Satisfiability modulo theories for verifying MILP certificates

Runtian Zhou
Davidson College,
Davidson, NC 28035
dazhou@davidson.edu

Haoze Wu
Stanford University,
Stanford, CA 94305
haozewu@stanford.edu

Hammurabi Mendes
Davidson College,
Davidson, NC 28035
hamendes@davidson.edu

Jonad Pulaj
Davidson College,
Davidson, NC 28035
jopulaj@davidson.edu

Abstract

Correctness of results returned from mixed-integer linear programming (MILP) solvers is highly desirable, particularly in the context of applications such as hardware verification, compiler optimization, or machine-assisted theorem proving. To this end, VIPR is the first recently proposed certificate format for answers produced by MILP solvers. We design a checker for VIPR certificates by encoding the format's inference rules as Satisfiability Modulo Theories (SMT) instances and show the equivalence of the certificates' correctness and the satisfiability of the corresponding SMT instances. In addition, we test the viability of this approach on benchmark instances found in the literature with the Z3 and cvc5 solvers.

1 Introduction

Computational mixed-integer linear programing (MILP) is widely considered a successful blend of algorithmic improvement with advancements in hardware and compilers (Bixby 2012), where current state-of-the-art MILP solvers can solve problems with millions of variables and constraints (Koch et al. 2022).

Demand for MILP solvers is largely driven by applications in industry, hence underlying numerical computations typically rely on floating-point arithmetic for efficiency. For most such applications, in order to achieve a high degree of numerical stability, floating point computations are combined with numerical error tolerances. However, MILP solvers are also used in experimental mathematics to find counterexamples (Lancia, Pippia, and Rinaldi 2020; Pulaj 2020) or provide numerical evidence (Stolee 2013; Kenter and Skipper 2018). For theorem proving correctness is tantamount and additional safeguards are needed to counter the use of inexact floating-point arithmetic and/or programming or algorithmic errors. The best known example in this case is the verification in Isabelle/HOL of the infeasibility of thousands of linear programs involved in the proof of Kepler's conjecture (Hales 2005).

Another set of tools that have been successfully deployed in automated theorem proving tasks are Boolean satisfiability (SAT) solvers and Satisfiability modulo theories (SMT) solvers (De Moura and Bjørner 2009). For example, SAT solvers have had spectacular success in settling a number of long-standing mathematical problems including the Erdos discrepancy conjecture (Konev and Lisitsa 2015), the Boolean Pythagorean triples conjecture (Heule, Kullmann, and Marek 2016), and the determination of the fifth Schur number (Heule 2018). On the other hand, SMT solvers are increasingly used as subroutines for proof assistants (Böhme and Nipkow 2010; Ekici et al. 2017) to automatically resolve proof goals. In those applications, SAT/SMT solvers are often required to produce proofs, which can be independently checked by proof checkers to guarantee the correctness of the solver outputs. One theory particularly relevant to MILP is the theory of Linear Real Arithmetic, for which proof production is supported in state-of-the-art SMT solvers such as Z3, cvc5, and MathSAT5.

Although verification of answers returned by MILP solvers is not as well-established as in analogous solvers in the SAT/SMT community, significant steps forward have been made. VIPR is the first, recently proposed branch-and-cut certificate format for MILP instances, designed with simplicity in mind, which is composed of a list of statements that can be sequentially verified using a limited number of inference rules (Cheung, Gleixner, and Steffy 2017; Eifler and Gleixner 2023). This certificate format was used in the proof of special cases of Chvátal's conjecture (Eifler, Gleixner, and Pulaj 2022), and in the proof of the 3-sets conjecture (Pulaj 2023). The VIPR certificate format is currently implemented in the latest versions of exact SCIP (Cook et al. 2011), together with a C++ checker for the corresponding certificates.

Our motivation for this work is leveraging advances in SMT solvers for the verification of certificates for MILP instances. To this end we design a checker for VIPR certificates by encoding the format's inference rules as SMT instances. In addition, we implement a straightforward transformation from MILP instances to SMT instances. We note that tools like SMTcoq (Ekici et al. 2017) may facilitate further verification of answers produced by SMT solvers in interactive theorem provers, thus providing the highest levels of confidence in certificates for MILP instances. All associated code with this project is freely available on GitHub.

In particular, our main contributions are:

- We describe the logic of transforming MILP instances in MPS format (Cplex 2009) into SMT instances in SMT2 format (Barrett et al. 2010), and more importantly we describe the logic of transforming VIPR certificates into SMT instances in SMT2 format.

- We show the equivalence of VIPR certificates and the transformed SMT instances, and evaluate the viability of our certificate checker based on benchmarks instances used in the literature (Cheung, Gleixner, and Steffy 2017).

The rest of this paper is organized as follows. Section II presents the logic of transformation from MILP instances and VIPR certificates to SMT instances. Section III proves the equivalence of VIPR certificates and the transformed SMT instances. Section IV presents an evaluation of our design tested on known benchmarks. Section V concludes the paper, and the appendix contains more proof details of the claims in Section III.

2 Methodology

Transforming MILP instances to SMT instances

As already mentioned in the Introduction, for suitable applications in the MILP domain where additional safeguards are warranted, translating from MILP instances to SMT instances may be advantageous. However, to the best of our knowledge, we are not aware of any such general translation tools that are publicly available.

Two common file formats for MILP problems are the LP file format, which is human-readable and analogous to the algebraic form of linear constraints, and the MPS file format, which is more machine-friendly with higher precision for constraint coefficients. In MPS format, constraints are specified in a matrix, where each column corresponds to a variable and each row corresponds to a constraint. For each variable, its coefficient at each row is specified, otherwise the coefficient is assumed to be zero. In our implementation of the the transformation, we input MILP instances in MPS format and output equivalent SMT instances in SMT2 format.

