Google Research Awards Proposal

Title: **Constraint Solving for Software Reliability and Text Processing**

Principal Investigator: **Viktor Kuncak** , `http://lara.epfl.ch/~kuncak`
viktor.kuncak@epfl.ch
Assistant Professor, School of Computer and Communication Sciences
Head of the Lab for Automated Reasoning and Analysis
Swiss Federal Institute of Technology (EPFL), Building INR 318
Station 15
CH-1015 Lausanne, Switzerland

Contacts at Google: **Monika Henzinger**
(PI received a special invitation to submit a proposal from **Juan Vargas**)

Abstract

We will develop and implement new algorithms for constraint solving and apply them to construct two classes of tools: 1) bug finding and verification tools building on tools such as Java PathFinder [33] and Jahob [38]; 2) tools for deep semantic analysis of texts containing a mix of English, source code, and mathematical formulas. The starting point for our techniques are constraint solving algorithms developed in the rapidly expanding field of *satisfiability modulo theories* (SMT) [15, 3, 4, 10]. We will use state-of-the art techniques to implement an SMT constraint solver in the Scala programming language [30, 1] running on JVM platforms. One of the distinguishing features of our constraint solver will be the ability to analyze rich constraints that include not only quantifiers, numerical domains and data structures, but also lambda expressions and recursive definitions.

To be effective for program analysis, the constraint solver will have a native support for transition systems describing program semantics. This will enable it to tackle sources of exponential explosion in context-sensitive and path sensitive analyses such as symbolic execution [21, 20]. Such analyses can identify a wide range of bugs, many of which can cause crashes and security problems [37, 9].

In the area of text processing, we expect our constraint solver to enable efficient reasoning about rich semantic domains arising in computational semantics of natural language [8, 18, 19, 23]. This capability will make the solver a useful component of tools for creating semantically annotated text and post-processing search results in scientific and engineering domains. To evaluate this hypothesis, we will develop a tool for analyzing text whose subject is explanation of source code, programming language semantics, compilation, and program analysis [36, 28, 25].

Two keywords from the list of suggested topics: security, natural language processing

Additional keywords: automated reasoning, program analysis, education innovation

Technical Description

General Background. A new generation of automated reasoning tools has proved effective in a number of domains, including identification of security problems in operating system code [37, 9], analysis of security protocols [35], web services [6], and device drivers [7], as well as classification and analysis of widely used ontologies [2, 16, 34]. An approach that proved particularly effective for software analysis is satisfiability modulo theories (SMT) [15, 3, 4, 10]. Among reasons for effectiveness of this approach are recent advances in SAT solvers, and efficient algorithms (decision procedures) for constraints in theories such as unification [27] and linear arithmetic [11].

An Opportunity. Current SMT techniques are efficient as subroutines of larger program analysis algorithms, but fail to directly handle analysis of loops, recursion, and concurrency, leaving those constructs to an outside analysis engine. Encouraged by the effectiveness of solvers for set constraints [12, 22] and our experience with Java Pathfinder [33], we believe we can design more efficient algorithms by migrating more functionality of program analyses into constraint solvers.

We also believe that SMT solving technology has a great potential in precise analysis of texts containing natural language. Although the computational semantics and natural language processing community are aware of the advantages of finite model finders, resolution provers, and description logic reasoners [8, 23, 26, 14, 31], no system we know has taken advantage of increasingly sophisticated SMT solvers.

Goals and Stages of the Proposed Research. We propose to extend the SMT solving techniques, making them applicable 1) in a wider range of software analysis problems and 2) in the field of text analysis. Our research will proceed in three stages, each stage lasting approximately one year and resulting in the development of tools: Stage 1) a new SMT solver; Stage 2) bug finding tools; Stage 3) deep text analysis. We describe these stages and their expected results below.

Stage 1: A New SMT Solver. We will implement our own SMT constraint solver infrastructure that will enable us to push the limits of current SMT techniques. Our design will focus on the ability of the solver to find and represent (possibly infinite) *models* of constraints. To address the challenges in building a solver, we will use a high-productivity programming language Scala, and involve in the development additional students, working in synergistic research activities.

