"Proof Certification / Formal Verification for CP"

Tip ten Brink

1. Introduction

Constraint Programming (CP) is a paradigm often used to solve combinatorial problems. CP has a number of important characteristics:

- It employs "declarative" problem modelling, where the logical representation of the problem is separate from the actual method used for solving it
- It has a strict separation between methods to reduce the search space (through deduction referred to as "propagation") and methods to explore that search space efficiently

The general type of problem solved by most CP solvers is the constraint optimization problem (COP). The COP is a generilization of the constraint satisfaction problem (CSP), which is a decision problem that asks whether there exists a solution that satisfies the constraints. The CSP is NP-complete[1]. Note that if we can solve a CSP, if we assume our objective function is not too costly to compute, we can in many cases expect to also solve the related COP[2] In the worst case, these problems take exponential time. Even small problems can be computationally intensive and scaling up a problem, which we usually want to do for real-world problems, inevitiably leads to high computational costs. Solvers therefore generally use every tool available to them to squeeze out performance.

This includes many algorithmic optimizations, which make solvers increasingly complex. The methods they use include, but are not limited to, Lazy Clause Generation (LCG), search restarts, and efficient data structures such as the two-watched literal scheme. Furthermore, they are generally written in system programming languages (often C or C++), to allow for greater control over memory and highly efficient machine code due to the use of optimizing compilers.

All this leads to the fact that modern CP solvers contain many types of bugs. TODO: Sources for the occurrence of bugs. Let us consider a solver with idealized behavior that, when given a CSP, does one of the following things. We consider only a CSP instead of a COP based on our earlier discussion. The behaviors are as follows:

1. It crashes
2. It runs forever without returning a solution
3. It returns a solution
4. It returns that no solution exists (it is unsatisfiable)

Other behaviors are possible, for instance if it contains malicious code and infects a user's computer. Many solver also have many other additional outputs, but these behaviors should

[1] See e.g. [1]. To see it is clearly at least NP-hard, consider that a CSP with only Boolean variables and clausal constraints is equivalent to SAT.

[2] This is not the case in general, but if we are talking about a reasonably well-behaved finite domain (e.g. in many discrete problems), we can find the solution to the optimization problem in polynomial time with e.g. binary search.

cover most of the behavior related to solving the actual problem and will be useful to motivate our research.

If the solver crashes, we know we have a bug. However, if it crashes we can never make a wrong decision based on the solver, because it tells us nothing. However, if we need to make a decision within a certain time, crashes could lead to us having to make a random decision, so there are scenarios in which we want to guarantee our solver to be crash-free. We will not consider this case further.

In the second scenario, again, we have no information. Maybe we have a bug in our solver (e.g. there exists some while loop that never terminates), or maybe our problem is simply too difficult. However, this can in many cases be solved by using some kind of application tracing to see what code the solver is running.

In the third scenario, we get a result that our solver claims is a solution. But what if there is a bug? Remember that a CSP is NP-complete. Therefore, it is in NP and we can verify whether or not a solution satisfies the constraints in polynomial time. If it turns out that the solution does not satisfy the constraints, we are in a similar situation as when it crashes or runs forever: namely, we have no information (other than that we have a bug in our solver).

The fourth scenario is more tricky. How do we know if there is a bug in our solver? For complicated problems, a solver can look at hundreds of thousands (or many more) solutions. We could check whether every one of these attempted solutions and verify whether or not they might actually be solutions after all. However, this is quite expensive and still doesn't answer the question whether or not our solver truly eliminated every possible solution. If we return to the general case in which we are looking for an optimal solution, maybe we added a stricter bound to a previous solution and now the solver says it is unsatisfiable. If the solver contains no bugs, that means we are done. But how can we know it contains no bugs?

There are three main approaches to answer this question with high certainty. First of all, we can test (and/or fuzz) the solver. This can prove that for specific inputs (namely, the ones we test on), our solver provides correct answers. However, if we solve a new problem, it is still possible the result is incorrect. The other two approaches, proof certification and full formal verification, can answer this question more exhaustively. More precisely, they answer the two following questions:

1. Given that I have run the solver on this instance and it returns unsatisfiable, is this instance really unsatisfiable? (answered by proof certification)
2. If I run my solver on any instance and it returns that the instance is unsatisfiable, is the instance really unsatisfiable? (answered by formally verifying that the solver is *complete*, which generally requires formally verifying the entire solver, hence *full* formal verification)

1.1. **Full verification.**

We consider full formal verification of the solver out of scope. Two solvers, which focus only on SAT and are not general CP solvers, are of interest as for the reason why. First is IsaSAT by

Fleury et al. [2], which is developed in Isabelle. It is the fastest currently known fully verified SAT solver TODO: Add citation from SAT competition and implements a CDCL scheme with many optimizations. Pumpkin, the CP solver developed at the Algorithmics group at TU Delft, is written in Rust. This eliminates entire classes of bugs (providing memory safety and data race-freedom, if certain conditions are met). Of even greater interest then is CreuSAT [3], which is also written in Rust but deductively verified using Creusot [4], which leverages the Why3 verification platform (see e.g. [5]).