At a high level, our transformation from MPS to SMT2 is straightforward: we use `declare-fun` statements to declare variables and `assert` statements to capture constraints or bounds specified in the original MPS file. Although it is likely that further optimizations could improve the performance of transformed instances in SMT solvers, our simple translation provides a useful baseline for comparison. It enables us to experimentally examine the advantages of checking the correctness of a VIPR certificate for a MILP instance via an SMT solver versus directly solving the translated MILP instances in an SMT solver.

MPS files encode most instance information in four sections: ROWS, RHS, COLUMNS, and BOUNDS. The ROWS section specifies the type of each constraint (>=, <=, or =). The RHS section specifies the right hand side value of each constraint, which is assumed to be a constant. The COLUMNS section specifies for each variable its corresponding coefficient in each constraint. The BOUNDS section specifies the upper and lower bounds of each variables.

Figure 1 presents the logic of transformation from these four general sections. First, each variable specified in the COLUMNS section will be encoded as a `declare-fun` statement. Furthermore, sections ROWS, RHS and COLUMNS encode information on each constraint in the corresponding MILP problem, and each such constraint is transformed into an `assert` statement. Finally, sections COLUMNS and BOUNDS encode lower and upper bounds for each variable, and each such bound is transformed into an `assert` statement.[1]

Transforming VIPR certificates to SMT instances

Next we describe the design of the checker for VIPR certificates. At a high level, a VIPR certificate contains two main parts: one that describes the original MILP problem (constraints, variable types, etc.) and an inference part that describes the reasoning of the certificate (assumptions and inferenced constraints). As part of our checker design we declare a fixed variable in SMT2 format for each constant (coefficient) specified in the corresponding VIPR certificate, so that the symbolic computation of the inferenced reasoning can be wholly captured in an SMT solver.

When transforming the system part into an SMT instance, unlike the transformation from MPS format, we don't use `assert` statements to determine whether each constraint of the original MILP instance holds. Instead we encode each coefficient in a given constraint as a fixed variable. Furthermore when transforming the inference part, we also encode every coefficient in each inferenced constraint. Finally, the reasoning part of each inferenced constraint is transformed into several `assert` statements which check inequalities between constants.

In short, our transformation ensures that SMT solvers are only burdened with the work involved in the symbolic computation of the reasoning of the certificate. Thus we seek to reduce the computational burden to SMT solvers by mainly checking formulas with fixed variables. [2]

A standard VIPR certificate file contains 7 sections: VAR, INT, CON, RTP, SOL, OBJ, and DER. Figure 2 provides an overview of the logic of transformation from these sections to SMT2 statements. The VAR and INT sections are transformed into `assert` and `declare-fun` statements specifying types of variables in the original MILP problem. The CON section is transformed into `assert` and `declare-fun` statements specifying type, rhs (right hand side value), and coefficients of each constraint in the original MILP problem. The SOL section is transformed into `assert` and `declare-fun` statements specifying variable assignment in each solution given by the MILP solver, and `assert` statements verifying that these variable assignments do satisfy the constraints specified in the CON section. The OBJ section is transformed into `assert` and `declare-fun` statements specifying the coefficients of each variable in the objection function. The DER section is transformed into `assert` and `declare-fun` statements specifying the type, rhs, and coefficients used in each inferenced (derived) constraint and the reasoning of that constraint. In addition,

[1] Our implementation can also handle non-standard statements in MPS format, such as the RANGE section and the INDICATOR section.

[2] A relatively expensive operation we encode in our checker is Chvátal-Gomory rounding. Since the SMT2 format doesn't natively provide a rounding operation, we enforce it by encoding several `assert` statements.

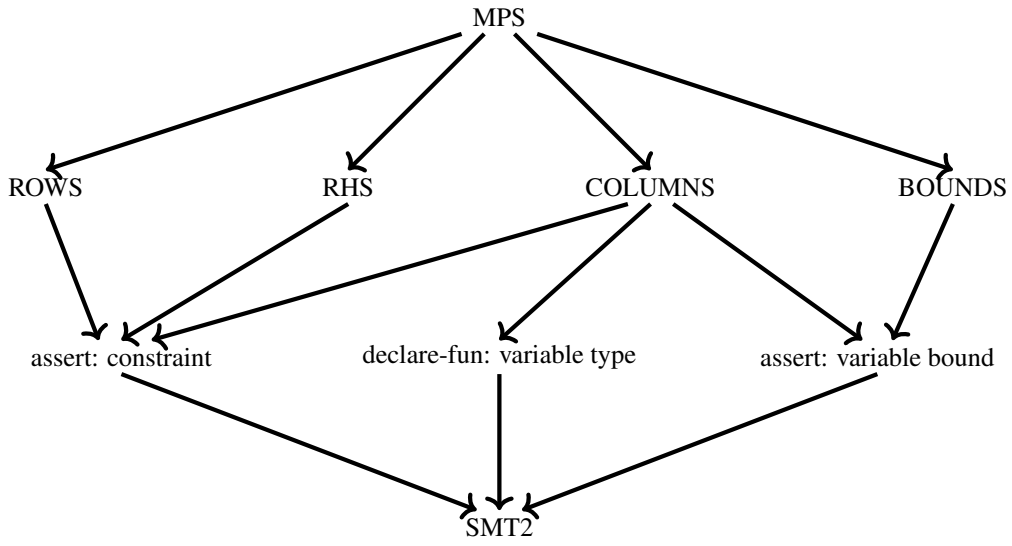

Figure 1: Transformation from MILP instance (MPS format) to SMT instance (SMT2 format)

