Extending Benefits of Static Code Analysis Tools

NASA IV&V Software Assurance Tools (SWAT)
Outline

• Setting Context
• Identifying Types of Code Defects
• Preparing for Analysis
• Maximizing Tool Effectiveness
• Selecting Appropriate Tools
• Customizing to Support Analysis
• Leveraging Historical Analysis Data
• Reviewing Ongoing Efforts
• Taking Questions
Setting Context

SWAT is working to maximize the effectiveness of IV&V’s static code analysis capabilities.

- SWAT is now actively participating in the Static Code Analysis Working Group to help facilitate improvements in the use of static code analysis tools within the IV&V Program.
- SWAT is working with IV&V projects to gather representative source code and creating sample test code when necessary for a library to support evaluation of both current and additional tools.
- SWAT is responding to requests from projects regarding specific capabilities by verifying current tools with test code and evaluating additional tools for use.

- Working to better understand the capabilities of current static code analysis tools.
- Evaluating how current tools are being applied and identifying opportunities to improve the process of using them.
- Identifying gaps in our current static analysis capabilities and evaluating solutions to improve IV&V’s analysis coverage.
Identifying Types of Code Defects

Categories of code defects

- Syntax
- Logic
- Run-time
Identifying Types of Code Defects

Syntax

Caught during compile time, e.g. code is attempting to use an undeclared variable

May include style or certain coding standards violations that are caught through basic parsing and “pattern matching”
Identifying Types of Code Defects

May be caught during functional requirements based testing, i.e. does the code do what it was supposed to?

High value findings when IV&V is able to identify or confirm logic defects through either manual analysis or use of dynamic analysis resources such as the ITC JSTAR lab.

Analysts have indicated that, from their experience, digging deeper into understanding the code when reviewing static code analysis results can provide a side benefit of identifying other unrelated logic defects.
Identifying Types of Code Defects

Run-time

- Occurs during program execution, e.g. out of bounds array access ("buffer overflow"), division by zero, non-initialized local variables, integer overflows, illegal dereferenced pointer
- High value findings, as these defects may result in program crash or undefined behavior that could ultimately result in the loss of the mission or impact the accomplishment of a mission objective
- Underlying tool technology (e.g. abstract interpretation) and project configuration impacts the effectiveness of identifying certain defects
- Static code analysis tools can help detect dangerous run-time defects in the code or even prove the absence of certain run-time defects when using formal methods-based approaches
Preparing for Analysis

What code do I expect to get and what does it do?

What is the development environment for this build of code, and how will it affect my analysis?

What are the characteristics of the target architecture on which this code was written to run?
Preparing for Analysis

What code do I expect to get and what does it do?

Understanding of requirements implemented in the code

Form an expectation of interfaces that should be present in the code build

Identify the important characteristics of the architecture that could be reflected in the source code such as CPU, operating system, key addresses, and memory layout

What are the characteristics of the target architecture on which this code was written to run?
Preparing for Analysis

Understanding of requirements implemented in the code

Form an expectation of interfaces that should be present in the code build

Identify the important characteristics of the architecture that could be reflected in the source code such as CPU, operating system, key addresses, and memory layout

- Know what to expect when you receive code (or recognize if you didn’t get what was expected!)
- Create a team “cheat sheet” of important characteristics of the architecture that could be reflected in the source code.
- Determine what prior projects may be relevant to the analysis
- Check with ITC JSTAR lab for any available project simulations that could support understanding of the architecture for analysis
Preparation for Analysis

What is the development environment for this build of code, and how will it affect my analysis?

- Understand the characteristics of the developer’s environment
- Identify differences between the developer’s environment and your analysis environment
- Consider most appropriate tools to aid in analysis
Preparing for Analysis

- Understand the characteristics of the developer’s environment
- Identify differences between the developer’s environment and your analysis environment
- Consider most appropriate tools to aid in analysis

- Identify dependencies that may exist in source code
- Identify build artifacts and settings to expect, with respect to the build system used by the developer
- Leverage previous experience with the same development environment if possible
- Identify what static analysis tools the development project is using to prevent overlap where possible
Preparing for Analysis

**Developer’s Build Environment**
- **Developed Source Code**
- **3rd Party Source Code / Libraries** (VxWorks, standard libraries, etc.)
- **Build System**
  - **Compiler** (GNU, Wind River Diab, javac, etc.)
  - **IDE** (Wind River Workbench, Eclipse, etc.)
  - **Build Artifacts** (makefiles, build logs, etc.)

**IV&V Analysis Environment**
- **Developed Source Code**
- **3rd Party Source Code / Libraries**
- **Build System**
  - **Compiler**
  - **IDE**
  - **Build Artifacts**
- **IV&V Static Analysis Tool** (e.g., Klocwork, FlexeLint)
- **IV&V Code Browsing Tool** (e.g., Understand, Eclipse)

Build information gathered directly from the developer or through IV&V analysis.
Preparing for Analysis

Development Environment Characteristics

- Developed in Wind River Workbench
- Uses the GNU toolchain
- Windows desktop environment
- Regular source code Linting during builds

- Source code with VxWorks dependencies and version information
- Makefiles to build the code with the VxWorks GNU make tool
- Source code and makefiles with file references that assume case insensitivity
- Source code that may be more effectively analyzed with some tool other than FlexeLint (because of the developer-initiated lint during the build process)

• Create cheat sheet with processor, memory, and VxWorks characteristics
• Evaluate projects with similar mission characteristics to get an idea of what will be delivered
Maximizing Tool Effectiveness

That wasn’t too bad. I guess I’ll…

Now that I have code, what do I need to know to maximize the effectiveness of my static code analysis tools?

