KU LEUVEN

DTAI Declarative Languages & Artificial Intelligence

Fuzz Testing of Constraint Programming

ing. Ruben Kindt
Prof. dr. T. Guns, Ir. I. Bleukx

Motivation

Given a program that returns the solutions on mathematical and logical problems with complex constraints, how can we test that it always give the correct solution?

Code review?	(Hard to find complex combinations bugs, Second pair of eyes)
Regression tests?	(No focus on finding new bugs, Once made is automatic)
Unit tests?	(No bigger picture view, Easier to make)
Integration tests?	(Combinatorial explosions to test, Better at finding complex bugs)
Fuzz tests?	(Inf. runtime needed, Excels in testing complex combinations inputs)

Problem

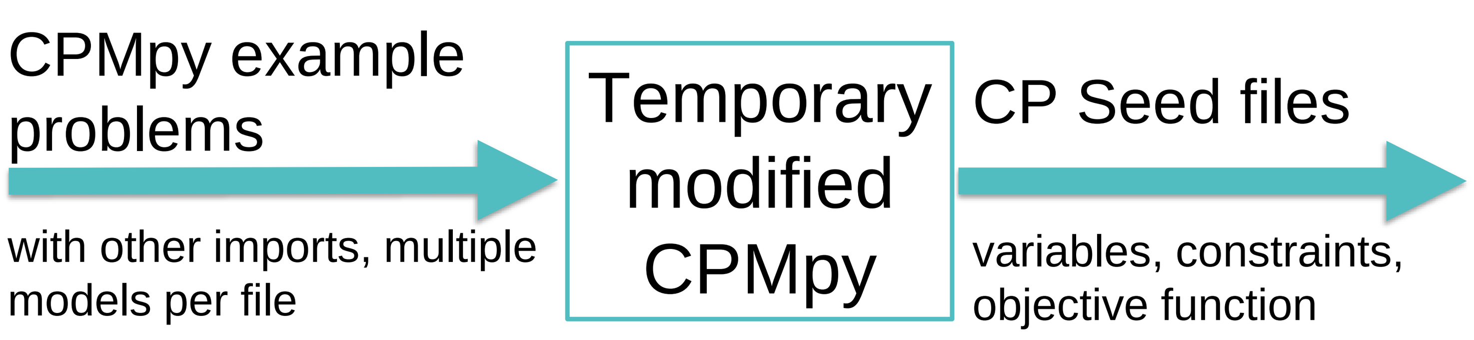
Bugs are practically unavoidable and always unwanted, especially when a user can not easily double-check the result, which is the case in constraint programming (CP). On top of that, the possible expressivity of CP results in complex combinations of constraints to model a problem, these combinations of constraints may have never been seen by a CP-solver and therefore could include untested code and bugs.
With fuzz testing we can create new problems to test and with smart design we can even know the true solution of the problem without having to search for it.

Background

CP is used to give solutions to mathematical and logical problems, these problems are made of constraints. In order to convey the problem to the solver, modeling languages have been created like MiniZinc and CPMpy.
Fuzz testing is a way of creating new and complex inputs, in order to test the software.

Approach

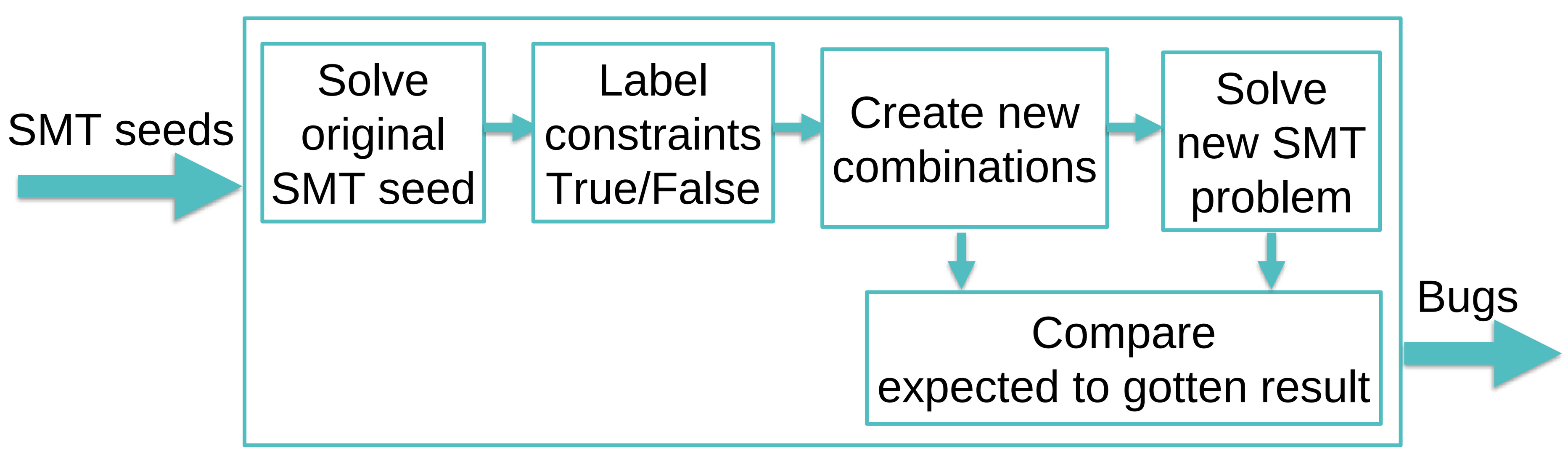
In order to create new inputs we need seed-inputs to modify. Since, generating problems from zero often results in the parser complaining that the problem does not make sense. We want to test deeper in the program, not just the parser.