RTP, SOL, OBJ, and DER sections together contribute to the `assert` statement specifying the lower and upper bounds of the optimum value of the objective function, when one exists.[3]

Next we provide more details with respect to the tranformations of each sections of the VIPR certificates. The VAR and INT sections specify the type (real or integer) of variables in the original MILP problem. The type of each integer is encoded as a boolean variable with fixed value, using a `declare-fun` statement and an `assert` statement. Let's take a look at an example of the VAR and INT sections in a VIPR certificate.

```
VAR 2
  x y
INT 1
    1
```

The VAR section shows that the original MILP problem has two variables, x and y, and our checker will index starting from 0. Thus x will be interpreted as x_0 and y as x_1. The INT section shows that only the variable with index 1 is an integer variable. Thus x_1 is an integer variable and x_0 is a real variable. This information is encoded as:

```
(declare-fun is_intx0 () Bool)
      (assert (not is_intx0))
(declare-fun is_intx1 () Bool)
           (assert is_intx1).
```

where is_intx0 is a boolean variable representing whether x_0 is an integer variable or not. Notice that VIPR does not distinguish boolean variables from integer variables or bound constraints from general constraints. The OBJ section specifies the coefficients of variables in the objective function. Each coefficient is encoded as a real variable in SMT2 format using a `declare-fun` statement and an `assert` statement. Let's take a look at an example of the OBJ section in a VIPR certificate:

```
OBJ min
  1 0 2.
```

The first number of the second line, 1, means there is only one variable in the objective function with a non-zero coefficient. The next number, 0, is the index of this variable. The next number, 2, is the coefficient of this variable. Thus the original MILP problem is a minimization problem and its objective function is $2x_0$. In SMT2 format, this information is encoded as:

```
(declare-fun obj0 () Real)
    (assert (= obj0 2.0)).
```

We use set structures in c++ to store the indexes of variables with non-zero coefficients in the objective function.

The RTP section specifies the result the result of the MILP solver. It either states the system is infeasible, or it gives the upper and lower bounds of the objective function. These information will be stored during the transformation for later use and will not be encoded into SMT2 format directly.

The CON section specifies the constraints in the original MILP problem (including bounds of variables). Here is an example line in the CON constraint:

```
CX G 1 2 0 2 1 3.
```

The first term, CX, is the name of the constraint. Constraints (including original constraints and inferenced constraints) in a VIPR certificate will be given index starting from 0 in the transformation. If this CX is the first line of CON section, then it will be referred to as c_0, meaning the first constraint,

[3]For further details on the VIPR format, please refer to (Cheung, Gleixner, and Steffy 2017).

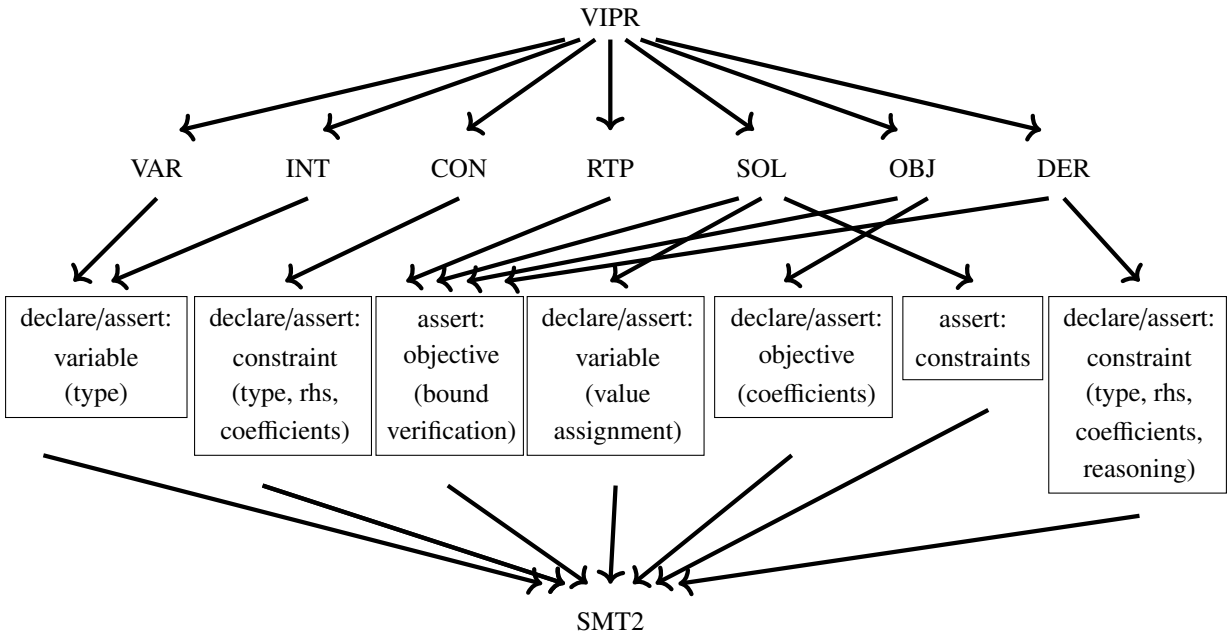

Figure 2: Transformation from VIPR format to SMT2 format

and the name CX will be disregarded. The next term, G, means c_0 is a >= constraint. In SMT2 format this is encoded as:

```
(declare-fun cs0 () Real)
    (assert (= cs0 1.0))
```

where $cs0$ specifies the type of the constraint with index 0. If it is a >= constraint, $cs0$ is assigned value 1.0. If it is a <= constraint, $cs0$ is assigned value -1.0. If it is an = constraint, $cs0$ is assigned value 0.0. This information will be used later to check the validity of inferenced constraints as linear combinations of other valid constraints.

