Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

TOBIAS ACHTERBERG[1]
TIMO BERTHOLD[2]
STEFAN HEINZ[2]
THORSTEN KOCH[2]
KATI WOLTER[2]

Constraint Integer Programming: Techniques and Applications

[1] ILOG Deutschland, Ober-Eschbacher Str. 109, 61352 Bad Homburg, Germany, tachterberg@ilog.de
[2] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany, {berthold,heinz,koch,wolter}@zib.de

ZIB-Report 08-43 (October 2008)

Constraint Integer Programming: Techniques and Applications

Tobias Achterberg   Timo Berthold*   Stefan Heinz*
Thorsten Koch   Kati Wolter**

October 31, 2008

Abstract

This article introduces *constraint integer programming* (CIP), which is a novel way to combine constraint programming (CP) and mixed integer programming (MIP) methodologies. CIP is a generalization of MIP that supports the notion of general constraints as in CP. This approach is supported by the CIP framework SCIP, which also integrates techniques for solving satisfiability problems. SCIP is available in source code and free for noncommercial use.

We demonstrate the usefulness of CIP on three tasks. First, we apply the constraint integer programming approach to pure mixed integer programs. Computational experiments show that SCIP is almost competitive to current state-of-the-art commercial MIP solvers. Second, we demonstrate how to use CIP techniques to compute the number of optimal solutions of integer programs. Third, we employ the CIP framework to solve chip design verification problems, which involve some highly nonlinear constraint types that are very hard to handle by pure MIP solvers. The CIP approach is very effective here: it can apply the full sophisticated MIP machinery to the linear part of the problem, while dealing with the nonlinear constraints by employing constraint programming techniques.

1 Introduction

In the recent years, several authors showed that an integrated approach of constraint programming (CP) and mixed integer programming (MIP) can help to solve optimization problems that were intractable with either of the two methods alone [21, 36, 56]. Different approaches to integrate CP and MIP into a single framework have been proposed, [7, 11, 20, 33, 51, 52] amongst others.

Most of the existing work followed the concept of extending a CP framework by basic MIP techniques. In contrast, this paper introduces a way to incorporate CP specific solving methods and its strong modeling capability into the sophisticated MIP solving machinery.

A previous version of this paper can be found in [3].

* Supported by the DFG Research Center MATHEON *Mathematics for key technologies* in Berlin.

** Supported by the DFG Priority Program 1307 "Algorithm Engineering".

This is achieved by a very low-level integration of the two concepts. The constraints of a CP usually interact through the domains of the variables. As in [11, 20, 51, 52], the idea of *constraint integer programming* (CIP) is to offer a second communication interface, namely the *linear programming (LP) relaxation.* Furthermore, the definition of CIP restricts the generality of CP modeling as little as needed to still gain the full power of all primal and dual MIP solving techniques.

Therefore, CIP is well suited for problems that contain a MIP core complemented by some nonlinear constraints. As an example for such a problem type, the property checking problem is presented in Section 6.

The concept of constraint integer programming is realized in the branch-and-cut framework SCIP. It combines solving techniques from CP, MIP, and the field of solving satisfiability problems (SAT) such that all involved algorithms operate on a single search tree, which yields a very close interaction. A detailed description of the concepts and the software can be found in [2].

The plugins that are provided with the standard distribution of SCIP suffice to turn the CIP framework into a full-fledged MIP solver. In combination with either SoPlex [58] or CLP [25] as LP solver, it is currently one of the fastest noncommercial MIP solvers, see [45] and the results in Section 4. Using Cplex [34] as LP solver, the performance of SCIP is even comparable to state-of-the-art commercial codes.

As a library, SCIP can be used to develop branch-cut-and-price algorithms, and it can be extended to support additional classes of nonlinear constraints by providing so-called constraint handler plugins. We present a solver for the chip design verification problem as one example of this usage.

SCIP is freely available in source code for academic and noncommercial use and can be downloaded from `http://scip.zib.de`. The current version 1.1.0—as of this writing—has interfaces to five different LP solvers and consists of 275 640 lines of C code. The code is actively maintained and extended.

The article is organized as follows: in Section 2, we introduce constraint integer programs. Section 3 presents the building blocks of the constraint integer programming framework SCIP. In Sections 4–6, we demonstrate the usage of SCIP on three applications. First, we employ SCIP as a stand-alone MIP solver. Second, we count optimal solutions with SCIP. Third, we use SCIP as a branch-and-cut framework to solve chip design verification problems. Computational results are given in Sections 4, 5 and 6.4.

2 Constraint Integer Programming

The hope of integrating CP, MIP, and SAT techniques is to combine their advantages and to compensate for their individual weaknesses. A constraint program is defined as follows.

Definition (constraint program)**.** *A* constraint program *is a triple* $CP = (\mathfrak{C}, \mathfrak{D}, f)$ *with* $\mathfrak{D} = \mathcal{D}_1 \times \ldots \times \mathcal{D}_n$ *representing the domains of finitely many variables* $x_j \in \mathcal{D}_j$, $j = 1, \ldots, n$, $\mathfrak{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ *being a finite set of constraints* $\mathcal{C}_i : \mathfrak{D} \to \{0, 1\}$, $i = 1, \ldots, m$, *and* $f : \mathfrak{D} \to \mathbb{R}$ *being the objective function. It consists of solving*

$$(CP) \quad f^\star = \min\{f(x) \mid x \in \mathfrak{D},\ \mathfrak{C}(x)\},$$

with $\mathfrak{C}(x) :\Leftrightarrow \forall i = 1, \dots, m : \mathcal{C}_i(x) = 1$*. A CP where all domains* $\mathcal{D} \in \mathfrak{D}$ *are finite is called a* finite domain constraint program *(CP(FD)).*

Note that, with a slight abuse of notation, we use the abbreviation CP for the term constraint programming as well as for the term constraint program. The same holds for MIP and CIP. In case the meaning is not clear from context we use the long versions.

To solve a CP(FD), the problem is recursively split into smaller subproblems, thereby creating a branching tree and implicitly enumerating all potential solutions. At each subproblem, domain propagation is performed to exclude further values from the variables' domains.

Due to the very general definition of a CP, solvers have to rely on constraint propagators, each of them exploiting the structure of a single constraint class. Usually, the only communication between the individual constraints takes place via the variables' domains. An advantage of CP is, however, the possibility to model the problem more directly than in MIP, using very expressive constraints, which maintain the structure of the problem. In MIP, we are restricted to linear constraints, a linear objective function, and integer or real-valued domains. A mixed integer program is defined as follows.

Definition (mixed integer program)**.** *Given a matrix* $A \in \mathbb{R}^{m \times n}$*, vectors* $b \in \mathbb{R}^m$*, and* $c \in \mathbb{R}^n$*, and a subset* $I \subseteq N = \{1, \dots, n\}$*, the corresponding* mixed integer program $MIP = (A, b, c, I)$ *is to solve*

$$(MIP) \quad c^\star = \min \{c^T x \mid Ax \leq b,\ x \in \mathbb{R}^n,\ x_j \in \mathbb{Z} \text{ for all } j \in I\}.$$

Note, that MIPs in maximization form can be transformed to minimization form by multiplying the objective function vector by -1. Similarly, "$\geq$" constraints can be multiplied by -1 to obtain "$\leq$" constraints. Equations can be replaced by two opposite inequalities.

Like CP solvers, most modern MIP solvers recursively split the problem into smaller subproblems. However, the processing of the subproblems is different. Because MIP includes only one type of constraints, MIP solvers can apply sophisticated techniques that operate on the subproblem as a whole. Usually, for each subproblem, the LP relaxation is solved, which is constructed from the MIP by removing the integrality conditions. The LP relaxation can be strengthened by cutting planes which use the LP information and the integrality restrictions to derive valid linear inequalities that cut off the solution of the current LP relaxation without removing feasible MIP solutions. The LP relaxation usually gives a much stronger bound than the one that is provided by simple dual propagation of CP solvers. Solving the LP relaxation usually requires much more time, however.

Satisfiability problems is also a very specific case of CPs with only one type of constraints, namely Boolean clauses. The Boolean truth values *false* and *true* are identified with the values 0 and 1, respectively, and Boolean formulas are evaluated correspondingly.