Expected Outcomes and Results for Stage 1: 1) a definition of a rich constraint language used as an input for the solver, building on [24]; and translators from other relevant input formats [3, 29, 32] into our language; 2) an SMT solver design and implementation in Scala [30, 1], including the following modules: a) a SAT solver, b) uninterpreted function symbols, c) term algebra (unification), d) integer linear arithmetic, and e) rewriting and instantiation techniques for comprehensions, quantifiers, and lambda expressions.

Stage 2: New Bug Finding Techniques and Tools. Building on the ability of our SMT solver to find models of complex formulas, we will develop precise techniques for modular checking of safety properties such as user assertion violations, uncaught exceptions, null dereferences and array bounds errors. We will deploy these techniques within symbolic execution module of the Java PathFinder tool [33], providing an expressive alternative to current SMT solver interfaces. The resulting system will be able to identify bugs that depend on deep semantic properties and user-specified assertions, making it complementary to FindBugs [17]. We expect these developments to produce a useful tool for improving software reliability and to establish a baseline against which we will compare the outcome of the next step.

In the next step we will develop techniques enabling our constraint solver to perform program analysis tasks currently done outside constraint solvers. These tasks involve inductive reasoning and are currently addressed using techniques such as loop unrolling, procedure inlining, and iterative fixpoint computation (e.g. predicate abstraction with abstraction refinement). We will define encodings of transition systems and recursive definitions into formulas of higher-order logic, and incorporate algorithms into the constraint solver to handle such constraints. The starting point for these algorithms are the corresponding program analysis techniques, but their integration into the core of our SMT solver will open up the possibilities for more efficient algorithms. In particular, SAT solver will take charge of the case analysis and finite search, the mechanisms for unification and quantifier handling will enable symbolic execution, simplification rules will reduce the size of path conditions, and representation using formulas will enable demand-driven and bi-directional analysis. Compared to existing constraint solvers for program analysis, our solver will support a richer class of constraints, allowing it to directly encode program semantics, and making construction of future precise program analysis tools for a wide range of languages substantially easier.

The constraint solver will include abstraction mechanisms for formulas, using (among others) the ideas developed in PI's PhD dissertation [24, Section 4.3], [38].

Expected Outcomes and Results for Stage 2: 1) deployment of our SMT solver within Java PathFinder; 2) design and implementation of algorithms that integrate SMT solving and program analysis within our constraint solver; 3) rules for encoding program representations into constraints handled by our solver.

Stage 3: Constraint Solving in the Analysis of Text Semantics. Having developed robust techniques for reasoning about formulas and reasoning about software, we will leverage these results in the area of analysis of documents that contain English text. We will examine texts whose subject is software itself, minimizing the problem of commonsense reasoning. Examples of such texts include comments in software source code, English explanations of algorithms and data structures, widely used texts on program semantics [36, 28], and lecture notes of the PI [25].

Our text analysis algorithms will address problems such as coreference resolution, word-sense disambiguation, and entailment. Unlike highly stylized controlled language approaches [13], we will avoid unjustified simplifying conventions and support a broad range of English constructs, relying on robust approaches that support ambiguity and underspecification [26, 8, 18]. While we remain open to statistical approaches to capture syntactic cues, our focus will be the semantic analysis supported by our constraint solver. Among the advantage of our constraint solver based on SMT techniques will be simultaneous support for both model finding and sound logical inference, efficient reasoning about quantitative and unification constraints, and support for compositional semantics thanks to lambda expressions in the input language. Given the amount of software artifacts, the proposed deep semantic analysis will have numerous applications. Our initial evaluation will focus on the uses of such analysis in authoring semantically annotated documents.

Expected Outcomes and Results for Stage 3: We will develop a tool that analyzes documents contained English text and source code, derives its semantic representation, detects semantic inconsistencies and performs entailment checks. We will deploy the tool in the form of a semantic wiki and use it to develop lecture notes for our courses.

