Constraint Integer Programming: a New Approach to Integrate CP and MIP

Tobias Achterberg[1], Timo Berthold[2], Thorsten Koch[2], and Kati Wolter[2]

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

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 from SAT solving. SCIP is available in source code and free for non-commercial use.
We demonstrate the usefulness of CIP on two 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 employ the CIP framework to solve chip design verification problems, which involve some highly non-linear 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 non-linear 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 [15, 25, 40]. Different approaches to integrate CP and MIP into a single framework have been proposed, [5, 9, 14, 22, 36, 37] amongst others.

Most of the existing work follows the concept of augmenting a CP framework with basic MIP techniques, namely LP relaxations and sometimes cutting planes. In contrast, this paper introduces a way to incorporate CP specific solving methods and its strong modeling capability into the sophisticated MIP solving machinery. This is achieved by a low-level integration of the two concepts. The constraints of a CP usually interact through the domains of the variables. Similar to [9, 14, 36, 37], the idea of *constraint integer programming (CIP)* is to offer a second communication interface, namely the 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 non-linear constraints. As an example for such a problem type, the property checking problem is presented in Section 5.

The concept of constraint integer programming is realized in the branch-and-cut framework SCIP. It combines solving techniques for CP, MIP, and *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 [42] or CLP [17] as LP solver, it is the fastest non-commercial MIP solver that is currently available, see [32] and our results in Section 4. Using Cplex [23] as LP solver, the performance of SCIP is even comparable to the today's best 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 non-linear 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 non-commercial use and can be downloaded from http://scip.zib.de. The current version 1.00—as of this writing—has interfaces to five different LP solvers and consists of 223 178 lines of C code. The code is actively maintained and extended, and we hope to be able to make further improvements.

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 and 5, we demonstrate the usage of SCIP on two applications. First, we employ SCIP as a stand-alone MIP solver, and second, we use SCIP as a branch-and-cut framework to solve chip verification problems. Computational results are given in the Sections 4 and 5.4.

2 Constraint Integer Programs

Most solvers for CP, SAT, and MIP are based on dividing the problem into smaller subproblems and implicitly enumerating all potential solutions. Because MIP is a very specific case of CP, MIP solvers can apply sophisticated techniques that operate on the subproblem as a whole, for example solving the linear programming (LP) relaxation or generating cutting planes.

In contrast, due to the very general definition of CPs, 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, using very expressive constraints, which maintain the structure of the problem.

On the other hand, SAT is also a very specific case of CP with only one type of constraints, namely Boolean clauses. Such a clause 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.

The hope of integrating CP, SAT, and MIP techniques is to combine their advantages and to compensate for their individual weaknesses. We propose the following slight restriction of a CP, which allows the application of MIP solving techniques, to specify our integrated approach:

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

$$\text{(CIP)} \quad c^\star = \min\{c^T x \mid \mathcal{C}_i(x) = 1 \text{ for all } i = 1, \ldots, m, \\ x \in \mathbb{R}^n,\ x_j \in \mathbb{Z} \text{ for all } j \in I\}$$

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:

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

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

Restriction (1) ensures that the remaining subproblem after fixing all integer variables is always 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 corresponding 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 non-linear 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 with finite domains for all variables can be modeled as a CIP.

(b) Every MIP can be modeled as a CIP.

3 The SCIP Framework

SCIP 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 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 are usually 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 CIPs. 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. SCIP provides its own memory management and plenty of statistical output.

Besides the infrastructure, all main algorithms of SCIP are implemented as external plugins. In the remainder of this section, we will describe the most important types of plugins and their role for solving CIPs.

3.1 Constraint Handlers

Since a CIP consists of constraints, the central objects of SCIP are the *constraint handlers*. Each constraint handler represents the semantics 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 turn SCIP into 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, namely

- 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 distribution of SCIP includes the 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.

3.2 Domain Propagation

Constraint propagation is an integral part of every CP solver [8]. 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 a restricted version of the preprocessing algorithm that is used before starting the branch-and-bound process to simplify the problem instance (see, e.g., [38] or [20]).

Besides the integrality restrictions, there is only one type of constraints in a MIP, namely the linear constraints. In contrast, CP models can include a large variety of constraint classes with different semantics and structure. 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, SCIP features two dual domain reduction methods that are driven by the objective function, namely the *objective propagation* and the *root reduced cost strengthening* [33].

3.3 Conflict Analysis

Current state-of-the-art 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 [31]. 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 *non-chronological backtracking*. A similar idea in CP are *no-goods*, see e.g., [39].

