Path Invariants in CPACHECKER

Thomas Stieglmaier
University of Passau
Email: stieglma@fim.uni-passau.de

***Abstract*—This work evaluates and describes possibilities how invariants can be generated for certain locations in an infeasible counter-example path. The generated path invariants can then be used as precision increment in the refinement step of the predicate analysis in CPACHECKER. Although several CPACHECKER-independent invariant generators exist, they are not (well) usable for generating invariants out of complex C programs. Therefore we use the built-in invariant generator of CPACHECKER. The evaluation shows that using invariants instead of interpolation has a noticeable impact on the results. Independent of the specific configuration and the used solver an improvement of about 1.5 % could be achieved, although the generated invariants do not have the expected shape in most cases. For selected programs, the amount of needed refinements could be reduced from over 100 to one.**

***Index Terms*—CEGAR, Invariants, Model-Checking, Path Invariants, Predicate Abstraction**

I. INTRODUCTION

Predicate abstraction is one of the analyses implemented as a CPA in CPACHECKER. Like many other CPAs it is usually used in combination with CEGAR and in the refinement step of CEGAR interpolation is used to compute the necessary precision increment to refute the counterexample. The one major drawback of CEGAR in combination with interpolation is that when a loop is on the error path potentially indefinitely many refinements are needed to prove the infeasibility of the found error location. When using loop invariants instead of interpolation for the computation of the precision increment this drawback can be removed.

This work is divided into 6 chapters:

- The introduction,
- the background with all necessary information about used techniques,
- the implementation of path invariants [BHMR07] in CPACHECKER,
- the evaluation of the path invariants implementation with several different invariant generation approaches used during refinement in the Predicate CPA,
- the description of the encountered restrictions and challenges
- and the conclusion together with an outlook on possible additions and improvements.

II. BACKGROUND

In this chapter we provide information about the concept of path invariants and some invariant generators we will be using in our implementation later on.

A. Path Invariants

As written in the introduction, the limitation of CEGAR is, that loops contained in an infeasible error path can potentially lead to an infinite amount of necessary refinements due to the predicates that are chosen to increment the precision for the next CEGAR iteration. To overcome this problem the strengths of CEGAR and invariant synthesis can be combined, thus invariants are computed only for a program constructed out of the error path found with CEGAR and these invariants are used for refinement instead of interpolation.

This is called a path invariant [BHMR07]. The smallest syntactic subprogram created out of the error path that is used for the invariant generation is called path program. It may contain loops and therefore often stands for a group of infeasible error paths that would be found upon unrolling the loop. So when using the computed invariants in the refinement step, potentially infinitely many infeasible error paths can be excluded at once from the analysis.

B. Invariant Generators

InvGen is an invariant generator for imperative programs [GR09]. It uses a technique based on boolean combinations of linear inequalities over the program variables, also called templates, to compute invariants that prove that error locations are not reachable, thus the output is either a valid invariant (it proves the non-reachability) or InvGen fails. The so computed invariants are composed of linear arithmetic constraints. In order to use InvGen, the program for which the invariants should be generated has to be a set of transition relations written in Prolog. There is also an experimental frontend [1] which translates simple C-programs [2] into Prolog. The frontend features non-deterministic variables which is a necessary addition for our purpose of using it with code generated by CPACHECKER.

Daikon is, in contrast to InvGen, an invariant detector [EPG+07]. Daikon follows a dynamic approach of detecting likely program invariants, where dynamic means that the program for which invariants should be generated is run, and Daikon observes the values of the variables the program has in a given state. Then the properties that were true over all observed runs are reported as invariants. Daikon is written in Java and can detect properties in programs written in C/C++, C#, Eiffel, F#, Java, Perl and Visual Basic. All programs have

[1] See http://www.tcs.tifr.res.in/~agupta/invgen/frontend.html for further information.

[2] Simple means in this case that there may only be one function (the main function), no function calls, no arrays and no pointers.

to be instrumented at first. For Java, Daikon comes packed with Chicory [3] which should be used for this case, but more important for this work is C. The C instrumenter is Kvasir [4] which is a plugin for valgrind. In order to use it, the C program has to be compiled with special debugging symbols This is — compared to InvGen — easier usable and applicable to more programs, as there are no limitations on the given program code. A major drawback is the dynamic approach, this means that any non-determinism will lead to endlessly running programs for which we can not compute invariants.

