

Using Ada for software development tools. RapiTime, a case study.

Dr. Guillem Bernat



Outline

- Rapita Systems
- RapiTime toolset
 - Command line tools
 - C and Ada source code analysis
 - Trace processing
 - Written in Ada
 - GUI
 - Eclipse Report viewer
 - Written in Java
- JavAda: Ada to Java integration.
- Summary: Lessons learnt, glory and pain

Rapita Systems Ltd.

- Who?
 - Provider of high quality Tools and Services for timing analysis of real-time systems
- RapiTime Toolset for WCET (Worst-Case Execution Time) analysis and optimisation
 - RapiTime *plugins*:
 - RapiTime C, Ada, and object code specific versions
 - RapiCover: On target test coverage (up to MC/DC)
 - Eclipse RapiTime viewer
- Customers in
 - Avionics:
 - BAE Systems (Hawk jet Trainer),
 - Honeywell (Boeing 787 Dreamliner flight controls),
 - Liebherr (Superjet100 flight controls),...
 - Space:
 - European Space Agency, Cache analysis for LEON2
 - Thales-Alenia Space,
 - Telecomms: Ericsson
 - Automotive: AUDI, BMW, Mentor Graphics,

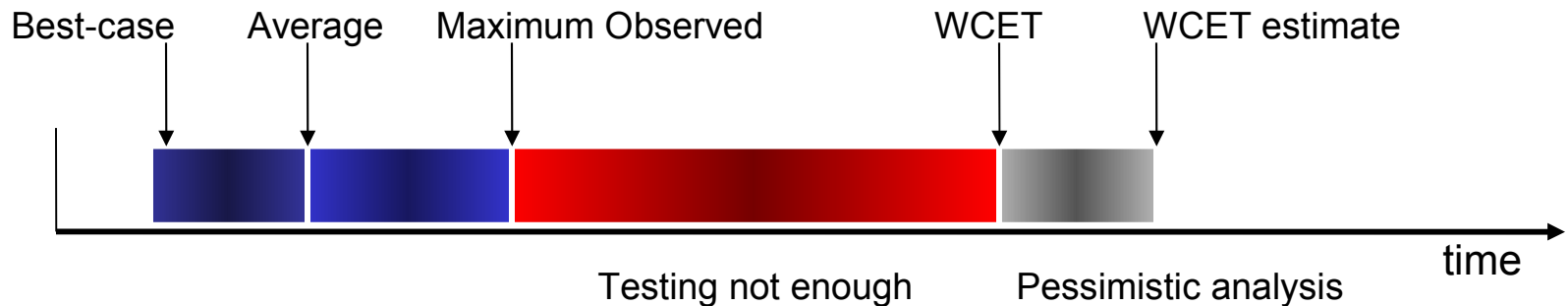
+ RapiTime™

Worst-case execution time analysis +

Terminology

Terminology for execution time:

- **Best-Case** execution time
 - **Average** execution time
 - **Maximum** observed (usually too low, testing not enough)
 - **WCET**: *longest time a piece of code can take to run* (possibly non computable)
 - **WCET estimate**
 - (**Response time**: including interference from other tasks and interrupts.)
- **RapiTime**: *Software tool that computes a WCET estimate based on integration of measurements and program analysis.*



Key Features

- **Accurate** WCET
 - Ensures timing constraints are met without the need for over-specified hardware.
- Identifies code on the **worst-case path**
 - **Colourized Source Code**
- Analysis of **worst case hotspots. Optimisation opportunities**
 - Target code that contributes most to the WCET
 - Case study: optimised WCET by 23% by changing 1.2% of code
 - Profiler of the longest paths in the program.
- **Code coverage** analysis
 - Ensures testing covers all sub-paths
- **Execution Time Profiles**
 - Show **variability** of execution times due to hardware effects
- **Minimal Porting Costs**