SCIP generalizes conflict analysis 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 the 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, we proceed in the same fashion as SAT solvers: we construct a conflict graph, choose a cut in this graph, and produce 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 Q 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 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 the current LP solution $\check{x}$ but do not cut off feasible solutions from Q. Thus, the current solution $\check{x}$ is *separated* from the convex hull of integer solutions Q_I by the *cutting plane* $a^T x \leq b$, i.e., $\check{x} \notin \{x \in \mathbb{R} \mid a^T x \leq b\} \supseteq Q_I$.

The theory of cutting planes is very well covered in the literature. For an overview of computationally useful cutting plane techniques, see [20, 30]. A recent survey of cutting plane literature can be found in [27].

SCIP features separators for knapsack cover cuts [10], complemented mixed integer rounding cuts [29], Gomory mixed integer cuts [21], strong Chvátal-Gomory cuts [28], flow cover cuts [35], implied bound cuts [38], and clique cuts [26, 38]. Detailed descriptions of the cutting planes algorithms integrated into SCIP and an extensive analysis of their computational impact can be found in [41].

Almost as important as finding cutting planes is the selection of the cuts that actually should enter the LP relaxation. Balas, Ceria, and Cornuéjols [11] and Andreello, Caprara, and Fischetti [6] proposed to base the cut selection on *efficacy* and *orthogonality*. The efficacy is the Euclidean distance of the cut hyperplane to the current LP solution, and an orthogonality bound makes sure that the cuts added to the LP form an almost pairwise orthogonal set of hyperplanes. SCIP follows these suggestions.

3.5 Primal Heuristics

Primal heuristics have a significant relevance as supplementary procedures inside a MIP solver: they help to find 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 can tighten the problem formulation.

Overall, there are 23 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 for 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 the primal heuristics implemented in SCIP and an in-depth analysis of their computational impact can be found in [12], an overview is given in [13].

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, 3].

SCIP implements most of the discussed branching rules, especially *reliability branching* which is currently the most effective general branching rule for MIP. Using SCIP, it is possible to implement arbitrary branching schemes such as branchings that create more than two subproblems or branching on constraints.

SCIP offers several node selection strategies as default plugins. *Depth first search* always chooses a child of the current node as the next subproblem to be processed 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. Additionally, it has the benefit that successively solved subproblems are very similar, which reduces the subproblem management overhead.

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

Best Estimate search was suggested by Forrest et al. [19]. It estimates the minimum value of a rounded solution in each subproblem 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.

3.7 Presolving

Presolving is a way to transform the given problem instance into an equivalent instance that is (hopefully) easier to solve. The most fundamental presolving concepts for MIP are described in [38]. For additional information, see [20].

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 such as implications or cliques from the model which can later be used, for example for branching or cutting 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 modern SAT solvers, but have not been used so far for solving MIPs.

It is often the case that cutting planes, strong branching [7], and reduced cost strengthening in the root node identify fixings of variables that have not been detected during presolving. These fixings can trigger additional presolve 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.

4 SCIP as a MIP Solver

With the default plugins that are included in the distribution, SCIP can be used as a stand-alone MIP solver. Some of the plugins have been described in Section 3. In this section we evaluate the performance of SCIP for solving MIPs.