CPACHECKER is a tool for configurable software verification [BK11]. Since the addition of an analysis using k-induction with continuously-refined invariants [BDW15] it is possible to generate invariants with CPACHECKER itself. In contrast to InvGen and Daikon the generated invariants are not computed in a special way, but can be obtained by running an analysis. The disjunction of all states that were reached for a certain location are the invariant for this location. This is still an over-approximation, as the call-stack and the exact path to the program are abstracted. The advantage of using CPACHECKER as invariant generator instead of other invariant generators is, that it is already implemented and almost ready to use.

III. IMPLEMENTATION

In this chapter we describe the implementation of the invariant-based precision refinement. Additionally we discuss different configurations and some heuristics that have huge effect on the performance of the analysis.

A. Code

This section focuses on explaining the necessary changes to CPACHECKER for being able to generate and use path invariants during precision refinement.

1) Path program export: In order to compute invariants for a certain path programs with external programs such as InvGen or Daikon we need to export the computed infeasible counterexample as a valid C program. CPACHECKER already has capabilities to generate C code from a given error path which is in turn used with CBMC [5]. However the code generated for CBMC does not contain loops — they are unrolled — and can therefore not be used for invariant generation, as we do explicitly want to have the loops there. So a new export feature was added which is able to generate code that contains loops in the form of `gotos` and `labels`. This feature is not working for all programs reliably, e.g. there may be bugs when it comes to implicit loops (loops which were `labels` and `gotos` before). This is however not a problem as some experiments showed that InvGen and Daikon both cannot be used for our purpose, InvGen because the C frontend is not working even with the smallest and simplest examples and Daikon because of the lack of support for non-determinism. More on this can be found in Chapter V.

2) Invariant Generation with CPACHECKER*:* In contrast to generating invariants with InvGen or Daikon we do not need a C program for generating the invariants with CPACHECKER internally. Instead we can directly use the `CPAInvariantGenerator`. This class then runs another analysis out of which the invariants are computed. The analysis is run on a given CFA with also given options and CPAs. As CFA the CFA of the overall analysis can be used although it does contain more than only the error path which should be analyzed. In order to narrow the analysis to the error path we create an additional specification for the invariant generation analysis which only allows transitions along the error path to be done. The generation of the specification automaton enforcing this is part of the code added within this work [6]. The `CPAInvariantGenerator` can then be asked to return invariants for every program location which is on the error path.

3) Predicate CPA: The code of the Predicate CPA remains almost unchanged. Instead of directly creating the `PredicateCPARefiner` there is now the option to create a `PredicateCPARefinerWithInvariants` instead. The latter is a subclass of `PredicateCPARefiner`. The invariant refinement can then be done with different approaches, either by running the internal invariant generator and using the generated invariants directly to refine the precision, or by running the `KInductionInvariantGenerator`. When using the k-induction approach we do not want to use the generated invariants, but we do interpolation several times and test the interpolants on inductivity. Afterwards we use the inductive interpolants if some were found, otherwise we fallback and do either use the other invariant generation approach or we use interpolation. For both cases we need to create the precision increment out of all computed invariants. Thus we only need the invariants for the locations where the Predicate CPA created an abstraction.

B. Configuration Options

In this section we provide detailed information about the important configuration options which can be used to adjust the invariant usage during refinement to ones needs or turn it off completely. All options need to be prefixed with **cpa.predicate.refinement.** to use them in configuration files. Besides those options there are also some that are needed for debugging, which provide e.g. extra output files, those are not mentioned here.

- **useInvariantRefinement** determines if any refinement based on path invariants should be done, thus if this is set to `false` all other options concerning invariant refinement do not have an impact on the analysis at all.

[3] See http://plse.cs.washington.edu/daikon/download/doc/daikon.html#Chicory for further information.

[4] See http://plse.cs.washington.edu/daikon/download/doc/daikon.html#Kvasir for further information.

[5] CBMC is a bounded model checker. More information can be found at http://www.cprover.org/cbmc/.

[6] The code referred to can be found in the class `ARGUtils` the relevant method is `producePathAutomatonWithLoops`.

