CPAchecker: A Tool for Configurable Software Verification*,**

Dirk Beyer[1,2] and M. Erkan Keremoglu[2]

[1] University of Passau, Germany
[2] Simon Fraser University, B.C., Canada

Abstract. Configurable software verification is a recent concept for expressing different program analysis and model checking approaches in one single formalism. This paper presents CPAchecker, a tool and framework that aims at easy integration of new verification components. Every abstract domain, together with the corresponding operations, implements the interface of configurable program analysis (CPA). The main algorithm is configurable to perform a reachability analysis on arbitrary combinations of existing CPAs. In software verification, it takes a considerable amount of effort to convert a verification idea into actual experimental results — we aim at accelerating this process. We hope that researchers find it convenient and productive to implement new verification ideas and algorithms using this flexible and easy-to-extend platform, and that it advances the field by making it easier to perform practical experiments. The tool is implemented in Java and runs as command-line tool or as Eclipse plug-in. CPAchecker implements CPAs for several abstract domains. We evaluate the efficiency of the current version of our tool on software-verification benchmarks from the literature, and compare it with other state-of-the-art model checkers. CPAchecker is an open-source toolkit and publicly available.

1 Overview

The field of software verification is a fast growing area, and researchers contribute new ideas and approaches with enormous pace. The more new approaches are discovered, the more difficult it is to understand the essential insight or the fundamental difference that makes a new approach good and better. Experimental evaluation is often a deciding factor for whether or not a new approach is considered an advancement of the field. But it requires a considerable engineering effort to actually build the software infrastructure for evaluating verification algorithms. Adapting a suitable parser front-end and transforming the abstract syntax tree into a format that is convenient for verification algorithms is one example. The interaction with a theorem prover is yet another issue that needs to be considered. There are successful approaches in program analysis as well as in model checking, but these techniques are rarely combined; the reason is that

* This research was supported by the Canadian NSERC grant RGPIN 341819-07.
** A preliminary version of this paper appeared as Technical Report SFU-CS-2009-02 in 2009.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 184–190, 2011.

it is indeed extremely difficult to combine them. Most published approaches are not even comparable, because the choice of the parser front-end, the choice of the theorem prover, and the choice of the pointer-alias analysis algorithm in the corresponding tool implementation, considerably influence the performance and precision of the new verification algorithm. When evaluating a performance comparison of two approaches, it is often difficult to identify what the new approach contributes and what is due to the different environment. In practice, it was so far extremely difficult to perform an experimental performance evaluation of one component while keeping all other components constant.

Configurable program analysis (CPA) provides a conceptual basis for expressing different verification approaches in the same formal setting. The CPA formalism provides an interface for the definition of program analyses, which includes the abstract domain, the post operator, the merge operator, and the stop operator [4]. Consequently, the corresponding tool implementation CPAchecker provides an implementation framework that allows the seamless integration of program analyses that are expressed in the CPA framework. The comparison of different approaches in the same experimental setting becomes easy and the experimental results will be more meaningful.

Availability. The source code and all benchmark programs for CPAchecker are available online at `http://cpachecker.sosy-lab.org`. The tool is free software, released under the Apache 2.0 license.

Related Tools. In many respects, CPAchecker is similar to Blast [3]. Our predicate analysis is also based on lazy abstraction and interpolation-based refinement. The novelty of CPAchecker is that it is easy to configure. For example, the tool can run a predicate analysis using single-block encoding (SBE), like Blast [3] and Slam [1], but also using large-block encoding (LBE) [2] or even adjustable-block encoding (ABE) [5]. The advantage of the new tool over tools that implement a separated abstract-check-refine loop, like Slam [1] and SATabs [8], is that the on-the-fly approach allows the design of more flexible, and more efficient, algorithms (like ABE [5]). We have integrated the bounded model checker CBMC [7] into CPAchecker for a bit-precise path-feasibility check.

