# **Abstract**

Comparison of Why3/KeY/OpenJML to see how they compare in installation/setup/functionality/ease of use/tools/verification/results. Give a recommendation of the best tool going forward and reasons why.

* **Motivation:** Reviewing the popular current verification tools (Why3,KeY, Spec#) to determine their relevance and applicability against the new proposed standard tool of OpenJML (JML).
* **Goal:** Determine the most favourable verification tool for non-experts and provide an analysis of the current tools. (Pro’s/Cons) The companies do not communicate or collaborate with each other so a comparison between them could be of use to future students.
* **Result:** Recommendation of verification tool as well as a guide on how to use/setup all available tools.

# First-Order Logic

# First-Order Language

# SMT Provers

Goals ?

* Z3
* Alt-Ergo
* Coq
* CVC4
* CVC3

# Why3 (Why + WhyML)

**Installation:**

* **OS**: Ubuntu
* **Process:** [)](https://www.lri.fr/~marche/MPRI-2-36-1/install.html)
* **Review:** Installed Why3, Alt-Ergo, Z3 and Coq using above link with relative ease using the online instructions. Ubuntu version for Z3 was for 14.02 and I had to search for the 16.02 releases myself.

**References:**

1. *Deductive Program Verification with Why3 - A Tutorial; Jean-Christophe Filliatre; May 2016*
2. *Why3 – Where Programs meet Provers; Jean-Christophe Filliatre and* Andrei Paskevich,
3. *The Why3 Platform; Francois Bobot, Jean-Christophe Filliatre, Claude Marche, Guillaude Melquiond, Andrei Peskovich; January 2018*
4. *Lets Verify This with Why3; Francois Bobot, Jean-Christophe Filliatre, Claude Marche, Andrei Peskovich*
5. *The Krakatoa verification tool for Java Programs; Claude Marche; January 2018*

**Purpose:**

* A set of tools to perform deductive program verification.
* Deductive verification means that we express the correctness of a program as a mathematical statement and then we prove it. [1]
* Uses Why and Why3 together to form programs that can be verified using the standard weakest precondition calculus to extract conditions. (\*expand)
* VC Generator produces proof obligations that need to be discharged to prove that a program respects it specification [4]
* Automatically verifies that recursive definitions are terminating by using lexicographic order of arguments that guarantees a structural descent (Only supports algebraic types) [4]
* Non algebraic types have to axiomised or defined as programs where termination is proved by variants [4]
* WhyML does not separate interface and implementation
* Verification conditions are generated using a standard weakest-precondition procedure [4]
* All aliases must be known statically at the time of verification condition generation in order to apply the Hoare-style rule for assignment without the need for heap memory
  + Consequence is that recursive data types cannot have mutable components

**Uses**: [1]

1. Use only to verify programs using the theorem provers. (Logic only)
2. Verify algorithms/data structures through WhyML
3. Verify programs from a mainstream language eg Java, C, Ada

**Why:**

* Logic Language (used to specify programs, eg pre-post conditions, invariants)
  + First Order Logic
    - Polymorphism
    - Algebraic Data Types
    - Inductive Predicates

**WhyML:**

* Programming Language
  + First-Order ML-like language
    - Imperative features
    - Pattern matching
    - Exceptions

1. Types

* Categories
* Built-In Types (int, real and tuples)
* Algebraic Data Types (polymorphic lists , binary trees)

**type** map a’ b’

**type** i\_map ‘y = map int

* User-defined type (Can be non-interpreted or be a synonym for a type expression)

**type** list a’ = Nil | Cons a’ (list a’)

**type** tree a’ = Empty | Node (tree a’) a’ (tree a’)

* Record types (Special case of algebraic types with a single unnamed constructor and named fields)

**type** queue a’ = { front: list a’; rear: list a’ }

* Extension of pure types of the specification language
  + Mutable state of a computation is exclusively embodied in mutable fields of record data types

**type** ref a’ = { **mutable** contents: a’ }

* + A program type can be provided with an invariant

**type** array a’

**model** { length: int; **mutable** elts: map int a’ }

**invariant** { 0 <= **self**.length }

* ‘**model**’ makes the ‘**type** array a’’ an abstract data type
* ‘elts’ cannot be accessed from a program; accessed only from inside specifications
  + A record field (argument of a constructor in an algebraic type) can be declared ghost.

