
# 4GL Call Graph Analysis

## Eliminate Dead Code (and More) Using Automated Call Graph Analysis

**Greg Shah**
**Golden Code Development**

**Friday November 6, 2020**

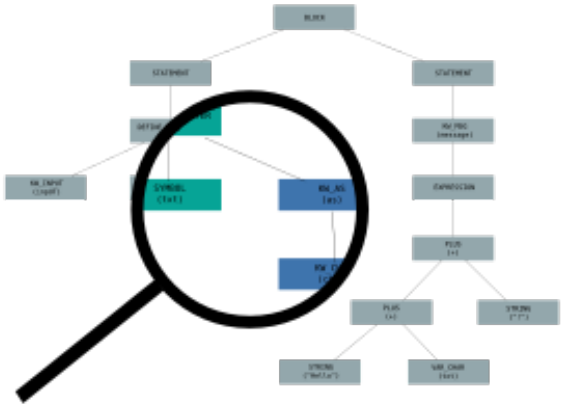
# Agenda

- Background
- Handling Flexibility and Scale
- Defining and Refining the Call Graph
- Call Graph Reports
- Visualization
- How to Get Started
- Planned Improvements

FWD analytics

# Background

# FWD is an **open source** toolset for **modernizing** 4GL applications.



## Explore

Tools for reporting, statistics and analysis, which expose the inner workings of applications.

## Transform

Fully automated transformation and modernization of entire applications without a manual rewrite.

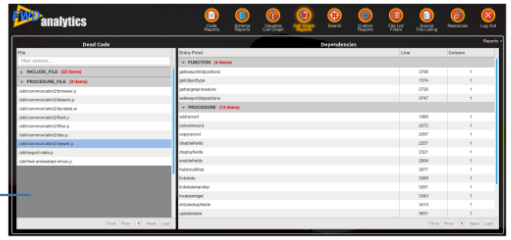

## Transcend

Compatible replacement for OpenEdge® which leverages an enhanced runtime, an upgraded 4GL language and Java to modernize and evolve applications.

FWD analytics Tools for reporting, statistics and analysis, which **expose the inner workings of an application**.

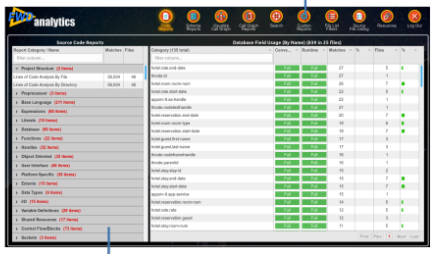
Search for arbitrary 4GL syntax to find exact answers to questions that were previously difficult or impossible to answer.

Call Graph Analysis determines program dependencies, missing code and explores all reachable code paths. Reports allow identification and removal of dead code reducing application size by 25% to 40%



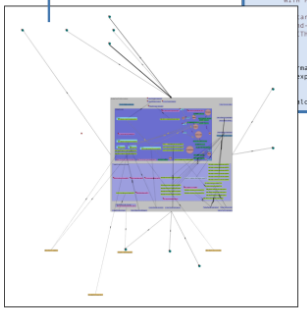
# Explore

Build custom reports to meet specific needs.

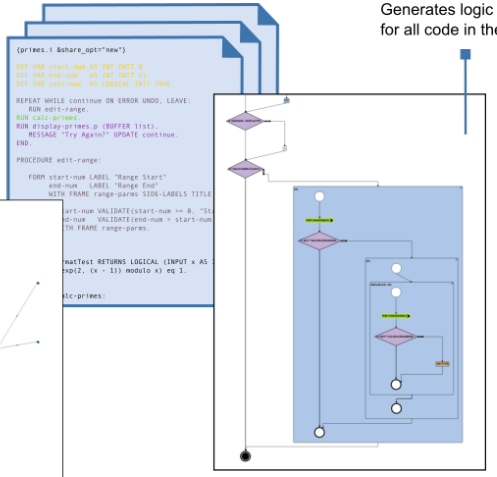


Built-in reports and statistics provide hundreds of predefined views of the code and schemata.

Visualize and interactively explore the call graph.



Generates logic flow charts for all code in the application.



**Advantages**

- Reduce development effort.
- Improve code quality.
- Deeply understand existing code.
- Compensate for missing documentation,
- Empower developers to more capably handle:
  - The most complex refactoring and modernization projects.
  - Making changes at scale, even with the largest of applications.

# Why Use Call Graph Analysis?

- Answer difficult questions using built-in reports:
  - What code can be reached from a given location?
  - What dependencies exist for a given program?
  - What code is missing from the project?
  - What code is dead (unreachable)?
- Most applications have between 25% and 40% dead code. Eliminating dead code saves significant future development time.
- Extend it with custom tools or reports.