2 Architecture and Implementation

Figure 1 shows an overview of the CPAchecker architecture. The central data structure is a set of control-flow automata (CFA), which consist of control-flow locations and control-flow edges. A location represents a program-counter value, and an edge represents a program operation, which is either an assume operation, an assignment block, a function call, or a function return (we do not consider more complex operations due to a well-known reduction called C intermediate language [1]). Before a program analysis starts, the input program is transformed into a syntax tree, and further into CFAs. The current version of CPAchecker uses the parser from the CDT, a fully functional C and C++ plug-in for the

[1] Available at `http://www.cs.berkeley.edu/~necula/cil`

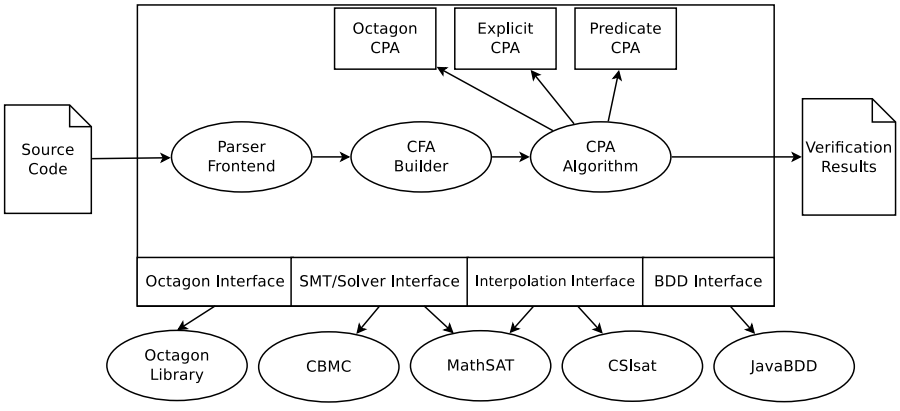

Fig. 1. CPAchecker — Architecture overview

Eclipse platform. Our framework provides interfaces to SMT solvers and interpolation procedures, such that the CPA operators can be written in a concise and convenient way. Currently we use MathSAT as an SMT solver, and CSIsat and MathSAT as interpolation procedures. We use CBMC as a bit-precise checker for the feasibility of error paths, JavaBDD as a BDD package, and provide an interface to an Octagon representation as well.[2]

The CPA algorithm is the core of CPAchecker: it performs the reachability analysis, and operates on an object of the abstract data type CPA, i.e., the algorithm applies operations from the CPA interface without knowing which concrete CPA it is analyzing [4]. For most configurations, the concrete CPA will be a composite CPA, which implements the combination of different CPAs. In order to extend CPAchecker by integrating an additional CPA for a new abstract domain, only two steps are necessary. First, an entry in the global properties file is necessary in order to announce the new CPA for composition. Second, the new CPA needs to implement the interface *CPA*, and implementations of all CPA operation interfaces need to be provided. Figure 2 shows the interaction: The CPA algorithm (shown at the top in the figure) takes as input a set of control-flow automata (CFA) representing the program, and a CPA, which is in most cases a *Composite CPA*. The interfaces correspond one-to-one to the formal framework [4]. The elements in the gray box (top right) in Fig. 2 represent the abstract interfaces of the CPA and the CPA operations. The two gray boxes at the bottom of the figure show two implementations of the interface *CPA*, one is a *CompositeCPA* that can combine several other CPAs, and the other is a *LeafCPA* (cf. the Composite design pattern). For example, suppose we want to implement a CPA for shape analysis. We would provide an implementation for *CPA*, possibly called *ShapeCPA*, and implementations for the operation interfaces that are presented in the top-right box. If we want to experiment with several different merge operators, we would provide several different implementations of *Merge*

[2] Tools available at `http://mathsat4.disi.unitn.it`, `http://www.sosy-lab.org/~dbeyer/CSIsat`, `http://www.cprover.org/cbmc`, `http://javabdd.sourceforge.net`, `http://www.di.ens.fr/~mine/oct`