Definition (satisfiability problem)**.** *Let* $\mathfrak{C} = \mathcal{C}_1 \wedge \dots \wedge \mathcal{C}_m$ *be a logic formula in conjunctive normal form on Boolean variables* $x_1, \dots, x_n$*. Each clause* $\mathcal{C}_i = \ell_1^i \vee \dots \vee \ell_{k_i}^i$ *is a disjunction of literals. A literal* $\ell \in L = \{x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n\}$ *is either a variable* x_j *or the negation of a variable* $\bar{x}_j$*. The task of the* satisfiability problem *(SAT) is to either find an assignment* $x^\star \in \{0,1\}^n$*, such that the formula* $\mathfrak{C}$ *is satisfied, i.e., each clause* $\mathcal{C}_i$ *evaluates to* 1*, or to conclude that* $\mathfrak{C}$ *is unsatisfiable, i.e., for all* $x \in \{0,1\}^n$ *at least one* $\mathcal{C}_i$ *evaluates to* 0.

Modern SAT solvers also use a branching scheme to split the problem into smaller subproblems and they apply *Boolean constraint propagation* on the subproblems, which is a special form of domain propagation. In addition, they analyze infeasible subproblems to produce *conflict clauses.* These help to prune the search tree later on. Furthermore, SAT solvers support periodic restarts of the search in order to revise the branching decisions after having gained new knowledge about the structure of the problem instance.

Boolean clauses can easily be linearized, but the LP relaxation is rather useless, as it cannot detect the infeasibility of subproblems earlier than domain propagation. Therefore, SAT solvers mainly exploit the special problem structure to speed up the domain propagation algorithm.

To specify our approach of integrating CP, MIP, and SAT solving techniques, we propose the following slight restriction of CP, which allows the application of MIP solving techniques:

Definition (constraint integer program)**.** *A constraint integer program* $\mathit{CIP} = (\mathfrak{C}, I, c)$ *consists of solving*

$$(\mathit{CIP}) \quad c^\star = \min\{c^T x \mid \mathfrak{C}(x),\ x \in \mathbb{R}^n,\ x_j \in \mathbb{Z} \text{ for all } j \in I\}$$

with a finite set $\mathfrak{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ *of constraints* $\mathcal{C}_i : \mathbb{R}^n \to \{0,1\}$*,* $i = 1, \ldots, m$*, a subset* $I \subseteq N = \{1, \ldots, n\}$ *of the variable index set, and an objective function vector* $c \in \mathbb{R}^n$*. A CIP has to fulfill the following additional condition:*

$$\forall \hat{x}_I \in \mathbb{Z}^I\ \exists (A', b') : \{x_C \in \mathbb{R}^C \mid \mathfrak{C}(\hat{x}_I, x_C)\} = \{x_C \in \mathbb{R}^C \mid A' x_C \le b'\} \qquad (1)$$

with $C := N \setminus I$*,* $A' \in \mathbb{R}^{k \times C}$*, and* $b' \in \mathbb{R}^k$ *for some* $k \in \mathbb{Z}_{\ge 0}$.

Restriction (1) ensures that the remaining subproblem after fixing all integer variables always is a linear program. This means that in the case of finite domain integer variables, the problem can be—in principle—completely solved by enumerating all values of the integer variables and then solving the remaining LPs.

Note, that this does not forbid quadratic or even more involved expressions. Only the remaining part after fixing (and thus eliminating) the integer variables must be linear in the continuous variables. Furthermore, the linearity restriction of the objective function can be compensated by introducing an auxiliary objective variable z that is linked to the actual nonlinear objective function with a constraint $z = f(x)$. Analogously, general variable domains can be represented as additional constraints.

Therefore, every CP that meets Condition (1) can be represented as a CIP. Especially, the following proposition holds.

Proposition. *The notion of constraint integer programming generalizes finite domain constraint programming and mixed integer programming:*

(a) Every CP(FD) can be modeled as a CIP.

(b) Every MIP can be modeled as CIP.

Proof. The notion of a constraint is the same in CP as in CIP. The linear system $Ax \leq b$ of a MIP is a conjunction of linear constraints, each of which is a special case of the general constraint notion in CP and CIP. Therefore, we only have to verify Condition (1).

For a CP(FD), each variables x_j, $j = 1, \dots, n$ has a finite domain and can therefore be equivalently represented as integers. The only noninteger variable is the auxiliary objective variable z, i.e., $x_C = (z)$. Therefore, Condition (1) can be satisfied for a given $\hat{x}_I$ by setting

$$A' := \begin{pmatrix} 1 \\ -1 \end{pmatrix} \quad \text{and} \quad b' := \begin{pmatrix} f(\hat{x}_I) \\ -f(\hat{x}_I) \end{pmatrix}.$$

For a MIP, partition the constraint matrix $A = (A_I, A_C)$ into the columns corresponding to the integer variables I and to the continuous variables C. For a given $\hat{x}_I \in \mathbb{Z}^I$ set $A' := A_C$ and $b' := b - A_I \hat{x}_I$ to meet Condition (1). □

3 A Framework to Solve Constraint Integer Programs

SCIP (Solving Constraint Integer Programs) is a framework for constraint integer programming. It is based on the branch-and-bound procedure, which is a very general and widely used method to solve discrete optimization problems.

The idea of *branching* is to successively divide the given problem instance into smaller subproblems until the individual subproblems are easy to solve. The best of all solutions found in the subproblems yields the global optimum. During the course of the algorithm, a *branching tree* is created with each node representing one of the subproblems.

The intention of *bounding* is to avoid a complete enumeration of all potential solutions of the initial problem, which usually are exponentially many. If a subproblem's lower (dual) bound is greater than or equal to the global upper (primal) bound, the subproblem can be pruned. Lower bounds are calculated with the help of a relaxation which should be easy to solve. Upper bounds are found if the solution of the relaxation is also feasible for the corresponding subproblem.

Good lower and upper bounds must be available for the bounding to be effective. In order to improve a subproblem's lower bound, one can tighten its relaxation, e.g., via domain propagation or by adding cutting planes (see Sections 3.2 and 3.4, respectively). Primal heuristics, which are described in Section 3.5, contribute to the upper bound.

The selection of the next subproblem in the search tree and the branching decision have a major impact on how early good primal solutions can be found and how fast the lower bounds of the subproblems increase. More details on branching and node selection are given in Section 3.6.

SCIP provides all necessary infrastructure to implement branch-and-bound based algorithms for solving constraint integer programs. It manages the branching tree along with all subproblem data, automatically updates the LP relaxation, and handles all necessary transformations due to presolving problem modifications, see Section 3.7. Additionally, a cut pool, cut filtering, and a SAT-like conflict analysis mechanism, see Section 3.3, are available. Capabilities to handle symmetries, see Section 3.8, can be realized via specific constraint handlers.

Furthermore, SCIP provides its own memory management and plenty of statistical output.

Besides the infrastructure, all main algorithms are implemented as external plugins. In the remainder of this section, we will describe the key ingredients of a branch-and-bound based CIP solver and discuss their role for the solving process.

3.1 Constraint Handlers

Since a CIP consists of constraints, the central objects of SCIP are the *constraint handlers*. Each constraint handler represents the semantic of a single class of constraints and provides algorithms to handle constraints of the corresponding type. The primary task of a constraint handler is to check a given solution for feasibility with respect to all constraints of its type existing in the problem instance. This feasibility test suffices to provide an algorithm which correctly solves CIPs with constraints of the supported types. To improve the performance of the solving process, constraint handlers may provide additional algorithms and information about their constraints to the framework, besides others

- presolving methods to simplify the problem's representation,
- propagation methods to tighten the variables' domains,
- a linear relaxation, which can be generated in advance or on the fly, that strengthens the LP relaxation of the problem, and
- branching decisions to split the problem into smaller subproblems, using structural knowledge of the constraints in order to generate a well-balanced branching tree.