- **timeForInvariantGeneration** is the maximal amount of time the invariant generation may take, if the limit is exceeded the invariant generation is terminated and interpolants are used as fall-back for refining the precision. The default is unlimited.
- **maxInvariantGenerationsPerLoop** is a threshold for the number of invariant generations which may be done maximally for any loop over all found error paths. If the threshold is reached, thus the loop is too often in error paths out of which the precision increment needs to be computed, we directly fall back to interpolation.
- **useStrongInvariantsOnly** signals whether only those invariants should be used where the invariant generator was capable of refuting the counterexample itself, or if the invariants should be used disregarding the analysis result of the invariant generator. Default is `true` as invariants which are not strong would definitely lead to a repeated counterexample and therefore an interpolation in the next turn, which we can also do right at this point.
- **atomicInvariants** signals if the found invariants should be split up into atoms for incrementing the precision or if the whole invariant should be taken at once. Default is `false`, due to the mostly very large invariants found, splitting them up into atoms leads to an explosion of predicates in the precision which in turn makes the analysis slower.
- **useKInduction** indicates whether the approach which checks interpolants on inductivity should be used before generating other invariants. This option is set to `true` as default.
- **fallbackToInvGen** signals whether after a failed inductivity check of interpolants with k-induction other invariants should be generated or if directly interpolants should be used. The default is `true`.
- **bmcConfig** is the path to the configuration file of the bounded-model-checking analysis configuration which should be used for k-induction.
- **kInductionTries** is the number of times new interpolants should be computed for checking them on inductivity. This option does only work if the computed interpolants for a given path are different each time they are generated. This may be possible or not depending on the used SMT solver. In our case, where SMTInterpol [7] and MathSAT 5 [8] were used, this is the case.

IV. Evaluation

In this chapter we will show the results of our evaluation. It is based on several different configurations that were used for invariant generation, and shows that overall using invariants for refining error-paths which contains loops is beneficial.

A. Notice on Comparability of the Results

While running the benchmarks we noticed an issue which is caused by the randomness of the SMT solvers. This issue makes the results we get incomparable. While for the configuration `predicateAnalysis` which we use as baseline the SMT solver is used once per refinement to compute the precision increment, it may be used more often while doing invariant generation, this can lead to different interpolants for exactly the same error-path. The interpolants are in some cases better and in some cases worse. To overcome this problem we did the benchmarks four times overall with small differences, at first, we did use two different SMT solvers, SMTInterpol and MathSAT 5. Then instead of calling the SMT solver only once we call it three times in order to have different interpolants with the `predicateAnalysis` baseline if possible. For all runs the baseline is worse than all configurations that use path invariants generated by the `CPAInvariantGenerator`. For the runs using k-induction for invariant generation better results could only be achieved with SMTInterpol, not with MathSAT 5. Thus our hypothesis is, that even though the SMT solver has some influence, still the invariants impact the results in a more significant way. In the following we will provide the results for both runs, and also show the differences.

B. Configurations

As mentioned in Chapter III the invariant generators we use can be set up with different analyses inside, the only constraint is that some methods have to be implemented for it. So we chose to use all analyses that are currently supported in the invariant generators and discuss the different results in our evaluation. All used configurations are based on the `predicateAnalysis` configuration. Multiedges are disabled as well as the initial static refinement. **useStrongInvariantsOnly** is set to true, to avoid repeated counterexamples during refinement [9]. For all configurations besides the one called **kInd** the k-induction approach to invariant generation is disabled. The tested configurations are:

base	the baseline analysis: `predicateAnalysis`
inv	uses the Invariant CPA for invariant generation
apron	uses the Apron CPA for invariant generation
interval	uses the Interval CPA for invariant generation
policy	uses Policy CPA for invariant generation
value	uses the Value CPA for invariant generation
kInd	this configuration uses the k-induction approach

As mentioned in Section IV-A these configurations were combined with MathSAT 5 and SMTInterpol, also the option to do additional (unnecessary) SMT solver calls was used. Thus overall we have done the evaluation 28 different configurations.

C. Evaluation Environment

The evaluation was performed on machines with a 2.6 GHz Octa Core CPU (Intel Xeon E5-2650 v2) and 135 GB of RAM. The operating system is an Ubuntu 14.04 (64-bit) with a Linux 4.2.0-23 kernel. For the Java support OpenJDK 1.7 is used. The CPAchecker revision for the evaluation is 19367 (*trunk*).

[7] More information about SMTInterpol can be found at https://ultimate.informatik.uni-freiburg.de/smtinterpol/.

[8] More information about MathSAT 5 can be found at http://mathsat.fbk.eu/.