Verifying the full solver would require fully re-engineering Pumpkin to make it more amenable to verification, or writing a new CP solver from scratch. An alternative would be to add CP support to either IsaSAT or CreuSAT, but this would also be a huge engineering effort and require assistance from their original authors. Furthermore, while especially IsaSAT is quite performant, it still lags very far behind state-of-the-art SAT solvers, according to the latest SAT competitions. TODO: Add some additional related work here.

1.2. **Proof certification.**

In the previous section we argued that answering the second question with certainty is not a good approach for this project. We now discuss answering instead the first question, which is one we can only ask *after* running a particular instance and producing some kind of output. It does not aim to prove general properties of the solver.

Instead, answering that question involves a technique where solvers produce proofs of unsatisfiability (proof logging), which can then be formally verified (proof certification or proof verification). We will use the term *proof certification* to refer to the combined approach TODO: check the terminology here.

TODO: add citations and longer introduction about proof logging in SAT, including about DRUP This approach has long been standard for dedicated SAT solvers. A proof generation framework was recently introduced in the Pumpkin CP solver by Flippo et al. [6].

Recently, the authors' (PhD students Konstantin Sidorov and Maarten Flippo, under Emir Demirović) continued efforts include building a fully verified proof checker, that can natively understand CP reasoning. This checker takes a proof log and returns whether it is valid. This checker is written in Coq and currently has support for a limited number of propagators. TODO: Compare this to earlier approach with VeriPB

A natural extension to their work, which would fit well within the scope of this project, is extending their work with support for additional propagators (see also below).

1.3. **Partial verification of correctness.**

The approaches discussed in the previous two sections can really answer the two posed questions with high certaintity. The trusted base includes generally the kernels of the formal verification tools and whether or not the formal definitions match the intended definitions. However, if we return to the original problem: we are looking to prevent bugs in CP solvers.

Instead of wishing to answer the question whether or solver is complete with certainty, we can focus on formally verifying the most error-prone *components* of the solver. This means we

can never prove actual completeness of the solver, but we can make it significantly less likely that it contains errors, not dissimilar to the way that the safety guarantees of Rust eliminate entire classes of bugs.

The difficulty in verification often lies in optimizations used to represent the internal state as well as the search algorithm (CDCL and the two-watched literal scheme in particular), because they are very close to the completeness question. If we let go of that requirement, many components have properties more amenable to verification. We name a few below:

- Propagators, arguably the most important component of a CP solver, these can apply specialized reasoning to dramatically reduce the search space. They generally represent constraints. Formally modeled, propagators are a promising target, because they have well-defined inputs and outputs (they map domains to domains) and have a number of properties that could be verified, see also ??. The properties include: they have to be *decreasing* and *monotonic* and must be *correct* for a constraint (not remove any assignments that satisfy a constraint) and must be *checking*.
- Encodings/decompositions: constraints can often be represented in multiple ways. For example, a linear inequality can be encoded using clauses and additional Boolean variables. Or a global cumulative constraint can be decomposed into many individual constraints.
- Conflict analysis. To avoid exploring parts of the tree that a solver has already seen before, after a conflict it adds redundant constraints that will make it avoid similar subproblems. Conflict analysis involves building an implication graph between assigned values and determining the first unique implication point (UIP, 1UIP since we take the first). What a UIP is can be formally modelled and the input (a conflict and the trail) and output (a learned clause) is also clear. However, 1UIP is a very central part to the solver's architecture and heavily reliant on the exact representation used for the trail and on how the solver records the implication graph (does it do it lazily, does it generate it from scratch from the trail, etc.). Furthermore, it is plausible that that mistakes in conflict analysis would lead to very clear bugs that would be caught even without formal verification.
- Clause minimization. Compact clauses are usually more valuable. There are a number of tricks to minimize clauses that could maybe be verified. TODO: Look at this more
- Restarts, clause deletions, others: the solver uses a lot of heuristics, whose properties are not always well-understood theoretically. When would we call a restart "correct"?