The standard distribution of SCIP already includes a constraint handler for linear constraints that is needed to solve MIPs. Additionally, some specializations of linear constraints like knapsack, set partitioning, or variable bound constraints are supported by constraint handlers, which can exploit the special structure of these constraints in order to obtain more efficient data structures and algorithms. Furthermore, SCIP provides constraint handlers for logical constraints, such as AND, OR, and XOR constraints and for nonlinear constraints, like SOS1, SOS2, and indicator constraints.

3.2 Domain Propagation

Constraint propagation is an essential part of every CP solver [10]. The task is to analyze the set of constraints of the current subproblem and the current domains of the variables in order to infer additional valid constraints and domain reductions, thereby restricting the search space. The special case where only the domains of the variables are affected by the propagation process is called *domain propagation*. If the propagation only tightens the lower and upper bounds of the domains without introducing holes it is called *bound propagation*.

In mixed integer programming, the concept of bound propagation is well-known under the term *node preprocessing*. Usually, MIP solvers apply restricted versions of the preprocessing algorithms, that are used before starting the branch-and-bound process, to simplify the subproblems [30, 53].

Besides the integrality restrictions, there are only linear constraints in a MIP. In contrast, CP models can include a large variety of constraint classes with different semantics and structures. Thus, a CP solver usually provides specialized constraint propagation algorithms for every single constraint class.

Constraint based (primal) domain propagation is supported by the constraint handler concept of SCIP. In addition, it features two dual domain reduction methods that are driven by the objective function, namely the *objective propagation* and the *root reduced cost strengthening* [46].

3.3 Conflict Analysis

Most MIP solvers discard infeasible and bound-exceeding subproblems without paying further attention to them. Modern SAT solvers, in contrast, try to learn from infeasible subproblems, which is an idea due to Marques-Silva and Sakallah [44]. The infeasibilities are analyzed in order to generate so-called *conflict clauses*. These are implied clauses that help to prune the search tree. They also enable the solver to apply so-called *nonchronological backtracking*. A similar idea in CP are *no-goods*, see [54].

Conflict analysis can be generalized to CIP and, as a special case, to MIP. There are two main differences of CIP and SAT solving in the context of conflict analysis. First, the variables of a CIP do not need to be of binary type. Therefore, we have to extend the concept of the conflict graph: it has to represent bound changes instead of variable fixings, see [1] for details.

Furthermore, the infeasibility of a subproblem in the CIP search tree usually has its reason in the LP relaxation of the subproblem. In this case, there is no single conflict-detecting constraint as in SAT or CP solving. To cope with this situation, we have to analyze the LP in order to identify a subset of all bound changes that suffices to render the LP infeasible or bound-exceeding. Note that it is an $\mathcal{NP}$-hard problem to identify a subset of the local bounds of *minimal cardinality* such that the LP stays infeasible if all other local bounds are removed. Therefore, we use a greedy heuristic approach based on an unbounded ray of the dual LP, see [1].

After having analyzed the LP, the algorithm works in the same fashion as conflict analysis for SAT instances: it constructs a conflict graph, chooses a cut in this graph, and produces a conflict constraint which consists of the bound changes along the frontier of this cut.

3.4 Cutting Plane Separators

Besides splitting the current subproblem into two or more easier subproblems by branching, one can also try to tighten the subproblem's relaxation in order to rule out the current LP solution $\check{x}$ and to obtain a different one. The LP relaxation can be tightened by introducing additional linear constraints $a^T x \leq b$ that are violated by $\check{x}$ but do not cut off feasible solutions of the subproblem. Thus, the current solution $\check{x}$ is *separated* from the convex hull of the feasible solutions of the subproblem by the *cutting plane* $a^T x \leq b$.

The theory of cutting planes is very well covered in the literature. For an overview of computationally useful cutting plane techniques, see [30, 42]. A recent survey of cutting plane literature can be found in [39].

SCIP features separators for knapsack cover cuts [12], complemented mixed integer rounding cuts [41], Gomory mixed integer cuts [32], strong Chvátal-Gomory cuts [40], flow cover cuts [50], implied bound cuts [53], and clique cuts [37, 53]. Detailed descriptions of these cutting plane algorithms and an extensive analysis of their computational impact can be found in [57].

Almost as important as finding cutting planes is the selection of the cuts that actually enter the LP relaxation. Balas, Ceria, and Cornuéjols [13] and Andreello, Caprara, and Fischetti [8] proposed to base the cut selection on *efficacy* and *orthogonality*. The efficacy is the Euclidean distance of the corresponding hyperplane to the current LP solution. An orthogonality bound ensures that the cuts added to the LP form an almost pairwise orthogonal set of hyperplanes. SCIP follows these suggestions. Furthermore, it considers the parallelism w.r.t. the objective function.

3.5 Primal Heuristics

Primal heuristics have a significant relevance as supplementary procedures inside a MIP solver: they aim at finding good feasible solutions early in the search process, which helps to prune the search tree by bounding and allows to apply more reduced cost fixing and other dual reductions that tighten the problem formulation.

Overall, there are 24 heuristics integrated into SCIP. They can be roughly subclassified into four categories:

- *Rounding heuristics* try to iteratively round the fractional values of an LP solution in such a way that the feasibility of the constraints is maintained or recovered by further roundings.
- *Diving heuristics* iteratively round a variable with fractional LP value and resolve the LP, thereby simulating a depth first search (see Section 3.6) in the branch-and-bound tree.
- *Objective diving heuristics* are similar to diving heuristics, but instead of fixing the variables by changing their bounds, they perform "soft fixings" by modifying their objective coefficients.
- *Improvement heuristics* consider one or more primal feasible solutions that have been previously found and try to construct an improved solution with better objective value.

Detailed descriptions of primal heuristics for mixed integer programs and an in-depth analysis of their computational impact can be found in [17], an overview is given in [18].

3.6 Node Selection and Branching Rules

Two of the most important decisions in a branch-and-bound algorithm are the selection of the next subproblem to process (*node selection*) and how to split the current problem Q into smaller subproblems (*branching rule*).

The most popular branching strategy in MIP solving is to split the domain of an integer variable x_j, $j \in I$, with fractional LP value $\check{x}_j \notin \mathbb{Z}$ into two parts, thus creating two subproblems $Q_1 = Q \cap \{x_j \leq \lfloor \check{x}_j \rfloor\}$ and $Q_2 = Q \cap \{x_j \geq \lceil \check{x}_j \rceil\}$.

Methods to select such a fractional variable for branching are discussed in [2, 5]. SCIP implements most of the discussed branching rules, especially *reliability branching* which is a very effective general branching rule for MIPs.

In CP, it is also common to branch on constraints. A typical example is branching on *special ordered sets* (SOS) [15], where multiple variables are fixed to zero simultaneously, thus creating two special ordered sets of roughly half the size of the original SOS. SCIP supports arbitrary branching schemes such as branching on constraints or branchings that create more than two subproblems, which may be used for variables or constraints with small finite domains. In particular, the SOS1 and SOS2 constraint handlers branch on their corresponding constraints.

Besides a good branching strategy, the selection of the next subproblem to be processed is an important step of every branch-and-bound based search algorithm.

Depth first search always chooses a child of the current node as the next subproblem or backtracks to the most recent ancestor with an unprocessed child, if the current node has been pruned. Depth first search is the preferred strategy for pure feasibility problems like SAT problems. Additionally, it has the benefit that successively solved subproblems are very similar, which reduces the subproblem management overhead. In particular, this speeds up resolving the LP relaxation by the dual simplex algorithm.

Best bound search aims at improving the global dual bound as fast as possible by always selecting a subproblem with smallest dual bound of all remaining leaves in the tree. Given that the branching rule is fixed [1], there exists a node selection rule of best bound search type that leads to a minimal number of nodes that need to be processed.

Best estimate search was suggested by Forrest et al. [29]. For each subproblem, it estimates the minimum value of a rounded solution and chooses a node with minimal estimate. The aim is to quickly find good feasible solutions. However, this node selection strategy may perform very poor in improving the global dual bound.

The default node selection strategy of SCIP is a combination of these three strategies: it performs depth first search for a few consecutive subproblems after which a node with best estimate is chosen. At a certain frequency, a node with smallest dual bound is selected instead of a node with best estimate. This strategy is also referred to as *interleaved best bound/best estimate search with plunging*.