WCET Analysis: approaches

- Extensive testing :
 - Measuring *end to end* execution time of the program
 - High level:
 - (-) No knowledge of the structure of the program
 - (-) Not sure that the longest path has been tested
 - (-) Expensive, exhaustive testing not feasible in practice
 - Low Level:
 - (+) Observes real hardware and software
 - (+) No knowledge/assumptions of the hardware needed
- Static WCET
 - Analysis of the program without running it. Timing model of CPU.
 - High-level:
 - (+) finds the longest path + program structure
 - Low-level:
 - (-) HW is (very) difficult to model and validate (DSP, MCU, buses, etc)
 - (-) Pessimistic (simplistic?)
 - (-) No information on how likely the worst-case behaviour is
- RapiTime Solution: Hybrid approach:
 - *Measurement Based WCET analysis ...*

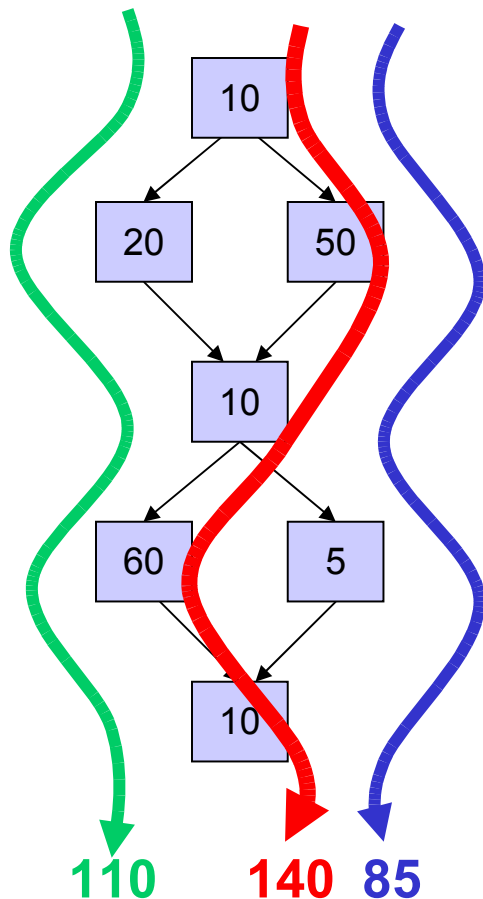
WCET Analysis: approaches

- *Measurement-based WCET analysis: RapiTime*
 - “The best model of a system is the system itself”
- Combines best features of both approaches
 - High-level:
 - (+) Use static analysis techniques to find the longest path through a program
 - Low-level:
 - (+) Determines timings from actual measurements
 - (+) Full distributions of execution times
 - (+) No “model” approximations!
 - (+) Easy and quick to port
- Integration of functional testing and timing analysis
 - Enables comparing WC measured with WC computed
- Automatic tool support

RapiTime “minimal example”

