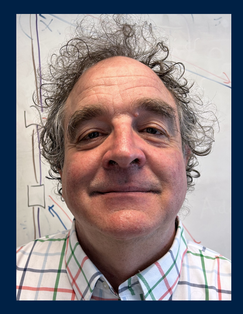
Shellac: compiler sketching with rewrite rule synthesis

VSTTE 2022

Christopher Chen, Margo Seltzer, and Mark Greenstreet
Computer Science Department
The University of British Columbia
Vancouver, Canada
{cchen2, mseltzer, mrg}@cs.ubc.ca

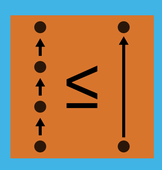
Code & paper repository



Problem & Example

Specification languages make it possible to design and analyze concurrent programs and distributed systems, but the task of actually lowering specs to working code is often left as an engineering exercise, **fraught with peril:**

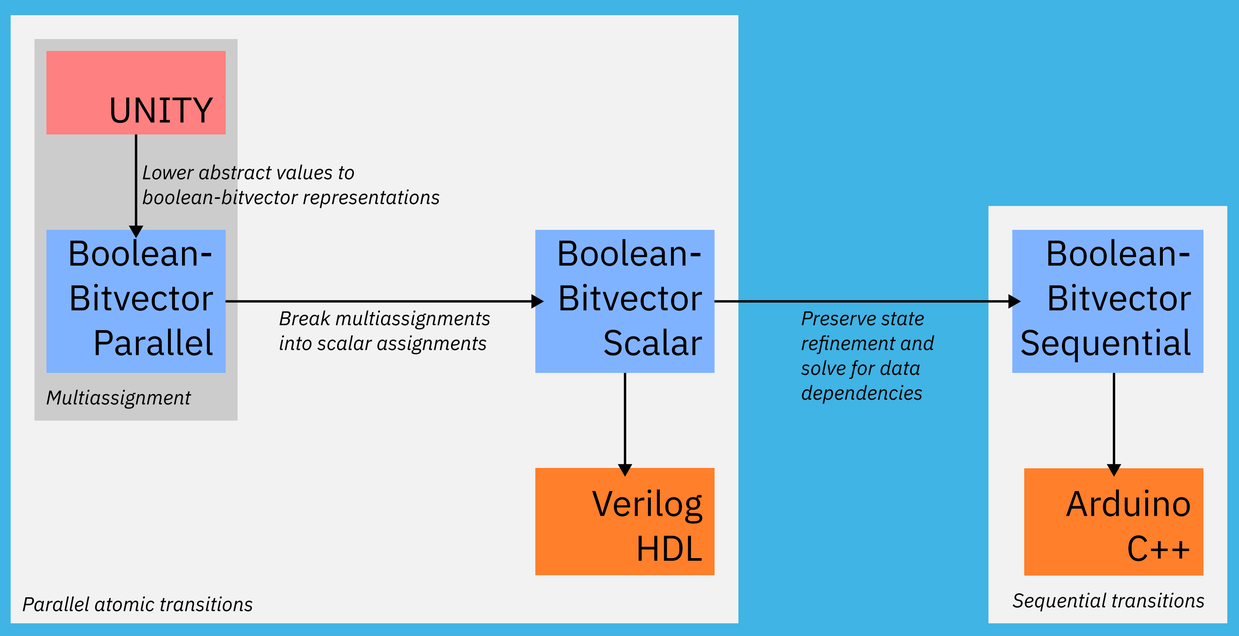
Satisfy any fairness constraints implied by the nondeterministic specification



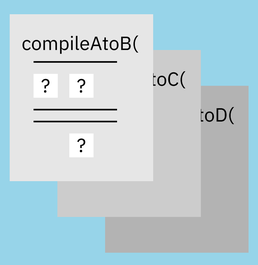
Show refinement if the implementation introduces visible states

Account for undefined behaviours introduced in the lowering process

Consider a compiler from UNITY[1] to Arduino and Verilog for message-passing programs that progressively lowers the program through syntax-directed passes:



Question & Approach

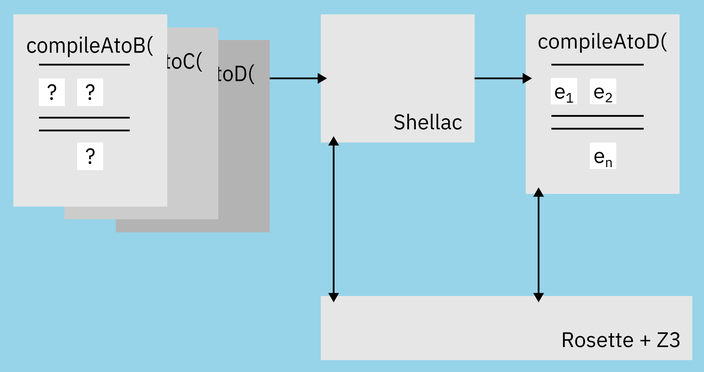


Given a partial program sketch of a compiler, can we synthesize rewrite rules so that the complete compiler preserves safety and liveness properties?

At a high level, Shellac uses an embedding of our languages of interest in Rosette[2], symbolically executes each source operator, and extracts a verification condition that the corresponding target expression(s) must satisfy.

The verification condition that the synthesizer ingests includes preconditions to avoid undefined behaviours in the target language, e.g., integer overflow.

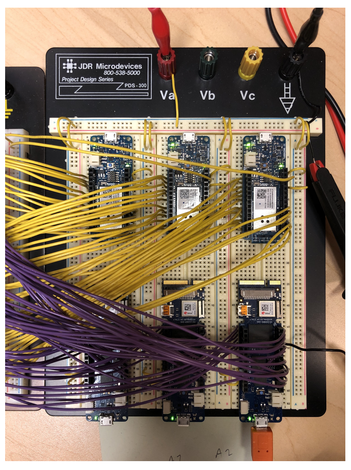
Each rewrite rule also includes pre and post-conditions, which the compiler uses to find valid orderings of sequential assignments to satisfy refinement and data dependencies.



Result

Shellac is able to synthesize rewrite rules from UNITY with channels to a boolean-bitvector intermediate representation and our final targets in minutes.

The resulting compiler can translate a specification of a single round of Paxos[3] to Arduino C++ and Verilog in seconds.



Future Work

The class of message-passing programs (e.g., actors) is an attractive and active area for program analysis in general and synthesis in particular.

Can we take the lessons learned from automating the construction of a compiler and apply them to building self-optimizing, efficient, highly concurrent, provably-correct systems?

1. Mani Chandy, K., Misra, J.: Parallel program design: a foundation. Addison-Wesley, Reading (1988). 2. Torlak, E., Bodik, R.: A solver-aided host lightweight symbolic virtual machine for solver-aided host languages. SIGPLAN Not. 49(6), 530-541 (Jun 2014). 3. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. 16(2), 133–169 (May 1998).