3.7 Presolving

Presolving transforms the given problem instance into an equivalent instance that is (hopefully) easier to solve. The most fundamental presolving concepts for MIP are described in [53]. For additional information, see [30].

The task of presolving is threefold: first, it reduces the size of the model by removing irrelevant information such as redundant constraints or fixed variables. Second, it strengthens the LP relaxation of the model by exploiting integrality information, e.g., to tighten the bounds of the variables or to improve coefficients in the constraints. Third, it extracts information from the model such as implications or cliques which can be used later for branching and cutting

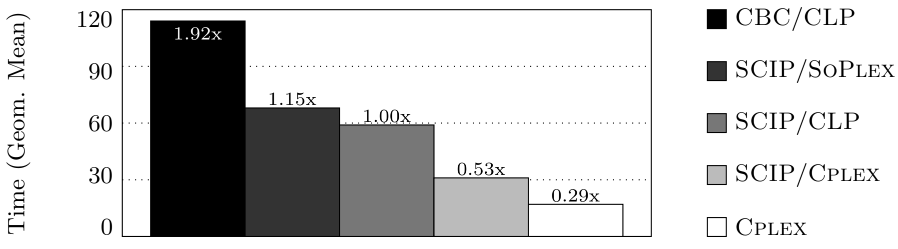

Figure 1: Ratios of the geometric means of the solving times depicted in Table 1. All solvers are compared to SCIP/CLP

.

plane separation. SCIP implements a full set of *primal* and *dual* presolving reductions for MIP problems, see [1].

Restarts differ from the classical presolving methods in that they are not applied *before* the branch-and-bound search commences, but abort a running search process in order to reapply other presolving mechanisms and start the search from scratch. They are a well-known ingredient of SAT solvers, but have not been used so far for solving MIPs.

Cutting planes, primal heuristics, strong branching [9], and reduced cost strengthening in the root node often identify fixings of variables that have not been detected during presolving. These fixings can trigger additional presolving reductions after a restart, thereby simplifying the problem instance and improving its LP relaxation. The downside is that we have to solve the root LP relaxation again, which can sometimes be very expensive.

Nevertheless, the above observation leads to the idea of applying a restart directly after the root node processing if a certain fraction of the integer variables has been fixed during the processing of the root node. In our implementation, a restart is performed if at least 5 % of the integer variables have been fixed.

3.8 Symmetry Handling

In mathematical programming, symmetries within a problem formulation usually detoriate the performance of state-of-the-art solvers.

There is a variety of symmetry handling techniques in constraint programmming, an overview is given in [31]. In integer programming, there are only few approaches to deal with generic symmetries, see [38, 49, 43] besides others. All of them consist of a global approach which operates on the LP relaxation.

Both, constraint specific symmetry handling and a global approach, which for example uses an additional symmetry breaking constraint, may be used in constraint integer programming. We are currently in the process of implementing symmetry handling capabilities into SCIP.

4 Solving MIPs with CIP Techniques

In Section 2, we proved that mixed integer programming is a specific case of constraint integer programming. As discussed in the previous section, most of the MIP solving algorithms can be generalized to the much richer class of

Table 1: Results for five MIP solvers on the Miplib 2003. If a solver hit one of the limits, we report the primal-dual gap in percent instead of the solving time in seconds.

	SCIP/CLP		SCIP/Cplex		SCIP/SoPlex		Cplex		CBC/CLP	
Name	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
10teams	166	19.5	250	9.3	2844	114.8	140	3.6	142	9.7
aflow30a	1756	22.9	1887	15.5	3393	34.0	2885	14.4	142 k	713.6
air04	75	124.4	165	73.2	129	215.9	405	14.1	840	85.2
air05	272	89.4	278	32.4	178	101.6	430	8.5	1085	61.8
cap6000	2543	5.2	3108	3.9	3024	6.5	4066	0.8	21 k	52.3
disctom	5682	1528.1	1	4.0	1	14.3	1	6.3	1	3.9
fiber	13	1.1	16	0.9	22	1.2	60	0.2	90	4.7
fixnet6	12	3.5	16	2.1	13	5.5	37	0.7	250	6.4
gesa2-o	8	1.5	5	1.4	5	1.5	514	1.1	1683	36.8
gesa2	7	1.6	11	1.6	6	1.4	112	0.3	403	9.6
manna81	1	1.0	2	1.1	2	1.6	1	0.1	1	1.0
mas74	3216 k	1715.4	3140 k	1071.7	2862 k	3138.0	2673 k	256.9	3643 k	2454.2
mas76	331 k	131.9	327 k	84.5	314 k	166.4	403 k	45.5	494 k	273.2
misc07	33 k	51.2	17 k	17.7	29 k	38.6	15 k	14.9	22 k	91.1
mod011	1477	314.0	1749	89.2	2136	537.4	74	34.8	500	68.8
modglob	29	0.9	149	1.1	118	1.3	164	0.1	2621	9.9
mzzv42z	808	1103.3	2159	343.2	1025	3565.9	324	42.2	238	121.0
nw04	6	51.8	34	44.8	205	580.3	189	27.1	1888	69.4
opt1217	1	0.5	1	0.4	1	0.4	1	0.1	1	0.4
p2756	80	2.3	118	2.2	36	2.5	11	0.3	34	2.7
pk1	255 k	172.2	219 k	89.9	229 k	204.7	193 k	117.0	214 k	160.0
pp08a	603	2.2	709	1.4	145	2.9	702	1.2	6705	33.5
pp08aCUTS	376	2.5	300	2.0	540	3.0	1400	1.6	1899	19.6
qiu	11 k	210.2	9150	66.9	13 k	283.1	2130	20.5	30 k	440.7
rout	21 k	83.3	22 k	34.5	16 k	60.0	9611	15.9	207 k	741.4
set1ch	15	0.7	15	0.6	20	0.9	307	0.3	80 k	903.6
vpm2	607	2.3	364	1.1	222	1.7	1153	0.4	309	5.0
aflow40b	153 k	2.9 %	309 k	2418.1	88 k	3.3 %	479 k	1.1 %	320 k	6.4 %
arki001	301 k	<0.1 %	1034 k	2666.8	148 k	<0.1 %	2133 k	<0.1 %	205 k	<0.1 %
fast0507	1207	0.8 %	2655	1250.8	842	0.9 %	4060	981.3	11 k	2801.9
harp2	4610 k	<0.1 %	9453 k	<0.1 %	706 k	0.2 %	379 k	238.1	900 k	<0.1 %
mzzv11	2778	2011.0	2503	449.8	64	0.9 %	1	85.7	6146	1002.7
net12	7337	64.3 %	3568	783.7	351	125.5 %	1225	51.2 %	547	77.3 %
noswot	306 k	235.7	574 k	242.5	262 k	144.8	4698 k	4.7 %	1857 k	4.7 %
swath	226 k	27.4 %	333 k	15.0 %	184 k	27.3 %	227 k	1626.0	480 k	36.6 %
tr12-30	690 k	<0.1 %	1024 k	3523.4	981 k	<0.1 %	426 k	761.5	28 k	1.1 %
Geom. Mean	1581	59.5	1641	31.4	1131	68.5	1542	17.0	4379	114.4
Solved Instances	29		34		28		32		29	
$\geq$ 10 % faster	—		25		6		31		8	
$\geq$ 10 % slower	—		1		18		1		19	

constraint integer programs. In this section, we show, that this still preserves the computational power for MIP solving. We evaluate the performance of the constraint integer programming solver SCIP for solving MIPs.

With the default plugins that are included in the distribution 1.1.0, SCIP can be used as a stand-alone MIP solver. Some of the plugins have been described in Section 3.

We tested SCIP 1.1.0 running on a 2.66 GHz Intel Core 2 Quad (32 bits) with 4 GB RAM and 4 MB cache, using CLP 1.8stable [25] as underlying LP solver. We set a time limit of one hour and a memory limit of 3 GB. For comparison, we applied the same test with SCIP 1.1.0 using Cplex 11.0 [34] or SoPlex 1.4.0 [58] to solve the LPs, Cplex 11.0 as stand-alone MIP solver,

Table 2: Results of five MIP solvers on the Miplib 2003 (continued). (⋆) For `markshare1` and `markshare2`, we report the primal bound instead of the primal-dual gap; the dual bound is zero in all cases.