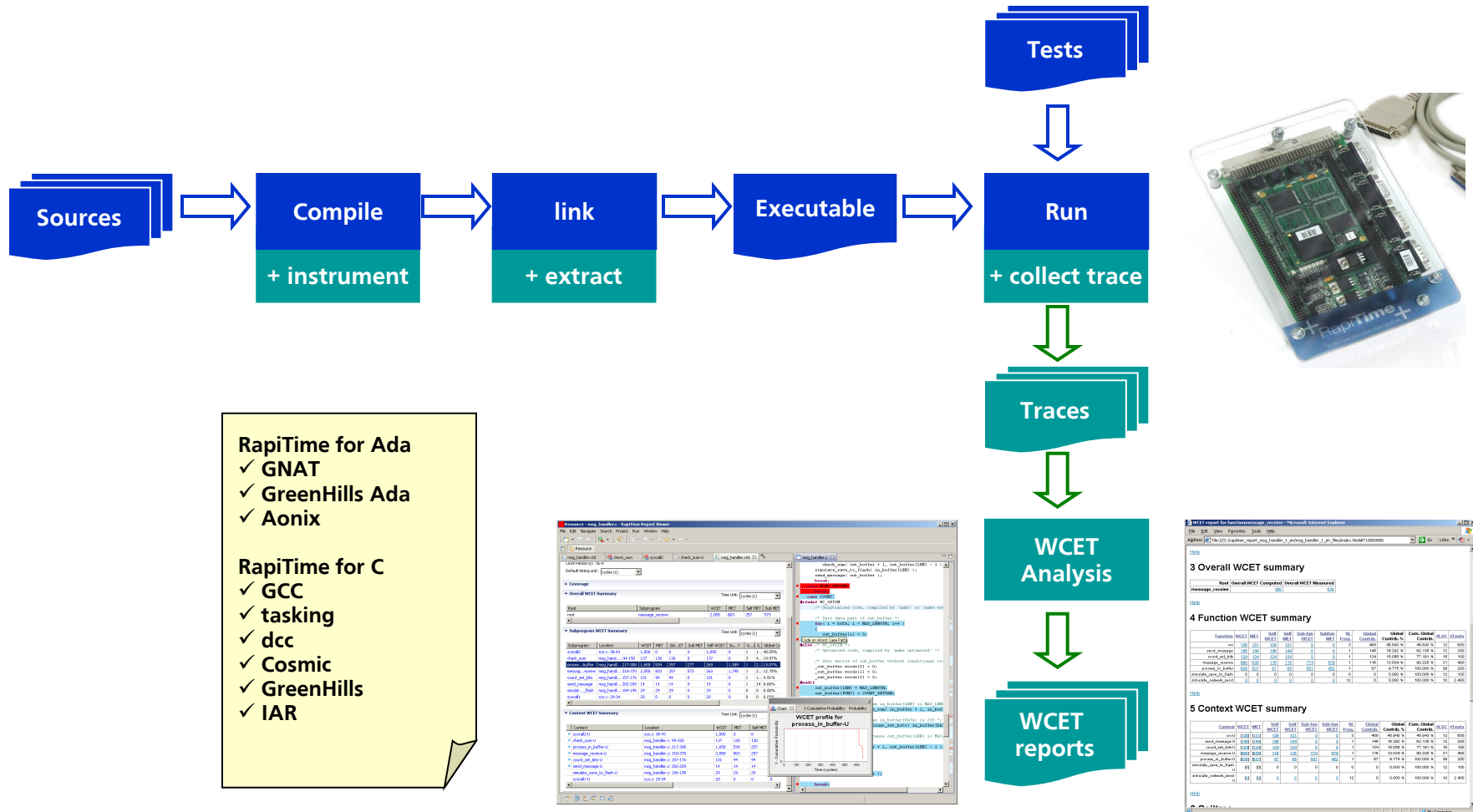
```
if a then  
    f1; /* 20us */  
else  
    f2; /* 50us */  
end if  
  
...  
if b then  
    f3; /* 60us */  
else  
    f4; /* 5us */  
end if;
```

RapiTime “minimal example”



- Tested, the “green” and “blue” paths
- Maximum measured is 110,
- RapiTime builds the structure of the program
 - Not tested all possible paths
 - But worst path computed “statically”
- And determines that the “red” path is possible, and computes WCET of 140.
- NOTE!: Green and Blue tests provide 100% MC/DC coverage -> But not enough for timing!
- Annotations can be added to remove infeasible paths from the analysis

RapiTime Analysis Process and toolchain

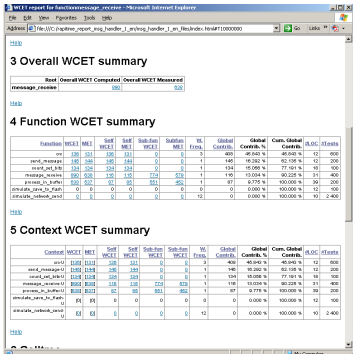
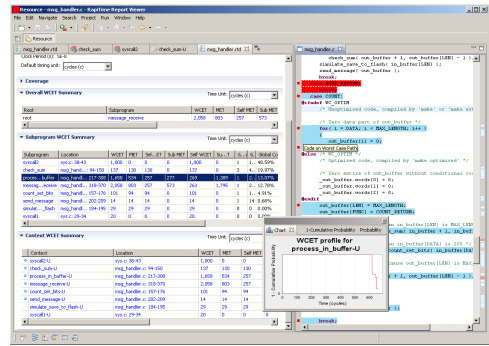


