The UCLID Decision Procedure[⋆]

Shuvendu K. Lahiri and Sanjit A. Seshia

Carnegie Mellon University, Pittsburgh, PA
shuvendu@ece.cmu.edu, Sanjit.Seshia@cs.cmu.edu

Abstract. UCLID is a tool for term-level modeling and verification of infinite-state systems expressible in the logic of counter arithmetic with lambda expressions and uninterpreted functions (CLU). In this paper, we describe a key component of the tool, the decision procedure for CLU. Apart from validity checking, the decision procedure also provides other useful features such as concrete counterexample generation and proof-core generation.

1 Introduction

Decision procedures for fragments of first-order logic form the core of many automatic and semi-automatic verification tools. Applications include microprocessor verification (e.g., [3]) and predicate abstraction-based software verification (e.g. [1]). Decision procedures also find use as components of higher-order logic theorem provers, such as PVS [10].

UCLID [4, 15] is a tool for modeling and verifying infinite-state systems expressible in a logic called CLU. The logic is a decidable fragment of first-order logic with restricted lambda expressions, uninterpreted functions and equality, counter arithmetic (i.e. addition by constants) and ordering ($<$). Thus, the only arithmetic constraints permitted in this logic are of the form $T_1 \bowtie T_2 + c$ where T_1 and T_2 are integer expressions and $\bowtie \in \{<, =\}$.

One of the key components of the tool is an efficient decision procedure for CLU. Apart from the logic it handles, there are several distinguishing features of the decision procedure that set it apart from other decision procedures such as SVC [2], CVC [14] and ICS [6]:

- *Eager translation to SAT:* The decision procedure performs a satisfiability-preserving translation of the first-order formula to a Boolean formula, which in turn is checked with a Boolean Satisfiability (SAT) solver. This is in contrast to other SAT-based procedures (e.g., [5, 14]), which compute a Boolean abstraction of the first-order formula and *lazily* refine the abstraction based on inconsistent SAT assignments. In contrast, UCLID performs an *eager* translation.
- *Integer interpretation:* Most queries in hardware and software verification require using an integer interpretation of symbols. However, most available decision procedures are not complete for integers even if one restricts oneself to CLU logic. UCLID, on the other hand is complete for integers, which, e.g., makes it extremely useful in reasoning about systems with arrays.

[⋆] This research was supported by the Semiconductor Research Corporation, Contract RID 1029 and by ARO grant DAAD 19-01-1-0485.

R. Alur and D.A. Peled (Eds.): CAV 2004, LNCS 3114, pp. 475–478, 2004.
© Springer-Verlag Berlin Heidelberg 2004

- *Reducing the domain of interpretation:* The decision procedure exploits optimizations that allow it to interpret symbols over smaller domains by analyzing formula structure. The small model property for CLU permits considering only a finite but often small set of values for the integer symbols in the formula. This set is further reduced by exploiting *positive equality* [3, 8].

The tool has been implemented in Moscow ML and contains around 30K lines of code. It can interface to both SAT solvers and BDD packages. The CLU formulas are internally represented using a directed acyclic graph (DAG) structure which facilitates effective sharing of common subexpressions. The DAG storage manager uses heuristics to detect and collapse certain semantically equivalent but syntactically distinct expressions.

2 CLU Logic via Examples

Consider an example of a valid CLU query which contains uninterpreted functions and lambda expressions for arrays where *ITE* stands for the *if-then-else* construct. Below, the first three lines define temporary names for sub-expressions, and the `decide` command is used to invoke the decision procedure.

```
t1  :=   f(a) !=  f(b) ;
m’  :=   Lambda x. ITE(x = a, 0, m(x)) ;  (* m’ <- m[a:=0] *)
t2  :=   t1 => (m’(b) = m(b)) ;

decide (t2); (* is t2 valid? *)
```

Here is an example[1] that cannot be modeled using traditional select-update arrays, since an arbitrary number of entries in the array `m` gets updated in a single step.

```
t1  :=   f(m(b)) =  f(m(a)) ;
m’  :=   Lambda x. ITE(m(x) < a+1, a, m(x)) ;
t2  :=   t1 => (m’(b) = m(b)) ;

decide (t2); (* is t2 valid? *)
```

This is an example of an invalid formula. The tool produces a counterexample which looks as follows:

```
+++ Counter-Examples Found : Formula Not Valid +++
a=23, b=32, m(23)=18, m(32)=22, f(22)=3, f(18)=3
```

This is a partial interpretation to all the function symbols which are relevant to the counterexample. The *concrete* counterexamples have been found extremely useful for debugging and verifying non-trivial systems [9].