	SCIP/CLP		SCIP/Cplex		SCIP/SoPlex		Cplex		CBC/CLP	
Name	Nodes	Gap [%]	Nodes	Gap [%]	Nodes	Gap [%]	Nodes	Gap [%]	Nodes	Gap [%]
a1c1s1	47 k	25.2	136 k	22.9	41 k	24.8	117 k	5.7	33 k	48.8
atlanta-ip	516	27.1	4939	8.5	43	22.5	2107	9.0	435	—
dano3mip	89	25.6	2071	24.8	179	30.4	2165	20.9	5301	30.9
danoint	183 k	3.8	634 k	3.1	141 k	3.9	390 k	2.9	206 k	3.1
ds	49	619.2	672	351.1	95	670.2	803	428.4	565	1773.4
glass4	2591 k	72.2	4258 k	90.8	1842 k	77.4	1998 k	16.7	636 k	45.9
liu	204 k	160.7	1129 k	161.1	282 k	137.1	143 k	107.9	16 k	204.3
mkc	428 k	2.0	921 k	1.7	482 k	2.0	121 k	0.2	452 k	1.6
momentum1	157	—	3679	—	300	—	7143	25.1	1144	—
momentum2	324	—	5691	27.9	377	—	3060	40.6	1536	—
msc98-ip	24	49.9	1273	12.1	1	—	915	12.1	307	—
nsrand-ipx	110 k	6.3	419 k	4.5	150 k	6.3	113 k	1.1	88 k	3.3
protfold	242	—	9100	—	1126	—	8689	47.7	14 k	51.4
rd-rplusc-21	49 k	—	46 k	>10 000	11 k	>10 000	14 k	>10 000	7621	—
roll3000	169 k	2.3	434 k	0.9	104 k	1.3	395 k	1.7	28 k	1.4
seymour	15 k	3.2	40 k	3.0	8596	3.5	58 k	2.2	13 k	3.8
sp97ar	2654	4.5	25 k	3.8	2651	4.6	112 k	1.0	8186	14.9
stp3d	1	—	6	—	1	—	1	—	1	—
timtab2	1986 k	80.0	2295 k	68.7	1488 k	72.5	692 k	58.4	145 k	165.5
markshare1	23 M	7⋆	32 M	4⋆	32 M	6⋆	11 M	5⋆	14 M	6⋆
markshare2	18 M	17⋆	28 M	12⋆	24 M	11⋆	10 M	11⋆	12 M	17⋆

and CBC 2.20 with CLP 1.8stable [25] as LP solver. Cplex is a very fast commercial MIP solver, CBC is a very fast open-source MIP solver. We used the provided default settings for all solvers. As test set we chose the 60 instances of the Miplib 2003 [6]. We left out the instances `momentum3`, `t1717`, and `timtab1` for which at least one of the solvers returned a wrong answer or reported an error.

Tables 1 and 2 compare the results of the five solvers. The first part of Table 1 lists all instances which were solved to optimality by all solvers, the second part those which were solved by at least one solver, and Table 2 those for which all solvers reached the time limit. Note, that the memory limit was not reached in any test run.

For each instance, listed in the "Name" column, the tables show the number of branch-and-bound nodes and the time in seconds needed to solve it with each of the five solvers. For instances which could not be solved within the time limit, we report the primal-dual gap in percent instead of the solving time. The primal-dual gap is defined as $\gamma = (\hat{c} - \check{c})/\inf[\check{c}, \hat{c}]$ with $\hat{c}$ being the upper (primal) and $\check{c}$ being the lower (dual) bound. The symbol "—" indicates instances for which no feasible solution was obtained within the time limit.

The results are summarized at the bottom of Table 1. There were 27 instances, for which all solvers were able to prove optimality within the time limit. The row "Geom. Mean" reports the geometric means of the number of branch-and-bound nodes and the solving times taken over these instances. Figure 1 visualizes the relations of the geometric means of the solving times. There were 36 instances, for which at least one solver was able to prove optimality within the time limit. The row "Solved Instances" states the number of solved instances for each individual solver. The rows "$\geq 10\,\%$ faster" and "$\geq 10\,\%$ slower" give the number of instances for which the solver was at least 10 % faster and at least 10 % slower than SCIP/CLP, respectively.

Although SCIP supports the much more general concept of constraint integer programming, it is competitive to state-of-the-art commercial and noncommercial MIP solvers. On this test set, SCIP/CLP turned out to be the fastest noncommercial solver w.r.t. the geometric mean of the solving times taken over those instances, which were solved by all solvers. SCIP/CLP was 3.5 and SCIP/Cplex only 1.84 times slower than Cplex. Furthermore, SCIP/CLP solved only three instance less, SCIP/Cplex even two instances more than Cplex within the time limit.

5 Counting Optimal Solutions with CIP Techniques

It is possible to enrich a branch-and-bound algorithm to be capable to count or enumerate all feasible or all optimal solutions of a given problem instance (branch-and-count). With release 1.1.0 we extended the framework SCIP to be able to solve counting problems of constraint integer programs.

Branch-and-bound algorithms can be adapted to enumerate all feasible solutions of a given problem instance by traversing the whole search tree and collecting all feasible solutions step-by-step. If we can deduce and construct *all* solution vectors contained in a subtree, it can be pruned without explicitly enumerating all leaves. The two most simple structures are subtrees which have no solutions and subtrees for which any variable assignment constitutes a feasible solution. These subtrees are called *infeasible subtrees* and *unrestricted subtrees* (see [4]). These two cases are handled in SCIP.

In this section, we regard the problem of computing the number of optimal solutions of integer programs. We compare the constraint integer programming approach to existing methods, in particular those of azove [16], Cplex [26], LattE [27], and zerOne [23]. We selected all instances contained in the Miplib [19] which only have integer variables. This gives a set of 30 instances. In a first step, we computed the optimal objective value $c^\star$ for each instance and added an additional linear constraint $c^T x = c^\star$ to the integer program. Thus, for the resulting problem, each feasible solution is an optimal solution.

We ran our tests in the same computational environment as in the previous Section 4. The results are presented in Table 3. For each instance listed in the "Name" column, the table reports the number of optimal solutions ("#Solutions") and the running time in seconds of each solver. If a solver did not solve an instance, due to an error or hitting one of the limits, we indicate this by an "—". None of the solvers could solve the instances `cracpb1` and `p2756`. Therefore, we do not list them in Table 3.

Column "SCIP/Cplex" gives the results for SCIP 1.1.0 with Cplex 11.0 as underlying LP solver. We used the setting "emphasis/counter.set" which is provided by SCIP. In order to ensure a correct counting process, it disables features which may cut off feasible, but suboptimal solutions and result in finding solutions twice, such as dual presolving and primal heuristics, respectively,

For azove, we present the results of two different versions: 1.1 and 2.0. Here, we used the "-c" option in order to count the number of feasible solutions

Table 3: Result for counting the number of optimal solutions.