RapiTime for Ada

- ✓ GNAT
- ✓ GreenHills Ada
- ✓ Aonix

RapiTime for C

- ✓ GCC
- ✓ tasking
- ✓ dcc
- ✓ Cosmic
- ✓ GreenHills
- ✓ IAR



Source code instrumentation

- **cins:**
 - C source code instrumenter
 - Input : preprocessed C source file + annotations
 - Output: instrumented C source file
 - C parser written in Ada.
 - Uses Aflex and Ayacc
- **cextract**
 - C source code analyser
 - Input: instrumented C source file
 - Output: XSC: RapiTime internal representation of the structure of a program
- C Compiler flavours
 - Handling non-standard compiler specific language extensions
 - Done mainly by pre-preprocessing the token list

Source code : Ada

- **gnatins**
 - ASIS based Ada source code instrumenter
 - And structure extractor
 - Similar to cins
 - Handling compiler flavors
- Desiderata
 - Some errors and limitations identified in ASIS
 - Fixed very quickly!

Program analysis

- **xstutils**
 - Post-processing tree structure
 - Heavyweight tree manipulation routines
 - Input: collection of structures of programs (XSC)
 - Output: “Executable” parsing engine
 - Using custom made container library
 - Gradually converting it using Ada05 container library
- Results:
 - Serialization of complex data structures (*a la* ML)
 - Strong type checking is excellent!
 -

Trace processing

- **traceutils**
 - Input:
 - flexible trace format (txt, binary dumps, logic analyzer outputs, simulators etc) (can be **very** large)
 - Trace filters
 - Output: common RapiTime compressed trace format
 - Main issues:
 - Handling very large traces (gigabytes)
 - Fast and Memory efficient
-

Trace processing

- **traceparser**
 - Processing trace against structure of program
 - Input : traces + structure of the program
 - Output: measured execution times
- Desiderata
 - Very efficient file_IO
 - Optimization options
 -

