Emergent Trend

• In recent years, NASA projects have begun using generative programming approaches
  – At least one high-profile project ultimately abandoned this approach
  – Others seem poised to be successful (but are still in development)
• This presentation is not an endorsement of generative programming, but rather a survey to hopefully benefit IV&V
Code Generators Benefits

• Minimize Risk
• Enforce use of best practices, standards and guidelines
• Higher Quality
  – Avoid traditional, manual coding errors
• Highly Optimized
  – Single-purpose
  – Specialized
• Consistency
• Increased Productivity
• Extensibility
• Reuse
• Predictable Code

(not a complete list, nor in any particular order)
Modern Code Generation Background

• Generative Programming “Black and White Book”
  – Feature modeling
  – Component libraries and frameworks
  – Focus on configuration knowledge
  – Simplify interfaces
• Practical UML Statecharts
  – Event-driven
    • Useful for lower power consumption
  – Good for embedded systems
  – Includes open source generator
• Model Driven Architecture with Executable UML
  – Comprehensive approach
  – Antiquated?
• Eclipse ecosystem: JMerge, JET, others
• Many, many others...
Typical Code Generation Approaches

• Structured document as source
  – aka Textual Domain Specific Language (aka External DSL)
  – Structured document might be XML or CSV

• UML model as source
  – Often using one or more UML Profiles and Action Languages
  – NOT Model-Driven Architecture (MDA)
    • Usually only generating parts of a larger system

• Commonly used for controller code

• Internal DSL or Fluent Interface
  – Libraries which morph syntax into domain specificity
  – Not really code generation, but a recent, related trend
  – Example:

manager.newCommand().with(4, "SYS").with(5, "DAT").optional().priorityLow();
Code Generator Inputs (Activities)

• Original approach was often a flowchart
  – Rather weak
    • Leads to inefficient, redundant, inherently procedural code

• Benefits of UML Activities
  – Standard UML2 syntax
  – Component and OO paradigm support
  – Code generation concepts included in metamodel
    • Control Flows
    • Object Flows
    • Tokens
  – Tools provide model audit / model checking capabilities
  – Extensible via UML Profiles
    • Optimize for special purpose
    • Influence generated code
  – Action Language support

• Challenges of UML Activities
  • Standard UML2 Syntax is bloated
    – Foundational Subset for Executable UML Models (fUML) helps
• Benefits of UML State Machines
  – Standard UML2 Syntax
  – Simple graphical model
  – State machines well-known
    • Harel Statecharts
  – Transform to/from textual representation
  – Tools provide model audit / model checking capabilities
  – Extensible via UML Profiles
    • Optimize for special purpose
    • Influence generated code

• Challenges of UML State Machines
  – Standard UML2 Syntax is bloated
    • Supports multiple kinds of statemachine: Mealy, Moore, Harel, etc.
  – Several “standards” to choose from
Challenges for Traditional V&V (1 of 2)

- Static Code Analysis of little use on generated code
  - Complexity metrics not applicable
  - Maintainability metrics not applicable

  1. Restrict to simple control flow constructs.
  2. Give all loops a fixed upper-bound.
  3. Do not use dynamic memory allocation after initialization.
  4. Limit functions to no more than 60 lines of text.
  5. Use minimally two assertions per function on average.
  6. Declare data objects at the smallest possible level of scope.
  7. Check the return value of non-void functions, and check the validity of function parameters.
  8. Limit the use of the preprocessor to file inclusion and simple macros.
  9. Limit the use of pointers. Use no more than two levels of dereferencing per expression.
  10. Compile with all warnings enabled, and use one or more source code analyzers

  - Not applicable to generated code
  - Ensure generator enforces
Challenges for Traditional V&V (2 of 2)

- Varying approaches and/or specifications
  - Projects sometimes use several code generators
  - Textual specification usually requires manual inspection
    - Automation possible if text is sufficiently structured
  - Model specification can be automated
- Change in V&V Methods for code inspections
  - Manual inspection of generated code of dubious value
  - Automated, static inspection of generated code of dubious value
Challenges for IV&V (1 of 2)

• Multiple code generation schemes and/or tools
  – Challenge for IV&V to cover multiple code generators
  – Additional work to thoroughly understand multiple code generators

• Access to code generator(s)
  – Timely access to the code generator is essential
  – Quality of IV&V is hindered if we must rely on the generated code only

• Integration approach
  – How is the project integrating generated and manual code?
    • Dependency analysis
    • Clear “Separation of Concerns?”
    • Low Coupling and High Cohesion?
Challenges for IV&V (2 of 2)

- Requirement traceability
  - Difficult, if not impossible, to trace from generated code back to originating requirement(s)
- Change in V&V Methods for code inspections
  - Manual inspection of generated code of dubious value
  - Automated, static inspection of generated code of dubious value
- Compare locally generated code against release version
  - Catch any post-code generation, well-intended “hacks”
Benefits for IV&V

• Input model analysis
  – Automated
    • if IV&V has the model and the modeling tool
    • Or, if IV&V can transform the model
• Extend code generator for IV&V purposes
  – One of the lurking benefits of code generators is the ability to use the same input to generate multiple outputs
    • Customize generator for a different language (e.g., C, C++, Java, PHP)
    • Customize generator for local test environment
    • Customize generator for Q2 and Q3 handling
• Reuse previous IV&V results against reused code generator
Code Generator Certification?

- Compilers
  - Originally, compiler output is not trusted
  - After testing and certification, compiler output is trusted and, importantly, becomes a V&V tool

- Code generators are similar to compilers
  - Certification of a code generator would enable its use as a V&V tool
  - Write code generator unit tests
    - Enable IVV-specific analyses
      - Q2 and Q3 handling
      - Non-functional requirement evaluation
  - Extend code generator with instrumented output
    - Enable IVV-specific analyses
      - Q2 and Q3 handling
      - Non-functional requirement evaluation
Certifying a Code Generator

- IV&V mini-project to “certify” a code generator
- Goals
  - Identify design constraints and/or assumptions encapsulated in the code generator
    - Memory management, array handling, indexing, etc.
    - Fault Handling
    - Off-nominal behavior (Q2 and Q3)
    - Etc, etc
  - Understand implications of input model and effect on generated code
  - Assemble audits for input model for quality and completeness checking
Room for Improvement

• Design Patterns for input models
  – Library of well-formed StateMachines for general purpose components
  – We see reuse of code generators
    • But not reuse of input models

• NASA IV&V library of code generators
  – Internal reference library of known code generators
    • Including test suites and model audits
  – Facilitate knowledge transfer across organization
  – Useful for subsequent projects
Summary

• Find out which code generation approach is being used
  – Answers many basic questions
  – Helps to focus the IV&V efforts
• Exert more analysis effort on input models
  – Watch for generator’s “assumptions”
    • Memory allocation / releasing strategies
    • String lengths / limits
    • Array handling
    • Fault handling
    • Extended language features
    • Conformance to applicable standards and regulations
• Complications when tracing design to implementation
  – Some “design” features are implemented within the code generator
    – They may not necessarily conform to the published project standards or design constraints