Name	#Solutions	SCIP/Cplex	azove 1.1	azove 2.0	Cplex	LattE	zerOne
air01	2	0.3	1.9	—	0.1	—	1.0
air02	1	3.0	—	—	2.1	—	21.0
air03	1	72.0	—	—	41.4	—	51.0
air04	8	151.8	—	—	2521.2	—	—
air05	2	54.7	—	—	—	—	—
air06	1	163.6	—	—	2.5	—	72.0
bm23	1	0.1	1.2	—	0.1	—	1.0
diamond	0	0.0	0.0	0.0	0.0	0.0	0.0
enigma	2	0.1	—	—	0.9	—	3.0
l152lav	1	26.3	—	—	220.9	—	24.0
lp4l	24	0.8	—	—	2.4	—	5.0
lseu	2	1.7	5.6	—	0.6	—	12.0
misc02	4	0.0	—	—	0.1	—	0.0
misc03	24	0.5	—	—	2.6	—	0.0
misc07	72	17.4	—	—	95.1	—	0.0
mod008	6	6.0	8.2	—	1.4	—	9.0
mod010	128	20.3	—	—	9.9	—	134.0
p0033	9	0.0	0.0	—	0.0	—	1.0
p0040	1	0.0	0.4	—	0.0	—	0.0
p0201	4	0.3	—	—	2.0	—	58.0
p0282	1	0.3	—	—	5.7	—	—
p0548	151 165 440	135.9	—	—	—	—	—
pipex	1	0.1	14.0	—	0.0	—	0.0
sentoy	1	0.1	—	—	0.1	—	0.0
stein9	54	0.0	0.0	0.0	0.0	11.0	0.0
stein15	315	0.1	0.0	0.0	0.1	—	0.0
stein27	2 106	2.7	0.1	—	3.2	—	5.0
stein45	70	53.9	300.7	—	36.1	—	105.0
Solved Instances		28	12	3	26	2	24

implicitly. For the solver Cplex 11.0, we set the number of threads to one and set the integrality tolerance to zero to ensure a proper result. Finally, column "LattE" gives the results of the tool LattE macchiato 1.2-mk-0.9 and the last column reports the results of zerOne 1.81 with Cplex 11.0 as LP solver.

In this test, the branch-and-bound based solvers Cplex, zerOne, and SCIP were superior to the other solvers. This is probably due to that general mixed integer programs are in favor for this type of tools. Cplex solved more instances than zerOne; SCIP solved more instances than Cplex. The instances `air05` and `p0548` were only solved by SCIP. For the latter one, the unrestricted subtree detection [4] was essential.

6 Property Checking with CIPs

One of the key technologies in the design of integrated circuits is the verification of the correctness of the chip design [35]. An important aspect of this process is the so-called *property checking problem*, which consists of verifying that certain inherent properties of the chip design are satisfied.

Today's techniques validate these properties on the so-called *gate level* by transforming the properties into Boolean clauses and hence modeling the property checking problem as a SAT instance. However, complex arithmetic operations like multiplication lead to difficult SAT instances with involved interrelationships between the variables, which are hard to solve for current SAT solvers.

Our approach is to tackle the problem on a higher level, the *register transfer level*. The property checking problem at the register transfer level can be formulated as a constraint integer program on bit vector variables $\varrho \in \{0, \dots, 2^{w_\varrho - 1}\}$ of width w_ϱ. The constraints $r = C(x, y, z)$ model the circuit operations.

For each bit vector variable ϱ, we introduce single bit variables $\varrho_b \in \{0, 1\}$, with $b \in \{0, \dots, w_\varrho - 1\}$, for which *linking constraints*

$$\varrho = \sum_{b=0}^{w_\varrho - 1} 2^b \varrho_b \tag{2}$$

define their correlation. In addition, we consider the following circuit operations: ADD, AND, CONCAT, EQ, ITE, LT, MINUS, MULT, NOT, OR, READ, SHL, SHR, SIGNEXT, SLICE, SUB, UAND, UOR, UXOR, WRITE, XOR, ZEROEXT with semantics as defined in [1, 22]. All these operations can be expressed as specific constraints according to the CIP paradigm.

6.1 Constraint Programming Techniques

For the bit linking constraints (2) and for each type of circuit operation, we implemented a specialized constraint handler which includes a domain propagation algorithm that exploits the special structure of the constraint class. In addition to considering the current domains of the bit vectors ϱ and the bit variables ϱ_b, we exploit knowledge about the global equality or inequality of bit vectors or bits, which is obtained in the preprocessing stage of the algorithm.

Some of the domain propagation algorithms are very complex. For example, domain propagation of MULT constraints uses term algebra techniques to recognize certain deductions inside its internal representation of a partial product and overflow addition network. Others, like the algorithms for SHL, SLICE, READ, and WRITE constraints, involve reasoning that mixes bit- and word-level information.

6.2 Mixed Integer Programming Techniques

Because property checking is a pure feasibility problem, there is no natural objective function. However, the LP relaxation usually detects infeasibility of local subproblems much earlier than domain propagation.

Table 4 shows the linearizations of circuit operation constraints that are used in addition to the bit linking constraints (2) to construct the LP relaxation of the problem instance. Very large coefficients like 2^{w_r} in the ADD linearization can lead to numerical difficulties in the LP relaxation. Therefore, we split the bit vector variables into words of 16 bits and apply the linearization to the individual words. The linkage between the words is established in a proper fashion. For example, the overflow bit of a word in an addition is added to the right hand side of the next word's linearization. The relaxation of the MULT constraint involves additional variables y_n and r_n which are "nibbles" of y and r with $L = 8$ bits.

No linearization is generated for SHL, SLICE, READ, and WRITE constraints. Their linearizations are very complex and would dramatically increase the size of the LP relaxation, thereby reducing the solvability of the LPs. For example,

Table 4: LP relaxation of circuit operations. l_ϱ and u_ϱ are the lower and upper bounds of a bit vector variable ϱ.

Name	Operation	Linearization
and	r = AND (x,y)	$r_b \le x_b,\ r_b \le y_b,\ r_b \ge x_b + y_b - 1$
or	r = OR (x,y)	$r_b \ge x_b,\ r_b \ge y_b,\ r_b \le x_b + y_b$
xor	r = XOR (x,y)	$x_b - y_b - r_b \le 0,\ -x_b + y_b - r_b \le 0,$
		$-x_b - y_b + r_b \le 0,\ x_b + y_b + r_b \le 2$
unary and	r = UAND (x)	$r \le x_b,\ r \ge \sum_{b=0}^{w_x-1} x_b - w_x + 1$
unary or	r = UOR (x)	$r \ge x_b,\ r \le \sum_{b=0}^{w_x-1} x_b$
unary xor	r = UXOR (x)	$r + \sum_{b=0}^{w_x-1} x_b = 2s, \qquad s \in \mathbb{Z}_{\ge 0}$
equal	r = EQ (x,y)	$x - y = s - t, \qquad s,t \in \mathbb{Z}_{\ge 0},$
		$p \le s,\ s \le p(u_x - l_y), \qquad p \in \{0,1\},$
		$q \le t,\ t \le q(u_y - l_x), \qquad q \in \{0,1\},$
		$p + q + r = 1$
less than	r = LT (x,y)	$x - y = s - t, \qquad s,t \in \mathbb{Z}_{\ge 0},$
		$p \le s,\ s \le p(u_x - l_y), \qquad p \in \{0,1\},$
		$r \le t,\ t \le r(u_y - l_x),$
		$p + r \le 1$
if then else	r = ITE (x,y,z)	$r - y \le (u_z - l_y)(1 - x)$
		$r - y \ge (l_z - u_y)(1 - x)$
		$r - z \le (u_y - l_z)\, x$
		$r - z \ge (l_y - u_z)\, x$
add	r = ADD (x,y)	$r + 2^{w_r} o = x + y, \qquad o \in \{0,1\}$
multiply	r = MULT (x,y)	$v_{bn} \le u_{y_n} x_b,\ v_{bn} \le y_n, \qquad v_{bn} \in \mathbb{Z}_{\ge 0}$
		$v_{bn} \ge y_n - u_{y_n}(1 - x_b)$
		$o_n + \sum_{i+j=n} \sum_{l=0}^{L-1} 2^l v_{iL+l,j}$
		$= 2^L o_{n+1} + r_n, \qquad o_n \in \mathbb{Z}_{\ge 0}$

a straight-forward linearization of the SHL constraint on a 64-bit input vector x that uses internal ITE-blocks for the potential values of the shifting operand y already requires 30944 inequalities and 20929 auxiliary variables.

6.3 SAT Solving Techniques

Conflict analysis is particular useful on feasibility problems like property checking. By applying reverse propagation, one or more conflict constraints can be extracted from the conflict graph of an infeasible subproblem. In our implementation, we use the *1-FUIP* [59] rule for generating conflict constraints. In addition to *1-FUIP* conflict constraints, we extract clauses from *reconvergence cuts* [59] in the conflict graph to support *nonchronological backtracking* [44].