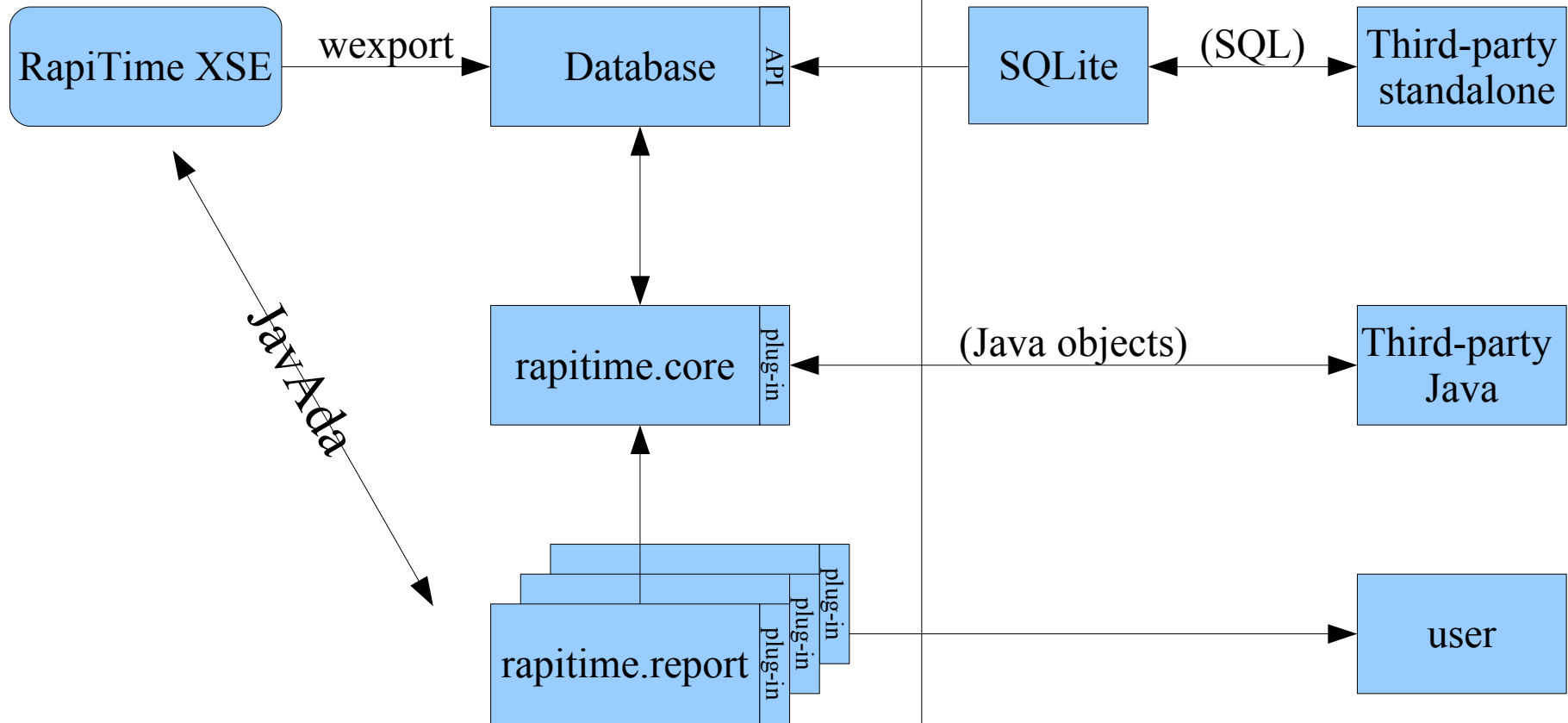
WCET analysis

- wcalc:
 - WCET calculation engine
 - Input: Measured execution times + XSE
 - Output: Computed WCET (and profiles)
 - Complex traversals on expanded call graphs
- wexport
 - Generation of timing results in unified format
 - Input: XSE with timing data
 - Output: SQL database (SQLite)
 - Thin binding
 - Desiderata: more standard third party library integration

RapiTime GUI

- Eclipse integration requested by customers
 - RapiTime Eclipse
 - Visualization of large amount of timing data
 - Interaction with other software plug-ins
 - Code editors
 - Code browsers
 -
- **reportviewer**
 - Eclipse plug-in (java)
 - Queries SQL database (SQLite, hibernate)
 - Generic Java object model derived from Ada model
 - Based on JADA (java to Ada mapper)
 -

