CPA-BAM-BnB: Block-Abstraction Memoization and Region-Based Memory Models for Predicate Abstractions

(Competition Contribution)

Pavel Andrianov[1(✉)], Karlheinz Friedberger[2], Mikhail Mandrykin[1], Vadim Mutilin[1], and Anton Volkov[1]

[1] Institute for System Programming of the Russian Academy of Sciences, Moscow, Russia
andrianov@ispras.ru
[2] University of Passau, Passau, Germany

Abstract. Our submission to SV-COMP'17 is based on the software verification framework CPAchecker. Combined with value analysis and predicate analysis we use the concept of block-abstraction memoization with optimization and several fixes relative to the version of SV-COMP'16. A novelty of our approach is usage of BnB memory model for predicate analysis, which efficiently divides the accessed memory into memory regions and thus leads to smaller formulas.

1 Software Architecture

The framework CPAchecker can be used for software verification. Following the concept of Configurable Program Analysis (CPA) [1], each abstract domain is implemented in its own CPA, e.g., common tasks like tracking the program location or the call stack are implemented in their own CPAs. The CPAs in the framework can be combined to build an efficient and more precise approach like value analysis or predicate analysis. A configurable algorithm like CEGAR uses the CPAs to verify reachability and memory-safety properties.

CPAchecker is a Java program that uses the Eclipse CDT[1] to parse C source code, and the JavaSMT library[2] [2] to query SMT solvers like SMTInterpol[3], for deciding the satisfiability of formulas and generating interpolants.

2 Verification Approach

Our configuration uses two orthogonal approaches, block-abstraction memoization (BAM) and BnB memory model, to speedup the analysis. These approaches are explained in the following.

The research was supported by RFBR grant 15-01-03934.

[1] https://eclipse.org/cdt.
[2] https://github.com/sosy-lab/java-smt.
[3] https://ultimate.informatik.uni-freiburg.de/smtinterpol.

A. Legay and T. Margaria (Eds.): TACAS 2017, Part II, LNCS 10206, pp. 355–359, 2017.
DOI: 10.1007/978-3-662-54580-5_22

2.1 Block-Abstraction Memoization with Value Analysis and Predicate Analysis

BAM [3,4] implements modular verification by dividing the program into blocks and analyzing them separately. The block size matches function calls, i.e., a block starts at a function entry and ends at the corresponding function exit. The analysis uses a cache to reuse block abstractions, such that whenever a block that has been already analyzed is visited again, the stored result from the cache is applied. BAM uses a nested analysis to track variables and assignments. In our configuration BAM executes value analysis and predicate analysis in a parallel manner, because this was found to be a very effective approach for finding bugs and verifying programs with BAM. Figure 1 shows the control flow of our approach. After finding a counterexample path, two precise counterexample checks are applied, one for each analysis. For a spurious path we apply a refinement, for a feasible path we report a violation witness.

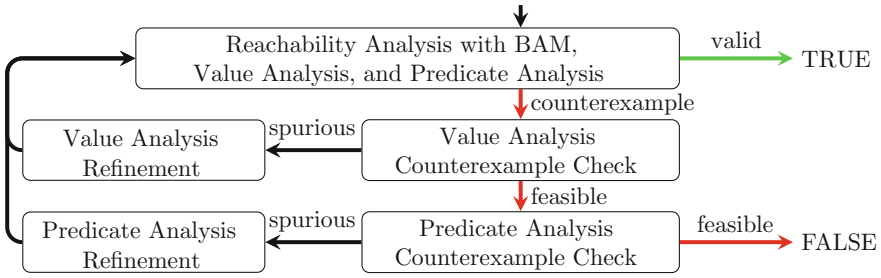

Fig. 1. Control flow for BAM with value analysis and predicate analysis

2.2 Modeling Memory with Memory Regions

BnB is a memory model based on ideas of Bornat and Burstall [5,6]. The model is implemented into the predicate analysis, which uses uninterpreted functions for mapping memory locations to memory values. An uninterpreted function f is a mathematical function, i.e. it satisfies the axiom $\forall a.\forall b.(a = b \Rightarrow f(a) = f(b))$.

In a program a memory location is represented by an lvalue expression, e.g., a pointer dereference $*p$. Assignments to lvalues change the memory state and are modeled by introducing a new uninterpreted function having the new memory value for the changed memory location and the same memory values for the unchanged ones. For example, if we have an assignment for a pointer dereference $*p = expr$, we model it by introducing a new function f_{new} with $f_{new}(p) = formula(expr)$. At the same time we should add retention conditions stating equality of memory values for the unchanged memory locations of this assignment. As far as we may not know the memory location for an lvalue expression during analysis, the retention conditions C are represented as a conjunction of disjunctions for each memory location a:

$$C := \bigwedge_{a \in \{A_1, \ldots, A_N\}} (p = a \vee f_{new}(a) = f_{old}(a)) \,,$$

where p is an lvalue expression, $A_1, \ldots, A_N$ are memory locations, $A_i \neq A_j$ for $i \neq j$, and f_{old} and f_{new} are uninterpreted functions for old and new memory states. The complexity of C highly depends on the number of memory locations.