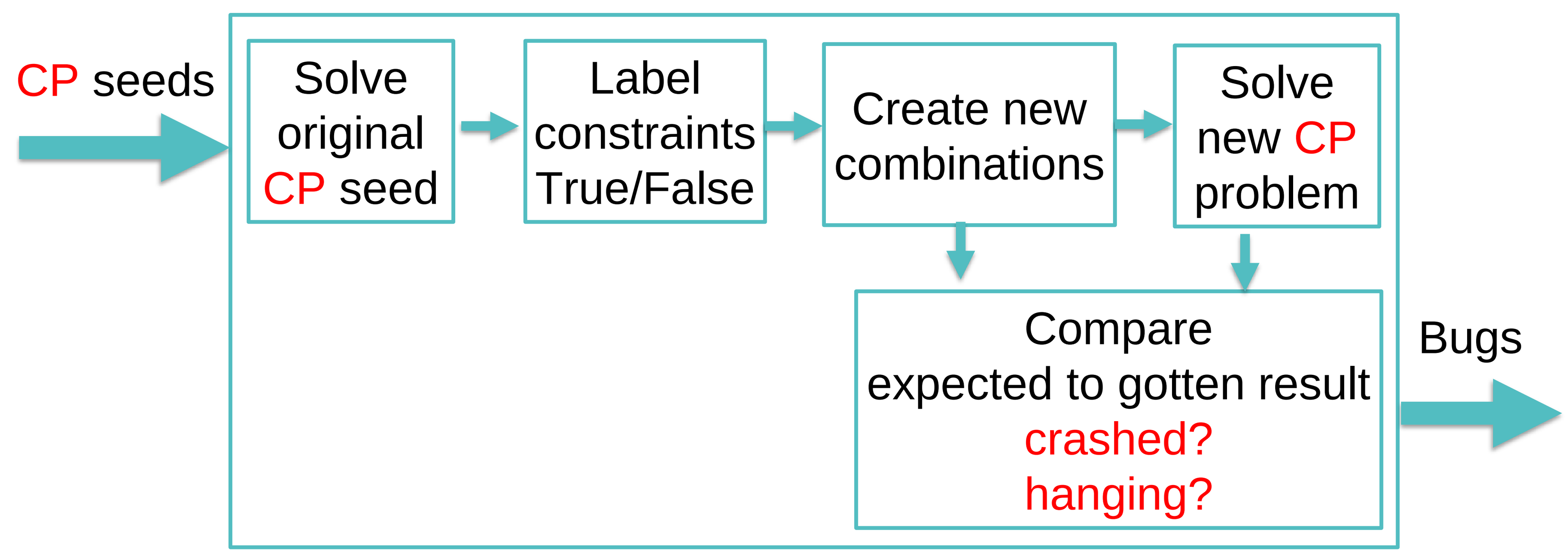
Technique 1: CTORM

The first technique starts from an existing SMT fuzz. SMT is used to determine the satisfiability of formulas. This fuzz tester is called STORM[1], and here we convert it to a CP fuzz tester in order to test CPMpy, hence the name CSTORM (from CPMpy-STORM).

