CPAchecker: A Tool for Configurable Software Verification *

Dirk Beyer and M. Erkan Keremoglu

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, is required to implement the interface of configurable program analysis (CPA). The main algorithm is configurable to perform a reachability analysis on arbitrary combinations of existing CPAs. The major design goal during the development was to provide a framework for developers that is flexible and easy to extend. We hope that researchers find it convenient and productive to implement new verification ideas and algorithms using this 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. We evaluate the efficiency of our tool on benchmarks from the software model checker Blast. The first released version of CPAchecker implements CPAs for predicate abstraction, octagon, and explicit-value domains. Binaries and the source code of CPAchecker are publicly available as free software.

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 frontend 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 being that it is indeed extremely difficult to combine them. Most published approaches are not even comparable, because the choice of the parser frontend, the choice

* This research was supported by the Canadian NSERC grant RGPIN 341819-07.

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 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 [5]. 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 (valid). The tool can be seen as a set of components that are loosely dependent on each other and that are easy to substitute.

In many respects, CPAchecker is similar to Blast [4]. For example, we implemented a predicate abstraction and an explicit-value analysis [6]. However, Blast has several limitations that we need to eliminate, most prominently, that the architecture and the design are not flexible enough to implement a pure CPA-based analysis. As in the Blast project already, many ideas were taken from Slam [2].

The source code, executables, and all benchmark programs for CPAchecker are available online at `http://www.sosy-lab.org/~dbeyer/CPAchecker`. The tool is free software, released under the Apache 2.0 license. CPAchecker is an open-source implementation of the framework of configurable program analysis (CPA). We hope that other researchers can integrate new techniques for software verification into CPAchecker and that software-verification technology becomes more accessible for practitioners using this platform.

We are currently working on integrating the bounded model checker CBMC as theorem prover, for a more precise counterexample analysis. Furthermore, we are working on a formulation of bounded model checking as CPA and look forward to integrating partial bounded model checking into CPAchecker. We have also evaluated different choices for the amount of control-flow that is considered in one single abstract-successor computation [3]. Also, we would like to integrate test-case generation and a specification language.

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) (similar to control-flow graphs [1]), which consist of control-flow locations and control-flow edges. A location represents a program-counter value, and an edge represents a program

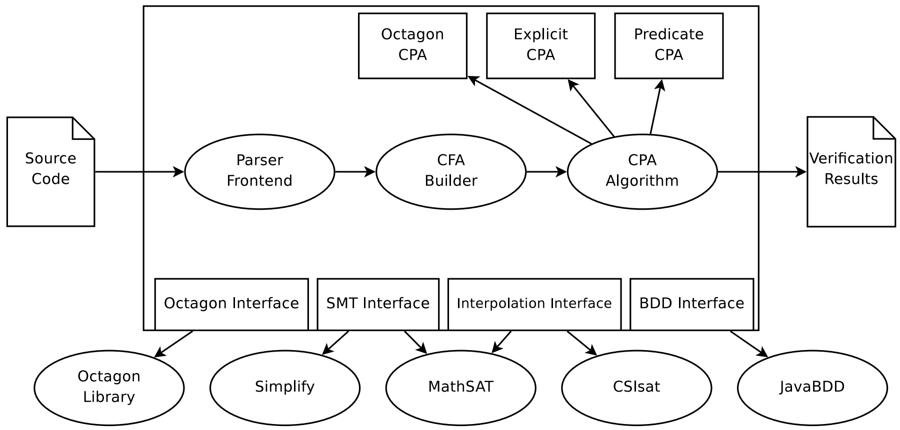

Fig. 1. CPAchecker — Architecture overview

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 [8]). 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 [1], a fully functional C and C++ IDE plug-in for the 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 Simplify[2] and MathSAT[3] as SMT solvers, and CSIsat[4] and MathSAT as interpolation procedures. We use JavaBDD[5] as BDD package and provide an interface to an Octagon[6] representation as well.