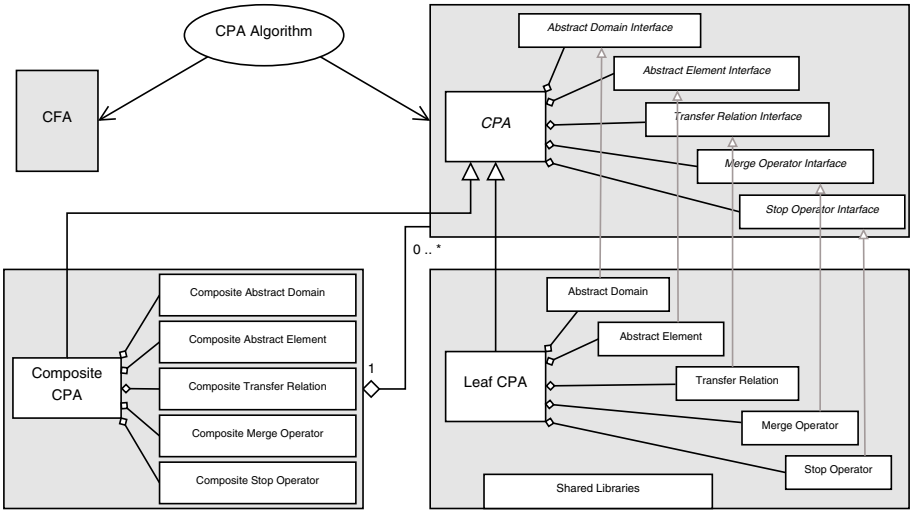

Fig. 2. CPAchecker — Design for extension

Operator Interface that can be freely configured for use in various experiments. Note that a user-defined CPA can be either a *CompositeCPA* (composing other CPAs) or a *LeafCPA* (used stand alone or as part of a *CompositeCPA*).

3 Experimental Evaluation

In this section, we compare CPAchecker with several existing tools. Our goal is to show that our CPA-based tool not only contributes a great flexibility by its configuration possibilities, but also that it can significantly improve the performance due to the possibility of constructing interesting analysis configurations.

Benchmarks. We experimented with three sets of benchmark verification problems. The first set consists of simplified, partial Windows device drivers; the second set consists of simplified versions of the state machine that handles the communication in the SSH suite. Different numbers in a program name indicate different simplifications that were applied to the source code. Both sets of benchmarks were taken from the Blast repository [3]. The third set consists of SystemC programs from the supplementary web page of SyCMC [6] [4]. The string `BUG` in the program name indicates that the program contains a defect. All benchmarks and tools are available online at `http://www.sosy-lab.org/~dbeyer/cpa-tool`.

Reporting. Table 1 shows the verification results for four tools. All experiments were performed on a machine with a 3.2 GHz Quad Core CPU and 16 GB of RAM. The operating system was Ubuntu 10.10 (64 bit), using Linux 2.6.35 as kernel and OpenJDK 1.6 as Java virtual machine. The first column reports the

[3] `http://www.sosy-lab.org/~dbeyer/Blast`
[4] `https://es.fbk.eu/people/roveri/tests/fmcad2010`