# Why Use Call Graph Analysis?

- Reduce development effort.
- Improve code quality.
- Deeply understand and explore existing code.
- Compensate for missing documentation.
- Empower developers to more capably handle:
  - The most complex refactoring, transformation and modernization problems; AND
  - Making changes at scale, even with the largest of applications.

FWD analytics

# Handling Flexibility and Scale

# Your Source Is Not Helping

- Programmatic analysis of an application needs to be aware of the 4GL language syntax.
- 4GL syntax is exceptionally tricky in this regard.
- Non-Regular Code
  - **different text, same meaning**
  - Case Insensitivity, Keyword Abbreviations, Database Symbol Abbreviations, Optional DB Name Qualifiers, Keyword Synonyms, Flexible Options/Clauses Ordering...
- Ambiguous Code
  - **same text, different meaning**
  - Punctuation and Keyword Overloading, 19 Namespaces, Unreserved Keywords...

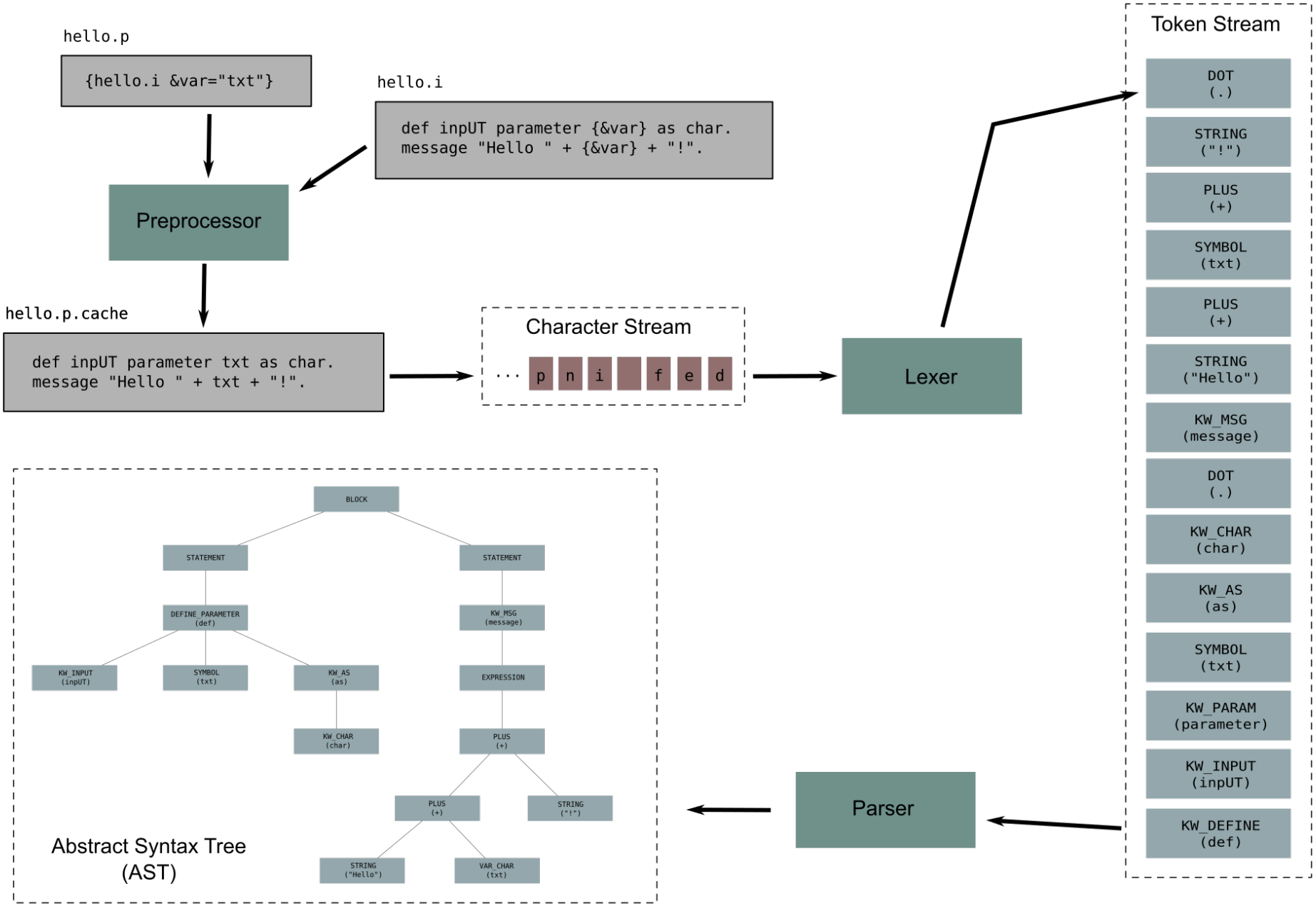
# Syntax Flexibility Increases Cost

- The more flexible the language's grammar, the larger the set of possible valid constructs that can be written by the programmer.
- This may lead to marginal time savings when creating new programs.
- That increased flexibility magnifies the long term cost of reading, maintenance, debugging, support and refactoring.
- **Greater flexibility in syntax results in greater long term cost over the life cycle of an application.**