The central algorithm is the program-analysis algorithm that performs the reachability analysis [5]. (CPAchecker actually implements CPA+, i.e., CPA with precision adjustment, but we skip this detail for better presentation.) The analysis algorithm 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. For most configurations, the concrete CPA will be a composite CPA [5], which implements the combination of several 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 interface for CPA needs to be implemented, and implementations of

---

1 Available at `http://www.eclipse.org/cdt`
2 Available at `http://secure.ucd.ie/products/opensource/Simplify`
3 Available at `http://mathsat4.disi.unitn.it`
4 Available at `http://www.cs.sfu.ca/~dbeyer/CSIsat`
5 Available at `http://javabdd.sourceforge.net`
6 Available at `http://www.di.ens.fr/~mine/oct`

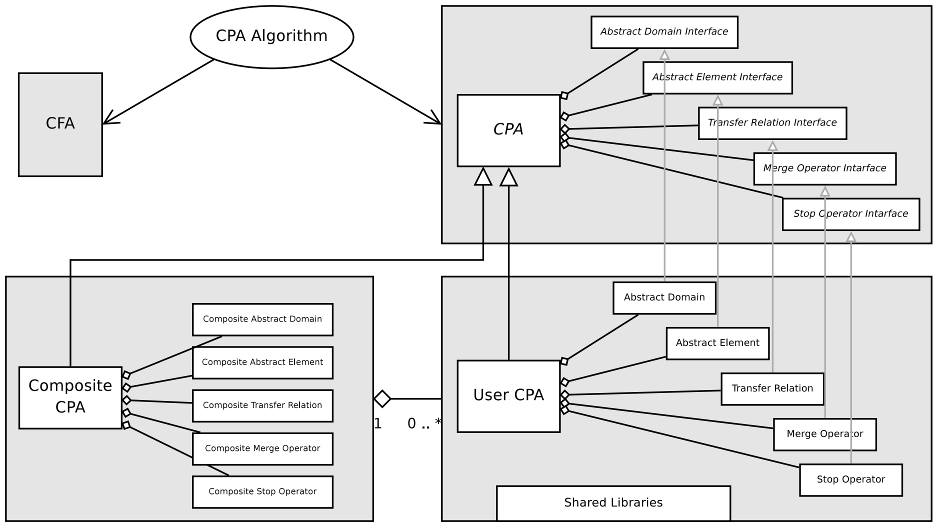

Fig. 2. CPAchecker — Design for extension

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 [5].

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 CPA interfaces, one is a *Composite CPA* that can combine several other CPAs, and the other is a *User CPA*. 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 on the right. If we want to experiment with several different merge operators, we would provide several different implementations of *Merge Operator Interface* that can be freely configured for use in various experiments.

3 Experiments

We report experiments in order to demonstrate that the tool implementation performs reasonably well on well-known benchmark examples. We pick a configuration for program analysis that was previously used [6], namely, the combination of an explicit-value analysis and a predicate-abstraction. Explicit-value analysis, also known as constant propagation, keeps track of values of integer variables. The predicate abstraction is based on Cartesian abstraction and lazy abstraction [7]. We run the analysis on various verification problems for simpli-

Table 1. Performance results; runtime given in seconds of processor time; the numbers in the column headings are the threshold values

Program	0	2	3	5	∞
cdaudio_simpl1	>1200.00	525.90	74.65	8.43	2.96
cdaudio_simpl1_BUG	167.67	88.45	17.09	3.28	0.62
diskperf_simpl1	>1200.00	>1200.00	36.95	21.19	280.10
floppy_simpl3	110.38	104.02	21.94	11.91	0.88
floppy_simpl3_BUG	42.33	37.55	7.98	2.37	0.35
floppy_simpl4	199.22	173.92	30.17	11.22	1.43
floppy_simpl4_BUG	42.95	36.15	8.03	2.16	0.36
kbfiltr_simpl1	13.77	4.59	3.50	1.02	0.42
kbfiltr_simpl2	30.89	9.98	5.48	1.83	0.89
kbfiltr_simpl2_BUG	16.17	5.76	1.24	0.73	0.32

