp. 226–231.

[5] Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. *JACM*, 7, 1960, pp. 201–215.

[6] Grasseli, A., Luccio, F.: A Method for Minimizing the Number of Internal States in Incompletely Specified Sequential Networks. *IEEE Trans. on Electronic Computers*, vol. X, June 1965, pp. 350–359.

[7] Hachtel, G.D., Rho, J.-K., Somenzi, F., Jacoby, R.: Exact and Heuristic Algorithms for the Minimization of Incompletely Specified State Machines. *Proc. EDAC*, Amsterdam 1991, pp. 184–191.

[8] Larabee, T.: Test Pattern Generation Using Boolean Satisfiability. *IEEE Trans. on CAD*, vol. 11, January 1992, pp. 4–15.

[9] Lengauer, T.: Combinatorial Algorithms for Integrated Circuit Layout. *Wiley-Teubner*, Stuttgart-New York, 1990.

[10] McCluskey, E.J.: Introduction to the Theory of Switching Circuits. *McGraw-Hill*, New York, 1965.

[11] Pearl, J.: Heuristics: Intelligent Search Strategies for Computer Problem Solving. *Addison-Wesley Publishing Co.*, Reading, Mass., 1984.

[12] Petrick, S.R.: A Direct Determination of Irredundant Forms of a Boolean Function from the Set of Prime Implicants. *Techn. Rept. No AFCRC-TR-56-110, AF Cambridge Research Center*, Bedford, Mass., 1956.

[13] Pipponzi, M., Somenzi, F.: An Iterative Algorithm for the Binate Covering Problem. *Proc. EDAC*, Glasgow 1990, pp. 208–211.

[14] Reeves, C.R. (ed.): Modern Heuristic Techniques for Combinatorial Problems. *Blackwell*, Oxford, 1993.

[15] Sangiovanni-Vincentelli, A.L., El Gamal, A., Rose, J.: Synthesis Methods for Field Programmable Gate Arrays. *Proc. IEEE*, vol. 81. no. 7, July 1993, pp. 1057–1083.

[16] Selman, B., Levesque, H., Mitchell, D.: GSAT: A New Method for Solving Hard Satisfiability Problems. *Technical Report*, AT&T Bell Laboratories, 1992.

[17] Servít, M.: A Heuristic Method for Solving Weighted Set Covering Problems. *Digital Processes*, vol. X, 1975, pp. 177–182.

[18] Servít, M., Zamazal, J.: Covering Problem, *Final Report DC-93-04*, Department of Computers, Faculty of Electrical Engineering, CTU Prague, March 1993.

[19] Servít, M., Zamazal, J.: Heuristic Approach to Binate Covering Problem. *Proc. EDAC*, Brussels 1992, pp 123–129.

Biographies

MICHAL SERVÍT received his Ing. (M. S.) degree in Electronics and CSc. (Ph. D.) degree in Computer Science from the Czech Technical University, Prague, in 1966 and 1987 respectively. After some time in industry he returned to his Alma Mater in 1971. He is currently a Docent (Associate Professor) and a Deputy Head of the Department of Computers, Czech Technical University in Prague. His research interests include CAD in electronics, VLSI design automation, algorithm theory, graph theory and automata theory. He authored more than 70 papers in thos research fields and he served as programme/organizing committee member or chairman at various conferences focused on VLSI design.

JAN ZAMAZAL received his Ing. (M. S. in Computing) degree from the Czech Technical University, Prague, in 1988. Since 1989 he has been a full time Ph. D. student at the Department of Computers, CTU Prague, in 1992 he became Assistant Professor. His research interests include CAD tools, VLSI design and algorithms for CAD tools.