The two most promising candidates are propagators and encodings. We would want to tightly integrate them with an existing solver, for which Pumpkin is the most natural candidate considering our experience in Rust and the fact it is developed here in the Algorithmics group. Therefore we are mainly interested in using tools to directly verify Rust code.

Below we compare a number of the different tools:

1.3.1. *Focused on safe Rust.*

1.3.1.1. *Aeneas.*

Aeneas [7] is a verification toolchain for Rust. Their goal is to that verifying Rust programs requires as little as possible reasoning about memory, so that the focus can lie on the functional behavior instead. This does require them to not rely on interior mutability or unsafe code.

The use a new approach to borrows and controlled aliasing TODO: explain the latter? translating a subset of Rust programs (using the MIR, see `https://blog.rust-lang.org/2016/04/19/MIR.html`) into a language that is in essence, functional and pure.

The semantics of this language are fully defined and do not model memory (again, it is designed to have a functional nature). They do this because they claim that Rust's references serve no semantic purpose and that due to Rust's strong ownership discipline, Rust programs are *essentially* functional.

These functional semantics allow translating the language further into a pure lambda calculus, which can then be handed off to a number of different backends (including F^*, Coq, Lean), where one can reason about various program properties.

Pros: requires minimal dealing with memory, which is often very hard to verify; allows directly translating the program into an equivalent in a backend, which can then use whatever tooling is available there, in essence you have a fast Rust program but then can do all the verification in a backend that has that as a first-class feature

Cons: large amount of Rust features are not available, including function pointers/closures, no nested loops, some complex structures involving borrows; no annotations, which means you miss a lot of the way you structure your code in Rust; limited automation, although some backends like F* do provide it.

1.3.1.2. *Prusti.*

Prusti [8], which builds on the Viper verification infrastructure [9], is another tool for verifying Rust code.

It models Rust in terms of a separation logic, which is an extension of Hoare logic developed in the late 1990s and early 2000s with deep support for modelling resource ownership. This logic makes it easier to reason about shared mutable structures. In particular, it translates Rust programs into the Viper intermediate language, which also supports other programming languages like Java, Python and Go.

What makes it unique is that it leverages Rust's rich type system and ownership guarantees to generate a large portion of the Viper reasoning required to model Rust automatically. These are also verified automatically using Viper's automation backends (generally with an SMT solver). This creates a so-called "core proof", leaving users of Prusti to focus on the functional properties of their code.

Pros: powerful logic that can express a large part of Rust; powerful automation through Rust that discharges most memory handling; intermediate language is very expressive; also verifies parts of the guarantees of Rust

Cons: logic is quite complex, assertion/specification language in code seems a bit limited (need to investigate this further); limited support for dynamic dispatch (this is true for most verification tools); verifying part of the guarantees of Rust can lead to long verification times

1.3.1.3. *Verus.*

Verus [10] is a specialized verifier for Rust. It uses a concept called "linear ghost types" to handle

- Flux
- Hax (?)

Able to also work for unsafe Rust:

- Kani
- Verifast

Approaches and terms to understand:

- Model checking
- Symbolic execution
- Separation logic
- Prophecies

1.4. **Formal verification terms.**

1.4.1. *Symbolic execution.*

Conceptually, when using "symbolic execution", a program is run not on a concrete input but on an input that is allowed to be anything. Program operations are then performed on each symbol and conceptually when a program branches, the symbolic execution engine follows both until termination or some other stopping condition.

1.4.2. *Model checking.*

Model checking is closely related to symbolic execution (in some sources model checking is said to aim for sound analysis while symbolic execution aims for complete analysis, where sound analysis means that when the checker says some property holds, it indeed holds, while a complete analysis says that when some property holds, the checker will report that it holds).

2. Suggested approach

Verifying Rust code is still in its early stages and the tooling still has quite some ways to go. Both building a proof checker, which will need to understand and verify inferences made by CP propagators, and verifying Rust propagators, will require building a formal model of the underlying constraints.

Building such a model is often easier in a language closer to the language that we use to reason about their abstract properties, namely the language of mathematics. A proof assistant like Coq is more suitable for building such a model. Once the model is more refined, it should be able to provide information on how we could write verified code.

