A Correct-by-Decision Solution for Simultaneous Place and Route

Alexander Nadel

Intel Corporation, P.O. Box 1659, Haifa 31015, Israel
alexander.nadel@intel.com

Abstract. To reduce a problem, provided in a human language, to constraint solving, one normally maps it to a set of constraints, written in the language of a suitable logic. This paper highlights a different paradigm, in which the original problem is converted into a set of constraints and a decision strategy, where the decision strategy is essential for guaranteeing the correctness of the modeling. We name such a paradigm Correct-by-Decision. Furthermore, we propose a Correct-by-Decision-based solution within a SAT solving framework for a critical industrial problem that shows up in the physical design stage of the CAD process: simultaneous place and route under arbitrary constraints (design rules). We demonstrate the usefulness of our approach experimentally on industrial and crafted instances.

1 Introduction

Nowadays, constraint solvers are widely applied to solve a rich variety of problems. Normally, reducing a problem, provided in a human language, to constraint solving involves the following steps:

1. Choosing the logic and the constraint solver most suitable for the problem at hand
2. Reducing the problem to a set of constraints in the chosen logic, querying the solver, and mapping its results back to the original problem.

We call this generally accepted paradigm of reducing a problem to constraint solving *Correct-by-Constraint Paradigm (*`CBC`*)*.

Recently, a different paradigm has been identified and applied [8, 12]. After choosing the logic and the solver, the problem is reduced to a set of constraints and a *decision strategy* for the solver. Both the constraints and the decision strategy are essential for guaranteeing correctness. We call such a paradigm the *Correct-by-Decision Paradigm (*`CBD`*)*. The difference between `CBC` and `CBD` paradigms is that in `CBD` the decision strategy is essential for guaranteeing the correctness of the modeling (in other words, reverting to the default decision strategy of the solver renders the modeling incorrect).

In [8, 12], `CBD`-based solutions within a SAT solving framework were shown to outperform `CBC`-based solutions by four orders of magnitude for two industrial applications that show up in the physical design stage of Computer-Aided-Design (CAD): clock routing [8] and routing under constraints (`RUC`) [12]. Several factors were responsible for such an effect. First, the number of propositional clauses was drastically reduced. This is because many of the constraints were not required, their role being fulfilled by the decision strategy. In particular, the number of clauses was reduced by two orders of magnitude for clock routing [8]. Second, applying a problem-aware decision strategy helped make convergence substantially faster (a similar effect was observed in earlier works [3, 18]). In addition, the decision strategy proved to be essential in meeting problem-specific optimization targets, e.g., reducing the routing cost in [12].

Interestingly, the widely applied scheme of incremental SAT solving under assumptions [7, 13, 2, 14] can be thought of as an early unintentional application of `CBD`. Incremental SAT solving under assumptions is used when the initial problem can be solved with a series of SAT invocations over an incrementally constructed SAT instance, each time under different assumptions. It was found that instead of creating the SAT solver instance from scratch and using unit clauses to model assumptions, it is more effective to keep the solver instance alive and model the invocation-specific unit clauses as *assumptions*, that is, the first decisions taken by the solver. Such an approach falls into `CBD` category, since dropping the assumptions would render the approach unsound. Other examples of early unintentional `CBD` applications include [6] and [10], where a custom decision strategy was applied for efficient array reasoning and implicative simultaneous satisfiability, respectively.

The goal of this paper is twofold:

1. *To highlight the usefulness of* `CBD` *and provide more insight into it.* While previous works [8, 12] identified that applying a decision strategy to simulate constraints is useful for their specific problems, we highlight `CBD` as a stand-alone paradigm. We wish to raise the awareness about `CBD` among the community, so that applying `CBD` for solving existing and new problems might become an explicit option to be considered by researchers.

2. *To propose a* `CBD`*-based solution to the critical industrial problem of Simultaneous Place and Route under Constraints (*`PRUC`*).* Both placement and routing are sub-stages in the physical design stage of CAD. Devices are placed in the placement stage and then connected (routed) in the routing stage (see [19, 1] for a survey of the currently used approaches to both placement and routing). The eventual routing solution has to meet design rules which specify restrictions originating in manufacturing requirements. Currently placement and routing are carried out separately. The separation simplifies the physical design process significantly, but can very negatively impact execution time and solution quality. This is because the placer may come up with a solution that cannot be routed at all or cannot be routed cleanly w.r.t the design rules. In such a case additional place-and-route iterations are carried out, slowing down the process. Moreover, convergence is still not guaranteed. We propose a formal modeling and a `CBD`-based so-

lution for the problem of *simultaneous* place and route under arbitrary bit-vector constraints that can be applied to model simultaneous place and route under arbitrary design rules for integrated circuits [19] and printed circuit boards [1]. The advantages of simultaneous place and route were realized in previous work [15, 16], where ad hoc place and route solutions were applied to two specific types of FPGA designs. [12] proposed a `CBD`-based solution for the routing stage only.

The rest of the paper is built as follows.

Sect. 2 sketches a `CBD`-based solution for the problem of Path-Finding under Constraints (`PFUC`) [12, 4], that is, the problem of finding a path between two given vertices in a graph in the presence of constraints. Sect. 2 is important for two reasons. First, in it we illustrate the usefulness of `CBD` on a relatively simple application. Second, the `PFUC` solution that it sketches is a sub-component in both the `RUC` [12] and our proposed `PRUC` solutions.

Sect. 3 briefly reviews [12]'s approach to solving the `RUC` problem. This provides the background for our formulation and solution for the `PRUC` problem, which is introduced in Sect. 4. Sect. 5 provides the experimental results, and Sect. 6 concludes our paper.

We assume that the reader is familiar with the basics of bit-vector solving and SAT solving. See [9] for a recent overview.