6.4 Computational Results

We examined the computational effectiveness of the described CIP techniques on industrial benchmarks obtained from verification projects conducted together with Infineon and OneSpin Solutions. The specific chip verification algorithms were incorporated into SCIP. All calculations given is this section were performed in the same environment as described in Section 4. We used version 1.1.0 of SCIP with Cplex 11.0 as LP solver. For reasons of comparison,

Table 6: Biquad properties.

Property	Meth	variant A	B	C
g_checkgpre	SAT	12.9	32.2	16.5
	CIP	27.4	22.3	34.4
g2_checkg2	SAT	3575.7	—	—
	CIP	253.8	203.2	202.3
g25_checkg25	SAT	0.0	1.5	1.6
	CIP	102.3	34.6	32.3
g3_negres	SAT	0.0	0.0	0.0
	CIP	0.5	0.0	0.0
gBIG_checkreg1	SAT	115.1	70.7	69.8
	CIP	230.2	73.3	67.3

Table 5: ALU properties. (time in seconds)

Prop	Meth	register width 5	10	15	20	25	30	35	40
muls	SAT	0.3	3560.2	—	—	—	—	—	—
	CIP	0.0	0.0	0.1	0.2	0.6	1.3	2.8	7.5
neg_flag	SAT	0.1	46.1	—	—	—	—	—	—
	CIP	1.3	4.5	11.2	23.4	50.2	99.0	187.0	336.8
zero_flag	SAT	0.0	0.0	0.1	0.1	0.2	0.2	0.3	0.4
	CIP	1.4	6.7	17.9	7.2	58.0	71.0	233.1	381.6

Table 7: Multiplier properties. (time in seconds)

Layout	Meth	register width 6	7	8	9	10	11	12	13	14
booth	SAT	0.3	2.1	13.0	74.0	422.7	3488.4	—	—	—
signed	CIP	11.1	47.1	63.3	257.9	127.9	1156.1	1101.7	—	2693.4
booth	SAT	0.4	1.6	11.4	60.7	442.3	1876.0	—	—	—
unsgnd	CIP	7.4	43.6	73.3	253.8	302.8	1360.7	2438.7	3046.9	—
nonbth	SAT	0.3	2.2	13.3	74.5	596.3	3004.7	—	—	—
signed	CIP	7.3	34.7	55.5	199.9	263.1	420.3	876.1	1568.8	1847.3
nonbth	SAT	0.2	1.2	10.4	50.2	470.1	2454.6	—	—	—
unsgnd	CIP	2.3	19.6	58.3	119.5	222.5	511.9	703.7	1258.5	2454.9

we also solved the instances with SAT techniques on the gate level. We used MiniSat 2.0 [28] to solve the SAT instances obtained after a preprocessing step.

The experiments were conducted on the valid properties included in the following sets of property checking instances: *ALU* (an arithmetical logical unit which performs ADD, SUB, SHL, SHR, and signed and unsigned MULT operations), *Biquad* (a DSP/IIR filter core obtained from [48] in different representations), and *Multiplier* (gate level net lists for Booth and non-Booth encoded architectures of signed and unsigned multipliers).

Tables 5–7 compare the results of MiniSat and our CIP approach on the valid properties. For each property or layout and each input register width or variant, the tables show the time in seconds of the two algorithms needed to solve the instance. Results marked with “—” could not be solved within the time limit. The experiments show that our approach outperforms SAT techniques for proving the validity of properties on circuits containing arithmetics. For invalid properties, which are not shown in the tables, our algorithm usually is inferior to SAT techniques for finding counter-examples. This is due to the much more involved procedures employed in the CIP approach, which may cause much more computational effort per branch-and-bound node.

7 Future Research

Further research in the field of constraint integer programming should enrich the variety of CIP solver features like counting feasible solutions, see Section 5, and help to deal with typical issues of mathematical programming, like numerical inaccuracy, see Section 7.2, and complex symmetries, see Section 3.8. We also plan to extend the spectrum of problem classes which can be handled by CIP methods, see Sections 7.1 and 7.3.

7.1 Nonconvex Mixed Integer Nonlinear Programming

Given twice continuously differentiable functions $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$, vectors $l, u \in \mathbb{R}^n$, and a subset $I \subseteq N = \{1, \dots, n\}$, the general *mixed integer nonlinear program* (MINLP) is to solve

$$\min \{f(x) \mid g(x) \leq 0,\ l \leq x \leq u,\ x \in \mathbb{R}^n,\ x_j \in \mathbb{Z} \text{ for all } j \in I\}.$$

For MINLPs where the functions f and g are *convex*, several software packages are available. Most of these solvers extend an LP based branch-and-cut approach for MIPs by linearizing nonlinear functions whenever appropriate. For *nonconvex* MINLPs, there are numerous application specific approaches. Only few codes exist, however, that can handle nonconvex MINLPs in general, among them the solvers BARON [55] and LaGO [47]. The latter implements a convexification based branch-and-cut algorithm. Here, nonconvex quadratic terms are convexified by applying the so-called alpha-underestimating technique, while nonconvex nonquadratic functions are first underestimated by a quadratic function, which is then convexified.

Bringing together the expertise of SCIP and LaGO, we are developing a general purpose LP based branch-and-cut solver for nonconvex MINLPs. Applications are, among others, in fare planning, in mine production planing, and in the optimization of the design and operation of complex energy conversion systems.

7.2 Exact Constraint Integer Programming

As most standard MIP solvers, SCIP is based on *floating-point arithmetic*, which may cause slight rounding errors in arithmetic operations. For most applications, this error-proneness can be neglected—the computed solutions meet the accuracy requirements. The situation changes fundamentally, if CIPs are used to study theoretical problems, if pure feasibility problems are considered, and if wrong answers can have legal consequences. For such applications, an exact solver is required.

We are developing an approach for the exact solution of CIPs. Extending the framework SCIP, we want to provide a solver that always produces correct answers for both feasible and infeasible CIP instances, i.e., an exact optimal solution or an exact infeasibility certificate.

7.3 Solving Scheduling Problems with CIP Techniques

Scheduling is an optimization problem where jobs or duties have to be assigned to resources minimizing, e.g., the overall makespan. A variety of solving methods for difficult scheduling problems has been proposed in the constraint programming literature [14] as well as in the integer programming literature [24]. The strong CP propagation methods for, e.g., precedence constraints are a powerful tool for constructing high-quality feasible solutions, while MIP techniques like column generation allow to solve very large scale problem instances.

As scheduling problems seem to be more tractable by CP techniques, but often additional instance specific requirements can be better dealt with by MIP solvers, we are developing a constraint integer programming approach, which incorporates solving features from both fields.

8 Conclusion

We introduced constraint integer programming, a new approach to integrate constraint programming and mixed integer programming. We presented various solving techniques from different fields of mathematical programming and showed how they can be used to solve constraint integer programs and how they are incorporated into the SCIP framework. Although SCIP is able to solve the much broader class of constraint integer programs, it is currently one of the fastest MIP solvers and competitive to state-of-the-art commercial and noncommercial solvers.

Taking the chip design verification problem as an application, we demonstrated the usefulness of constraint integer programming techniques to solve problems which were intractable by previous nonintegrated approaches. We presented new computational results and suggested possible directions of future development in the field of constraint integer programming.

References

[1] T. Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4(1):4–20, 2007. Special issue: Mixed Integer Programming.

[2] T. Achterberg. *Constraint Integer Programming.* PhD thesis, Technische Universität Berlin, 2007.

[3] T. Achterberg, T. Berthold, T. Koch, and K. Wolter. Constraint integer programming: A new approach to integrate CP and MIP. In L. Perron and M. A. Trick, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 5th International Conference, CPAIOR 2008*, volume 5015 of *Lecture Notes in Computer Science*, pages 6–20, 2008.

[4] T. Achterberg, S. Heinz, and T. Koch. Counting solutions of integer programs using unrestricted subtree detection. In *Proc. of CPAIOR-08*, volume 5015 of *Lecture Notes in Computer Science*, pages 278–282, 2008.

[5] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33:42–54, 2005.

[6] T. Achterberg, T. Koch, and A. Martin. MIPLIB 2003. *Operations Research Letters*, 34(4):1–12, 2006.