The next term, 1, is the right hand side value of this constraint. In SMT2 format, this is encoded as:

```
(declare-fun crhs0 () Real)
    (assert (= crhs0 1.0)).
```

The next term, 2, means that there are two variables in this constraint with non-zero coefficients. The last four terms mean that x_0 has coefficient 2 and x_1 has coefficient 3. In SMT2 format, this is encoded as

```
(declare-fun c0x0 () Real)
    (assert (= c0x0 2.0))
(declare-fun c0x1 () Real)
    (assert (= c0x1 3.0)).
```

Thus constraint $c1$ corresponds to inequality $2x_0 + 3x_1 \geq 1$.

So far, we have outlined how the information of the original MILP problem is encoded into SMT2 format in our checker. The last two sections, SOL and DER, contain crucial information for the validity of a VIPR certificate. Together they ensure the verification of the MILP solver reported results in the RTP section. We'll discuss more detailed mechanics of this transformation in the next section and the Appendix.

3 Validity of VIPR transformation

The design of the derivations in the VIPR certificates, which in turn aim to verify the result returned by a MILP solver, is based on well-understood principles in the MILP domain such as the branch-and-cut method and split cuts (Cheung, Gleixner, and Steffy 2017). The VIPR certificate format "flattens" the associated branch-and-bound tree explored by MILP solvers and renders it into a series of statements (corresponding to lines) that can be verified sequentially. We formalize this in the following proposition. We say that the SOL section of the VIPR certificate format is *valid* if it contains at least a solution that satisfies all constraints in the CON section and satisfies the bounds in the RTP section, or it contains no solutions and the RTP section reports infeasibility. Furthermore we say that the DER section of the VIPR certificate format is *valid* if every listed constraint in the section is derived[4] from previously listed constraints in the section.

Proposition 1. *The result specified in the RTP section of a VIPR certificate is valid, if and only the SOL and DER sections are valid.*

Therefore, in order to show an equivalency (in terms of correctness) between a VIPR certificate and the analogous transformation into an SMT instance, it suffices to show the

[4]We discuss the details of such derivations in the Appendix when examining the transfomation that corresponds to Lemma 2.

test set	inst	chk time (s)	transformation time (s)	cvc5 time (s)	vipr size (MB)	SMT2 size (MB)	size ratio
easy-all	46	155.6	522.9	N\A	711.3	14567.0	17.6
easy-cvc5-solved	31	10.2	30.8	859.4	101.7	1229	14.1
easy-cvc5-memout	15	456.0	1631.8	N\A	1971.1	42132.1	24.9
hard-all	9	34.6	150.5	N\A	378.0	4422.6	11.3
hard-cvc5-solved	7	12.6	59.1	1176.0	147.6	1619.1	8.0
hard-cvc5-memout	2	112.0	470.3	N\A	1185.0	14234.5	22.7
test set	inst	chk time (s)	transformation time (s)	Z3 time (s)	vipr size (MB)	SMT2 size (MB)	size ratio
easy-Z3-solved	32	15.3	50.9	15602.0	177.4	1971.8	13.9
easy-Z3-memout	14	476.2	1700.3	N\A	1931.5	43355.8	25.9
hard-Z3-solved	5	0.9	12.1	34.9	9.2	94.6	6.4
hard-Z3-memout	4	76.8	323.5	N\A	839.0	9832.5	17.3

Table 1: Aggregated computational results over 55 instances from vipr author (average value for each criteria).

equivalency of the transformation of the SOL and DER sections. More specifically, given the SOL and DER sections of a given instance, we can transform then into a series of statements in SMT2 format, and we claim that these statements will not generate a contradiction if and only if the SOL and DER sections are valid.

To this end, we will treat the validity of each section separately in the following two lemmas.

Lemma 1. *The SOL section is valid, if and only if the transformed statements in SMT2 format do not yield a contradiction.*

For details of this lemma, please refer to the Appendix.

Lemma 2. *The DER section is valid, if and only if the transformed statements in SMT2 format do not yield a contradiction.*

For details of this lemma, please refer to the Appendix.

Theorem 1. *The result specified in the RTP section of a VIPR certificate is valid, if and only if the transformed statements in SMT2 format do not yield a contradiction.*

Theorem 1 follows directly from Proposition 1, Lemma 1, and Lemma 2.

4 Experiments

In this section, we evaluate the proposed proof-checking workflow on MILP benchmarks studied by the authors of VIPR (Cheung, Gleixner, and Steffy 2017). We focus on each of the benchmarks that meets the following criterion: 1) a certificate (of either infeasibility or both lower and upper bounds) was produced; 2) the corresponding MPS (i.e., a standard format for MILP problem) file is publicly available. This amounts to 55 benchmarks in total. Following the strategy in the original paper, we partition the benchmark set into "easy" and "hard".

Our implementation includes 1) a MPS2SMT transformer that transforms MPS file format to the SMT-LIB format; 2) a VIPR2SMT transformer that transforms the VIPR proof format to the corresponding SMT problem in SMT-LIB format. Both transformers were written in C++. The former contains 655 lines of code (LOC), and the latter contains 570 LOC. Our implementation does not use any floating-point operations. In contrast, the existing VIPR proof checker (Cheung, Gleixner, and Steffy 2017) is written in 1685 lines of C++ code, and uses arithmetic operations that are not verified. This gives us reason to believe that our implementation is a trustworthy alternative to the existing proof-checker. We will make our tool publicly available.