2 Path-Finding under Constraints

Consider the problem of *path-finding*, that is, finding a path from the source $s \in V$ to the target $t \in V$ in an undirected non-negatively weighted simple graph $G = (V, E)$.

Path-finding can be solved using the Dijkstra algorithm. Alternatively, one can reduce path-finding to propositional satisfiability as shown in Fig. 1. The formula in Fig. 1 can easily be translated to Conjunctive Normal Form (CNF), that is, a set of propositional clauses*, and solved with a SAT solver. The active edges in the model returned by the solver (that is, the edges whose activity variables are assigned the value 1) comprise the sought-for path from s to t. Note that the neighbor constraints guarantee that s is connected to t.

Solving path-finding with a SAT solver is overkill since it can be solved with a polynomial algorithm, however SAT may be useful if the solution should satisfy additional user-given constraints. Consider the problem of Path-Finding under Constraints (`PFUC`) [12] defined as follows.

In `PFUC`, the input comprises a graph $G = (V, E)$, a source $s \in V$, a target $t \in V$, and an arbitrary bit-vector formula $F(E \cup V \cup A)$, where:

1. E are Boolean variables representing the edge activity
2. V are Boolean variables representing the vertex activity

* Neighbor constraints are cardinality constraints, which can easily be translated to CNF (see [5] for a review).

1. Create a Boolean variable for every vertex and edge representing its *activity*, where a vertex/edge is active iff it appears on the solution path from s to t.
2. Create *edge consistency* constraints: for each edge $e = (v, u)$, create the following constraint: $e \implies v \wedge u$ (i.e., both vertices of an active edge are active).
3. Create *neighbor constraints*: ensure that each vertex v has exactly n active neighbor edges, where:
   (a) n=0 if the vertex is inactive
   (b) n=1 if v is the source or the target
   (c) n=2 if v is an active *internal* vertex (that is, neither the source nor the target)

Fig. 1: Modeling path-finding with constraints

3. A is a set of arbitrary auxiliary Boolean and bit-vector variables

Given a model for F, α, let $E^\alpha \subseteq E$ be the subset of edges assigned to 1 in α and $V(E^\alpha) \subseteq V$ be the subset of the vertices touched by E^α. We call the graph $G(V(E^\alpha), E^\alpha)$ the *solution graph* induced by α.

In PFUC, the output should be either a model α for F, such that there exists a path from s to t in the solution graph, or UNSAT, if no such model exists. If the problem is satisfiable, a desirable optimization requirement is to decrease the overall cost of the active edges.

PFUC can be solved by applying a SAT solver using the following two sets of clauses as input: those generated by path-finding encoding in Fig. 1 and the input bit-vector formula F translated to clauses. The problem is that such a CBC-based solution does not scale [12]. This is because neither the decision heuristic nor the conflict analysis components of the SAT solver are aware of the high-level problem at hand. Below we sketch a CBD-based solution that is based on the solution provided in [12]. It had previously been hinted at in [4].

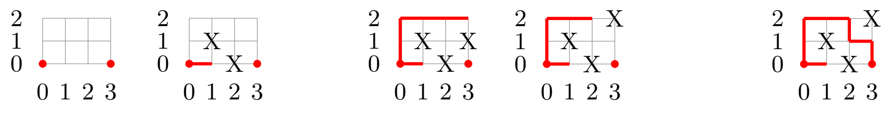
First, activity variables and edge consistency constraints (entries 1 and 2 in Fig. 1) are created. The neighbor constraints (entry 3 in Fig. 1) are no longer required. Instead, the decision strategy will guarantee that there exists a (short) path from s to t in the solution graph.

Second, SAT solver's decision strategy is replaced by a strategy that builds an explicit walk from s to t using an incremental shortest-path algorithm, such as the Ramalingam-Reps algorithm [17] in [4] or the lazy A*-based flow in [12]. The new decision strategy strategy constructs such a walk in a stack $\pi = \left\{ s \xrightarrow{e_1} v_1 \dots v_{k-1} \xrightarrow{e_k} l \right\}$. At the beginning, π contains the source s only. The algorithm will extend π following the suggestions of the shortest-path algorithm, where the shortest-path algorithm is allowed to use active and unassigned edges only. More specifically, assume that the solver has to take a decision; assume also that e is the next edge suggested by the shortest-path algorithm to be picked to connect the latest π's vertex l to t. If e is unassigned, the algorithm will push e to the back of π and activate e (by taking the decision which assigns e's activity variable the value 1). If e has already been assigned 1 by Boolean Constraint Propagation (BCP), e is pushed to the back of π, and the shortest-path algorithm is queried for the next edge. After the SAT solver backtracks, the algorithm pops any unassigned edges from the back of π. After the walk from s to t is completed, the solver deactivates any unassigned edges to reduce the solution cost and then reverts to the default decision heuristic.

Third, the conflict analysis of the solver is extended as follows. Whenever there is no longer any path from the latest π's vertex l to t (because some vertices were deactivated as a result of propagation in F), the solver records a new clause that comprises the negation of the *conflict cut*–a subset of the inactive vertices that blocks any path from l (and s) to t.

Fig. 2 illustrates the algorithm. Consider the initial position in Fig. 2a. The algorithm starts by extending π rightward from $s = (0,0)$ towards $t = (3,0)$ by activating the edge $(0,0)-(1,0)$. After that decision, two vertices become inactive as a result of Boolean Constraint Propagation as shown in Fig. 2b. The solver has to backtrack to $(0,0)$ and continue the walk towards t. As shown in Fig. 2c, when the solver reaches the vertex $(3,2)$ there is a conflict as no path from $(3,2)$ to t exists. The conflict cut comprises the following set $\{(2,0),(3,1)\}$. Note that when the conflict cut's vertices are inactive, any path from s to t is blocked. The solver records the conflict cut as a clause, which triggers the SAT solver's conflict analysis and backtracking engines. The situation after backtracking is shown in Fig. 2d. Extending the walk from that point on to t is straightforward. The eventual solution is shown in Fig. 2e.