# Abstract Syntax Trees (ASTs)

- To enable proper analysis of code, we must transform the text into a data structure that represents the purest form of the code.
- ASTs represent the code's language syntax without syntactic sugar.  The result is regular and unambiguous.
- This allows the **meaning of the code** (its semantics) to be separated from the messiness of the representation (the highly varying syntax).

# File -> Char -> Token -> Tree



# Scale Multiplies Issues

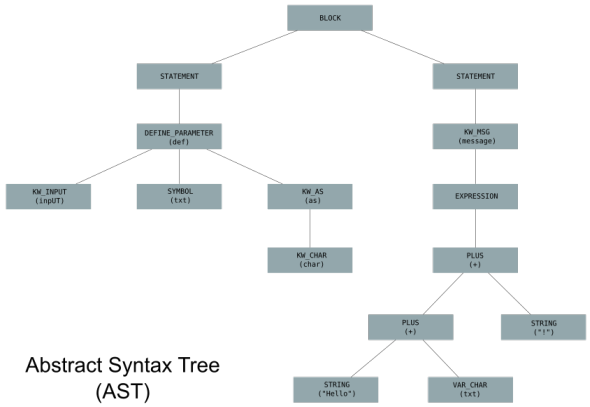
- Spread that syntax flexibility across tens of thousands of files and millions of lines of code.
- How do you find specific patterns?
- How do you even know if you have found all matches?
- Brute force is not enough.
- **Scale makes hard problems impossible (or at least impractical).**
- Automation is the only practical solution.

# TRee Processing Language (TRPL)

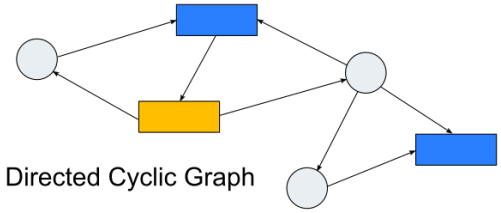
- FWD provides tools to parse an entire application.
- Each source file and each schema file (.df) will be represented as an AST.
- TRPL is the analysis and transformation toolset in FWD which can operate on the entire set of ASTs as a batch.
- When you process trees, it is commonly called a tree walk.
- TRPL includes an engine that handles the tree walking for programs written in the TRPL language.

# Trees Cannot Model a Call Graph

- Trees (ASTs) are “Directed Acyclic Graphs”:
  - a single root node
  - a directed relationship “downward” from parent node to child node
  - cycles (connections from a node to an already existing part of the tree) are not allowed
  - each child can have 1 and only 1 parent
  - each node can itself have 0 or more children
  - this is used to model a hierarchical problem
- Call Graphs require “Directed Cyclic Graphs”:
  - many root nodes
  - caller to callee directed relationships between nodes
  - cycles are allowed and expected
  - this is used to model a network problem




Abstract Syntax Tree (AST)



Directed Cyclic Graph

# What is the Call Graph?

- Tools for analysis of call paths and reachable code.
- Uses TRPL to process the ASTs for an entire application.
- Creates a graph database (Janus Graph) containing application vertices and edges:
  - **call targets** - callable blocks of code (functions, procedures, OO methods, triggers, etc)
  - **call sites** – invocation locations (RUN statements, function calls, etc)
  - Linkages between call sites and the call targets they invoke.
- Uses TinkerPop and the Gremlin graph traversal language to traverse the graph and calculate reports.