We compare the following three strategies:

1. Checking the VIPR proof with the original proof checker (Cheung, Gleixner, and Steffy 2017);
2. Translating the VIPR proof to a SMT query, and solve with a proof-producing SMT solver;
3. Translating the original MILP problem to a SMT query, and solve with a proof-producing SMT solver.

We consider two proof-producing SMT solvers, Z3 (De Moura and Bjørner 2008) and cvc5 (Barbosa et al. 2022), for solving the SMT queries. All experiments were conducted on a cluster of Intel(R) Xeon(R) Gold 6226R CPU (2.9Ghz). Each job was given one processor and 256G of RAM. The results are shown in Table 1.

Among the 55 SMT2 files transformed from the VIPR certificates, 17 of them forces cvc5 to require more than 256 GB and 18 of them forces Z3 reports memory error. Specifically, every transformed SMT2 file among the 46 easy tests with size less than 15 gigabytes can be solved by cvc5 and Z3. For the all test cases solved by cvc5 and Z3, they give the same results as the results from the $c++$ checker provided in (Cheung, Gleixner, and Steffy 2017), showing that all VIPR verificates are valid. Aggregated statistics of the computational performance is given in Table 1.

In Table 1, the inst column shows the number of tests in each test set. The chk time column shows the average checking time of the c++ checker, provided in (Cheung, Gleixner, and Steffy 2017). The transformation time column shows the average time our transformation program used to transform VIPR certificates into SMT2 format. The Z3/cvc5 time column shows the average time used by Z3/cvc5 solver to solve the generated SMT2 file. The size ratio column shows the average ratio of VIPR certificate size to SMT2 file size.

For rows labeled "memout", it contains tests where cvc5 solver requires more than 256 GB ram and Z3 reports memory error.

We observe that the size ratio arribute of each test set does not exhibit significant difference. This implies the transforming from VIPR certificates to SMT2 format generates a linear increase in the file size, as expected. We also observe that all tests with VIPR file size less than 200 MB can be solved by cvc5 and Z3 solver. Thus, our transformation techniques can be used to verify considerable amount of VIPR certificates.

We also performed these 55 tests on our program of direct transformation from MILP problems in MPS format to SMT problems in SMT2 format. In comparison to transformation from VIPR certificates, directly tranforming MPS format to SMT2 format is much faster, since the major work performed is parsing and no special data structures are used. For SMT2 files transformed from VIPR certificates of the 55 tests, 38 can be solved byc cvc5 in 4 hours and 26 can be solved by Z3 solver in 4 hours. In comparison, for the SMT2 files transformed from MPS format of the 55 tests, only 18 can be solved by cvc5 in 4 hours and 9 can be solved by Z3 solver in 4 hours. As expected, the time needed for cvc5 and Z3 solver to solve the SMT2 file transformed from MPS format is often much longer than the time needed for SMT2 files transformed from VIPR certificates, presenting the need for VIPR certificates.

5 Conclusions

In this work we seek to leverage advances in SMT solvers to verify certificates for MILP solvers. Thus we design a VIPR certificate checker that transforms the logic of the VIPR into an equivalent SMT2 format. We test the viability of our checker on a benchmark instances, and compare it to a straightforward translation of the MPS format to SMT2 format. For future directions we seek to introduce further optimizations for the MPS2SMT translation, and possibly leverage SMT solvers' incremental techniques to improve the efficiency of the VIPR2SMT direction.

References

Barbosa, H.; Barrett, C.; Brain, M.; Kremer, G.; Lachnitt, H.; Mann, M.; Mohamed, A.; Mohamed, M.; Niemetz, A.; Nötzli, A.; et al. 2022. cvc5: A versatile and industrial-strength smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 415–442. Springer.

Barrett, C.; Stump, A.; Tinelli, C.; et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, 14.

Bixby, R. E. 2012. A brief history of linear and mixed-integer programming computation. *Documenta Mathematica* 2012:107–121.

Böhme, S., and Nipkow, T. 2010. Sledgehammer: judgement day. In *Automated Reasoning: 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings 5*, 107–121. Springer.

Cheung, K. K.; Gleixner, A.; and Steffy, D. E. 2017. Verifying integer programming results. In *Integer Programming and Combinatorial Optimization: 19th International Conference, IPCO 2017, Waterloo, ON, Canada, June 26-28, 2017, Proceedings 19*, 148–160. Springer.

Cook, W.; Koch, T.; Steffy, D. E.; and Wolter, K. 2011. An exact rational mixed-integer programming solver. In *Integer Programming and Combinatoral Optimization: 15th International Conference, IPCO 2011, New York, NY, USA, June 15-17, 2011. Proceedings 15*, 104–116. Springer.

Cplex, I. I. 2009. V12. 1: User's manual for cplex. *International Business Machines Corporation* 46(53):157.

De Moura, L., and Bjørner, N. 2008. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 337–340. Springer.

De Moura, L., and Bjørner, N. 2009. Satisfiability modulo theories: An appetizer. In *Brazilian Symposium on Formal Methods*, 23–36. Springer.

Eifler, L., and Gleixner, A. 2023. Safe and verified gomory mixed integer cuts in a rational mip framework. *arXiv preprint arXiv:2303.12365*.

Eifler, L.; Gleixner, A.; and Pulaj, J. 2022. A safe computational framework for integer programming applied to chvátal's conjecture. *ACM Transactions on Mathematical Software (TOMS)* 48(2):1–12.

Ekici, B.; Mebsout, A.; Tinelli, C.; Keller, C.; Katz, G.; Reynolds, A.; and Barrett, C. 2017. Smtcoq: A plug-in for integrating smt solvers into coq. In *Computer Aiåded Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II 30*, 126–133. Springer.