After a solution is found, a simple post-processing algorithm can be applied to eliminate any cycles in the solution graph [12]. In our example this would have resulted in the active edge $(0,0)-(1,0)$ being deactivated.



(a) Start (b) Can't go right (c) A conflict (d) After backtracking (e) Done

Fig. 2: Solving `PFUC`: trace example. Each intersection of the grid lines comprises a vertex. Assume that $s = (0,0)$ and $t = (3,0)$ and that the following CNF formula is provided: $\neg(1,0) \vee \neg(2,0)$, $\neg(1,0) \vee \neg(1,1)$, $\neg(3,2) \vee \neg(3,1)$. Bold red edges correspond to π. "X" marks inactive vertices.

A `CBD`-based solution works well for the `PFUC` problem, since `PFUC` is comprised of two sub-problems, one better suited to a heuristic approach (finding a short path) and the other to constraint solving (solving the user-given constraints).

3 Routing under Constraints

This section sketches [12]'s formulation and `CBD`-based solution to the problem of Routing under Constraints (`RUC`), which is required as background to our `PRUC` solution.

Consider first the following *routing* problem. Let $G = (V, E)$ be a non-negatively weighted simple graph. Let $N_{i\in\{0\ldots m-1\}} \subseteq V$ be m pairwise disjoint non-empty subsets of G's vertices, called the *nets*, where the vertices of each net are called the *terminals*. A *routing* (comprising a solution for the routing problem) is a forest of *net routings* $E_{i\in\{0\ldots m-1\}} \subseteq E$, such that all N_i's terminals are connected in N_i's net routing and all the net routings are pairwise vertex-disjoint. See an input example in Fig. 4a and an example of a solution in Fig. 4b. In practice, solutions minimizing the overall routing cost are preferred. To solve the routing problem, heuristic approaches are commonly applied – see [1] for a survey.

Consider now the `RUC` problem. In `RUC`, the input comprises an instance of the routing problem (that is, a graph $G = (V, E)$ and nets $N_{i\in\{0\ldots m-1\}} \subseteq V$) and, in addition, a bit-vector formula $F(E \cup V \cup N \cup A)$, where the variables have the following semantics:

1. E are Boolean variables representing the edge activity
2. V are Boolean variables representing the vertex activity
3. N are bit-vector variables defined as follows: for every vertex v, a bit-vector variable $0 \leq nid(v) \in N < m$ of width $\lceil log_2 m - 1\rceil$ represents the unique net id of active vertices, where the *net id* of net N_i is the index i
4. A is a set of arbitrary auxiliary Boolean and bit-vector variables

Let R be an `RUC` instance and α be an assignment to $E \cup V \cup N \cup A$, then α satisfies R if and only if all the conditions in Fig. 3 hold.

1. *Routing correctness*: any two terminals of the same net N_i are connected in the solution graph $G(V(E^\alpha), E^\alpha)$ and any two terminals of two different nets are disconnected in $G(V(E^\alpha), E^\alpha)$
2. *Net boundary consistency*: for each vertex $v \in V : 0 \leq nid(v) < m$
3. *Terminal consistency*: for each terminal $t \in N_i : nid(t) = i$
4. *Net edge consistency*: for each edge $e = (v, u)$, $e \implies v \wedge u \wedge (nid(v) = nid(u))$ (i.e., both vertices of an active edge are active and they share the net id)

Fig. 3: `RUC` consistency

We now sketch [12]'s `CBD`-based solution for `RUC`.

Recall that any `CBD`-based solution is comprised of constraints and a decision strategy. In our case, the constraints part is composed of net boundary consistency, terminal consistency, and net edge consistency shown in Fig. 3. The routing correctness is ensured by the decision strategy.

The basic idea behind the decision strategy is to route the nets one by one, where, within each net, the terminals are routed one by one. Specifically, assume that the terminal t_i of net $N_i = \left\{t_0, t_1, \ldots, t_{|N_i|-1}\right\}$ is to be routed. Let the current *net vertices* be the set $V(N_i)$ of vertices, connected to the already routed terminals $\{t_0, \ldots, t_{i-1}\}$ by active edges. The algorithm will connect t_i to the net vertices using the `PFUC` algorithm of Sect. 2 (with some extensions to conflict analysis, discussed below).

Consider, for example, the problem in Fig.4a (no constraints are provided). Routing the blue net N_1, followed by the red net N_0 yields the solution in Fig. 4b. Assume the solver picked the other net ordering, i.e., that it started routing the red net N_0 first. That would result in the net conflict shown in Fig. 4c, where a *net conflict* is a situation in which a *conflicting net* N_c (N_1 in our example) is blocked by other nets and inactive vertices.

In case of a net conflict, let the *conflict cut* be a set of inactive vertices and vertices of net id *nid(v)* $\neq c$ which block the path from the current terminal of the conflicting net N_c to $V(N_c)$. In our example, the conflict cut is $\{(0,2), (1,2), (2,2), (3,2), (4,2)\}$. The solver will add a new clause, which disallows the conflict cut. The clause will contain the inactive vertices in the cut, and, in addition, any active vertex v of net id *nid(v)* $\neq c$ in the cut will contribute to the clause one bit on which the values of *nid(v)* and c differ. In our example, the net id comprises a single bit, and the clause will look as follows: $\{$*nid(0,2)*, *nid(1,2)*, *nid(2,2)*, *nid(3,2)*, *nid(4,2)*$\}$. Adding the clause will trigger the solver's conflict analysis and backtracking engines. In our case, it will block the red path shown in Fig. 4c. The solver will keep constructing and blocking red paths until a red path going above the blue terminal $(2,3)$, and thus not blocking the blue net N_1, will be found.