Therefore, I want to take the following approach:

- Implement support for two new types of constraints to the Coq proof checker, namely all-different and cumulative (prioritize the next steps if the second one turns out to be too difficult)
- Build and write out the formal model, try different approaches, really try to answer the question: "how can we best formally model CP propagation?"
- Use the gained intuition and insight to now (hopefully more easily) build Rust propagators and verify those. The extra few weeks of time before starting with this might also allow more investigation into the different verification approaches and even for them to improve
- (Maybe, has to be investigated if this is possible, might require some really nasty FFI work that is maybe more buggy than any unverified propagators) use these verified propagators in the Coq proof checker, which might lead to improved performance. My intuition is that the CP inferences will be the most expensive part of the proof checking.

2.0.1. *Short summary of the preconditions and logic that this project builds on.*

- **assume**: CSPs/COPs describe useful problems
- CSPs/COPs describe useful problems → CP solvers are useful
- CSPs/COPs are hard → CP solvers are complex → CP solvers have bugs
- CP solvers are useful + CP solvers have bugs → we want to reduce bugs in CP solvers
- **assume**: CP solver bugs are hard to prevent with testing alone / testing enough to prevent them has a prohibitive cost (1)
- we want to reduce bugs in CP solvers + (1) → we want to use formal verification
- **assume**: formally verifying correctness of a solver is hard (specifically, formally verifying completeness) (2)
- we want to use formal verification + (2) → we need to relax our correctness requirements
- **assume**: two promising ways to relax correctness requirements, formal verification of components, or formal verification of unsatisfiability of specific instances (proof certification) (3)
- **assume**: formally verifying propagators is a good component to verify
- formally verifying propagators is a good component to verify + (3) + we need to relax our correctness requirements → we want to formally verify propagators → we need to build a formal model of CP propagator reasoning for use in partial verification
- (3) + we need to relax our correctness requirements → we want to do proof certification for CP → we need to build a formal model of CP propagator reasoning for proof certification
- **assume**: building a formal model of CP propagator reasoning is easier in a general-purpose interactive theorem prover like Coq than doing it while also dealing with the immature Rust verification tooling (4)
- we want to do proof certification for CP → we want to build a verified proof checker

- **assume**: performance requirements for a proof checker are not as stringent for a proof checker (5)
- **assume**: building a program that is correct-by-construction is easier than verifying code written in a normal programming language (6)
- **assume**: Coq is a good language to build correct-by-construction programs (7)
- **assume**: building a formal model of CP propagator reasoning in Coq is useful for building a formal model of CP propagator reasoning for partial verification (8)
- we want to build a verified proof checker + (5) + (6) + (7) → we want to build a verified proof checker in Coq
- we need to build a formal model of CP propagator reasoning for use in partial verification + we want to build a verified proof checker in Coq + (8) → the proof certification approach in Coq is for now the most promising approach and is also useful for the partial verification approach

2.0.2. *Update.*

References

[1] A. K. Mackworth, "Consistency in networks of relations," *Artificial intelligence*, vol. 8, no. 1, pp. 99–118, 1977.

[2] M. Fleury, J. C. Blanchette, and P. Lammich, "A verified SAT solver with watched literals using imperative HOL," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2018, pp. 158–171. doi: 10.1145/3167080.

[3] S. H. Skotåm, "CreuSAT: Using Rust and Creusot to create the world's fastest deductively verified SAT solver," 2022.

[4] X. Denis, "Deductive Verification of Rust Programs," 2023. [Online]. Available: https://theses.hal.science/tel-04517581

[5] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich, "Why3: Shepherd your herd of provers," in *Boogie 2011: First International Workshop on Intermediate Verification Languages*, 2011, pp. 53–64.

[6] M. Flippo, K. Sidorov, I. Marijnissen, J. Smits, and E. Demirović, "A Multi-Stage Proof Logging Framework to Certify the Correctness of CP Solvers," in *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, P. Shaw, Ed., Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Aug. 2024, pp. 11:1–11:20. doi: 10.4230/LIPIcs.CP.2024.11.

[7] S. Ho and J. Protzenko, "Aeneas: Rust verification by functional translation," *Proceedings of the ACM on Programming Languages*, vol. 6, no. ICFP, pp. 711–741, Aug. 2022, doi: 10.1145/3547647.

[8] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust Types for Modular Specification and Verification," in *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM, 2019, pp. 1–30. doi: 10.1145/3360573.

[9] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*, 2016, pp. 41–62. doi: 10.1007/978-3-662-49122-5_2.

[10] A. Lattuada *et al.*, "Verus: Verifying rust programs using linear ghost types," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 286–315, 2023, doi: 10.1145/3586037.