Introducing the Overture Architecture Guide

Luis Diogo Couto

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

12th Overture Workshop
21 June 2014

Overture Development Challenges

- Large code base (400k+ LoC)
- Few current developers
- Need to attract new developers
  - How to attrack them?
  - Possible new projects
- Put them in position to succeed

Why an architecture guide?

- Help new developers orient themselves
  - Hard to just “dig into the code”
  - Not enough documentation
  - Provide top-down source for all developer information
- Conduct an architectural inspection
  - Architecture is too implicit
  - We may be unaware of various issues
  - Guide is an opportunity for architectural analysis

About the guide

- (Open source) development process a little chaotic
- Sophisticated architectural descriptions not well suited
- Architecture guide should be:
  - brief
  - easily maintainable
  - used to look-up information, not guide development
- Living document available at developer wiki (visit it!)
- Currently covers non-UI components of Overture

Overview

- 2 different architectural perspectives
- Each divides tool in different components (modules) and connections

  Source Code: architecture of the tool mirrors organisation of source code files

  Functionality: each component represents single high-level functionality, regardless of source code

The 2 perspectives

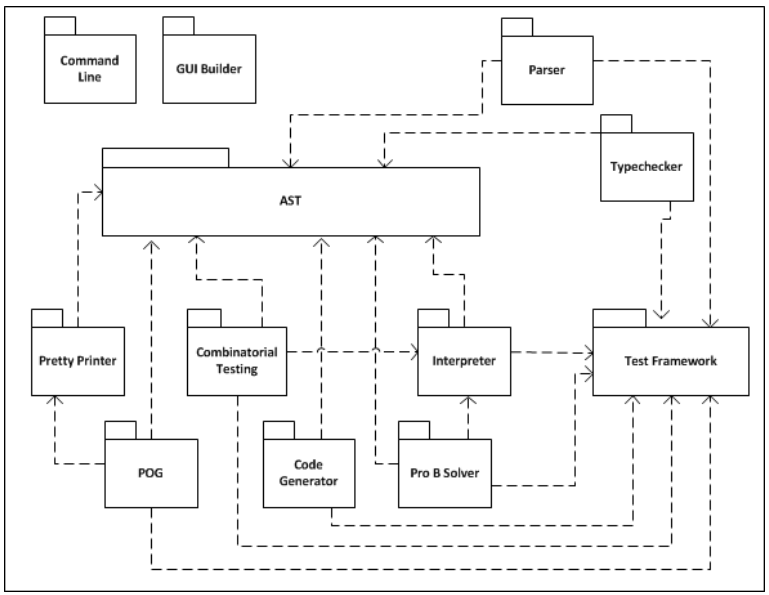


Figure : Source Code

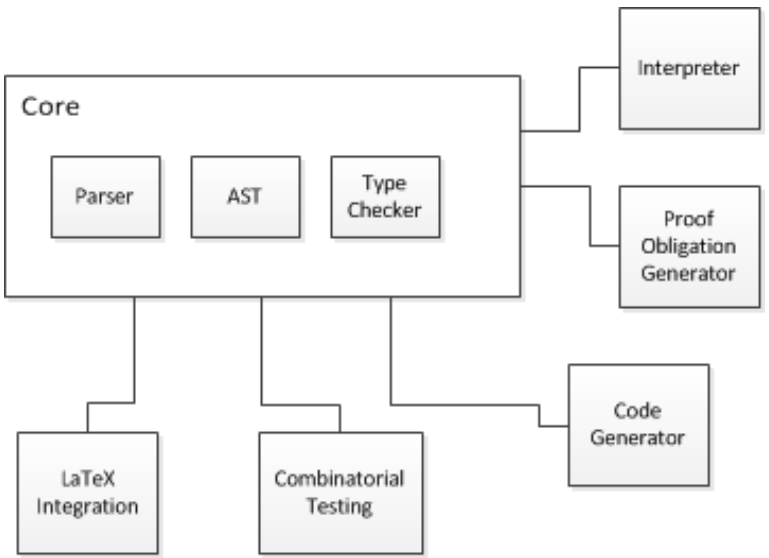


Figure : Functionality

Overview – Source code perspective

- Technical perspective; well-suited for developers
- Core modules in pure Java
  - Each provides fairly tight functionality
- IDE modules based on Eclipse
  - Provide GUI to interact with core modules
- Development cycle for new functionality:
  1. Develop core module
  2. Develop IDE module for access to core