Table 1. Performance experiments

Program	Result (expected)	CBMC		SATABS		BLAST		CPACHECKER Predicate		CPACHECKER Explicit	
cdaudio_simpl1	SAFE	2.5	✓	–	TO	120	✓	25	✓	5.0	✓
diskperf_simpl1	SAFE	–	ER	–	TO	78	✓	21	✓	–	TO
floppy_simpl3	SAFE	.27	✓	720	✓	68	✓	12	✓	4.0	✓
floppy_simpl4	SAFE	.56	✓	–	TO	95	✓	19	✓	4.2	✓
kbfiltr_simpl1	SAFE	.09	✓	20	✓	8.3	✓	4.8	✓	2.8	✓
kbfiltr_simpl2	SAFE	.18	✓	46	✓	12	✓	7.3	✓	3.4	✓
cdaudio_simpl1_BUG	BUG	2.3	✓	–	TO	57	✓	16	✓	3.7	✓
floppy_simpl3_BUG	BUG	.28	✓	210	✓	3.2	✓	8.9	✓	3.1	✓
floppy_simpl4_BUG	BUG	.63	✓	650	✓	3.2	✓	15	✓	3.3	✓
kbfiltr_simpl2_BUG	BUG	.20	✓	100	✓	6.4	✓	5.4	✓	2.9	✓
s3_clnt_1	SAFE	–	ER	49	✓	180	✓	6.9	✓	21	✓
s3_clnt_2	SAFE	–	ER	610	✓	240	✓	9.0	✓	19	✓
s3_clnt_3	SAFE	–	ER	630	✓	–	ER	11	✓	19	✓
s3_clnt_4	SAFE	–	ER	330	✓	150	✓	11	✓	21	✓
s3_srvr_1	SAFE	–	ER	130	✓	–	ER	28	✓	–	TO
s3_srvr_2	SAFE	–	ER	170	✓	–	ER	15	✓	–	TO
s3_srvr_3	SAFE	–	ER	120	✓	–	ER	14	✓	–	TO
s3_srvr_4	SAFE	–	ER	210	✓	–	ER	13	✓	–	TO
s3_srvr_6	SAFE	–	ER	–	TO	200	✓	77	✓	–	TO
s3_srvr_7	SAFE	–	ER	–	TO	–	ER	25	✓	–	TO
s3_srvr_8	SAFE	–	ER	–	TO	85	✓	310	✓	–	TO
s3_clnt_1_BUG	BUG	7.4	✓	15	✓	9.3	✓	4.9	✓	3.2	✓
s3_clnt_2_BUG	BUG	7.7	✓	18	✓	10	✓	5.0	✓	2.6	✓
s3_clnt_3_BUG	BUG	8.9	✓	20	✓	10	✓	4.7	✓	2.7	✓
s3_clnt_4_BUG	BUG	7.9	✓	18	✓	9.1	✓	4.7	✓	2.7	✓
s3_srvr_1_BUG	BUG	12	✓	15	✓	–	ER	4.4	✓	12	✓
s3_srvr_2_BUG	BUG	10	✓	13	✓	130	✓	4.2	✓	12	✓
bist_cell	SAFE	–	ER	21	✓	430	✓	280	✓	–	ER
kundu	SAFE	–	ER	51	✓	–	TO	800	✓	–	MO
mem_slave_tlm.1	SAFE	–	TO	46	✓	–	ER	650	✓	–	ER
mem_slave_tlm.2	SAFE	–	TO	110	✓	–	ER	–	TO	–	ER
mem_slave_tlm.3	SAFE	–	TO	230	✓	–	ER	–	TO	–	ER
mem_slave_tlm.4	SAFE	–	TO	480	✓	–	ER	–	TO	–	ER
mem_slave_tlm.5	SAFE	–	TO	–	TO	–	ER	–	TO	–	ER
pc_sfifo_1	SAFE	–	ER	3.0	✓	14	✓	7.7	✓	–	TO
pc_sfifo_2	SAFE	–	ER	2.9	✓	55	✓	14	✓	–	TO
token_ring.1	SAFE	–	ER	4.2	✓	36	✓	14	✓	–	ER
token_ring.2	SAFE	–	ER	18	✓	–	ER	420	✓	–	ER
token_ring.3	SAFE	–	ER	34	✓	–	ER	–	TO	–	ER
token_ring.4	SAFE	–	TO	76	✓	–	TO	–	TO	–	ER
token_ring.5	SAFE	–	TO	200	✓	–	TO	–	TO	–	ER
token_ring.6	SAFE	–	TO	420	✓	–	TO	–	TO	–	ER
token_ring.7	SAFE	–	TO	–	TO	–	TO	–	TO	–	ER
token_ring.8	SAFE	–	TO	–	TO	–	TO	–	TO	–	ER
toy	SAFE	–	ER	10	✓	–	TO	–	TO	–	ER
kundu1_BUG	BUG	70	✓	20	✓	88	✓	11	✓	2.6	✓
kundu2_BUG	BUG	350	✓	49	✓	230	✓	57	✓	3.0	✓
toy1_BUG	BUG	380	✓	12	✓	–	TO	560	✓	3.0	✓
toy2_BUG	BUG	330	✓	8.9	✓	–	TO	270	✓	2.9	✓
transmitter_BUG.1	BUG	14	✓	3.2	✓	11	✓	3.7	✓	2.4	✓
transmitter_BUG.2	BUG	55	✓	11	✓	86	✓	8.4	✓	2.6	✓
transmitter_BUG.3	BUG	190	✓	20	✓	330	✓	40	✓	2.8	✓
transmitter_BUG.4	BUG	510	✓	62	✓	670	✓	–	TO	2.9	✓
transmitter_BUG.5	BUG	–	TO	140	✓	–	TO	–	TO	3.3	✓
transmitter_BUG.6	BUG	–	TO	340	✓	–	TO	–	TO	3.3	✓
transmitter_BUG.7	BUG	–	TO	–	TO	–	TO	–	TO	3.4	✓
transmitter_BUG.8	BUG	–	TO	–	TO	–	TO	–	TO	3.6	✓
transmitter_BUG.9	BUG	–	TO	–	TO	–	TO	–	TO	3.8	✓
transmitter_BUG.10	BUG	–	TO	–	TO	–	TO	–	TO	4.1	✓
transmitter_BUG.11	BUG	–	TO	–	TO	–	TO	–	TO	4.4	✓
transmitter_BUG.12	BUG	–	TO	–	TO	–	TO	–	TO	4.5	✓
transmitter_BUG.13	BUG	–	TO	–	TO	–	TO	–	TO	5.1	✓