Hales, T. C. 2005. A proof of the kepler conjecture. *Annals of mathematics* 1065–1185.

Heule, M. J.; Kullmann, O.; and Marek, V. W. 2016. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *International Conference on Theory and Applications of Satisfiability Testing*, 228–245. Springer.

Heule, M. 2018. Schur number five. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Kenter, F., and Skipper, D. 2018. Integer-programming bounds on pebbling numbers of cartesian-product graphs. In *International Conference on Combinatorial Optimization and Applications*, 681–695. Springer.

Koch, T.; Berthold, T.; Pedersen, J.; and Vanaret, C. 2022. Progress in mathematical programming solvers from 2001 to 2020. *EURO Journal on Computational Optimization* 10:100031.

Konev, B., and Lisitsa, A. 2015. Computer-aided proof of erdős discrepancy properties. *Artificial Intelligence* 224:103–118.

Lancia, G.; Pippia, E.; and Rinaldi, F. 2020. Using integer programming to search for counterexamples: A case study. In *International Conference on Mathematical Optimization Theory and Operations Research*, 69–84. Springer.

Pulaj, J. 2020. Cutting planes for families implying frankl's conjecture. *Mathematics of Computation* 89(322):829–857.

Pulaj, J. 2023. Characterizing 3-sets in union-closed families. *Experimental Mathematics* 32(2):350–361.

Stolee, D. 2013. A linear programming approach to the manickam-miklos-singhi conjecture.

Appendix

In this section we provide justification for the two lemmas used to claim to the equivalence of the SMT transformation of the VIPR certificate format. We only argue for the transformation in one directions (from VIPR to SMT) as the other direction is obvious once the proposed transformation is understood.

Next we examine the transformation corresponding to Lemma 1. Each line of the SOL section specifies a set of value assignments for variables in the original MILP instance. It is valid if this value assignments satisfy the constraint specified in the CON section.

The following is a simple example of CON and SOL transformation. Below we display the portion of the VIPR certificate we seek to convert to an SMT2 statement.

```
CON 1 0
CX G 1 2 0 2 1 3
SOL 1
opt 1 1 1
```

The first number after CON is the number of constraints specified in the CON seciton. The second number is the number of bound constraints. However, the VIPR format does not distinguish bound constraints from general constraints, so we disregard this number. Thus the original MILP problem has only one constraint. As explained in Section II, this constraint is

$$2x_0 + 3x_1 \geq 1.$$

The first number after SOL is the number of solutions provided by the MILP solver. Notice that if the result of MILP solver presented in the RTP section is infeasible, then SOL must contain no solution. In our example, there is one solution to be verified, as specified in the next line. The first term, "opt", is the name of this solution. The next number is the number of variables that are not assigned value 0 in this solution. The next two number specify the index of this variable and its value assignment. Then, this information is transformed into SMT2 format as

```
(declare-fun sol0x1 () Real)
(assert (= sol0x1 1.0)).
```

Notice that solutions are also given indices start from 0. Thus these two statements mean x_1 is given value 1.0 in the solution with index 0. Next, we need to verify this value assignment satisfies the constraints in the CON section. In SMT2 format, this is encoded as

```
(assert (>= (* c0x1 sol0x1) crhs0))
```

. Finally, in a minimization problem, at least one solution needs to verify the upperbound specified in the corresponding RTP section. Suppose the corresponding RTP section is

```
RTP range 0 0.
```

Then, since no variable with non-zero coefficient in the objective function is assigned non-zero value in this solution, the objective function has value 0 in this solution. Thus in SMT2 it's encoded as

```
(assert (>= 0.0 0.0)).
```

If there are several solutions, to check that at least one of them verifies the upperbound specified in RTP section, we will use the "or" operator in SMT2 format to encode several inequalities in one assert statement. This concludes the trasformation associated with Lemma 1.

Next we examine the transformation corresponding to Lemma 2. Each line in the DER section contains three parts: the first part specifies the constraint in the same way as in the CON section; the second part is the reasoning of this constraint, specified within {}; the third part is largest index of later derived constraints that use the current constraint in its reasoning part. This index is used only to free up space during the transformation, and will not be encoded into SMT2 format. There are four possible reasoning formats within {} and we will explain transformation from each of them.

If the reasoning format is { asm }, then this constraint is an assumption. On a high level, the current constraint is used to open a branch for later reasoning and does not rely on any other constraints. The set structure of c is used to store indices of constraints that are assumptions. Each constraint also has a set storing the indices of assumptions it relies on. We define the only assumption used by an assumption constraint to be itself, and each constraint in the CON section uses no assumptions. Since there is no computation in the reasoning part, such a line will be encoded in the same way as any line in the CON section. For example,

```
AX L 0 1 0 1 { asm } -1
```

is transformed into

```
(declare-fun cs3 () Real)
(assert (= cs3 -1.0))
(declare-fun crhs3 () Real)
(assert (= crhs3 0.0))
(declare-fun c3x0 () Real)
(assert (= c3x0 1.0)).
```

If the reasoning format is { lin ...}, it means that the current constraint is implied by a linear combination of previous constraints. Consider the following example:

```
CQ G 1 0 { lin 3 0 1 3 -2 5 -3 } 12.
```

Suppose this is the 6th constraint in the VIPR file. The first number after "lin" means that there are three constraints

used in this linear combination. The next 3 pairs of numbers each specifies the index of an used constraint and the multiplier of it. Thus, the reasoning of c_6 is the linear combination $c_0 - 2c_3 - 3c_5$. The constraint part is encoded in the same way as any line in the CON section, but we need to add several other statements to check the validity of the reasoning part.