Overview – Functionality perspective

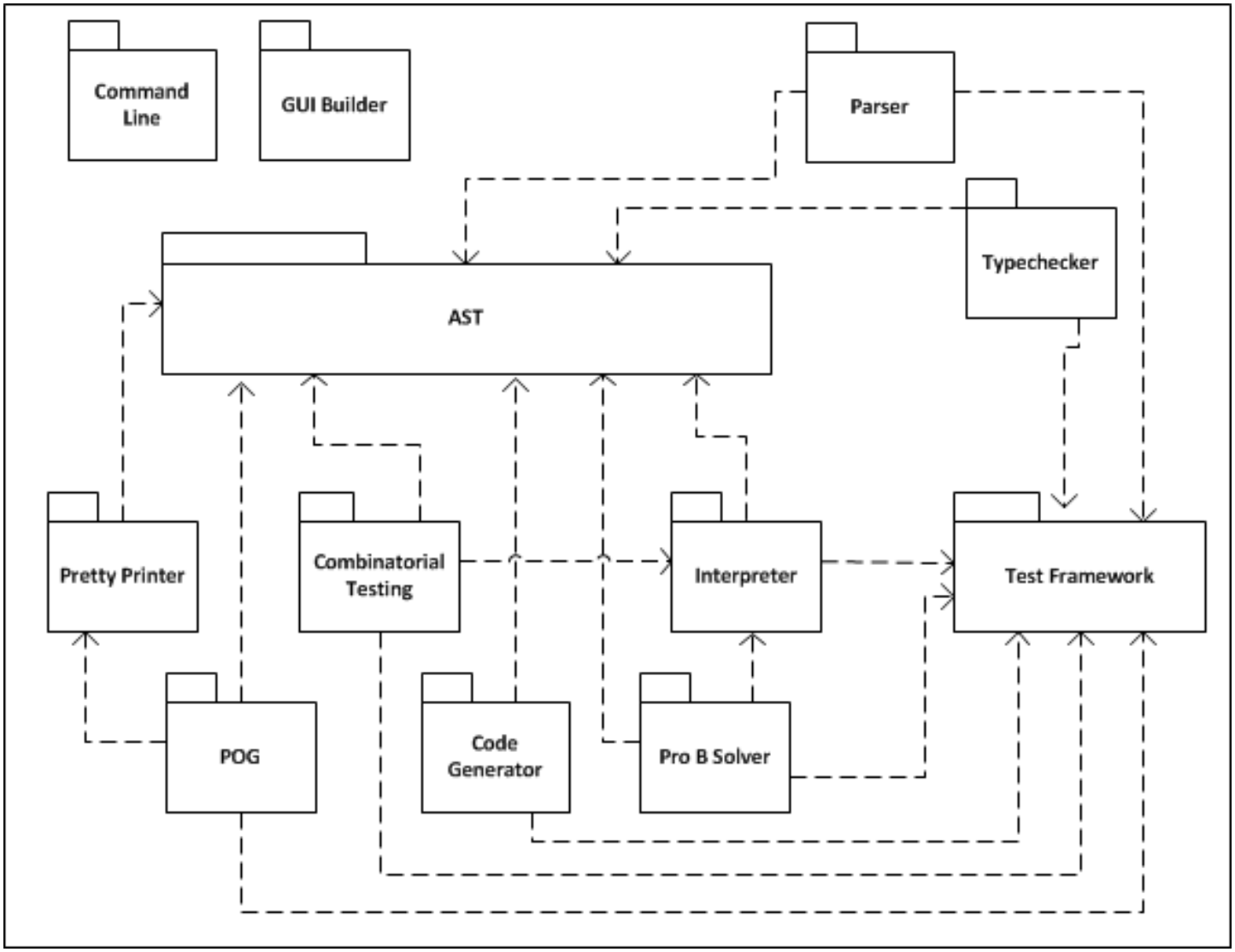
- Conceptual perspective; suited for non-developers
- Functionalities are divided into core and plug-ins
- Core functionality:
  - parser, type checker and Abstract Syntax Tree (AST)
  - construct VDM model from source
  - relied upon by others
- Plug-ins:
  - discrete functionalities
  - extensions to the core
  - examples: interpreter, code generator
  - easy to add new plug-ins

Core modules

Note: The architecture guide tends to follow the source code perspective.

- For each module, describe functionality and connections
- Connections correspond to dependencies

A somewhat simplified architecture



An example module description
Parser

- Constructs raw ASTs from VDM sources
- Handwritten, efficient but hard to maintain
- Various utility methods for test purposes
- Depends on:
  - AST, Test Framework
- Dependency of:
  - Type Checker, POG, Code Generator

Issues for new developers

- Assistant mechanism is highly extensible but challenging to use
  - Implement alternative mechanism (unlikely)
  - Supplement assistants with documentation
- “core” and “plug-in” are overloaded terms
  - *IDE module exposes the core of the core plug-in*
  - *A core plug-in has a core and a plug-in*
  - Confusing terminology. Can get used to it but could be better

Examples of design issues

- No canonical way to produce type checked AST
  - Many variations of the same functionality
  - Interact with parser directly, interact through type checker...
  - Introduce wrapper module for parser, type checker and AST
- Excessive interdependency between functionality modules
  - Almost all can be justified case-by-case
  - But the big picture is very messy
  - Lack of architecting module; introduce it

Conclusion

- Initial guide covers core modules
- Seems good but we need a “tester”
- Various issues (and solutions) encountered already
- In the future this guide will be an entry point for all other guides and tutorials
  - What should these guides be?
  - Who validates them?

Conclusion

- Initial guide covers core modules
- Seems good but we need a “tester”
- Various issues (and solutions) encountered already
- In the futu r guides and tutoria
  - What
  - Who v