Our example demonstrates that the order in which the nets are routed is critical. To speed up the algorithm, two net reordering algorithms were proposed in [12]: net swapping and net restarting. Both techniques are applied when a conflicting net N_c is blocked more times than a certain user-given threshold. Net swapping swaps the order between N_c and the latest net N_i blocking N_c. It then backtracks to the latest point where N_c is unassigned and starts routing N_c. Net restarting moves N_c to the top of the net list and carries out a full restart. The algorithm will start routing N_c right after the restart. The combination of these techniques (where the net restarting threshold is higher than the net swapping threshold) has been shown to be extremely efficient [12]. Note that applying either one of the net reordering techniques would have solved the example instance in Fig. 4a after the conflict shown in Fig. 4c without any further conflicts.

4 Simultaneous Place and Route under Constraints

As we have mentioned, the routing stage follows the placement stage in the CAD process. More specifically, placement lays out the user-given devices on a grid, and routing connects them. Below, for simplicity of presentation, we assume

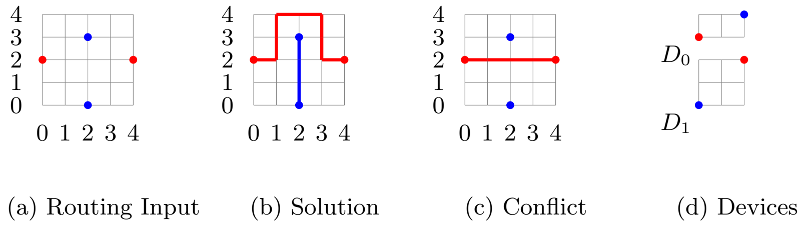


Fig. 4: A routing example on a 5×5 solid grid graph, given two nets of two terminals each $N_0 = \{(0,2),(4,2)\}$ and $N_1 = \{(2,0),(2,3)\}$. It also serves as a place & route example, given two devices $D_0 = \{(2,1)_1,(0,0)_0\}$ and $D_1 = \{(2,2)_0,(0,0)_1\}$. Assume the edges' weights are 1.

that the grid is 2-dimensional; our algorithms, however, are equally applicable to 3-dimensional grids.

The input to the Place and Route under Constraints (`PRUC`) problem contains the following components:

1. A rectangle $R = \{R_x, R_y\}$, serving as the grid.
2. A graph $G(V,E)$, whose each vertex $v_i \equiv (v_i^x \in [0, R_x - 1], v_i^y \in [0, R_y - 1]) \in V$ is a point in R. We do not apply any restrictions on the edges (normally, the edges are induced by the specific connectivity model of the input design).
3. A non-empty set of devices, where a *device* $D_i = \left\{L_i^0, \ldots, L_i^{|D_i|-1}\right\}$ is a set of leaves, each *leaf* $L_i^j = (x_i^j, y_i^j)_{n_i^j}$ containing the leaf's *relative coordinates* (x_i^j, y_i^j) w.r.t to the device's root and the leaf's *net id* n_i^j. See the definition of D_0 and D_1 in Fig. 4's caption and their illustration in Fig. 4d for an example. An optional *placement rectangle* $R_i \subseteq R$, where the device's root is to be placed, is provided for each device. R_i defaults to R.
4. A bit-vector formula $F(E \cup V \cup N \cup A)$ (the semantics of the variables being similar to those presented in Sect. 3 in the context of `RUC`).

Informally, given a satisfiable `PRUC` instance, a `PRUC` solver should return a placement for all the devices (where a vertex cannot be occupied by more than one terminal) and a solution for the `RUC` instance, induced by the placement. We need some more notations to formalize `PRUC`'s output.

Given a device D_i, the vertex $r_i \equiv (r_i[x], r_i[y]) \in V$ is a *potential root*, if, for each leaf $L_i^j = (x_i^j, y_i^j)_{n_i^j}$, the leaf's *terminal* $t(L_i^j, r_i) \equiv (r_i[x] + x_i^j, r_i[y] + y_i^j)$, given r_i, lies within R_i. The set of all potential roots of D_i is denoted by *roots(D_i)*.

The example device D_1 in Fig. 4 has the following potential roots, given a 5×5 solid grid graph:
roots(D_1) $= \{(0,0),(1,0),(2,0),(0,1),(1,1),(2,1),(0,2),(1,2),(2,2)\}$.

Placing a device D_i at an *actual root* vertex $r_i \equiv (r_i[x], r_i[y]) \in V$, means creating a terminal $t(L_i^j, r_i) \equiv (r_i[x] + x_i^j, r_i[y] + y_i^j)$ for each leaf $L_i^j = (x_i^j, y_i^j)_{n_i^j}$.

It must hold that the terminal $t(L_i^j, r_i)$ is active (i.e., the activity variable of the terminal's vertex is turned on) and that its net id matches that of the leaf (that is, $nid(t(L_i^j, r_i)) = n_i^j$). In addition, each vertex can serve as a terminal for at most one leaf (in other words, at most one device can occupy a vertex).

For example, the routing instance in Fig. 4a could have been created by placing the two devices in Fig. 4d on a 5×5 solid grid graph as follows: D_0 is placed at $(0, 2)$ (creating the red terminal (0,2) and the blue terminal $(2, 3)$), while D_1 is placed at $(2, 0)$ (creating the remaining two terminals).