[7] E. Althaus, A. Bockmayr, M. Elf, M. Jünger, T. Kasper, and K. Mehlhorn. SCIL – Symbolic Constraints in Integer Linear Programming. In *Algorithms – ESA 2002*, pages 75–87, 2002.

[8] G. Andreello, A. Caprara, and M. Fischetti. Embedding cuts in a branch&cut framework: a computational study with $\{0, \frac{1}{2}\}$-cuts. *INFORMS Journal on Computing*, 2007. to appear.

[9] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem.* Princeton University Press, Princeton, 2006.

[10] K. R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[11] I. D. Aron, J. N. Hooker, and T. H. Yunes. SIMPL: A system for integrating optimization techniques. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2004*, volume 3011 of *Lecture Notes in Computer Science*, pages 21–36, 2004.

[12] E. Balas. Facets of the knapsack polytope. *Mathematical Programming*, 8:146–164, 1975.

[13] E. Balas, S. Ceria, and G. Cornuéjols. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science*, 42:1229–1246, 1996.

[14] P. Baptiste, C. L. Pape, and W. Nuijten. *Constraint-Based Scheduling.* Kluwer, 2001.

[15] E. M. L. Beale and J. A. Tomlin. Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. In J. Lawrence, editor, *OR 69: Proc. of the Fifth International Conference on Operations Research*, pages 447–454, London, 1970. Tavistock Publications.

[16] M. Behle and F. Eisenbrand. 0/1 vertex and facet enumeration with BDDs. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[17] T. Berthold. Primal heuristics for mixed integer programs. Master's thesis, Technische Universität Berlin, 2006.

[18] T. Berthold. Heuristics of the branch-cut-and-price-framework SCIP. In J. Kalcsics and S. Nickel, editors, *Operations Research Proceedings 2007*, pages 31–36. Springer-Verlag, 2008.

[19] R. E. Bixby, E. A. Boyd, and R. R. Indovina. MIPLIB: A test set of mixed integer programming problems. *SIAM News*, 25:16, 1992.

[20] A. Bockmayr and T. Kasper. Branch-and-infer: A unifying framework for integer and finite domain constraint programming. *INFORMS Journal on Computing*, 10(3):287–300, 1998.

[21] A. Bockmayr and N. Pisaruk. Solving assembly line balancing problems by combining IP and CP. Sixth Annual Workshop of the ERCIM Working Group on Constraints, June 2001.

[22] R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *Proc. of the IEEE VLSI Design Conference*, pages 741–746, 2002.

[23] M. R. Bussieck and M. E. Lübbecke. The vertex set of a 0/1-polytope is strongly $\mathcal{P}$-enumerable. *Comput. Geom*, 11(2):103–109, 1998.

[24] N. Christofides, R. Alvarez-Valdes, and J. M. Tamarit. Project scheduling with resource constraints: A branch and bound approach. *European Journal of Operational Research*, 29(3):262–273, June 1987.

[25] COIN-OR. Computational infrastructure for operations research. `http://www.coin-or.org`.

[26] E. Danna, M. Fenelon, Z. Gu, and R. Wunderling. Generating multiple solutions for mixed integer programming problems. In M. Fischetti and D. P. Williamson, editors, *Integer Programming and Combinatorial Optimization*, volume 4513 of *Lecture Notes in Computer Science*, pages 280–294, 2007.

[27] J. A. De Loera, R. Hemmecke, J. Tauzer, and R. Yoshida. Effective lattice point counting in rational convex polytopes. *J. Symb. Comput.*, 38(4):1273–1302, oct 2004.

[28] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Proc. of SAT 2003*, pages 502–518. Springer, 2003.

[29] J. J. Forrest, J. P. H. Hirst, and J. A. Tomlin. Practical solution of large scale mixed integer programming problems with UMPIRE. *Management Science*, 20(5):736–773, 1974.

[30] A. Fügenschuh and A. Martin. Computational integer programming and cutting planes. In K. Aardal, G. L. Nemhauser, and R. Weismantel, editors, *Discrete Optimization*, volume 12 of *Handbooks in Operations Research and Management Science*, chapter 2, pages 69–122. Elsevier, 2005.

[31] I. Gent, K. Petrie, and J.-F. Puget. Symmetry in constraint programming. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, pages 329–376. Elsevier, 2006.

[32] R. E. Gomory. Solving linear programming problems in integers. In R. Bellman and J. M. Hall, editors, *Combinatorial Analysis*, Symposia in Applied Mathematics X, pages 211–215, Providence, RI, 1960. American Mathematical Society.

[33] J. N. Hooker and M. A. Osorio. Mixed Logical/Linear Programming. *Discrete Applied Mathematics*, 96-97(1):395–442, 1999.

[34] ILOG CPLEX. Reference Manual. `http://www.ilog.com/products/cplex`.

[35] International technology roadmap for semiconductors, 2005. `http://public.itrs.net`.

[36] V. Jain and I. E. Grossmann. Algorithms for hybrid MILP/CP models for a class of optimization problems. *INFORMS Journal on Computing*, 13(4):258–276, 2001.

[37] E. L. Johnson and M. W. Padberg. Degree-two inequalities, clique facets, and biperfect graphs. *Annals of Discrete Mathematics*, 16:169–187, 1982.

[38] V. Kaibel and M. E. Pfetsch. Packing and partitioning orbitopes. *Mathematical Programming*, 114(1):1–36, 2008.

[39] A. Klar. Cutting planes in mixed integer programming. Master's thesis, Technische Universität Berlin, 2006.

[40] A. N. Letchford and A. Lodi. Strengthening Chvátal-Gomory cuts and Gomory fractional cuts. *Operations Research Letters*, 30(2):74–82, 2002.

[41] H. Marchand. *A polyhedral study of the mixed knapsack set and its use to solve mixed integer programs.* PhD thesis, Faculté des Sciences Appliquées, Université catholique de Louvain, 1998.

[42] H. Marchand, A. Martin, R. Weismantel, and L. A. Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*, 123/124:391–440, 2002.

[43] F. Margot. Pruning by isomorphism in branch-and-cut. *Mathematical Programming, Series A*, 94:71–90, 2002.

[44] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions of Computers*, 48:506–521, 1999.

[45] H. Mittelmann. Decision tree for optimization software: Benchmarks for optimization software. `http://plato.asu.edu/bench.html`.

[46] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, 1988.

[47] I. Nowak and S. Vigerske. LaGO – a (heuristic) Branch and Cut algorithm for nonconvex MINLPs. *Central European Journal of Operations Research*, 16(2):127–138, 2008.

[48] `http://www.opencores.org`.

[49] J. Ostrowski, J. Linderoth, F. Rossi, and S. Smirglio. Orbital branching. In M. Fischetti and D. Williamson, editors, *Proc. of the 12th IPCO*, volume 4513 of *Lecture Notes in Computer Science*, pages 104–118. Springer-Verlag, 2007.

[50] M. W. Padberg, T. J. van Roy, and L. A. Wolsey. Valid inequalities for fixed charge problems. *Operations Research*, 33(4):842–861, 1985.

[51] P. Refalo. Tight Cooperation and Its Application in Piecewise Linear Optimization. In *Principles and Practice of Constraint Programming, CP 1999*, volume 1713 of *Lecture Notes in Computer Science*, pages 375–389, 1999.

[52] R. Rodosek, M. G. Wallace, and M. T. Hajian. A New Approach to Integrating Mixed Integer Programming and Constraint Logic Programming. *Annals of Operations Research*, 86(1):63–87, 1999.

[53] M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.

[54] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[55] M. Tawarmalani and N. V. Sahinidis. Global optimization of mixed-integer nonlinear programs: A theoretical and computational study. *Mathematical Programming*, 99:563–591, 2004.

[56] C. Timpe. Solving planning and scheduling problems with combined integer and constraint programming. *OR Spectrum*, 24(4):431–448, November 2002.

[57] K. Wolter. Implementation of cutting plane separators for mixed integer programs. Master's thesis, Technische Universität Berlin, 2006.

[58] R. Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus*. PhD thesis, Technische Universität Berlin, 1996.

[59] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.