# Key Elements

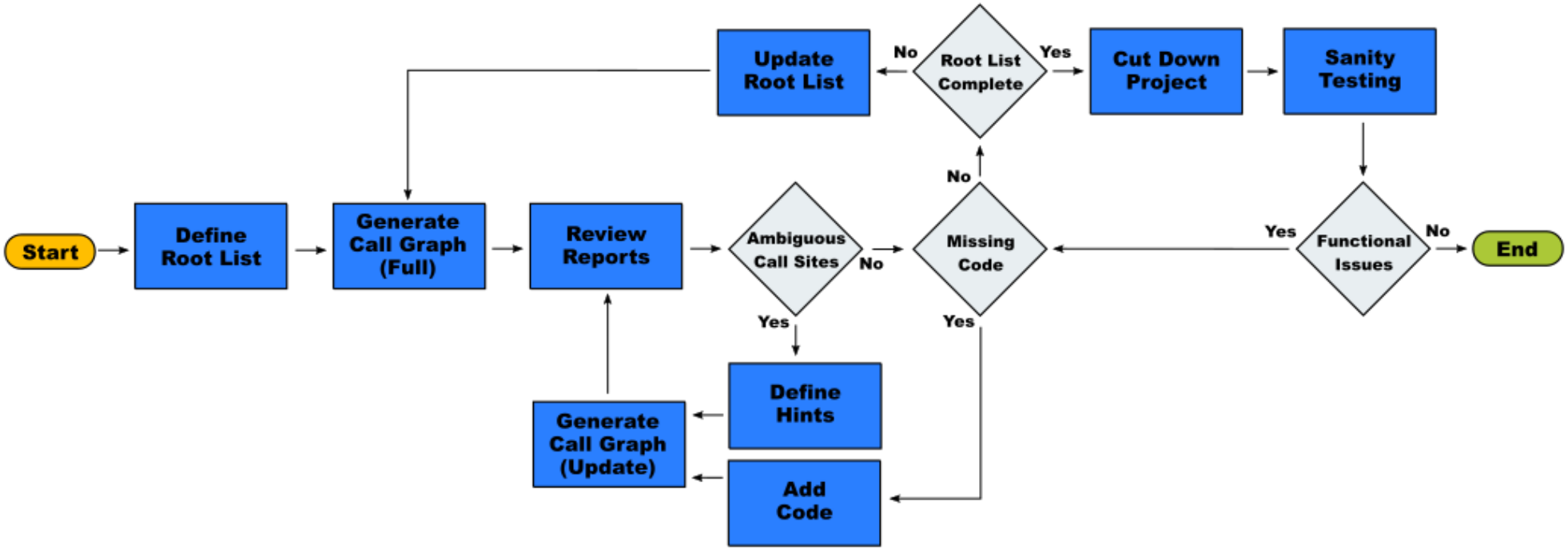
- A **graph node** (vertex) is represented by a portion of your application code, which can be:
  - Call sites, 4GL code that invokes other code (dynamic or static):
    - RUN statements, function calls, SUPER(), RUN SUPER
    - OO method calls, property getter/setter, constructors and destructors
    - Schema defined table triggers
    - UI statements raising events
  - Call targets, 4GL code being invoked by other code:
    - 4GL programs: functions, internal procedures, triggers
    - OO class hierarchy, methods, properties
    - UI triggers
    - Schema trigger procedures
  - External dependencies, for calls performed outside of 4GL code:
    - OS native APIs
    - OS process launch commands
    - R-code compiled programs
    - Socket, AppServer or Web Service
    - COM Automation, DDE Servers or OCX Controls
  - Missing code, created for call sites targeting non-existent 4GL code.
  - Include files

# Key Elements

- A **graph link** (edge) is represented by a dependency between two graph nodes:
  - Include dependencies.
  - Class hierarchy.
  - Raised events.
  - Function/procedure references to their definition (for IN SUPER or IN handle).
  - Call sites and their targets.
- **Root list**: the program files which act as entry points into your application.
- Automatic ADM/ADM2 call disambiguation.


# Defining and Refining the Call Graph

# Defining the Call Graph



# Iterative Process

1. Define the root list.
2. Run the call graph analyzer.
   - **full** generation (non-update mode) when the root list has been modified
   - **incremental** generation (update mode) when hints are added or new code is added
3. Review call graph reports for missing and ambiguous code.
4. Define hints to resolve the ambiguous code, change the root list or add/modify code to the project for missing programs.
5. Repeat from step 2 until there is nothing ambiguous or missing.
6. Use the call graph reports to eliminate dead code from the project.
7. Compile and test the resulting "cut down" distribution to sanity check the validity of the call graph.
8. If there are any issues that show some code that was thought to be dead is actually needed, add these program files back (with their associated hints) and repeat from step 2.
9. If no functional issues can be found in testing, the call graph is correct and complete.
10. The resulting source code represents your application with all the code needed to run correctly.

# Define the Root List

- Initially the Call Graph Analyzer will only process the programs in the root list and reachable external programs (e.g. via RUN statements).
- External programs not reachable from any of the root programs will not be processed (they are declared "dead").
- The root list must contain all programs (and only those programs) which are accessible from the ‘outside world’:
  - programs executed by a user, a batch process or via the appserver
  - programs ran from command line or shell scripts
  - programs targeted by schema triggers
  - programs configured in .pf files
- Example:

```
<?xml version="1.0"?>
<roots>
  <node filename="./abl/some/application/folder/some-external-program.p.ast" />
  <node folder="./abl/another/application/folder/" pattern="*.ast" recursive="true"/>
</roots>
```

# Ambiguous Call Sites

- Call sites which have their targets resolved at runtime are marked ambiguous (i.e. RUN VALUE, DYNAMIC-FUNCTION(char-expr), etc).
- Use the **Ambiguous Call Sites** built-in report, to see the ambiguous call-sites.
- Analyze how each ambiguous call site determines (calculates, looks up...) the real targets.
- Resolve these ambiguities via hints.
  - Build a **.hints** file for each ambiguous program.
  - Use the report to determine the hint IDs.
- Re-generate the call graph to apply the changes.