First, we need to make sure that the reasoning part only uses constraints already specified, i.e. the indices of the reasoning part should be less than current index 6:

```
(assert (< 0 6))
(assert (< 3 6))
(assert (< 5 6)).
```

Second, we need to make sure the linear combination indeed "forces" an inequality or equality. There are only three possibilities: the addition of equalities forces equality; the addition of >= forces >=; the addition of <= forces <=. Note that a negative multiplier reverse the direction of an inequality. Remember that we used $0.0, -1.0, 1.0$ to represent the sign of a constraint, i.e. $=, <=, >=$. Therefore, if a multiplier times the value of the sign gives negative value, then it becomes a <= inequality.

Given this, we can use three boolean variables to represent whether the linear combination forces an $=, <=, >=$ constraint:

```
(declare-fun cleq6 () Bool)
(declare-fun cgeq6 () Bool)
(declare-fun ceq6 () Bool)
(assert (= ceq6 (and (= (* 1.0 cs0) 0.0) (= (* -2.0 cs3) 0.0) (= (* -3.0 cs5) 0.0))))
(assert (= cleq6 (and (<= (* 1.0 cs0) 0.0) (<= (* -2.0 cs3) 0.0) (<= (* -3.0 cs5) 0.0))))
(assert (= cgeq6 (and (>= (* 1.0 cs0) 0.0)  (>= (* -2.0 cs3) 0.0) (>= (* -3.0 cs5) 0.0))))
```

where ceq6, cleq6, cgeq6 represent whether the linear combination in the reasoning part of c_6 forces equality, less or equal, or greater or equal. At least one of them has to be true so that the linear combination is valid, and we assign sign to this reasoning constraint correspondingly (use rs6 to represent $=, <=$ or $>=$):
(assert (ite ceq6 (= rs6 0.0) (ite cleq6 (= rs6 -1.0) (ite cgeq6 (= rs6 1.0) xfalse))))
where xfalse is always false by a statement at the beinning of the transformed SMT2 file.

Given that this linear combination is valid, let's calculate the coefficients of variables and right hand side value of the (reasoning) constraint forced by this linear combination:

```
(declare-fun r6x0 () Real)
(assert (= r6x0 (+ (* 1.0 c0x0) (* -2.0 c3x0) (* -3.0 c5x0))))
(declare-fun r6x1 () Real)
(assert (= r6x1 (+ (* 1.0 c0x1) (* -2.0 c3x1) (* -3.0 c5x1))))
(declare-fun beta6 () Real)
(assert (= beta6 (+ (* 1.0 crhs0) (* -2.0 crhs3) (* -3.0 crhs5))))
```

where r6x0 is the coefficient of x0 of this constraint, beta6 is the right hand side value of this constraint.

The last thing to check is that the reasoning constraint, if true, implies the current constraint (c_6).
In VIPR format, only 4 kinds of implication are allowed. Here is the description given by VIPR authors:

- A constraint that is $0 \geq \beta$ with $\beta > 0$, $0 \leq \beta$ with $\beta < 0$, or $0 = \beta$ with $\beta \neq 0$ is called an absurdity or falsehood. Such a constraint dominates any other constraint.
- The constraint $a^T x \geq \beta$ or $a^T x = \beta$ dominates $a'^T x \geq \beta'$ if $a = a'$ and $\beta \geq \beta'$.
- Similarly, the constraint $a^T x \leq \beta$ or $a^T x = \beta$ dominates $a'^T x \leq \beta'$ if $a = a'$ and $\beta \leq \beta'$.
- Finally, the constraint $a^T x = \beta$ dominates $a'^T x = \beta'$ if $a = a'$ and $\beta = \beta'$.

We can check whether at least one kind of implication holds with just one assert statement in SMT2 format, with structure like this:
(assert (or (reasoning constraint is absurdity) (and $a = a'$ (sign and right hand side value of reasoning and current constraints satisfy the three cases listed above)))))
For c_6 especially, this assertion is:

(assert (or (and (= r6x0 0.0) (= r6x1 0.0) (=> (= rs6 -1.0) (< beta6 0.0)) (=> (= rs6 1.0) (> beta6 0.0)) (=> (= rs6 0.0) (not (= beta6 0.0)))) (and (= r6x0 c6x0) (= r6x1 c6x1) (or (and (= beta6 crhs6) (= rs6 0.0) (= cs6 0.0)) (and (<= beta6 crhs6) (= cs6 -1.0) (<= rs6 0.0)) (and (>= beta6 crhs6) (= cs6 1.0) (>= rs6 0.0))))))

If the reasoning format is { rnd ...}, it means that the current constraint is implied by a linear combination of previous constraints, whose left hand side is an integer and the right hand side value is rounded to an integer. Consider the following example:

```
CE G 1 1 1 1 { rnd 1 9 1 } 11.
```

Suppose this is the 10th constraint in the VIPR certificate. Most information of this linear combination is encoded similar to the case of { lin ...}. In addition, we need to verify the reasoning constraint indeed can be rounded, i.e. the coefficients of real variables are 0, and the coefficients of int variables are integers. In SMT2 format, the type of all coefficients are originally set as real. To check that a coefficient is an integer, we simply assert another random integer variable to be equal to it under the to_real operator:
(declare-fun rndr10x0 () Int)
(assert (ite is_intx0 (= (to_real rndr10x0) r10x0) (= r10x0 0.0)))
where is_intx0 is the boolean variable representing whether x0 is an integer variable.