program name and the second column indicates the expected verification result. The entries in the table use the following conventions: run times are given in seconds of CPU time; TO and MO indicate that the run was terminated after 900 s of run time or 12 GB of memory were consumed, respectively; ✓ indicates that the expected verification result was correctly computed; ER indicates that the checker failed to return a verification result, i.e., it gave up for some reason, or it crashed.

Tools. Column CBMC reports the results obtained using the bounded model checker CBMC 3.9. The bound was set to 10 loop iterations (`--32 --error-label "ERROR" --unwind 10`), which detects many of the bugs and proves safety for five drivers (by computing the loop bound); in general, nothing can be said about safe programs. If CBMC reports a violated loop assertion, we add `--no-unwinding-assertions` to the options and re-run the analysis, trying to identify more bugs. Column SATABS is based on SATABS 2.6 with the standard configuration (`--32 --error-label "ERROR"`); an explicit CEGAR loop is performed. Column BLAST reports the results using BLAST 2.5 (with MATHSAT^2 as solver [2]), configured to employ lazy, interpolation-based refinement, the DFS algorithm for the state-space exploration, and the recommended predicate-search heuristic (`-craig 2 -dfs -predH 7 -nosimplemem -alias ""`). Column CPACHECKER was obtained using Revision 3330 of CPACHECKER, with two configuration options: on the left, we used predicate analysis where the adjustable-block encoding was configured for large blocks (`-config symbpredabsCPA-lbe.properties`) [5]; on the right, we used an explicit-value analysis that tracks explicit values for each variable, where CBMC is used to certify that an error path corresponds to a true bug by encoding the error path into a C program that is given to CBMC (`-config explicitAnalysis.properties`).

Summary. The general outcome of the evaluation is that CPACHECKER's predicate analysis outperforms BLAST in all but four cases. CPACHECKER (predicate) outperforms SATABS on all driver and ssh programs; for the SystemC programs, SATABS is better than CPACHECKER (predicate). However, for the programs with a bug, CPACHECKER can be started with an explicit-value analysis and compute the result within seconds (this is especially impressive for the SystemC programs with a bug). CBMC was most successful on the driver programs. There was no false-alarm, and no tool reported a violating program as safe.

Acknowledgments. We thank G. Endler, A. Griggio, T. Henzinger, A. Holzer, S. Löwe, A. v. Rhein, M. Tautschnig, G. Théoduloz, P. Wendler, and the BLAST developers for their direct and indirect contributions to the CPACHECKER project.

References

1. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: POPL 2002, pp. 1–3. ACM, New York (2002)
2. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD 2009, pp. 25–32. IEEE Computer Society Press, Los Alamitos (2009)

3. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transfer 9(5-6), 505–525 (2007)
4. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)
5. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: FMCAD 2010, pp. 189–197 (2010)
6. Cimatti, A., Micheli, A., Narasamdya, I., Roveri, M.: Verifying SystemC: A software model checking approach. In: FMCAD 2010, pp. 51–59 (2010)
7. Clarke, E., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
8. Clarke, E., Kröning, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)