[9] This has an implication for the **inv** configuration where we set `cpa.automaton.breakOnTargetState` to `1` instead of `0`. `0` means that although the error was reached the analysis keeps on generating invariants, this is not necessary for us, as we do not use the invariant anyway.

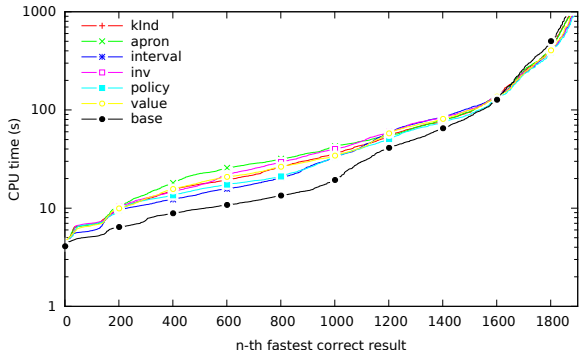

Fig. 1. Results of the evaluation while using SMTInterpol and no additional SMT solver calls.

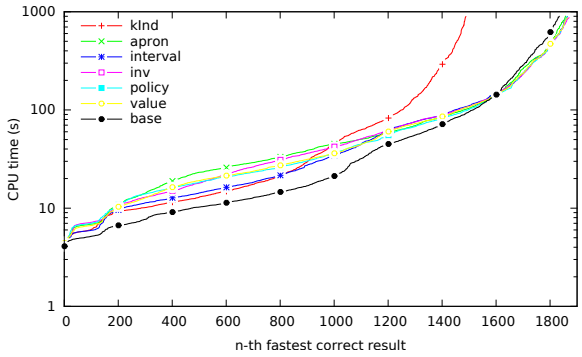

Fig. 2. Results of the evaluation while using SMTInterpol and 2 additional SMT solver calls.

Each single verification run was limited to 15 GB of RAM and to 900 s of run-time. The Java heap was set to 10 GB. For each verification run the overall amount of CPU time [10] and memory usage is measured. In all tables the time consumption will be given in hours with three significant digits unless specified differently, all other numbers are exact.

D. Benchmark Set

The used benchmark set contains all relevant files from the SV-Comp 2016 [11]. Thus we excluded the categories containing verification problems regarding memory safety, arrays, termination and concurrency. From the other categories we did only use those files that had at least one necessary refinement while running the **base** configuration. Only tasks where refinements are necessary can be influenced by our project so this step was done to filter our irrelevant data. In the end our benchmark set contains 3322 verification tasks.

E. Results

This section is divided into two major parts which will be looked at separately. On the one hand, the results with SMTInterpol as SMT solver, and on the other hand with MathSAT 5. In the end of this section we will do a small comparison of the performance across the used SMT solvers. The raw results tables can be found at http://students.fim.uni-passau.de/~stieglma/seminar/.

1) Results with SMTInterpol: For a quick overview Figs. 1 and 2 can be used. Both show the analyses done with SMTInterpol as solver. The only difference is, that Fig. 2 shows the analyses that did additional SMT solver calls, in order to have more reliable results for this evaluation. What can be seen is, that all analyses behave in mostly the same way as without the additional calls. The only difference here, is that the analysis using k-induction for invariant generation has a bug when doing additional calls [12]. Later on we exclude the configurations with the additional calls from the evaluation, the impact of the randomness in the solver is not as big as expected and comparing the performance while doing unnecessary computations does not provide useful information.

When looking at the consumed time it can be seen that the **base** configuration is significantly faster while analyzing the tasks which can be solved correctly by all solvers. The difference ranges from 3.7 h to 6.1 h. This is expected, as generating invariants takes much more time than interpolation and for larger problems, where the time consumption plays a significant role, the additional time spent in the invariant generation pays off and therefore more programs can be verified overall. Furthermore the time spent with one analysis over all tasks is the lowest in the **interval** and **policy** configuration. The time needed for invariant generation is between 2 h and 3 h, **interval** is the fastest with 2.04 h, **policy** needs 2.07 h.