# Ambiguous Call Sites

- For a **./abl/some/app/program.p**, create a **./abl/some/app/program.p.hints** file.
- For each ambiguous call-site, you will be able to:
  - specify multiple targets via the **string[]** datatype
  - specify a single target via **string** datatype
  - ignore this call-site, by specifying an empty array
- The **<hint_ID>** is provided in the **Ambiguous Call-Sites** report.
- Example:

```
<?xml version="1.0"?>
<hints>
   <!-- multiple targets -->
   <uast name="<hint_ID>" datatype="string[]">
      <array-val value="target1" />
      <array-val value="target2" />
   </uast>
   <!-- single target -->
   <uast name="<hint_ID>" datatype="string" value=”target1”/>
   <!-- just ignore this ambiguous call-site -->
   <uast name="<hint_ID>" datatype="string[]" />
</hints>
```

# Ambiguous Call Sites

- The hint name uses the `CALL_SITE_TYPE_#` format
- **`CALL_SITE_TYPE`** is the call site's token ID (like RUN_VALUE, KW_DYN_FUNC).
- **#** is a 0-based counter to distinguish multiple call sites of the same type.
- hint's value may be suffixed with an internal entry name, while the prefix is the external program name (for RUN_VALUE and KW_DYN_FUNC); use the ':' char as separator.
- Example:

```
<?xml version="1.0"?>
<hints>
  <!-- multiple targets -->
  <uast name="RUN_VALUE_0" datatype="string[]">
    <array-val value="program1.p" />
    <array-val value="program2.p" />
   <array-val value="program3.p:someInternalProc" />
  </uast>
</hints>
```

# Ambiguous Calls to 4GL Code

- Ambiguous call sites which target 4GL code need to be disambiguated for the call graph to be complete.
- Each call site may disambiguate to:
  - One or more external program files
  - One or more internal procedure or function in one or more external program files

| Call Site Type | Hint Name | Statement |
|---|---|---|
| RUN_VALUE | RUN_VALUE_# | RUN VALUE(expr) |
| RUN_VALUE_ON_SERVER | RUN_VALUE_ON_SERVER_# | RUN VALUE(expr) ON SERVER <server> |
| RUN_PORT_TYPE_VALUE_ON_SERVER | RUN_PORT_TYPE_VALUE_ON_SERVER_# | RUN VALUE(expr) SET <handle> ON SERVER <server> |
| KW_SUPER | KW_SUPER_# | SUPER() |
| RUN_SUPER | RUN_SUPER_# | RUN SUPER. |
| KW_DYN_FUNC | KW_DYN_FUNC_# | DYNAMIC-FUNCTION |
| INTERNAL_PROCEDURE | INTERNAL_PROCEDURE_# | PROCEDURE ... IN SUPER |
| FUNCTION | FUNCTION_# | FUNCTION IN SUPER<br>FUNCTION IN handle |
| METHOD_INVOCATION | METHOD_INVOCATION_# | DYNAMIC-NEW<br>DYNAMIC-INVOKE |

# Missing Code

- Use the **Missing Call-Site Targets** report to view missing 4GL code, which might mean:
  - problems in the code, where dead external programs were removed, but references to it still remain in other parts of programs (which may in turn be dead code)
  - external programs which were specified during call-site disambiguation, but are not part of the code set being parsed
  - case-sensitivity inconsistencies - a wrong case was specified in the p2j.cfg.xml file, and program file names can't be matched properly
  - AppServer programs invoked via RUN statement, which are not included in the code being parsed
  - functions or internal procedures part of the Possenet framework, which are called from ADM/ADM2 code which is not in use by your application
  - functions or internal procedures which are assumed to be invoked by the Possenet framework, but are optional and not in use by your ADM/ADM2 windows

# Missing Code

- Investigate each missing target.
- Possible resolutions:
  - If the program was removed incorrectly, add it back. This may happen while ambiguous call sites still exist, and the graph is not populated fully and correctly.
  - If some program file was never added to the file-cvt-list.txt or it doesn't exist in the source tree, add it..
  - If the call site is dead-code, remove the code.
  - If the case-sensitivity configuration in p2j.cfg.xml is incorrect fix the configuration.
- Parsing and call graph generation will have to be done again for all of these cases.