Formally, a satisfying assignment to a PRUC instance comprises:

1. A placement for each device D_i (that is, each D_i is mapped to an actual root).
2. A solution for the RUC instance, comprising $G(V, E)$, the formula $F(E \cup V \cup N \cup A)$, and nets $N_{k\in\{0\ldots m-1\}}$, where each net N_k is comprised of the terminals of all the leaves with net id k.

As in the case of PFUC and RUC, a desirable optimization requirement is to decrease the overall cost of the active edges. A PRUC should return UNSAT, given an unsatisfiable instance.

Note that RUC is a special case of PRUC, where all the devices are fixed (that is, their placement rectangle is fixed to one particular location). It was shown in [12] that RUC cannot be efficiently solved with either a CBC-based encoding or with the graph-aware solver Monosat [4]. Even more so, neither can PRUC be efficiently solved by these means. Below we introduce our CBD-based PRUC solution, which comprises a set of constraints (Sect. 4.1), a decision strategy (Sect. 4.2), and performance optimization heuristics (Sect. 4.3).

4.1 Constraints

For each device D_i and each potential root $r_i \in roots(D_i)$, we create a Boolean *potential root engagement* variable $e(r_i)$. Intuitively, $e(r_i)$ is 1 iff D_i is placed at r_i. Now we are ready to present the constraints:

1. Net boundary consistency and net edge consistency constraints are inherited from the RUC solution in Fig. 3.
2. *Placement consistency*: for each device D_i, exactly one of its potential root engagement variables holds.
3. *Leaf consistency*: for each device D_i and each potential root engagement variable $e(r_i)$: if $e(r_i)$ holds, then, for each leaf $L_i^j = (x_i^j, y_i^j)_{n_i^j} \in D_i$, the terminal vertex $t(L_i^j, r_i)$ is active, and it holds that $nid(t(L_i^j, r_i)) = n_i^j$.
4. *Placement uniqueness*: For every vertex v, if the vertex can serve as a terminal for more than one leaf, then at most one of the relevant potential root engagement variables (that is, the potential root engagement variables for roots, which, when used to place a device D_i, render v a terminal) holds. In our example in Fig. 4, the vertex $(4, 2)$ can serve as a terminal for the leaves $L_0^0 \equiv (2, 1)_1$ (if D_0 is placed at $(2, 1)$) and $L_1^0 \equiv (2, 2)_0$ (if D_1 is placed at $(2, 0)$). Hence, the following cardinality constraint is created for the vertex $(4, 2)$: at-most-1($e(r_0 \equiv (2, 1))$, $e(r_1 \equiv (2, 0))$).

Note that the high-level constraints are expressed in terms of standard bit-vector operators and cardinality constraints. Thus, the constraints can easily be translated to propositional clauses [9, 5].

4.2 Decision Strategy

Our basic decision strategy goes as follows. For each device D_i, in the user-given order, we place the device at some potential root r_i (by turning on r_i's potential root engagement variable) and then route the device. To route a device D_i we proceed as follows: for every leaf $L_i^j = (x_i^j, y_i^j)_{n_i^j} \in D_i$ in the user-given order, we connect the newly created terminal $t(L_i^j, r_i)$ to n_i^j's net vertices $V(n_i^j)$ (net vertices are all the vertices connected to the already routed terminals of net n_i^j by active edges) using the PFUC algorithm, explained in Sect. 2 (with some modifications to net conflict analysis, explained below).

Placement Heuristic We introduce our placement heuristic (that is, the heuristic the decision strategy uses for picking the actual root for the device) after providing some additional notations.

Assume the decision strategy is about to place a device D_i. We call a potential root $r_i \in$ *roots(D_i) available*, if its engagement variable $e(r_i)$ is unassigned or assigned 1[⋆]. For a device D_i, let *D_i's net id's* $N(D_i)$ be the set of the nets associated with D_i's leaves. A net N_h is *fresh* if N_h does not belong to $N(D_j)$ for any of the already placed devices D_j. D_i is *fresh* if all of D_i's nets are fresh (in other words, a device D_i is fresh if and only if none of net ids of the already placed terminals belongs to $N(D_i)$).

Assume we need to place a fresh device D_i. In this case we place the device as close to the center of the grid R as possible. More specifically, we strive to minimize the overall cost of the shortest paths in G from the terminals, created by placing the device, to the center[⋆⋆]. The reason we chose this heuristic is because the closer the device is to the border of the grid the more difficult it is to route (since the borders restrict the routing options). More specifically, our algorithm works as follows. We run Breadth-First-Search (BFS) starting at the center of the grid and working outwards towards its borders. We pick the first available root $r_i \in$ *roots(D_i)*, such that all D_i's terminals, given r_i, were visited by BFS. Note that the placement consistency constraints ensure that one of the roots must be available. Placement uniqueness constraints ensure that there exists a location, not occupied by another device.

Now assume we are about to place a non-fresh device D_i. We place D_i so as to minimize the overall cost of the paths from the terminals, generated by placing

[⋆] Let r_i be a potential root of a device D_i that is about to be placed by the decision strategy. The engagement variable $e(r_i)$ can already be assigned 1 at this point if Boolean Constraint Propagation (BCP) has been able to conclude that r_i is the only potential root of D_i where the device can be placed.

[⋆⋆] The center is the vertex $(R_x/2, R_y/2)$, if available, or, otherwise, the vertex whose Manhattan distance from $(R_x/2, R_y/2)$ is as small as possible.

D_i, to the net vertices of the non-fresh nets. Such a heuristic is useful both for increasing the odds that the placement will be routable and for decreasing the overall routing cost to meet the optimization requirement. More specifically, our algorithm works as follows. For every vertex v and non-fresh net N_h, we calculate the cost $c(v, N_h)$ of the shortest path from v to N_h's net vertices $V(N_h)$. This can be done by running BFS starting from $V(N_h)$ for each non-fresh net N_h. For every v, let $c(v)$ be the sum of the costs $c(v, N_h)$ across all the non-fresh nets. We place D_i at the root $r_i \in$ *roots(D_i)* which minimizes the sum of the $c(v)$'s for all D_i's terminals, given r_i. To further improve efficiency, one can run the BFS searches which start at the non-fresh nets in parallel and halt once the first available root is reached.