While the **base** configuration is able to verify 1856 correctly each of the other analyses is able to prove more than that. The number of correctly solved tasks increases by 14 to 30, which makes approximately 1 to 2% of the overall correctly analyzed tasks. Table I also shows that the overall amount of needed refinements is lower while using invariants during the refinement. This is most likely an effect of the slower analysis due to the invariant generation. The number of refinements for correctly solved programs by all configurations is lowest for **base** with 13288, **policy** is second with 13770 refinements. This means that the expected reduction of necessary refinements can not be achieved for all programs, e.g when invariant generation succeeds but the over-approximation while translating the invariants into formulas for our analysis is too coarse, this may lead to a repeated counterexample and therefore to a higher amount of refinements overall.

To highlight some cases where the computed invariants reduced the amount of needed refinements the benchmarks `loop-new/count_by_1_true-unreach-call.i` and `count_by_1_variant_true-unreach-call.i` can be used. With the **base** configuration the analysis times out and has needed over 120 refinements. With **policy** or **inv**

[10] This time measure refers to the CPU time of the whole verification run.

[11] The SV-Comp 2016 is a competition among automated software verifiers, more information can be found at http://sv-comp.sosy-lab.org/2016/.

[12] We could not fix this bug within an appropriate amount of time, it seems to have to do with threading, a thing we never touched while doing our implementation, so the bug has to be somewhere in the existing code we used.

Configuration	# correct		#incorrect		Overall CPU Time (h)			Inv Time (h)			# refinements		
	all	equal	true	false	correct	equal	overall	correct	equal	overall	correct	equal	overall
SMTInterpol													
base													
normal	1856	1789	5	32	37.8	30.0	321	0.00	0.00	0.00	15040	13288	71838
additional	1834	1783	5	34	38.1	32.2	330	0.00	0.00	0.00	13786	13023	63605
inv													
normal	1880	1789	5	31	44.5	35.5	323	2.95	2.75	6.66	15603	14586	68823
additional	1870	1783	5	34	45.3	36.7	330	3.44	3.24	12.8	15362	14319	61919
apron													
normal	1870	1789	5	32	44.4	36.1	324	2.76	2.59	6.51	15263	14393	67357
additional	1857	1783	5	34	44.6	37.0	330	3.16	2.99	11.5	14913	14079	60169
interval													
normal	1880	1789	5	31	43.0	33.7	321	2.22	2.04	5.56	16134	14268	70998
additional	1864	1783	5	34	43.0	34.8	328	2.61	2.44	11.0	14932	13991	60651
policy													
normal	1886	1789	5	32	42.1	32.9	319	2.26	2.07	6.47	14867	13779	68990
additional	1863	1783	5	34	42.4	34.6	328	2.85	2.66	10.9	14282	13649	60969
value													
normal	1874	1789	5	31	42.6	34.1	322	2.69	2.51	6.15	15081	14658	71060
additional	1864	1783	5	34	43.6	35.5	329	3.10	2.93	11.8	14837	14399	57044
kInd													
normal	1867	1789	5	32	42.4	34.1	323	3.31	3.01	10.5	14478	13536	67472
additional	1488	/	5	25	/	/	/	/	/	/	/	/	/
MathSAT 5													
base													
normal	1957	1904	5	29	31.7	28.2	288	0.00	0.00	0.00	9826	8831	33292
additional	1955	1916	5	33	36.9	32.9	295	0.00	0.00	0.00	9232	8896	31445
inv													
normal	1981	1904	5	29	39.9	34.6	290	3.07	2.90	4.71	11059	10295	33761
additional	1964	1916	5	32	42.9	39.1	299	3.27	3.18	5.25	10677	10380	34255
apron													
normal	1977	1904	5	30	38.1	33.4	288	2.66	2.55	4.36	10545	9773	34152
additional	1966	1916	5	32	41.2	37.7	295	2.85	2.77	4.84	10792	9924	34146
interval													
normal	1975	1904	5	30	36.7	32.2	288	2.25	2.11	3.78	10588	9876	34325
additional	1960	1916	5	32	40.1	36.9	297	2.39	2.32	4.14	10437	9928	33363
policy													
normal	1984	1904	5	30	37.2	32.0	286	2.24	2.08	4.65	10106	9382	31761
additional	1959	1916	5	32	40.8	37.5	298	2.58	2.51	4.36	9954	9663	33126
value													
normal	1977	1904	5	30	36.7	32.1	288	2.76	2.60	4.31	10918	10191	33764
additional	1961	1916	5	32	40.7	37.3	297	2.91	2.84	4.65	10569	10279	32411
kInd													
normal	1945	1904	5	29	34.4	31.4	265	3.04	2.90	5.85	9705	9187	31679
additional	1643	/	5	29	/	/	/	/	/	/	/	/	/

TABLE I
Overall Performance of all Analyses on the Benchmark Set

one, maximally two refinements are needed to analyze the program correctly. For **policy** there are six more cases which can be analyzed correctly instead of timing out and additionally needing less refinements.