	SCIP/Cplex		Cplex		SCIP/SoPlex		CBC/CLP	
Name	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
10teams	671	20.3	1	0.4	564	77.7	190	24.9
aflow30a	2353	13.5	3054	7.9	4293	35.6	30577	79.0
air04	334	98.9	263	8.2	159	189.7	565	172.1
air05	384	49.4	467	7.3	314	134.6	548	95.4
cap6000	3455	4.1	4227	0.7	2647	6.4	3390	7.1
disctom	1	85.4	1	6.0	1	64.4	1	4.2
fiber	24	1.1	60	0.2	12	1.3	40	2.2
fixnet6	26	1.6	71	0.6	10	2.8	114	3.4
gesa2-o	108	6.5	482	0.8	155	11.1	5695	32.6
gesa2	132	5.7	147	0.2	251	7.4	275	6.7
manna81	2	5.5	1	0.1	1	5.7	1	0.7
mas74	3275993	783.9	2673089	281.8	3036576	1582.8	4887385	2390.2
mas76	349635	73.4	398167	37.4	313718	118.0	687061	180.3
misc07	19719	15.2	25645	20.2	19831	27.7	29130	64.1
mod011	1751	76.8	54	20.7	2034	636.2	6318	132.4
modglob	21	0.9	183	0.1	3573	50.1	12664	26.3
nw04	457	92.7	283	29.2	49	369.5	22	12.5
p2756	45	2.6	11	0.2	109	3.3	37	1.4
pk1	219292	71.9	186390	81.7	226525	165.5	204094	81.8
pp08a	139	1.3	567	0.4	199	2.5	5087	31.3
pp08aCUTS	77	1.1	1102	1.1	109	2.6	5928	26.5
qiu	12653	76.9	7233	29.3	12973	337.5	31866	295.2
rout	11967	15.3	5260	8.8	10991	36.2	1011908	2219.9
vpm2	297	0.9	1619	0.4	1077	2.2	459	4.3
aflow40b	347845	2067.6	491380	2342.5	427125	2.2 %	1321287	4.0 %
danoint	1158489	4856.1	778939	4975.1	330296	3.5 %	683171	2.0 %
fast0507	1350	395.2	2941	555.0	1380	2407.0	7770	1.6 %
glass4	7335667	79.6 %	8939059	6595.8	322356	125.0 %	1729411	95.8 %
harp2	22481616	<0.1 %	316170	144.8	5732001	0.1 %	2589310	3448.6
mzzv11	3376	547.6	498	90.8	1545	0.6 %	2899	4.8 %
mzzv42z	761	302.9	298	33.5	1369	5243.8	5500	3.9 %
net12	5501	2139.0	2603	28.3 %	1411	—	12191	22.3 %
noswot	1510640	6110.8	8158083	4.7 %	495596	238.4	5713896	2.8 %
opt1217	3833790	16.3 %	1	0.1	3558191	16.6 %	20584953	17.7 %
set1ch	27	1.4	330	0.2	8825	18.9	1317890	0.5 %
tr12-30	909033	2600.7	212451	294.2	1259733	4433.7	506441	1.3 %
Geom. Mean	4101	58.0	2455	11.3	4224	136.5	12609	183.8
Solved Instances	33		34		29		25	
≥ 10 % faster	–		27		2		5	
≥ 10 % slower	–		6		30		29	

Table 1. Results of four 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.

We tested SCIP 1.00 running on a 3.00 GHz Intel Xenon with 8 GB RAM and 4 MB cache, using Cplex 11.0 [23] as underlying LP solver. We set a time limit of 2 hours and a memory limit of 4 GB. As a comparison we applied the same test with Cplex 11.0 as stand-alone MIP solver, with SCIP 1.00 using SoPlex 1.3.2 [42] to solve the LPs, and CBC 2.0 with CLP 1.6 [17] as LP solver. We used the provided default settings for all solvers. As test set we chose the 60 instances of the Miplib 2003 [4]. We left out the instances `arki001`, `protfold`, 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 four solvers. The first part of Table 1 lists the instances which were solved to optimality by all solvers, the second part those which were solved by at least one solver, Table 2 those for

Name	SCIP/CPLEX Nodes	SCIP/CPLEX Gap	CPLEX Nodes	CPLEX Gap	SCIP/SOPLEX Nodes	SCIP/SOPLEX Gap	CBC/CLP Nodes	CBC/CLP Gap
a1c1s1	426057	15.8 %	491631	5.7 %	115512	20.7 %	143591	41.0 %
atlanta-ip	11342	5.5 %	4011	8.1 %	10	—	350	—
dano3mip	9911	22.8 %	5565	18.8 %	123	24.1 %	12898	30.5 %
ds	4512	486.6 %	5760	314.2 %	310	511.3 %	456	1482.5 %
liu	3146152	135.4 %	319976	102.1 %	347383	159.3 %	157480	206.4 %
mkc	2396228	1.3 %	140170	0.2 %	1022181	0.9 %	961565	2.5 %
momentum1	6221	20.5 %	23623	18.7 %	1276	—	5158	20.2 %
momentum2	6004	28.7 %	6144	28.7 %	1260	—	5529	152.4 %
momentum3	11	—	140	466.5 %	1	—	1	—
msc98-ip	10301	0.7 %	1996	12.1 %	67	—	324	—
nsrand-ipx	592996	6.5 %	234970	1.1 %	381553	8.8 %	661104	2.0 %
rd-rplusc-21	84288	>10 000 %	35562	>10 000 %	71	—	11795	—
roll3000	1180987	0.6 %	1253352	0.4 %	201728	1.2 %	133378	3.8 %
seymour	103485	2.2 %	146297	1.9 %	2829	11.5 %	33374	5.9 %
sp97ar	86939	3.4 %	210446	0.8 %	36063	4.6 %	180426	2.5 %
stp3d	8	—	20	—	3	—	1	—
swath	429024	19.1 %	262088	19.3 %	257953	26.8 %	2352638	40.7 %
t1717	2665	50.2 %	64721	60.4 %	898	37.0 %	13016	76.9 %
timtab2	3095502	78.4 %	1736172	52.5 %	2420114	63.1 %	639547	102.8 %
markshare1	46 M	5	31 M	4	52 M	6	42 M	6
markshare2	42 M	9	25 M	12	40 M	9	48 M	10