Incoming Code!
Now that I have code, what do I need to know to maximize the effectiveness of my static code analysis tools?

- Understand the developer’s build process and gather build information.
- Ensure that macros and other information are defined correctly within your build.
- Review the checks enabled for your build to ensure that the checks of interest for the IV&V effort have been selected.

Preparing for the static code analysis effort adequately and properly configuring the build are necessary for the most complete and accurate analysis results.
Maximizing Tool Effectiveness

Understand the developer’s build process and gather build information

Build Logs

INC

Build

INC
Maximizing Tool Effectiveness

Understand the developer’s build process and gather build information

• Inspect developer build artifacts given by the project to get a feel for the build process
• Consider utilizing tools to help determine if we have the needed files for analysis, in addition to the build information (e.g., Understand, Eclipse, and SWAT script for creating a Klocwork build specification file)

• Once missing files have been identified, the analyst should:
  • Request missing files from the developer (common with third party libraries) as the ideal solution
  • Use other available library code if necessary
  • Stub out the missing code if absolutely necessary

• Inspect any build logs or other status produced by our tools, looking carefully for warning or error messages such as missing header files, make any necessary adjustments and execute the build again
Ensure that macros and other information are defined correctly within your build so that we are analyzing the correct code.

Not only does the build tell us WHICH files need to be analyzed, but it also tells HOW those files should be analyzed based on macro definitions passed to the compiler.

```c
void Fsw::setupTimer( void )
{
    bool setupSuccessful;

    #ifdef TARGETCPU
        setupSuccessful = false;
    
        // Perform setup of interrupt handling routine.
        // Lines snipped that do the work...
        // Create a task to handle the interrupt.
        // More lines snipped...
    #endif
}
```
Maximizing Tool Effectiveness

Ensure that macros and other information are defined correctly within your build

• Make sure that macro definitions and other information relative to the software being analyzed are added to the build information passed to our static code analyzers. Failure to do so may produce invalid or misleading results.

Most tools can:
• Operate with incomplete information and provide potential issues
• Execute the build and produce some output
Maximizing Tool Effectiveness

- Review the specific checks enabled for your build in the static analysis tool to ensure that the checks of interest for the IV&V effort have been selected.
- Consider also documenting what is NOT being checked for, if applicable.
- Consider running multiple builds with different sets of checks enabled.
- Consider enabling MISRA checks for critical flight software, but separate builds may be warranted in this instance due to the potential for a significant number of resulting messages.

Review the checks enabled for your build to ensure that the checks of interest for the IV&V effort have been selected.
Selecting Appropriate Tools

Which of the available tools are appropriate for my task?
Selecting Appropriate Tools

Which of the available tools are appropriate for my task?

- Project software characteristics
- Project development characteristics
- IV&V team analysis requirements
- IV&V Project scope
Selecting Appropriate Tools

Project software characteristics:
- Source code language
- Size of code base
- Ground vs. Flight software

Project development characteristics:
- Overlapping analysis with project
- Availability and completeness of project artifacts
Selecting Appropriate Tools

IV&V team analysis requirements:
- Characteristics of tools available
- Targeted defect types
- Experience of the team with specific tools
- Turn-around time

IV&V Project scope:
- Mission criticality
- Level of assurance and verification needed
Selecting Appropriate Tools

Static Source Code Analysis

Defect Finding Tools (e.g., Klocwork and FlexeLint)

*Static analysis allows the analyst to find specific defects and formal methods-based tools can prove the absence of certain defects across the source code or in targeted areas.

Formal Verification (e.g., Polyspace)

*Dynamic analysis allows the analyst to target specific threads of execution to demonstrate particular instances of defects or the absence of defects.

Testing & Simulation

Dynamic Analysis (ITC)

Diagram Key

Unproven Code

Specific Errors Identified

Proven to be Absent of Specific Errors

*Dynamic analysis can identify areas of the system that may warrant focused static analysis. Conversely, static analysis can identify areas of the system that may warrant dynamic analysis to prove the existence or absence of defects and improve understanding of the behavior of the system.
Customizing to Support Analysis

SWAT is doing things to help you too!
Customizing to Support Analysis

Additional benefits from static code analysis tools can be realized from customization

Understand
- Standards verification
- Metrics, best practices, other rules or conventions
- Perl API

Klocwork
- Steep learning curve
- Abstract Syntax Tree checks such as unnecessary negative value testing for unsigned integers

Polyspace
- Data range specification
- Simulate normal working conditions of software
- Support ICD verification
Leveraging Historical Analysis Data

Historical data can be used to further extend the benefits of static code analysis tools

SWAT is researching prior code defects that have been identified by IV&V to determine what tool or approach was utilized

Code level metrics and quality attributes

Focus future analysis
Reviewing Ongoing Efforts

**Static Code Analysis Working Group**

- Documenting lists of required checks that should be enabled per tool given the target domain
- Documenting lists of checks that can be safely disabled or output ignored per tool given the target domain

**SWAT**

- **Current Tool Evaluation**
  Comparing tool setup, use and resulting output for the same source code drop across various tools
- **New Tool Evaluation**
  Continuing evaluation of tools based on both additional capability (e.g. formal methods-based abstract interpretation) and target language support (e.g. Ada)
Taking Questions

Contact us at: ivv-swat@lists.nasa.gov