# Limitations

- Unsupported entry points or call-sites:
  - PUBLISH and SUBSCRIBE
  - ON ... PERSISTENT RUN triggers
  - READ-RESPONSE and CONNECT socket events
  - PROCEDURE-COMPLETE event for RUN ... ASYNC calls
  - OO class events
  - OCX Event Procedures
  - DDE-NOTIFY trigger
  - Exported web services.
  - Web Service and SOAP invocations are not disambiguated from normal procedure calls, so they will be matched the same as an internal procedure call.
  - Stored procedure invocations are not processed.
- .NET and OpenEdge classes are not differentiated. To determine if a class is an external dependency (.NET class) or a 4GL dependency (OO 4GL) the user should look at the class namespace or the dotnet-cls annotation to check if a class is .NET or 4GL.


# Call Graph Reports

# Ambiguous Call Sites

- Identifies all ambiguous call sites having targets computed via a dynamic expression, at runtime.
  - From user input.
  - Read from the database.
  - Read from some file or other configuration source.
  - Calculated.
  - Any combination of the above.
- A 4GL developer with application knowledge can use hints to disambiguate each call site.
- As long as there are ambiguous call sites existing in the report, the Dead Code report will be potentially incorrect.

# Dead Code

- Performs a Depth-First Search, starting with the program(s) in the root list.
- All unreachable code is reported as ‘dead’.
- Dead code may be:
  - External programs, internal procedures or user-defined functions.
  - OO methods, properties, class or interfaces.
  - Include files.
  - UI triggers.
- Details about the content of dead code:
  - For external programs, shows all defined internal entries and the call sites which are contained in this external program.
  - For include files, shows all its references. An include file may be part of a connected sub-graph, which is by itself fully dead.
  - For internal procedures,user-defined functions or UI triggers, shows all contained call sites.
  - For class or interface definitions, shows its defined methods and properties.
  - For method definitions, shows the contained call sites.
  - For property getter and/or setter definitions, shows the contained call sites.

# Missing Call Targets

- Find all 4GL code for which there is an attempt to invoke, but which does not exist:
  - External programs (via RUN statements).
  - User-defined functions (via DYNAMIC-FUNCTION).
  - Internal procedures (via RUN or RUN … ASYNC EVENT-PROCEDURE internal-proc).

# External Dependencies

- Find all calls outside of the 4GL code:
  - Procedure Library Member - the RUN statements which target compiled R-CODE legacy code
  - Web Service - the target of a RUN port-type SET hport ON SERVER h statement, which represents the name of the WSDL port type.
  - Shared Library - the OS shared library name to which the target of a PROCEDURE nativeApi EXTERNAL sharedLibName procedure belongs.
  - Native API - the API targeted by a PROCEDURE ... EXTERNAL procedure.
  - Child Process - the command for an OS process launching statement, like UNIX, BTOS, INPUT THROUGH.
  - Socket - the configuration for a connection to/from a socket, AppServer or Web Service.
  - COM Automation - the target of an automation object being created via a CREATE automation-object statement; it specifies the details about the created automation object.
  - DDE Server - the target of a DDE INITIATE statement; it specifies details about the DDE server.
  - OCX Control - the target of a loadControls method call; it specifies the name of the control file
- Analyze each dependency and decide if:
  - a user which requires access to native libraries will be deployed to the web or not
  - which network resources the clients need access to
  - which OS libraries require to be installed on client machines
  - linked to dead code, remove it from your installation process/documentation.

# Schema Triggers

- Find any inconsistencies related to:
  - **External program not a table trigger** – schema tables references a trigger program which is not defined as such.
  - **Trigger type mismatch** – trigger type at the schema table and at the trigger program are not the same.
  - **Different table** – schema trigger and the trigger program don’t specify the same table.
  - **Missing external program** – the schema trigger references a missing program.
  - **Dead table trigger** – trigger program is not used by any of the schema triggers.