STORM

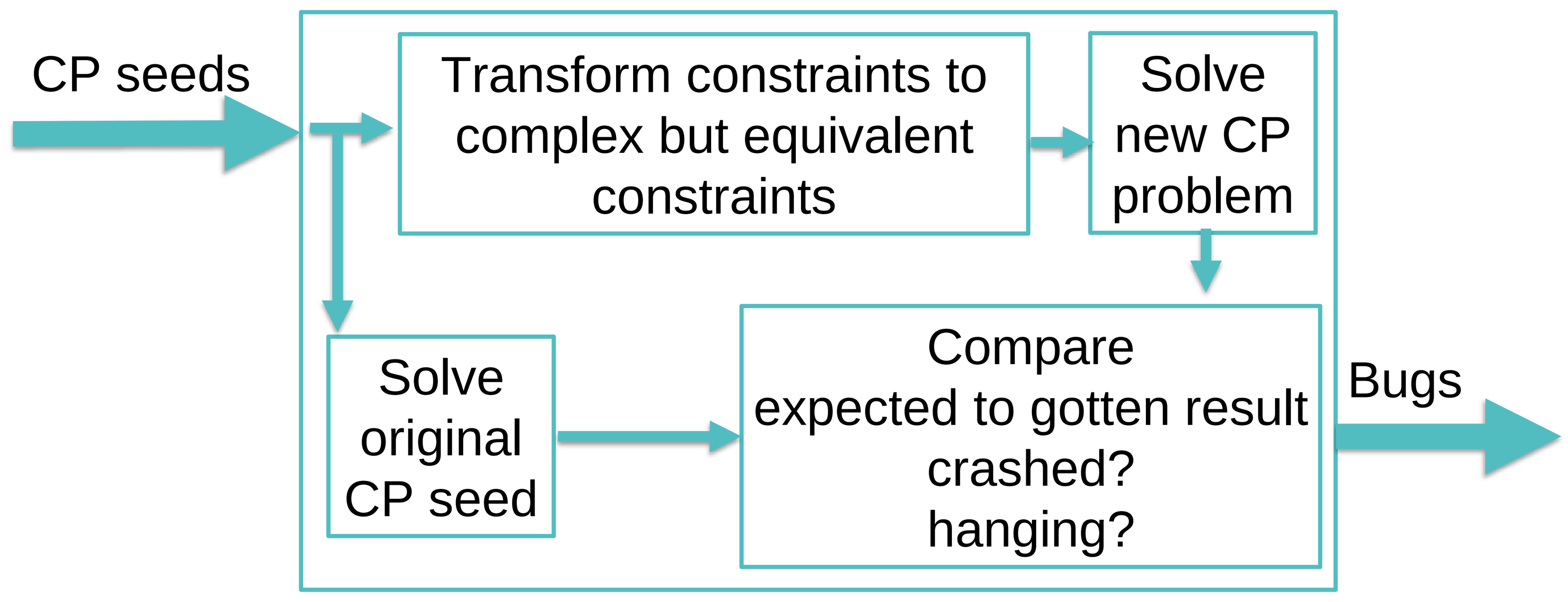


CTORM



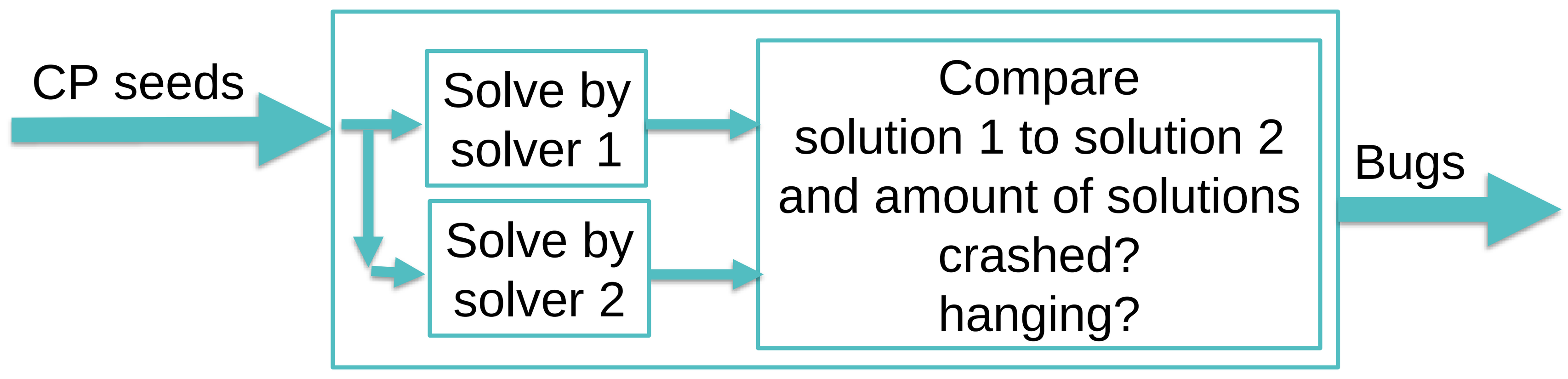
Technique 2: Metamorphic testing

This technique takes the constraints of the problem and turns them into equivalent but complex constraints. For example, a ‘variable1 == 4’ will be changed into ‘(variable1 >= 4) and (variable1 =< 4)’
In total 30 metamorphic transformations were implemented and can be reformed on already transformed constraints to built even more complex ones.



Technique 3: Differential testing

The last technique moves away from the fuzz testing world since no changes were made to the seed inputs. Instead of changing constraints, here the advantage of having multiple solvers is used.



Results

The table below shows the found bugs, around two-thirds of the bugs were the result of a crash, the others are more critical and result in a wrong output. The bugs found surrounding the OR-Tools solver were also found in the Gurobi solver this due to both solver sharing a substantial amount of code in the transformations of CPMpy. Of the techniques used CTORM found **10** bugs, metamorphic testing found the most bug at **13** and differential testing found **11** out of 19 total found bugs.

Location of bugs withing CPMpy	Crash, wrongly (un)sat or wrong nr of Solutions	Bug nr on GitHub	solver independent Bug	OR-Tools	Gurobi	MiniZinc subsolvers	PySAT subsolvers
Model	crash	145	Diff				
Model	UNSATISFIABLE	158	Meta				
Model	UNSATISFIABLE	161	CTORM, Meta				
Solver	crash	156				CTORM, Meta, Diff	
Solver Interface	crash	149			CTORM, Diff		
Solver Interface	crash	150					Diff
Solver Interface	crash	152	Meta, Diff				
Solver Interface	Wrong Nr of sol	153			Diff		
Solver Interface	crash	154				CTORM, Meta, Diff	
Solver Interface	crash	155			CTORM, Meta, Diff		
Solver Interface	crash	159			CTORM, Diff		
Solver Interface	crash	162				Meta	
Transformation	UNSATISFIABLE	142		CTORM	CTORM		
Transformation	crash	143		CTORM, Meta	CTORM, Meta		
Transformation	crash	157	Meta				
Transformation	crash	164				Meta	