Table 2. Results of four MIP solvers on the MIPLIB 2003 (continued). For the `markshare` instances we report the upper bound instead of the primal-dual gap; the lower bound is zero in all cases.

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

There were 36 instances, given in Table 1, for which at least one solver was able to prove optimality within the time and memory limit. For these instances, the results are summarized at the bottom of the table. 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, respectively, than SCIP-CPLEX. Although SCIP supports the much more general concept of constraint integer programming, it is still competitive to state-of-the-art MIP solvers. On this test set, SCIP-CPLEX can solve only one instance less than CPLEX within the limits.

5 Using SCIP for Property Checking

One of the key technologies in the design of integrated circuits is the verification of the correctness of the design [24]. One important aspect of this process is the so-called *property checking problem*, which means to verify that certain expected inherent properties of the chip design hold.

Today's techniques validate these properties on the so-called *gate level* by transforming the properties into Boolean clauses and hence the property checking problem into a SAT instance. However, complex arithmetic operations like multiplication lead to SAT instances with quite 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 (RT) level*. The property checking problem at RT level can be formulated as CIP on bit vector variables $\varrho \in \{0,\dots,2^{w_\varrho-1}\}$ of width w_ϱ. The constraints $r^i = C_i(x^i, y^i, z^i)$ model the circuit operations.

For each bit vector variable ϱ, we introduce single bit variables ϱ_b, $b = 0,\dots,w_\varrho - 1$, with $\varrho_b \in \{0,1\}$, 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 the semantics as defined in [16].

5.1 CP 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, the domain propagation of the MULT constraint 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, involve reasoning that mixes bit- and word-level information.

5.2 IP Techniques

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

Table 3 shows the linearizations of the 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 $W = 16$ 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

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

Table 3. LP relaxation of circuit operations. l_ϱ and u_ϱ are the lower and upper bounds of a bit vector variable ϱ.

constraint involves additional variables y_n and r_n which are "nibbles" of y and r with $L = \frac{W}{2}$ bits.

No linearization is generated for the 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, 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 30 944 inequalities and 20 929 auxiliary variables.

5.3 SAT 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* [43] rule for generating conflict constraints. In addition to the *1-FUIP* conflict constraints we extract clauses from *reconvergence cuts* [43] in the conflict graph to support *non-chronological backtracking* [31].

5.4 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 0.90i using Cplex 10.0.1 [23] as LP solver. All calculations were performed on a 3.8 GHz Pentium-4 workstation with 2 GB

Prop	Meth	register width 5	10	15	20	25	30	35	40
muls	SAT	0.5	—	—	—	—	—	—	—
	CIP	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.3
neg_flag	SAT	0.1	100.0	—	—	—	—	—	—
	CIP	0.8	3.6	11.6	36.3	81.8	136.6	218.4	383.5
zero_flag	SAT	0.0	0.0	0.1	0.1	0.2	0.4	0.5	0.6
	CIP	2.3	0.6	1.6	4.0	6.2	10.7	15.6	379.7

Table 4. ALU properties. (time in seconds)

Property	Meth	variant A	B	C
g_checkgpre	SAT	22.2	57.6	29.1
	CIP	14.2	12.3	15.3
g2_checkg2	SAT	—	—	—
	CIP	213.9	204.8	257.6
g25_checkg25	SAT	0.0	2.4	2.5
	CIP	29.7	22.4	24.2
g3_negres	SAT	0.0	0.0	0.0
	CIP	0.7	0.0	0.0
gBIG_checkreg1	SAT	287.2	157.3	159.6
	CIP	170.0	7.0	8.6

Table 5. Biquad properties.