# Dependencies

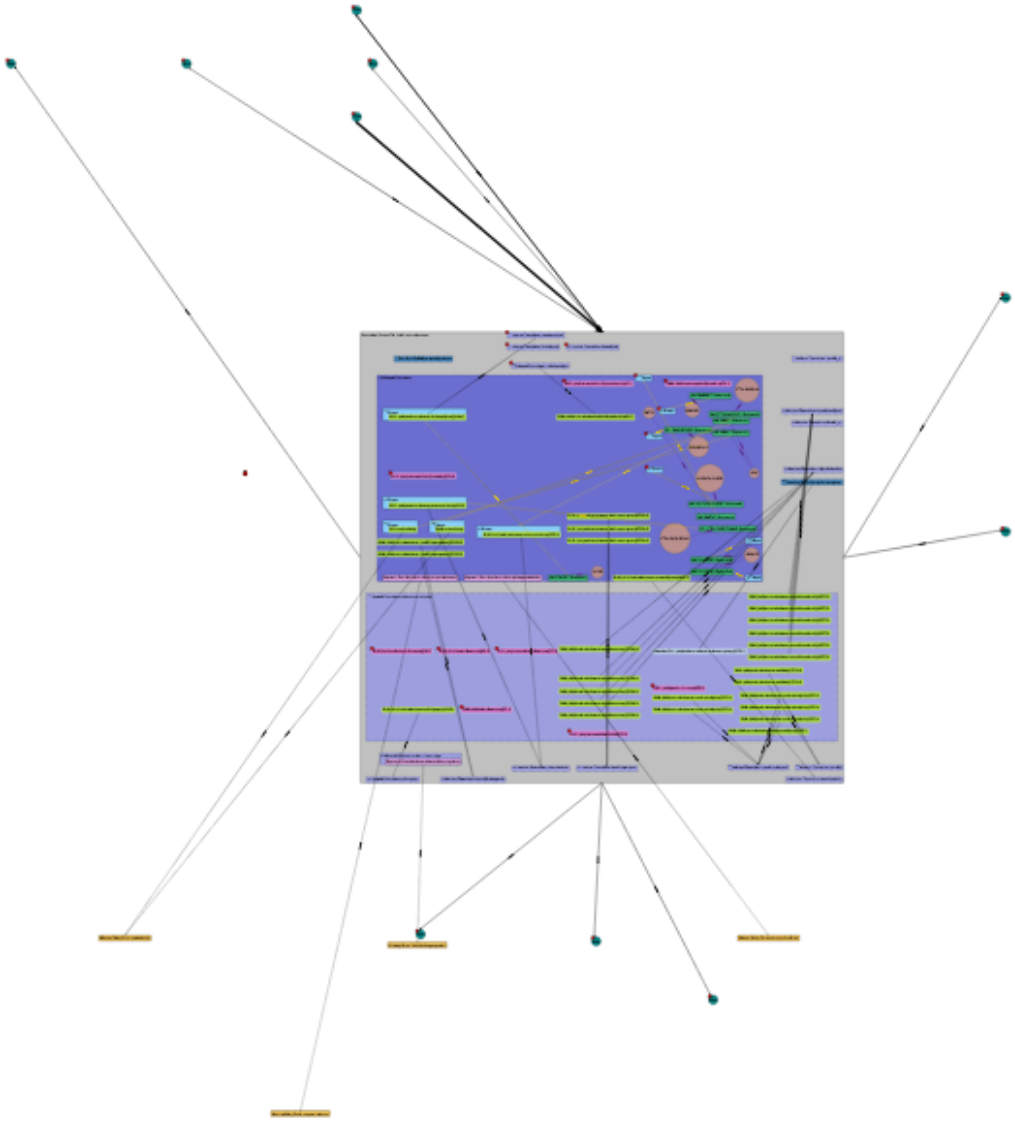
- Identify all dependencies for a certain file resource.
- A procedure or class file will show you:
  - Call sites which invoke code:
    - In other 4GL programs
    - Missing code
    - External targets
  - Include files which are used by this program.
  - References to other internal procedures or user-defined functions (i.e. IN SUPER functions references).
- An include file will show you the programs and location where it is included.
- A schema file will show you all triggers referenced by this schema.