To reduce the formula complexity we introduce memory regions representing disjoint sets of memory locations, i.e., a pointer associated with one memory region never references a memory location in another region. For each memory region R with $R \subseteq \{A_1, \ldots, A_N\}$ we introduce a separate uninterpreted function f^R. For each lvalue expression we associate a memory region R, such that an assignment to it changes only the memory locations from the associated region. Hence in retention conditions C' we consider only addresses a^R from a corresponding region R:

$$C' := \bigwedge_{a^R \in R} \left(p = a^R \vee f^R_{new}(a^R) = f^R_{old}(a^R)\right).$$

The retention conditions C' are less complex than C, because only a subset R of memory locations is used to construct the formula instead of all possible ones.

The previous implementation of the memory model [7] used *type regions* with an assumption that every memory location is always accessed with the same type. For this year we implemented *BnB regions*, which divide structure types into separate memory regions by field names. For each structure field we introduce a region defined by its name and structure type if we never take the memory address of that field. In that case we assume that the field is always accessed using field access expressions. Otherwise, if a memory address was taken, then somewhere in a program we may access this field with a pointer to a field type, thus we place such fields to a common memory region defined by the field type.

3 Strengths and Weaknesses

The contributed configuration is optimized for large programs where we need to ignore many irrelevant details. BAM is effective for the programs consisting of many functions, so that we can reuse block abstractions and have little overhead of BAM itself.

The BnB memory model benefits from separation of memory into memory regions for different fields. We have made experiments on 2795 tasks from the category *DeviceDriversLinux64* and the ratio *number of not addressed fields/number of fields* was 77%. According to the BnB memory model the majority of fields can be placed into separate regions. Thus the number of disjunctions in the resulting formulas becomes smaller. We have compared the results to the tool without BnB memory model. The CPU time was almost the same. With BnB memory model it proves 6 tasks more, but finds 5 less false, thus gets a little more points. In practice the BnB memory model may work slower if the program contains pointers for which memory was not allocated with standard memory allocation functions. In this case the analysis may prove more paths to be unreachable, thus requiring more refinements.

As far as BnB separates different fields into disjoint regions it knows that an assignment to one field does not change the other memory regions even if the pointer does not point to properly allocated memory.

Consider the following example:

```
p = not_malloc();
p->f = a;              // write access
q->g = b; p->h = c;    // updates of other fields
if (p->f != a) __VERIFIER_error();
```

The assignments to `q->g` and `p->h` do not change `p->f` and we can be sure that it still contains value `a`.

4 Setup and Configuration

We submit CPAchecker in version `1.6.1-svcomp17-bam-bnb` build from revision `ldv-bam:23987` for participation in the categories *DeviceDriversLinux64* and *Falsification*. The tool requires a Java 8 runtime environment and is available at: http://linuxtesting.org/downloads/CPAchecker-1.6.1-svcomp17-bam-bnb-unix.tar.bz2

CPAchecker has to be executed with the following command line:

```
scripts/cpa.sh -sv-comp17-bam-bnb -heap 10000m -spec prop.prp program.i
```

5 Project and Contributors

The CPAchecker project is open-source and developed by an international research group from Ludwig-Maximilian University of Munich, University of Passau, and Institute for System Programming of the Russian Academy of Sciences. We thank all contributors for their work. More information about the project (including a list of bugs in the Linux kernel found by LDV[4] with CPAchecker) can be accessed at https://cpachecker.sosy-lab.org.

References

1. 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). doi:10.1007/978-3-540-73368-3_51
2. Karpenkov, E.G., Friedberger, K., Beyer, D.: JavaSMT: a unified interface for SMT solvers in java. In: Blazy, S., Chechik, M. (eds.) VSTTE 2016. LNCS, vol. 9971, pp. 139–148. Springer, Heidelberg (2016). doi:10.1007/978-3-319-48869-1_11
3. Wonisch, D., Wehrheim, H.: Predicate analysis with block-abstraction memoization. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 332–347. Springer, Heidelberg (2012). doi:10.1007/978-3-642-34281-3_24

[4] http://linuxtesting.org/ldv.

4. Friedberger, K.: CPA-BAM: block-abstraction memoization with value analysis and predicate analysis. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 912–915. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49674-9_58
5. Bornat, R.: Proving pointer programs in Hoare logic. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000). doi:10.1007/10722010_8
6. Burstall, R.M.: Some techniques for proving correctness of programs which alter data structures. Mach. Intell. **7**, 23–50 (1972)
7. Löwe, S., Mandrykin, M., Wendler, P.: CPAchecker with sequential combination of explicit-value analyses and predicate analyses. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 392–394. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54862-8_27