ASSURANCE OF DOMAIN SPECIFIC LANGUAGES

ALWYN E. GOODLOE

NASA LANGLEY RESEARCH CENTER

DOMAIN SPECIFIC LANGUAGES

- Domain specific languages (DSL) are programming languages tailored to a specific application domain.
  - SQL – Relational databases.
  - LaTeX – Typesetting.
  - Matlab – Matrix algebra and linear systems.
- A program in a DSL is often sufficiently abstract to be a specification.
- DSLs can be stand-alone programming languages.

EMBEDDED DOMAIN SPECIFIC LANGUAGES

- Embedded Domain Specific Languages (EDSL) are embedded in a host language.
  - Cryptol - Cryptographic protocols.
  - Lava – Programming FPGAs.
- Parsing and type checking are handled by host language.
- EDSLs are usually defined as a library of high-level language.
- EDSL programs can be directly executable, or generate code in another language, like C or VHDL.

REPEATING BAD HABITS

- DSL designers often repeat the mistakes of general purpose language design.
  - Syntax that is difficult to parse.
  - No defined semantics and type system.
  - The language grows very complex with age as many people work on it.
- Complex DSLs lacking formal definition are very difficult to reason about informally and formally.
- Theme: You need a programming language expert on the team from the beginning not just domain experts who code.

RUNTIME VERIFICATION

- “Runtime verification is the discipline of computer science/engineering that deals with the study, development, and application of those verification techniques that allow **checking whether a run of a system under scrutiny satisfies or violates a given correctness property**” Leucker et.al.
- Runtime verification (RV) refers to the use of monitors to observe the behavior of a system and detect if it is inconsistent with a given specification.
- Lightweight formal method complements design-time approaches .

COPILOT: RV FRAMEWORK

- Haskell-based Embedded Domain Specific Language (DSL).
  - Takes advantage of the wonderful Haskell type system.
- Abstract functional specifications written in a Lustre-like language.
- Synthesize monitors targeting real-time embedded systems.
- Generates Misra-like C monitors.
  - Constant time, constant memory.
- Minimum instrumentation of system under observation source code.
- Samples the system under observation.
  - Can miss state changes if not sampled, but effective for cyber-physical systems.

DSL DESIGN PHILOSOPHY

- Challenge: A good DSL should encompass the features of modern programming languages that enable assurance while still being domain specific.
  - Sophisticated type systems catch errors.
  - Referential transparency enforces repeatability.
- Solution: Embedding the DSL in sophisticated typed functional languages such as Haskell and OCAML.

SPECIFYING AN EDSL

- Challenge: You cannot verify programs if there is no formal definition.
- Solution: Construct the necessary formal definitions.
- BNF Syntax.
- Typing rules.
- Axiomatic semantics.
- Denotational semantics.
- Operational semantics.
  - Can be executable.

IS YOUR PROGRAM CORRECT

- Challenge: It should be easy to assure DSL programs as they are more abstract, but in practice the abstractions used are often poorly defined and tool support is lacking.
- Solution: Apply the tools and techniques developed by computer scientists.
- Write and publish a mathematical semantics of the DSL.
- Build an interpreter so that users can experiment with their programs.
- Integrate proof tools like SMT solvers, model checkers, interactive provers to facilitate correctness proofs of the DSL program.

Focus on Assuring Generated Code

TRACEABILITY

- Challenge: Maintaining traceability from the generated code to the source code.
- Solutions: Build in support for traceability.
- Many code generators produce unreadable code. Use or build a code generator favoring readability over efficiency.
- Generate comments and assertions that make it easy to relate generated code to the DSL.
  - ANSI C Specification Language (ACSL) assertions for C code.
- Generate diagrammatic representations of relationships between source and generated code.

OVERFLOWS, TIMING, AND SUCH

- Challenge: Buffer overflows and numerical overflows as well as numerical errors and scheduling issues are a source of a wide range of problems in real-time embedded systems.
- Solution: Use existing tools where possible.
- Apply a collection of analysis tools to the generated code to ensure the absence of the buffer overflows, undefined behavior, numerical errors, and scheduling issues.
  - Abstract interpretation.
  - Dynamic Analysis (RV Match).
  - Worst case timing analysis tools.
- When possible do the analysis on the DSL.

EQUIVALENCE CHECKING I

- Challenge: Can we have a formal proof that the generated code is equivalent to the DSL program.
- Solution I: Model the semantics of source and target language in a theorem prover and built the translation and proof within the prover.
  - CompCert is C complier built in Coq.
- There are a number of academic efforts applying this approach, but such an approach requires experts at conducting interactive proof.

EQUIVALENCE CHECKING II

- Challenge: Can we have a formal proof that the generated code is equivalent to the DSL program.
- Solution: For small well defined DSLs, apply automated equivalence checking tools.
- Galois’ Software Assurance Workbench (SAW) can show that generated C code is indeed equivalent to a DSL specification.
  - Spec and C code get translated to an intermediate language that SMT solvers can apply equivalence checking decision procedures to.
  - C code is compiled to LLVM, symbolic execution is used to unroll loops, etc. and then C is translated to the intermediate language.
  - Copilot is typed functional language so translation to the intermediate language should be simple.

SUPPORT FOR TESTING

- Challenge: Testing the generated code.
- Solution: Apply approaches from the interaction of testing and programming languages.
- Property-based testing.
  - Generating random tests from specs. (Quickcheck).
- Unit testing for each module generated.
- Coverage analysis.

QUESTIONS?

Contact Information:
Alwyn E. Goodloe
a.goodloe@nasa.gov

IMPROVING OUR PROCESSES

- Copilot was developed as part of a decade-long research program into runtime verification.
    - Open source, NASA Class E software.
- We are in the process migrating Copilot framework to NASA Class D software.
    - Class D – Basic Science/Engineering Design and Research Technology Software.
- The generated monitors will be need to be NASA Class C software.
    - Class C –Intended for Mission Support or Aeronautic Vehicles.

SIMPLE COPILOT EXAMPLE

- Copilot stream language specification of Fibonacci numbers: 0, 1,1,2,3, 5, 8, …
- fib :: Stream Int32
- fib = [0, 1] ++ (fib + drop 1 fib)