Table 2. Statistical data observed during the experiments; a dash indicates that the experiment was aborted after 20 min; 'Preds' indicates the number of predicates used in the verification run, and 'Refines' indicates the number of refinement steps

Program	0		2		3		5	
	Preds	Refines	Preds	Refines	Preds	Refines	Preds	Refines
cdaudio_simpl1	-	-	81	332	12	76	2	11
cdaudio_simpl1_BUG	112	242	56	140	12	38	2	10
diskperf_simpl1	-	-	-	-	20	61	4	34
floppy_simpl3	81	219	51	167	20	51	4	21
floppy_simpl3_BUG	47	125	38	93	13	28	6	5
floppy_simpl4	96	307	54	219	20	58	4	19
floppy_simpl4_BUG	47	125	38	93	13	28	6	5
kbfiltr_simpl1	30	70	7	22	5	11	1	2
kbfiltr_simpl2	48	133	7	40	5	11	1	2
kbfiltr_simpl2_BUG	44	89	16	34	1	4	0	1

fied versions of Windows device drivers. The verification property is always a safety property (reachability of a certain error location under certain variable values) and is thus contained in the source code. The same program name ending with a different number indicates that the same program is present with a different simplification applied to the source code. If the program name ends with "BUG" then a defect was artificially introduced into the program.

The overall performance results obtained in our initial development phase of CPAchecker are satisfactory, although optimization was not the main design goal — rather we focussed on a portable and flexible environment to be used for many different analysis purposes. All experiments were performed on a GNU/Linux (Ubuntu 8.10) x86_32 machine with an Intel Core 2 Duo processor and 2 GB RAM. We limited the memory for the Java virtual machine to 1.8 GB and set the time limit for termination to 1200 s.

Table 1 shows the performance results for different configurations. The first column of the table lists the names of the programs. The next five columns report the runtimes for the analysis configuration where predicate abstraction and explicit-value analysis are used together. The threshold (the number in the column heading) indicates how many different explicit values where tracked for each variable (cf. [6] for the details). After reaching this threshold the value of the variable is set to $\top$, i.e., nothing can be said about the value of the variable in the explicit analysis. This might lead to an infeasible path and the predicate-abstraction domain discovers predicates in order to track the missing variables and to eliminate the infeasible program path. We experimented with five different threshold values, where 0 represents the extreme case of pure predicate abstraction-based analysis, and ∞ represents the extreme case of pure explicit-value analysis. Table 1 indicates that the best performance in total for this set of programs is achieved with a threshold of 5, which represents a good tradeoff between the expensive but abstract predicate abstraction and the simple but exploding explicit-value analysis. It is interesting to observe that pure predicate abstraction is not tractable for some of the experiments (time out reached).

Table 2 shows the number of predicates and the number of refinement iterations needed to obtain the verification result. Surprisingly, many facts can be tracked by explicit values, and thus the number of predicates in the abstract-successor computations is drastically reduced. Also, the number of refinements that are necessary to discover predicates is significantly reduced (note that many different refinements might discover the same predicate for different locations).

Acknowledgments. We thank Tom Henzinger, Ranjit Jhala, and Rupak Majumdar for the fruitful collaboration in the Blast project. Blast served as example for CPAchecker in several aspects. We also thank Alberto Griggio, Andreas Holzer, and Michael Tautschnig for their valuable comments and for their code contributions to CPAchecker. We thank Alberto especially for his contribution to the predicate-abstraction analysis.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. T. Ball and S. K. Rajamani. The Slam project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.
3. D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Proc. FMCAD*, 2009.
4. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transfer*, 9(5-6):505–525, 2007.
5. D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. CAV*, LNCS 4590, pages 504–518. Springer, 2007.
6. D. Beyer, T. A. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *Proc. ASE*. IEEE, 2008.
7. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.
8. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of C programs. In *Proc. CC*, LNCS 2304, pages 213–228. Springer, 2002.