In a VIPR certificate, we only allow rounding operation when the reasoning constraint is >= or <=. If the reasoning constraint is equality, either the right hand side is already integer so there is no need to round, or the rounding is invalid. Thus,

```
(assert (or (= rs10 -1.0) (= rs10 1.0))).
```

Remember the right hand side value of this reasoning constriant is beta10, and we round it up or down depending on the sign of the reasoning constraint:

(declare-fun rndbeta10 () Int)

(assert (ite (= rs10 -1.0) (and (≤ (to_real rndbeta10) beta10) (> (to_real (+ rndbeta10 1)) beta10)) (and (≥ (to_real rndbeta10) beta10) (< (to_real (- rndbeta10 1)) beta10))))

(declare-fun brndbeta10 () Real)

(assert (= brndbeta10 (to_real rndbeta10)))

where brndbeta10 is the rounded right hand side value, with real type.

Finally, let's used the same assert statements as last case to verify that the rounded reasoning constraint implies the current constraint:

(assert (or (and (= r10x0 0.0) (= r10x1 0.0) (=> (= rs10 -1.0) (< brndbeta10 0.0)) (=> (= rs10 1.0) (> brndbeta10 0.0)) (=> (= rs10 0.0) (not (= brndbeta10 0.0)))) (and (= r10x0 c10x0) (= r10x1 c10x1) (or (and (= brndbeta10 crhs10) (= rs10 0.0) (= cs10 0.0)) (and (<= brndbeta10 crhs10) (= cs10 -1.0) (<= rs10 0.0)) (and (>= brndbeta10 crhs10) (= cs10 1.0) (>= rs10 0.0))))))

The last type of reasoning is the form { uns i_1 l_1 i_2 l_2 }. This case is a bit complicated. Notice that each time we introduce an assumption constraint, it can be interpreted as if we open a branch in reasoning. In contrary, this reasoning form, "uns", closes two branches and goes back in the reasoning track.

Consider the 12th constraint as an example:

```
CF G 1 0 { uns 6 5 8 7 } 13
```

where c_{12} is $0 \geq 0, i_1 = 6, l_1 = 5, i_2 = 8, l_2 = 7$.

The constraint is encoded in the same way as in other cases. We also need to check the constraints used in the reasoning have been specified before:

```
(assert (< 6 12))
(assert (< 5 12))
(assert (< 8 12))
(assert (< 7 12))
```

Then, in c code, we use the set structure to check that l_1, l_2 are indeed indices of assumptions and they are used by c_{i_1} and c_{i_2}, correspondingly.

Now we need to verify that c_{i_1}, c_{i_2} both imply c_{12} in the four criteria specified before, with similar assert statements:

(assert (or (and (= c6x0 0.0) (= c6x1 0.0) (=> (= cs6 -1.0) (< crhs6 0.0)) (=> (= cs6 1.0) (> crhs6 0.0)) (=> (= cs6 0.0) (not (= crhs6 0.0)))) (and (= c6x0 c12x0) (= c6x1 c12x1) (or (and (= crhs6 crhs12) (= cs6 0.0) (= cs12 0.0)) (and (<= crhs6 crhs12) (= cs12 -1.0) (<= cs6 0.0)) (and (>= crhs6 crhs12) (= cs12 1.0) (>= cs6 0.0))))))

(assert (or (and (= c8x0 0.0) (= c8x1 0.0) (=> (= cs8 -1.0) (< crhs8 0.0)) (=> (= cs8 1.0) (> crhs8 0.0)) (=> (= cs8 0.0) (not (= crhs8 0.0)))) (and (= c8x0 c12x0) (= c8x1 c12x1) (or (and (= crhs8 crhs12) (= cs8 0.0) (= cs12 0.0)) (and (<= crhs8 crhs12) (= cs12 -1.0) (<= cs8 0.0)) (and (>= crhs8 crhs12) (= cs12 1.0) (>= cs8 0.0))))).

Finally, let's verify that at least one of c_{l_1}, c_{l_2} is true. A VIPR certificate forces this with three properties:

- the left hand side of c_{l_1} and c_{l_2} are the same;
- the left hand side of c_{l_1} and c_{l_2} have to be integers, i.e. all coefficients of real variables are 0, all coefficients of int variables are integers;
- the sign and right hand side value of c_{l_1} and c_{l_2} have to be something like <= B and >= $B+1$ so that it covers all the cases. Namely, if the left hand side is any integer, then exactly one of c_{l_1} and c_{l_2} has to be true.

To check the first property:

(assert (= c5x0 c7x0))
(assert (= c5x1 c7x1)).

To check the second property:

(declare-fun rndr12x0 () Int)
(assert (ite is_intx0 (= (to_real rndr12x0) c5x0) (= c5x0 0.0)))
(declare-fun rndr12x1 () Int)
(assert (ite is_intx1 (= (to_real rndr12x1) c5x1) (= c5x1 0.0))).

Finally, to check the third property:

(declare-fun rndbeta12 () Int)
(assert (or (and (= cs5 -1.0) (= cs7 1.0) (= (to_real rndbeta12) crhs5) (= (+ crhs5 1.0) crhs7)) (and (= cs7 -1.0) (= cs5 1.0) (= (to_real rndbeta12) crhs7) (= (+ crhs7 1.0) crhs5))))

where (= cs5 -1.0) checks whether c_5 is less or equal constraint.

In addition to the properties verified above, we need to verify that the last constraint implies the lower or upper bound of the objective function specified in the RTP section. If RTP states that the MILP is infeasible, then we check that this constriant is absurdity.

To show a constraint is absurdity, we can show that it implies another absurdity. In our program, we show that if this constraint is true, then 0 = 1 is true.