The logic also supports very limited quantifiers (at the cost of incompleteness) at the top-level of a formula. One can assert a universally quantified formula in the antecedent while deciding a CLU formula as follows:

[1] The syntax is slightly different for the actual tool.

```
decide((FORALL x,y. f(x) = f(y) => x = y) => f(a) != f(a+1));
```

This limited capability has been found very useful in practice, e.g., in automating non-trivial proofs for out-of-order processor verification with unbounded resources [9].

3 Decision Procedure

Operation. The decision procedure performs a series of transformations to reduce a first-order formula to a Boolean formula. The quantifiers are first eliminated using quantifier instantiation techniques [9]. The resulting CLU formula is translated to an equi-satisfiable Boolean formula using the following sequence of steps: (i) First, lambda expressions are removed using Beta-reduction; (ii) Second, function applications are replaced with symbolic constants using optimizations like exploiting positive equality; (iii) Finally, integer-valued symbolic constants are either instantiated over a finite domain (which is sufficient to preserve satisfiability) or atomic predicates (e.g. $x < y + 3$) over these symbolic constants are encoded using fresh Boolean variables and transitivity constraints are imposed [13]. The generated formula is checked using a SAT solver. Since the nature of encoding greatly affects the SAT solver's performance, UCLID employs problem-specific hybrid encoding strategies [12] to improve the quality of the final encoding.

Counterexample Generation. The assignment produced by the SAT solver over the Boolean variables to an assignment over the first-order symbols including function constants. First, assignments for the integer variables are constructed, and then for each function application, the arguments and the result of the application are evaluated from the integer variables that represent them.

Proof-Core Generation. Many SAT solvers generate an unsatisfiable core of Boolean variables. This can be used to generate a proof core for the original CLU formula. These variables can be mapped back to atomic predicates in CLU logic, since the mappings generated by the translation to SAT are preserved. The atomic predicates find use in, for instance, predicate discovery for predicate abstraction-based verifiers.

Benchmarking. We have benchmarked the decision procedure on a diverse set of verification benchmarks arising in verifying high-level microprocessor designs, cache coherence protocols, model checking software device drivers, and compiler validation. UCLID outperforms other decision procedures including SVC and CVC on these benchmarks; results may be found in a recent paper [12].

Extensions. The decision procedure code has also been used for performing symbolic predicate abstraction [7]. Ongoing work includes extending UCLID's logic to include quantifier-free Presburger arithmetic [11].

Acknowledgments

We are grateful to Randal E. Bryant for his invaluable support and feedback.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01)*, pages 203–213, 2001.
2. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Formal Methods in Computer-Aided Design (FMCAD '96)*, LNCS 1166, pages 187–201, 1996.
3. R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, January 2001.
4. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Computer-Aided Verification (CAV '02)*, LNCS 2404, pages 78–92, 2002.
5. Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Conference on Automated Deduction (CADE '02)*, pages 438–455, 2002.
6. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonizer and Solver. In CAV '01, LNCS 2102, pages 246–249, 2001.
7. S. K. Lahiri, R. E. Bryant, and Byron Cook. A symbolic approach to predicate abstraction. In *Computer-Aided Verification (CAV '03)*, LNCS 2725, pages 141–153, 2003.
8. S. K. Lahiri, R. E. Bryant, A. Goel, and M. Talupur. Revisiting positive equality. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, LNCS 2988, pages 1–15. Springer-Verlag, 2004.
9. S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 142–159, 2002.
10. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Conference on Automated Deduction (CADE '92)*, LNAI 607, pages 748–752, 1992.
11. S. A. Seshia and R. E. Bryant. Deciding quantifier-free Presburger formulas using parameterized solution bounds. In *19^{th} IEEE Symposium on Logic in Computer Science (LICS)*, July 2004. To appear.
12. S. A. Seshia, S. K. Lahiri, and R. E. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *40th Design Automation Conference (DAC '03)*, pages 425–430, June 2003.
13. O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with SAT. In *Computer-Aided Verification (CAV '02)*, LNCS 2404, pages 209–222, 2002.
14. Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 500–504. Springer-Verlag, 2002.
15. UCLID. Available at `http://www.cs.cmu.edu/~uclid`