Summary of Expected Outcomes and Results. The proposed research will result in new tools for analysis of software source code and for precise analysis of the semantics of certain text documents. The constraint solving techniques that we develop in the process will also be applicable to other domains, such as interactive theorem proving [29, 5]. We will implement all tools in Scala, so they will run on the Java platform. We will make the source code available under a range of permissive open-source licences. Following the previous methodology of PI, we will base these implementations on investigation of the underlying logical and algorithmic foundations, and we will document them by publications in relevant venues. We believe that the proposed research direction will also have important consequences in bridging the research communities of natural language processing, automated reasoning, and program analysis.

Budget Justification

The cost of one year of support for a PhD student at EPFL is approximately 55'000 CHF for 12 months. We are asking for the support of **33 months of PhD student salary**, which, assuming one summer internship, corresponds to a support for a PhD student over the period of three years. We are therefore asking for the total amount of 151'250 CHF, corresponding to, at the time of writing, **137'637 USD**. This support will enable the completion of the proposed three stages of research.

References

[1] The Scala programing language. http://www.scala-lang.org. Last accessed January 2008.

[2] FaCT++, an efficient description logic reasoner compatible with OWL DL and OWL 2. http://code.google.com/p/factplusplus/, 2008.

[3] C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). http://www.SMT-LIB.org, 2008.

[4] C. Barrett and C. Tinelli. CVC3. In *CAV*, volume 4590 of *LNCS*, 2007.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development–Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[6] A. Betin-Can, T. Bultan, and X. Fu. Design for verification for asynchronously communicating web services. In *WWW*, pages 750–759, 2005.

[7] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[8] P. Blackburn and J. Bos. *Representation and Inference for Natural Language*. CSLI Publications, 2005.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[10] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[11] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, volume 4144 of *LNCS*, 2006.

[12] M. Fähndrich. *BANE: A Library for Scalable Constraint-Based Program Analysis*. PhD thesis, University of California at Berkeley, May 1999.

[13] N. E. Fuchs, U. Schwertel, and S. Torge. Controlled natural language can replace first-order logic. In *14th IEEE Int. Conf. on Automated Software Engineering (ASE)*, 1999.

[14] U. Furbach, I. Glckner, H. Helbig, and B. Pelzer. LogAnswer - a deduction-based question answering system. In *4th Int. Joint Conf. Automated Reasoning (IJCAR)*, 2008.

[15] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *16th Conf. Computer Aided Verification*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.

[16] I. Horrocks. Using an expressive description logic: FaCT or fiction? In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 636–647, 1998.

[17] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA Companion*, pages 132–136, 2004.

[18] D. Jurafsky and J. H. Martin. *Speech and Language Processing*. Prentice Hall, 2 edition, 2008.

[19] H. Kamp and U. Reyle. A calculus for first order discourse representation structures. *Journal of Logic, Language and Information*, 5(3/4):297–348, 1996.

[20] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.

[21] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[22] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS '05: Proceedings of the 12th International Static Analysis Symposium*. London, United Kingdom, September 2005.

[23] K. Konrad. *Model Generation for Natural Language Interpretation and Analysis*. Springer, 2004.

[24] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

[25] LARA Research Group. Lara lecture notes. http://lara.epfl.ch/dokuwiki/doku.php#teaching, 2008.

[26] I. Lev. *Packed Computation of Exact Meaning Representations*. PhD thesis, Stanford University, 2007.

[27] R. Nieuwenhuis and A. Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580, 2007.

[28] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[29] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[30] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2007. Preprint available at http://www.artima.com/shop/forsale.

[31] O. Popescu. *Logic-Based Natural Language Understanding in Intelligent Tutoring Systemsn*. PhD thesis, CMU, 2005.

[32] G. Sutcliffe and C. B. Suttner. The TPTP problem library: CNF release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[33] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[34] A. Voronkov. Inconsistencies in ontologies. In *JELIA*, page 19, 2006.

[35] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In *CADE*, pages 314–328, 1999.

[36] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.

[37] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 243–257, Washington, DC, USA, 2006. IEEE Computer Society.

[38] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2008.