Net Conflict Analysis Our PRUC algorithm applies the PFUC algorithm, presented in Sect. 2, to connect the current terminal t of net N_h (created by placing leaf $L_i^j = (x_i^j, y_i^j)_{h \equiv n_i^j}$ of the currently routed device D_i) to $V(N_h)$. Assume that PFUC encounters a net conflict, that is, that all the paths from t to $V(N_h)$ are blocked by the conflict cut, which comprises inactive vertices and vertices of net id *nid(v)* $\neq h$. Our proposed conflict analysis algorithm is based on the conflict analysis algorithm of the RUC solution, described in Sect. 3.

Recall that the RUC solution records a conflict clause C comprising the inactive vertices in the cut, and that any active vertex v of net id *nid(v)* $\neq h$ in the cut contributes to the clause one bit on which the values of *nid(v)* and h differ. Note that, unlike in the case of RUC, where the terminals are static, the terminals are created by placing devices for PRUC. For the net conflict to occur, two dynamically created terminals of net N_h must be separated by the conflict cut. One such terminal is created by placing the currently routed device D_i, while the other terminal can belong to any previously placed device whose set of nets includes N_h. Hence, we must augment the conflict clause C with two additional literals:

1. The negation of the actual root's engagement variable $e(r_i)$ of the currently routed device D_i, and
2. The negation of the actual root's engagement variable $e(r_w)$ of one of the previously placed devices D_w, whose set of nets includes N_h.

Example Consider Fig. 4. Assume the algorithm is given a 5×5 grid and the devices D_0 and D_1 (in that order). The algorithm will start by placing the device D_0 as close as possible to the center. The minimal distance to the center is 3 for several potential roots, $(0, 2)$ included. Assume $(0, 2)$ is picked as the actual root, so D_0 is placed as shown in Fig. 4a. The set of net vertices is empty for both nets before placing D_0, so no routing is required for the newly created terminals. The algorithm will continue by placing D_1. The minimal combined distance to the net vertices of both nets is 7 for several roots, including $(2, 0)$. Assume the algorithm picks the root $(2, 0)$ for placing D_1. Fig. 4a reflects the situation after the placement. Next, the algorithm will route the red terminal $(4, 2)$ (of the leaf

$L_1^0 \equiv (2,2)_0$) to connect it to $V(N_0) = \{(0,2)\}$. The situation after completing this routing is shown in Fig. 4c.

The algorithm will then attempt to connect the blue terminal $(2,0)$ (of the leaf $L_1^1 \equiv (0,0)_1$) to $V(N_1) = \{(2,3)\}$. It will immediately encounter a net conflict (since the path from $(2,0)$ to $(2,3)$ is blocked by the red net N_0), and record a conflict clause. The clause will include the net id variables' bits $\{\mathit{nid(0,2)}, \mathit{nid(1,2)}, \mathit{nid(2,2)}, \mathit{nid(3,2)}, \mathit{nid(4,2)}\}$ and the negation of the actual root engagement variables for both devices, namely, $\neg e(r_0 \equiv (0,2))$ and $\neg e(r_1 \equiv (2,0))$.

The algorithm will then keep routing the terminals generated by the first placement it chose, and will, eventually, succeed. More specifically, the solver will keep constructing and blocking red paths until a red path going above the blue terminal $(2,3)$ (and thus not blocking the blue net N_1) is found (exactly as in the `RUC` case).

The basic `PRUC` algorithm presented so far is functional but inefficient. As our example demonstrates, the algorithm will preserve the current device ordering, leaf ordering and the initial placement picked by the algorithm. Such an approach is not sufficiently dynamic.

4.3 Performance Optimization Heuristics

We propose three performance optimization heuristics to improve the efficiency of the algorithm. As we shall see, each of the proposed techniques solves the example we discussed at the end of Sect. 4.2 right after the first conflict and without any further conflicts.

For the currently routed device D_i, let its *root decision level* be the decision level of its actual root engagement variable.

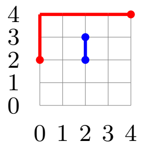
Consider each technique separately for now. We will discuss how to combine the heuristics a bit later.

Leaf Reordering *Leaf reordering* is about reordering the leaves of the currently routed device D_i when routing the terminal of a certain leaf L_i^j. Leaf reordering is applied after encountering a user-given number of conflicts, initialized with 0 each time the algorithm starts routing the device. The technique simply moves L_i^j to the beginning of the leaf list of D_i, backtracks to D_i's root decision level, and starts routing D_i in the new order.

Consider the conflict shown in Fig. 4c and discussed at the end of Sect. 4.2. Applying leaf reordering at that point would switch the order of D_1's leaves. The algorithm would then route the blue terminal $(2,0)$ first, followed by the red terminal $(4,2)$. That would yield the solution in Fig. 4b without further conflicts.

Device Replacement *Device replacement* is about trying out different placements for the currently routed device, rather than sticking to one particular placement. Device replacement works as follows: when a certain conflict threshold is reached, after the algorithm starts routing the currently routed device D_i, the algorithm backtracks to D_i's root decision level and places D_i at the *next* available root w.r.t to the current ordering: that is, either the distance from the center, if D_i is fresh, or, otherwise, the overall distance to the net vertices of the non-fresh nets of the device. If no next available root is available, replacement cycles back to the *first* root in the current ordering.