2) Results with MathSAT 5: As well as for the analyses with SMTInterpol we did the quantile plots also for the analyses using MathSAT 5. The Figs. 3 and 4 show the outcome. As well as for the analyses using SMTInterpol we exclude the configurations with the additional SMT solver calls from the evaluation.

It can be seen that the **base** analysis is for most programs a bit faster, but when it comes to solving more programs the configurations using invariant generation without k-induction are better. In Table I we show that depending on the configuration between 19 and 27 tasks more can be verified successfully. When comparing all configurations using MathSAT 5, the best configuration is **policy**. Second best results are achieved by **inv**. Both configurations generate constraints whose form is closest to the invariants we expected to be generated with external invariant generators. **apron** was able to correctly verify 1977 tasks, the same amount as **value**. While **apron** was intended to be a good invariant generator (due to the octagon domain [Min06] and the internally created constraints which model relations between variables in the form of $\pm x \pm y \leq c$) it is quite surprising the **value** did also achieve this result. As the explicit value analysis, which is used in this configuration, does not save any constraints, but only stores the exact value for a variable or no value at all, the invariants generated with this analysis were expected to be quite bad.

When looking at the number of refinements for the tasks solved correctly by all configurations, no analysis needed less refinements than the **base** configuration. The observed effect is already described in the evaluation of the analysis with SMTInterpol, due to repeated refinements — triggered by over-approximation during converting the invariants to formulas for our analysis — the overall amount of needed refinements increases for all configurations using invariant generation. Equal to the results of the **policy** configuration with SMTInterpol, **policy** with MathSAT 5 is able to analyze the program `loop-new/count_by_1_true-unreach-call.i` correctly and with over 100 refinements less needed (148 vs 1). The same applies to 6 more programs.

The consumed time is smallest for the **base** analysis on the correct results across all configurations, with a difference of approximately 4 h to 6 h. The invariant generation consumed between 2 h and 4 h which does not cover the additional runtime completely which is again a side-effect of repeated counterexamples. The time needed for invariant generation is overall lowest for the **interval** configuration, while for the correctly solved tasks of all solvers **policy** needs the smallest amount of time.

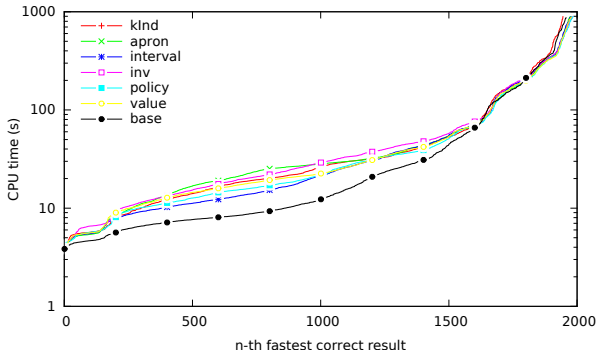

Fig. 3. Results of the evaluation while using MathSAT 5 and no additional SMT solver calls.

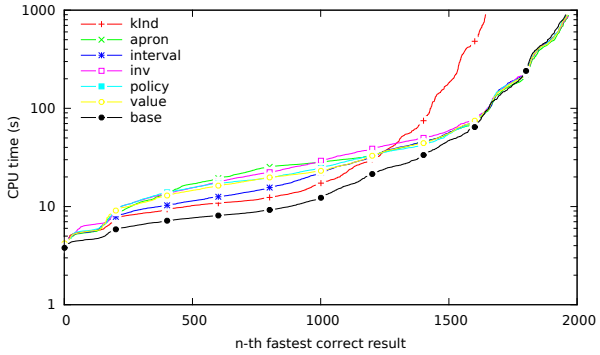

Fig. 4. Results of the evaluation while using MathSAT 5 and 2 additional SMT solver calls.