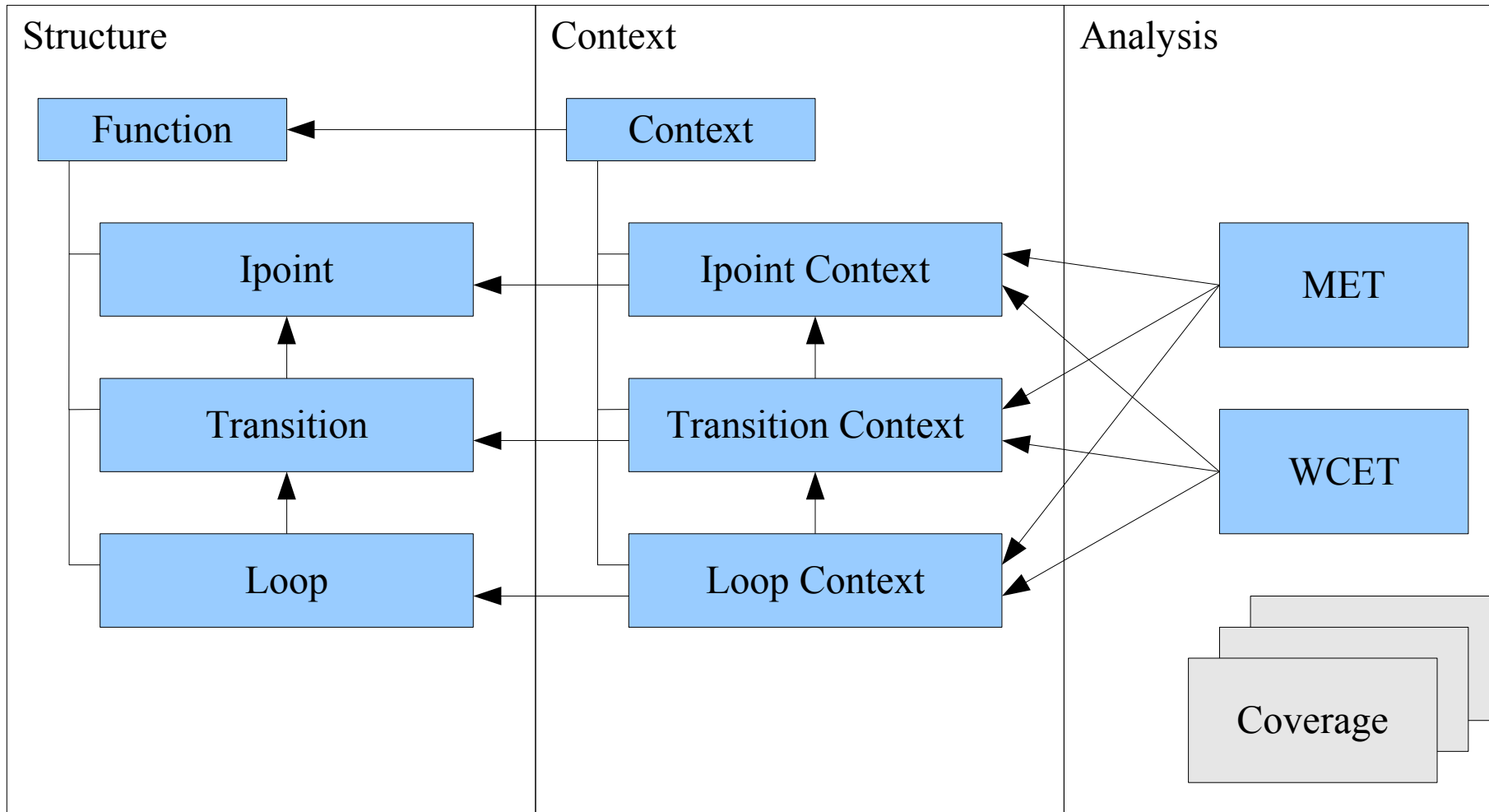
RapiTime Eclipse plugins



RapiTime Eclipse
Report Viewer

External access to
RapiTime data

Database Structure



JavAda

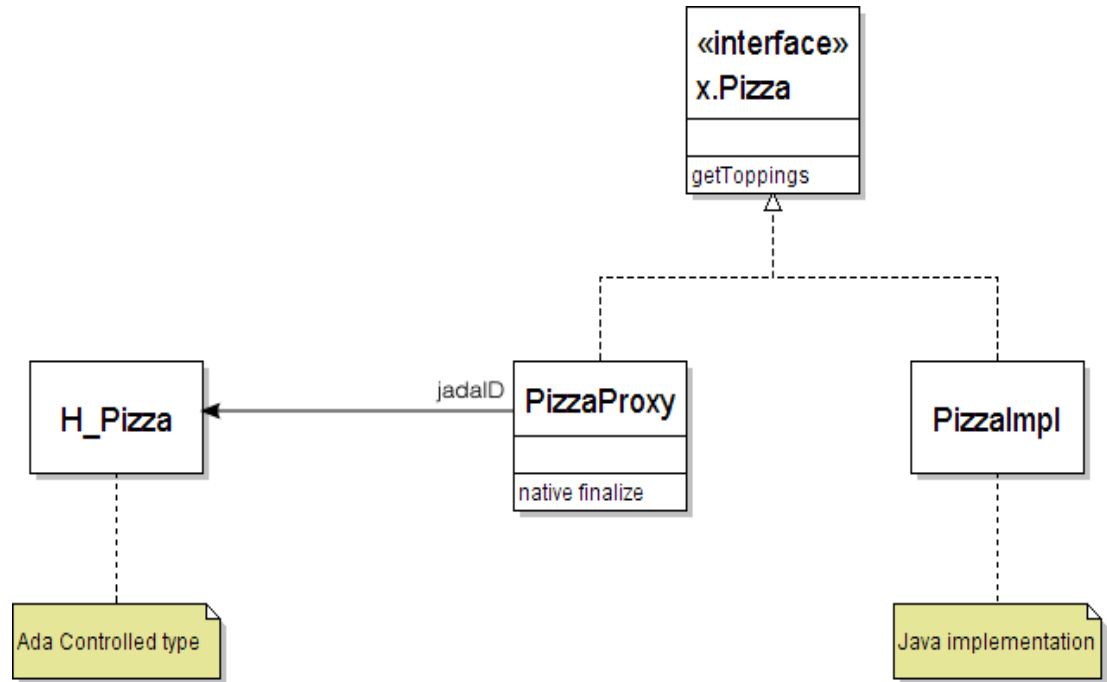
- Automatically generate bindings to allow Ada and Java to talk to each other
- OO features of Ada to create 'mirror' class hierarchies in Ada and Java.
- Loosely based on CORBA proxy/skeleton model
- Hides the interface from both Ada and Java
- Uses CORBA IDL (interface definition language) as a platform neutral description.
 - IDL parser from eclipse.sf.net to generate a model
 - Input to JET (java emitter template)
 - Builds Java classes and Ada
 -
 - <http://code.google.com/p/javada/>

Define IDL Interface

```
module x {  
    interface Toppings;  
  
    interface Pizza {  
        Toppings getToppings();  
    };  
};
```

Implementation

- Generated code tracks object references
- T_Pizza just implements the defined interface I_Pizza
- Java calls are delegated to native T_Pizza
- Ada calls are delegated to Java object



Experiences

Experiences: Good

- Don't *preach to the pope...* but
- Compiler catches lots of potential errors!
 - Run-time system does a very good job too
- Customer problems are easy to replicate and fix.
 - e.g. no chasing pointers
 - No “cores”
 - Fast turnaround fixes (<24 hours)
- Allows genuine co-development between engineers
 - Ada spec files very robust