Layout	Meth	register width 6	7	8	9	10	11	12	13	14
booth	SAT	0.4	3.3	21.0	135.4	935.1	—	—	—	—
signed	CIP	21.3	70.1	318.7	384.2	904.1	1756.2	2883.7	4995.9	3377.9
booth	SAT	0.5	2.5	17.9	102.9	879.0	4360.4	—	—	—
unsgnd	CIP	15.7	51.7	269.1	911.3	1047.6	2117.7	2295.1	4403.4	7116.8
nonbth	SAT	0.4	3.4	21.8	134.1	1344.1	—	—	—	—
signed	CIP	12.8	31.2	100.6	265.9	569.8	690.8	1873.0	1976.3	4308.9
nonbth	SAT	0.3	1.8	16.5	83.1	909.6	5621.5	—	—	—
unsgnd	CIP	3.6	22.4	111.2	214.0	335.4	1040.1	1507.5	2347.7	4500.2

Table 6. Multiplier properties. (time in seconds)

RAM. In all runs, we used a time limit of 2 hours. For reasons of comparison, we also solved the instances with SAT techniques on the gate level. We used MiniSat 2.0 [18] 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 [34] in different representations), and *Multiplier* (gate level net lists for Booth and non-Booth encoded architectures of signed and unsigned multipliers).

Tables 4–6 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 is usually inferior to SAT for finding counter-examples. This is due to the much more involved procedures employed in the CIP approach.

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. http://opus.kobv.de/tuberlin/volltexte/2007/1611/.
3. T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33:42–54, 2005.
4. T. Achterberg, T. Koch, and A. Martin. MIPLIB 2003. *Operations Research Letters*, 34(4):1–12, 2006. http://miplib.zib.de.
5. 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.
6. G. Andreello, A. Caprara, and M. Fischetti. Embedding $\{0, \frac{1}{2}\}$-cuts in a branch-and-cut framework: A computational study. *INFORMS Journal on Computing*, 19(2):229–238, 2007.
7. D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem.* Princeton University Press, Princeton, 2006.
8. K. R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.
9. 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.
10. E. Balas. Facets of the knapsack polytope. *Mathematical Programming*, 8:146–164, 1975.
11. 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.
12. T. Berthold. Primal heuristics for mixed integer programs. Master's thesis, Technische Universität Berlin, 2006.
13. T. Berthold. Heuristics of the branch-cut-and-price-framework SCIP. ZIB-Report 07-30, Zuse Institute Berlin, 2007. To appear in Operations Research 2007.
14. 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.
15. 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.
16. R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *Proceedings of the IEEE VLSI Design Conference*, pages 741–746, 2002.
17. Computational infrastructure for operations research. http://www.coin-or.org.
18. N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Proceedings of SAT 2003*, pages 502–518. Springer, 2003.
19. 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.
20. 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.

21. 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.
22. J. N. Hooker and M. A. Osorio. Mixed logical/linear programming. *Discrete Applied Mathematics*, 96-97(1):395–442, 1999.
23. ILOG CPLEX. Reference Manual. http://www.ilog.com/products/cplex.
24. International technology roadmap for semiconductors, 2005. http://public.itrs.net.
25. 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.
26. E. L. Johnson and M. W. Padberg. Degree-two inequalities, clique facets, and biperfect graphs. *Annals of Discrete Mathematics*, 16:169–187, 1982.
27. A. Klar. Cutting planes in mixed integer programming. Master's thesis, Technische Universität Berlin, 2006.
28. A. N. Letchford and A. Lodi. Strengthening Chvátal-Gomory cuts and Gomory fractional cuts. *Operations Research Letters*, 30(2):74–82, 2002.
29. 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.
30. 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.
31. J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions of Computers*, 48:506–521, 1999.
32. H. Mittelmann. Decision tree for optimization software: Benchmarks for optimization software. http://plato.asu.edu/bench.html.
33. G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988.
34. Opencores. http://www.opencores.org.
35. M. W. Padberg, T. J. van Roy, and L. A. Wolsey. Valid inequalities for fixed charge problems. *Operations Research*, 33(4):842–861, 1985.
36. 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.
37. 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.
38. M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.
39. 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.
40. C. Timpe. Solving planning and scheduling problems with combined integer and constraint programming. *OR Spectrum*, 24(4):431–448, November 2002.
41. K. Wolter. Implementation of cutting plane separators for mixed integer programs. Master's thesis, Technische Universität Berlin, 2006.
42. R. Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus*. PhD thesis, Technische Universität Berlin, 1996.
43. L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.