A Unified Framework for Handling Constraints During Program Analysis

Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind[1]
School of Computing, University of Utah
Salt Lake City, UT 84112
yyang | ganesh@cs.utah.edu

Abstract submitted to the Constraint Programming and Verification Workshop (ETAPS) 2004

We have been using constraints for concurrent program analysis in a number of research efforts (see http://www.cs.utah.edu/~yyang/publications.html for details). This talk will provide a summary of each effort, the results to date, and future extensions planned. In [Charme'03], we presented a non-operational approach to specifying and analyzing shared memory consistency models. The method uses higher order logic to capture a complete set of ordering constraints on execution traces, in an axiomatic style. A direct encoding of the semantics with a constraint logic programming language provides an interactive and incremental framework for exercising and verifying finite test programs. The framework has also been adapted to generate equivalent boolean satisfiability (SAT) problems. These techniques make a memory model specification executable, a powerful feature lacked in most non-operational methods. As an example, we provide a concise formalization of the Intel Itanium memory model and show how constraint solving and SAT solving can be effectively applied for computer aided analysis. Encouraging initial results demonstrate scalability for complex industrial designs.

In [IPDPS'03], we show how the above methods can be extended to cover a collection of well known memory models, including sequential consistency, coherence, PRAM, causal consistency, and processor consistency. We discuss the essential aspects of this framework called Nemos (Non-operational yet Executable Memory Ordering Specifications), which employs a uniform notation based on predicate logic to define shared memory semantics in an axiomatic as well as compositional style.

In a follow-up to [Charme'03], we pursue two directions. First, we seek a semantically elegant formulation of the problem, and provide a formal translation from a problem instance $\langle r, ops \rangle$, where r is a collection of higher order logic formulae defining the memory model and *ops* is a finite execution, to a QBF formula q. This approach does not yet scale to large problem sizes, but is semantically cleaner than our work in [Charme'03]. In our second direction, our aim is to scale the size of (assembly program) executions that can be handled so that we can apply our work to post-silicon verification and multiprocessor code optimization. In this approach, $\langle r, ops \rangle$ is translated into a functional program p that when run generates a Boolean formula b. Furthermore, we show that b consists of two parts b_1 and b_2 where b_1 can be pre-generated knowing only the assembly program length. If we pre-generate b_1 for various lengths, we can load them into an incremental SAT solver and save their process state (i.e., a checkpoint) on disk. Later when given an execution *ops*, b_2 is generated for it, and the saved image is run on b_2 for further execution. We study when this 'partial evaluation' approach yields payoffs. Showing the conformance of an execution to a memory model involves solving for a total order. In this context, we study two different SAT

[1]This work was supported by National Science Foundation Grant CCR-0081406 and SRC Contract 1031.001

encoding methods and choose one that works well in practice. The results are demonstrated on a formal specification of the Intel Itanium memory model.

In our latest effort, we explore the practicality of conducting program analysis for multithreaded software using constraint solving. By precisely defining the underlying memory consistency rules in addition to the intra-thread program semantics, our approach offers a unique advantage for program verification - it provides an accurate and exhaustive coverage of all thread interleavings for any given memory model. We demonstrate how this can be achieved by formalizing sequential consistency for a source language that supports control branches and a monitor-style mutual exclusion mechanism. Our choice of sequential consistency is merely to make our illustrations clearer; by adopting the Nemos approach discussed above, we can plug in *any* desired memory model in the broad range of memory models for which Nemos is appropriate. We then discuss how to formulate programmer expectations as constraints and propose three concrete applications of this approach: execution validation, race detection, and atomicity analysis. Finally, we describe the implementation of a formal analysis tool using constraint logic programming, with promising initial results for reasoning about small but non-trivial concurrent programs. To the best of our knowledge, this is the first effort that considers conducting program analysis in a framework where the underlying memory model is *an explicitly controllable parameter*—and not assumed tacitly to be sequential consistency as in all other works we know about. This may be of growing importance, given that many programs in real applications today cannot assume sequential consistency. Examples of programs that are already known to be in this class include: (i) multiprocessor garbage collectors that may run on shared memory multiprocessors that follow a weak shared memory model; (ii) multithreaded and re-entrant device drivers where the threads may run on co-processors.

In this paper, we provide a detailed explanation of each line of work, the results to date, and how all these analysis methods might be supported by a unified constraint-solving framework. We anticipate this framework to consist of efficient SAT engines and efficient compilation engines. Given a constraint expression e, an evaluator will analyze it and annotate each sub-expression with what type of evaluation it merits. The evaluation methods under consideration are SAT and direct execution, although this list could easily be enlarged to include various decision procedures also. We will then rewrite e to a form which allows multiple evaluation methods to be applied one after the other. For example, consider a constraint expression of the form $p \Rightarrow q$ where p has symbolic information ("free variables") and q has ground information (variables have been instantiated). Instead of translating $p \Rightarrow q$ to a Boolean formula, a more efficient method might to use the contrapositive form $\neg q \Rightarrow \neg p$, emit program code to evaluate q, and if this evaluation returns $true$ then to emit the Boolean SAT formulas for $\neg p$. We find that starting from a higher-order logic specification, such rewrites when judiciously applied result in efficient constraint evaluators that partly evaluate the constraints through execution and partly through SAT-like tools. We will discuss our preliminary ideas towards building such a tool.