In our example, the device replacement algorithm could have moved device D_1 to several possible locations equidistant from the existing nets, $(2, 2)$ being one of them. Assume that D_1 is placed at $(2, 2)$. The following solution would then be generated without any further conflicts:



Device Reordering *Device reordering* is about reordering the devices right after a certain conflict threshold is encountered after the algorithm starts routing the currently routed device D_i. The technique moves D_i to the top of the list of devices, carries out a full restart (that is, backtracks to decision level 0) and starts placing and routing in the new order, i.e., D_i will be the first one to be placed and routed.

In our example, switching the order of the devices, placing D_1 first followed by D_0, will still result in the placement in Fig. 4a. However, the algorithm will route without any further conflicts, since the blue leaf comes first for device D_0; hence the blue net N_1 will be routed first, followed by the red net N_0, yielding the solution in Fig. 4b.

Combining Performance Optimization Heuristics The three heuristics differ w.r.t to their locality.

Leaf reordering is the most local of the three, since it is applied given one particular device and one concrete placement. Device replacement is more global, since it can change the actual placement of the current device. Device reordering is the most global of the three techniques, since it changes the global device ordering, and also requires a full restart.

To combine the three heuristics, one can try them all out starting with the more local leaf reordering, followed by the more global device replacement, followed by the yet more global device reordering. To this end, the leaf reordering conflict threshold should be the smallest, followed by the device replacement threshold, and finally by the device reordering threshold.

For the combination to work, conflict counting towards the next device replacement and the next device reordering must *not* be restarted when leaf reordering occurs for the currently routed device. Likewise, conflict counting towards the next device reordering must not be restarted when device replacement occurs.

5 Experimental Results

Our experiments on industrial and crafted instances are described below. We used machines with 32Gb of memory running Intel® Xeon® processors with 3Ghz CPU frequency. The timeout was set to 20 minutes for all the experiments.

5.1 Industrial

We experimented with 48 clips of Intel's designs. Each such clip is currently solved by a proprietary industrial place and route flow that applies placement first followed by routing. On average it takes about 24 hours to solve one clip with 60 to 85 invocations of the place and route flow. Multiple invocations are needed since the placer sometimes fails to generate a routable routing instance.

In our experiments, we aim to demonstrate the potential usefulness of applying our algorithms in industrial settings. Another goal is to study the impact of applying and combining the three performance optimization heuristics presented in Sect. 4.3: leaf reordering, device replacement, and device reordering. We generated 48 different configurations by combining the following conflict thresholds for the three heuristics, where ∞ means that the technique is not applied at all:

```
1: for all $lro \in \{3, 10, 25, \infty\}$ do
2:     for all $drp \in \{3, 10, 25, 100, \infty\}$ do
3:         for all $dro \in \{3, 10, 25, 100, 1000, \infty\}$ do
4:             if ($lro = \infty$ or ($lro < drp$ and $lro < dro$)) and ($drp = \infty$ or $drp < dro$) then
5:                 Generate a configuration $(lro, drp, dro)$ with leaf reordering threshold $lro$, device replacing threshold $drp$, and device reordering threshold $dro$.
```

Note that based on the conclusions of Sect. 4.3 we let the leaf reordering conflict threshold always be the smallest one, followed by the device replacement threshold, and then by the device reordering threshold.

Table 1 shows the results for the best 12 configurations in terms of solved instances within the time-out of 20 minutes (the 'Time in sec.' column shows the overall run-time, where the time-out value of 1200 sec. is added for unsolved instances).

The best performing configuration, solving 44 / 48 instances within 20 minutes, is $(\infty, 3, 25)$. Hence, combining frequent device replacement with frequent device reordering yields the best results. Leaf reordering, on the other hand, does not contribute, based on these results. The configuration (∞, ∞, ∞) (that is, none of the three performance optimization heuristics is applied) does not appear in the table, since it solved only 14 instances.

Additional analysis revealed that the 4 instances that remained unsolved by the best configuration $(\infty, 3, 25)$ are solved by at least one of the following 3 configurations: $(\infty, 10, 100)$, $(\infty, 10, 25)$, and $(10, 25, 100)$. Interestingly, one of the instances is solved solely by two configurations which apply leaf reordering– $(10, 25, 100)$ and $(10, \infty, 25)$. This result hints that although leaf reordering was not found to be useful overall, it can still contribute to solving some instances.

The new CBD-based tool is as good as the existing industrial solution in terms of quality. The average wire length is almost identical; the difference is 0.6% (the new approach being slightly better). This is ultimately because both the existing heuristical solution and the new CBD-based tool are based on shortest-path algorithms. In addition, physical design experts have confirmed that the quality of the new tool is as good as that of the existing solution.

lro	*drp*	*dro*	Solved	Time in sec.
∞	3	25	44	16823
∞	∞	10	40	18641
∞	10	25	40	16463
∞	∞	25	40	16880
3	∞	10	39	19593
10	∞	25	39	18068
∞	3	10	38	17763
3	∞	25	38	16675
∞	10	100	36	24422
∞	∞	3	36	22623
10	25	100	35	24346
∞	3	100	35	23824

Table 1: Best Configs for Industrial

lro	*drp*	*dro*	Solved	Time in sec.
∞	3	10	30	1914
∞	3	25	30	8932
∞	∞	3	30	16046
3	10	25	29	10683
∞	3	100	29	12503
∞	∞	10	29	15459
3	10	100	28	19598
∞	10	25	27	15008
3	∞	25	22	17558
3	∞	10	20	16734
∞	25	100	19	24116
10	∞	25	18	27598

Table 2: Best Configs for Crafted

5.2 Crafted

This section analyzes the performance of our algorithms on PRUC instances we crafted. All the benchmarks and detailed results are publicly available at [11]. We pursued two goals:

1. To generate challenging yet solvable publicly available PRUC instances to encourage further PRUC research, and
2. To further analyze the performance of leaf reordering, device replacement, and device reordering.