3) Comparison of Performance across the used SMT Solvers: What can be seen from Table I is that all four **base** configurations are worse than their combination with any of the invariant generation configurations, besides **kind** with MathSAT 5. This is a good indication that our hypothesis holds. Apart from that we observed that all analyses using MathSAT 5 did only need approximately two thirds of the amount of refinements than analyses which were using SMTInterpol. This means that the interpolants found with MathSAT 5 have a higher quality as they lead to less refinements overall. Another implication of this observation is the higher amount of verification tasks solved by analyses using MathSAT 5, due to less refinements the average analysis is faster and therefore per tested configuration about 100 tasks, approximately 5 %, more could be solved correctly.

Overall we can say that the **policy** configuration is, independently of the used SMT solver, the best configuration when not doing additional calls. This configuration does not only provide the best results, it is also the fastest analysis overall and produces correct results with a drastically smaller amount of needed refinements on selected benchmarks from the `loops` category [13].

[13]The folders of the files are mainly `loop-new` and `loop-acceleration`.

V. Restrictions and Challenges

As mentioned in Chapter III the original idea was not to use the invariants generator inside CPAchecker but one or more external invariant generators. The encountered difficulties while trying to implement this approach are described here.

a) InvGen: In principal InvGen is the tool fitting best to the needs we have by implementing path invariants in CPAchecker. However the C frontend is very experimental and does not work on even simple files. No function calls are allowed, as well as arrays and pointers are forbidden, too. But also on files with only (non-deterministic) integer variables the frontend crashed. So InvGen cannot be used until either the C frontend is more stable, or CPAchecker provides output not in C but in Prolog, which is the native input language of InvGen.

b) Daikon: Daikon's restriction is the lack of support for non-deterministic variables. We tried to overcome this by using random numbers where necessary e.g. `while(nondet_int())` became `while(rand())` (which is not correct but the closest we could get) but this resulted in non-termination of Daikon as internally just the instrumented program itself is run which will most likely not terminate due to the random number used inside. So Daikon can also not be used for implementing path invariants in CPAchecker.

VI. Conclusion

In this work we showed that using invariants instead of or in combination to interpolants during precision refinement improves the overall performance of the analysis. Although invariant generation is complex and takes more time than interpolation the found invariants have a stronger influence on the analysis than interpolants. Therefore the analysis is able to check more programs correctly. Besides tasks where the necessary amount of refinements could be reduced, which was the aim of this work, there are also tasks where invariant generation does not help, and even leads to repeated (and therefore more) refinements. This could be issued by the form of the invariants generated with the built-in invariant generator in CPAchecker, but the lack of invariant generators that can be used with complex C programs and non-deterministic variables led to the decision to use it anyway.

For the future there are several possibilities, at first the invariant generation capabilities of CPAchecker could be improved and so the analysis which uses those invariants during precision refinement. Another option is to dump the counterexample trace which should be used for refinement in the native input format of InvGen such that the C frontend is not necessary. This however means that we need a C-to-Prolog converter.

Besides improving the generated invariants or using other invariant generators it is also possible to change the way the invariants are used for incrementing the precision. So additionally to the already implemented approaches it may make sense to combine invariants and interpolants at once or filter out the inductive parts of computed interpolants and only use these for the precision refinement.

References

[BDW15] Dirk Beyer, Matthias Dangl, and Philipp Wendler. *Boosting k-Induction with Continuously-Refined Invariants*. Proceedings of the 27th International Conference on Computer Aided Verification (CAV 2015, San Francisco, CA, USA, July 18-24)), LNCS 9206, pages 622–640. Springer-Verlag, Heidelberg, 2015.

[BHMR07] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. *Path invariants*. Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007, pages 300–309, 2007.

[BK11] Dirk Beyer and M. Erkan Keremoglu. *CPAchecker: A Tool for Configurable Software Verification.* Proc. CAV, LNCS 6806, pages 184–190. Springer, 2011.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. *The Daikon System for Dynamic Detection of Likely Invariants*. Sci. Comput. Program., volume 69, number 1-3, pages 35–45. Elsevier North-Holland, Inc., December 2007.

[GR09] Ashutosh Gupta and Andrey Rybalchenko. *InvGen: An Efficient Invariant Generator*. Proceedings of the 21st International Conference on Computer Aided Verification, CAV '09, pages 634–640, Berlin, Heidelberg, 2009. Springer-Verlag.

[Min06] Antoine Miné. *The Octagon Abstract Domain*. Higher Order Symbol. Comput., volume 19, number 1, pages 31–100. Kluwer Academic Publishers, March 2006.