 - Parallel development from Ada spec packages.
- Incredible productivity and Code quality
- AdaCore
 - Gnat compiler excellent support
 - GPS very productive (but a a bit unstable at times)

Experiences: (not that good)

- New staff not familiar with Ada (generally)
 - OO model is tricky
 - Not enough trained engineers
- Slower execution
 - Stack checking (13% overhead)
 - Run-time checks on
 - Overhead of abort defer regions (10% for minimal tasking)
 - Large memory footprint
 - Debugger struggles sometimes
 - TextIO is slow (use stream_IO)
- Hard to integrate with third party components
 - Lack of a standard library of components
 - Significant amount of development from scratch
 - Eclipse-> Required development of JavAda
 - Strong eclipse model needed
 - Lack of a strong GUI model

Benefits

- **Reduced development costs**
 - Systematic and **scientific approach**: Engineer timing correctness into systems rather than try and get timing bugs out
- **Shorter time to market**
 - Significant **reduction in time and effort** spent ensuring that systems meet their timing constraints
- **Minimised unit costs** in production
 - Extract maximum performance from the most cost-effective microprocessor variants
- **Improved product reliability**
 - Significant reduction in number of **timing bugs** going undetected in deployed systems
- **Protects Brand Image**
 - By maintaining high levels of reliability.