The instances were generated as follows.

First, we used a 100×100 solid grid graph. It was shown in [12] that reducing RUC (being PRUC's special case) to either bit-vector reasoning or Monosat [4] solver input does not scale to grids of such a size, even on simple instances.

Second, we used 5×5 devices with 4 leaves, each leaf using random relative coordinates (within the rectangle $(0,0)$–$(4,4)$) and a random net.

Third, according to our preliminary experiments, it made sense to use 10 devices and 40 nets for each instance in order to create instances which are challenging enough yet not too difficult to solve.

Fourth, each instance was augmented with $(C/100)*|V|$ binary clauses, where $|V| = 10000$ and C is a parameter. The clauses are generated as follows: pick a random vertex $v = (x, y)$ and another random vertex u sharing either the x or y coordinate with v, and add the clause $\neg v \vee \neg u$.

All in all, we generated 30 random instances as follows:

```
1: for all $C \in \{0, 15, 30\}$ do
2:     for all $i \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ do
3:         Generate a PRUC instance on a $100 \times 100$ grid with 10 $5 \times 5$ devices
           with 4 leaves each, each leaf using random relative coordinates within the
           rectangle $(0, 0)$–$(4, 4)$ and a random net with net id in the interval $[0 - 39]$.
           In addition, generate $(C/100) * 10000$ random binary clauses as described
           above.
```

We used the same 48 configurations we had experimented with in Sect. 5.1. Consider the results in Table 2, showing the best 12 configurations in terms of the number of instances solved. The absolutely best configuration, which solved all the instances 4.5 times faster than the next best configuration, is $(\infty, 3, 10)$. Hence, combining frequent device replacement with frequent device reordering yields the best results for crafted instances, similarly to our experiments with industrial instances, with an even smaller device reordering conflict threshold. Leaf reordering does not seem to be helpful for the crafted instances. The configuration (∞, ∞, ∞), which applies none of the three performance optimization techniques, solved none of the instances.

6 Conclusion

This paper highlights a novel paradigm for reducing a problem to constraint solving, which we called Correct-by-Decision (`CBD`). In `CBD`, the problem is reduced to a set of constraints and a decision strategy, where the decision strategy is essential for guaranteeing correctness. We saw that `CBD` is useful when the problem is composed of two interleaved sub-problems, one of which has an easy heuristical solution, while the other requires solving a set of constraints.

Furthermore, we proposed a `CBD`-based solution to a critical industrial problem that shows up in the physical design stage of the CAD process: simultaneous place and route under arbitrary constraints (design rules), which we called `PRUC`. We demonstrated that our approach can successfully cope with industrial instances and analyzed the performance of different heuristics we proposed on these instances. We also crafted challenging publicly available `PRUC` instances and studied the performance of our algorithms on these instances.

References

1. Nadine Abboud, Martin Grötschel, and Thorsten Koch. Mathematical methods for physical layout of printed circuit boards: an overview. *OR Spectrum*, 30(3):453–468, 2008.
2. Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In

Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 309–317. Springer, 2013.

3. Clark Barrett and Jacob Donham. Combining SAT methods with non-clausal decision heuristics. *Electr. Notes Theor. Comput. Sci.*, 125(3):3–12, 2005.
4. Sam Bayless, Noah Bayless, Holger H. Hoos, and Alan J. Hu. SAT modulo monotonic theories. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 3702–3709. AAAI Press, 2015.
5. Armin Biere, Daniel Le Berre, Emmanuel Lonca, and Norbert Manthey. Detecting cardinality constraints in CNF. In Sinz and Egly [20], pages 285–301.
6. Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. *JSAT*, 6(1-3):165–201, 2009.
7. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.
8. Amit Erez and Alexander Nadel. Finding bounded path in graph using SMT for automatic clock routing. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 20–36. Springer, 2015.
9. Liana Hadarean. *An Efficient and Trustworthy Theory Solver for Bit-vectors in Satisfiability Modulo Theories.* Dissertation, New York University, 2015.
10. Zurab Khasidashvili and Alexander Nadel. Implicative simultaneous satisfiability and applications. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, volume 7261 of *Lecture Notes in Computer Science*, pages 66–79. Springer, 2011.
11. Alexander Nadel. A correct-by-decision solution for simultaneous place and route: Benchmarks and detailed results. https://goo.gl/MNl1PE.
12. Alexander Nadel. Routing under constraints. In Ruzica Piskac and Muralidhar Talupur, editors, *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 125–132. IEEE, 2016.
13. Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In Alessandro Cimatti and Roberto Sebastiani, editors, *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 242–255. Springer, 2012.
14. Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Ultimately incremental SAT. In Sinz and Egly [20], pages 206–218.
15. Sudip Nag and Rob A. Rutenbar. Performance-driven simultaneous place and route for row-based fpgas. In *DAC*, pages 301–307, 1994.
16. Sudip K. Nag and Rob A. Rutenbar. Performance-driven simultaneous place and route for island-style fpgas. In Richard L. Rudell, editor, *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1995, San Jose, California, USA, November 5-9, 1995*, pages 332–338. IEEE Computer Society / ACM, 1995.
17. G. Ramalingam and Thomas W. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.
18. Ashish Sabharwal. Symchaff: A structure-aware satisfiability solver. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings, The Twentieth National*

Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA, pages 467–474. AAAI Press / The MIT Press, 2005.

19. Naveed A. Sherwani. *Algorithms for VLSI physical design automation.* Kluwer, 3 edition, November 1998.
20. Carsten Sinz and Uwe Egly, editors. *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*. Springer, 2014.