FWD analytics

# Visualization

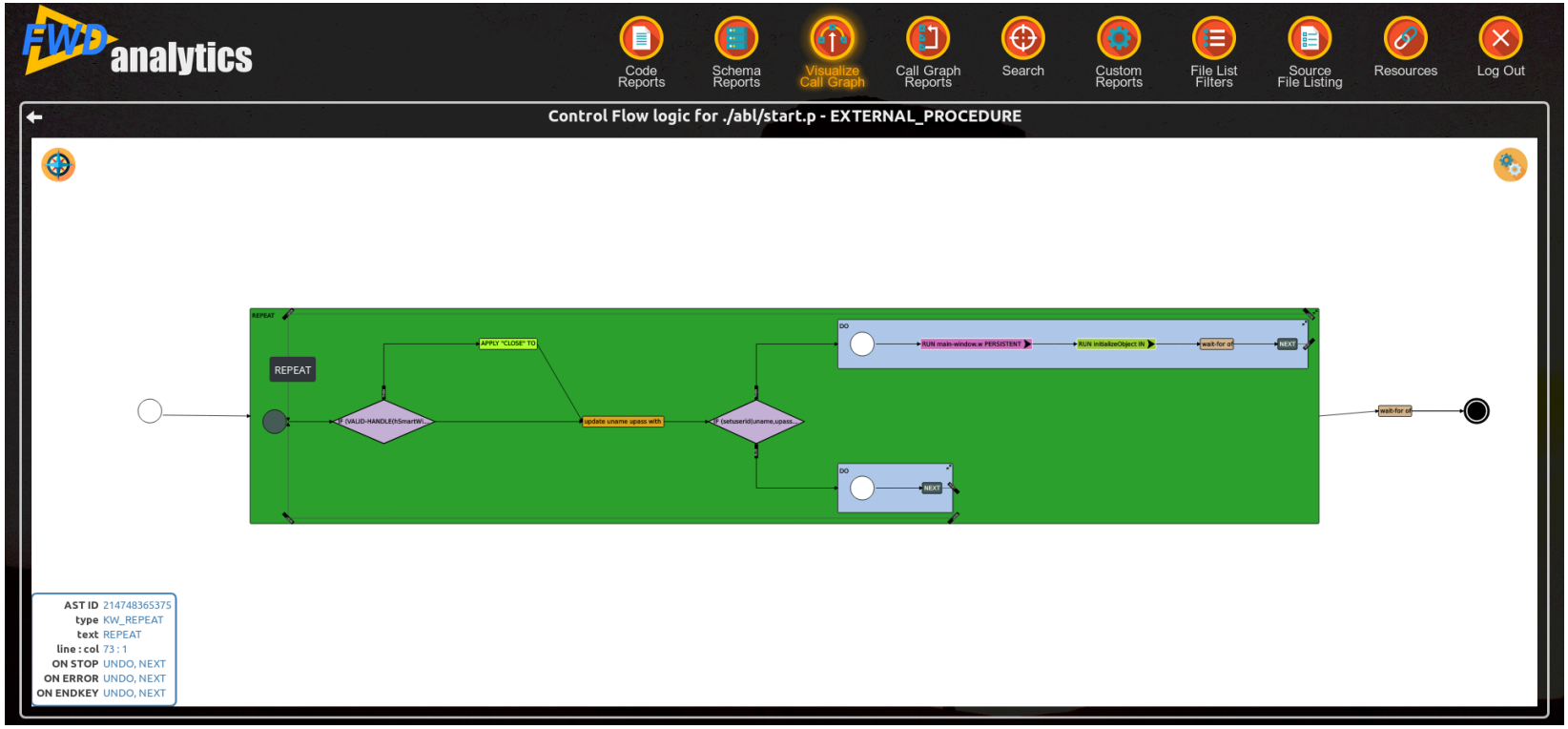
# Call Graph Visualization

- Live model of the call tree using a “force directed graph”.
- User can load the graph from arbitrary locations.
- Traverse to “More” links with SHIFT-click (load just that node) or CTRL-click (add node to current graph snippet).
- Use this to explore the application.
- Useful to identify macro patterns that would be hard to see by reading source code.
- Zoom with mouse wheel, pan with drag on background.
- Still in very active development, this is an early version.
- Drag nodes to move them around. Hover to see details.
- SHIFT-click on AST nodes to go to the source/AST view.



# Logic Flow Charts

- Live model of the logic flow chart for every callable block of code in the application.
- Accessed via the “Y” flow icon in the call graph visualization.
- Traverse to called code from call sites such as RUN.
- This is a form of documentation.
- Useful to identify macro patterns that would be hard to see by reading source code.
- Zoom with mouse wheel, pan with drag on background.
- Drag nodes to move them around. Hover to see details.
- SHIFT-click on AST nodes to go to the source/AST view.



FWD analytics

# How to Get Started

# How to Get Started

- Download and install FWD.
- Download one of the sample template projects (there is one for ChUI and one for GUI).
- Follow the “Getting Started” instructions to get the template project installed and configured for your application code, including placing your code and schemata into the template project.
- Run the `ant report_server` target.
- Start the report server.
- Access the server at port 9443 via a browser.
- Full details of this process and all documentation: **https://proj.goldencode.com/projects/p2j/wiki/Code_Analytics**

FWD analytics

# Planned Improvements

# Planned Improvements

- Add more built-in call-graph analysis and reports.  One example: identifying all locations that use a specific NEW SHARED variable (and the inverse).
- Instrumentation of 4GL code to capture hints via actual usage of the application (testing or real usage).
- Support for all remaining call sites and call targets not yet implemented.
- Improve automatic disambiguation of call targets, especially for ADM/ADM2 code.
- Web UI for encoding creating and defining the call graph including hints and the root list.
- Improve usability of the visualization.

# Find Us On the Web!

www.beyondabl.com

facebook.com/beyondabl

twitter.com/beyondabl

linkedin.com/company/fwd-project

youtube.com/channel/
UCk3pga7EKxAQVOV_CiYOR7g