Transformation	crash	165				Meta, Diff	
Transformation	UNSATISFIABLE	168			CTORM, Meta, Diff		
Transformation	(UN)SATISFIABLE	170		CTORM, Meta	CTORM, Meta		

Conclusion

None of the techniques got a perfect score, meaning that when looking for all bugs a combination of tools will be needed. As in the real world there is no silver bullet on bug catching. This does not take away the utility of each of the techniques used. CTORM is relatively easy to turn on and to forget about until it is done. Metamorphic testing can be used to guide the fuzz tester on a specific code area by choosing which metamorphic transformations used. Differential testing is easy to set up and to test between similar solvers.

Future Work

Most interesting is fuzz testing the configuration space of the solvers on top of fuzz testing the input, as discussed by Peisen Yao et al. [2]. For example, there could be bugs that only occur when certain optimizations are turned on or off like: dynamic symmetry breaking or others.

Acknowledgements

Firstly, I would like to thank professor dr. Tias Guns for the guidance and the proposal of this fascinating topic, ir. Ignace Bleukx for answering many questions, intensive thesis meetings, proofreading and the cleverness for coming up with the name of CTORM, dr. ir. Jo Devriendt for finding bugs within our bug finder, the rest of the CPMpy-team, Hakan Kjellerstrand for publishing a significant number of examples which we used as seeds, friends for proofreading even all the way back from industrial engineering on campus De Nayer like ing. Simon Vandevelde.
Finally, I would like to thank my family for the support during my further studies.

- Ruben Kindt

References

[1] Muhammad Numair Mansur et al. “Detecting critical bugs in SMT solvers using blackbox mutational fuzzing”. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2020, pp. 701–712.

[2] Peisen Yao et al. “Fuzzing smt solvers via two-dimensional input space exploration”. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2021, pp. 322–335.