**type** sparse\_matrix \_ =

{ **ghost** view : map (int,int) \_; ... }

* Ghost data can only be used in specifications and cannot modify non-ghost code

1. Function/Predicate
   * + Every function or predicate symbol has a polymorphic type signature.[4]

**function** get (map a’ b’) a’ : b’

* + - Both can be given definitions, possible mutually recursive[4]

**predicate** increasing (m: map int int) = **forall** i j: int. i < j ! get m i < get m j

**function** height (t: tree \_): int = **match** t **with**

| Node l \_ r ! 1 + max (height l) (height r)

| Leaf ! 0

**end**

* Function Prototypes
  + Useful for providing a usable interface for ‘model’ types like array

**val** ([]) (a: array a’) (i: int) : a’

**requires** { 0 <= i < a.length }

**reads** { a }

**ensures** { **result** = get a.elts i }

**val** ([] <-) (a: array a’) (i: int) (v: a’) : unit

**requires** { 0 <= i < a.length }

**writes** { a }

**ensures** { a.elts = set (**old** a.elts) i v }

* + [] defines a[i]
  + []<- defines a[i] <- v
  + **‘reads’** and ‘**writes’** specify the side effects

1. Terms and Formulas

* First order language is extended (Terms and Formulas)
  + Pattern Matching
  + ‘let’ expressions
  + Conditional expressions (allowed in terms)

**function** abs (x: int) : int =

**if** x >= 0 **then** x **else** -x

1. Theories

* Collections of pure logical definitions
  + 1. Lemmas (Provable statements)
    2. Axioms (Definitions of function)
* ‘use’ keyword imports theory library

**use import** list.List

1. Design by Contract

* Usual DBC verification system is used in WhyML language
  + Ensures
  + Requires
  + Invariant
  + Variant (used to ensure termination in recursive functions and while loops)
  + Asserts (Statically checked)

1. Programs

* ML syntax

**function** (!) (x: ref a’) : a’ = x.contents

**let** (!) (r:ref a’) : a’

**ensures** { **result** = !r }

= r.contents

**let** (:=) (r:ref a’) (v:a’) : unit

**ensures** { !r = v }

= r.contents v

**let** incr (r: ref int) : unit

**ensures** { !r = **old** !r + 1 }

= r := !r + 1

* + **‘function (!)’** = pure access function with name ‘!’
  + **‘let (!)’** = program function with name ‘!’
  + **‘let (:=)’** = program function with name ‘**:=**’
  + **‘let (incr)’** = program function with name ‘incr’
  + **!r** = dereferencing a variable
    - **Note: ‘!r’** in the pre/postconditions can only refer to the pure function, **‘function’**, as program symbols cannot appear in specification
    - In the program code, **‘!r’** will refer to current WhyML function **‘let (…)’**
    - **‘r := !r + 1’** would have thrown an error in the last program function if it had not been redefined. Otherwise it would be trying to access the pure access function from program code, which is illegal.

1. Modules

* WhyML declarations and definitions are grouped into modules, like pure logical theories
* May import logical theories or contain pure declarations

1. Verification

**Command Line Operations:**

* why3 prove $User/test.mlw
  + Check the file for syntax and termination point.
  + Prints the contents to the terminal.
  + No proof attempt is made
  + Returns line and character numbers if an error is found, eg unable to prove a termination point
* why3 prove –P alt-ergo $User/test.mlw
  + Proof check the file
  + Each goal in the file is verified
  + Returns ‘Valid’ if verified
  + Returns ‘Unknown’ if a goal could not be proven
    - Induction will cause an Unknown error to be returned and requires the Coq proof assistant to do the proof, as SMT Solvers cannot verify it.
* why3 ide $User/test.mlw
  + - Launches IDE
    - Displays code for the file
    - Displays each goal to be verified
    - Allows selection of SMT Prover
    - Can perform translations of goals to assist with proofs, eg while using Coq
    - Code Extraction: a verified WhyML program can be translated to a compilable correct-by-construction OCaml program [4]

**Online Operations**

# Design by Contract

**References:**

1. *Applying “Design by Contract”, Bertrand Meyer, 1992*

**Brief:**

* Reduce the need for defensive programming, therefore reducing code size, complexity and the capability for introducing bugs
* If the contract is precise and explicit, there is no need for redundant checks [6]
* Assertions: Specify the relationship between the client (caller) and the supplier [6] (routine/method)
  + Pre-conditions
  + Post-conditions
  + Variants
  + Invariants (per routine/ per loop)
  + Class Invariants (cover all class routines)
  + Asserts
* Pre-condition (Requires)
  + Expresses requirements that any call must satisfy if it is to be correct [6]
  + Method/Routine cannot execute if precondition is not satisfied
  + The absence of a precondition is the same as the clause ‘Requires True’ which is the weakest possible precondition [6]
  + A pre-condition violation indicates a bug on the client’s (caller) side [6]
  + The stronger the pre-condition, the heavier the burden on the client [6]
  + “*As long as the conditions on the use of a routine make sense, and the routines documentation states these conditions explicitly, the programmers will be able to use the routine properly by observing their part of the deal*”
* Post-condition (Ensures)
  + Expresses properties that are ensured in return by the execution of the call [6]
  + The absence of a precondition is the same as the clause ‘Ensures True’ which is the weakest possible postcondition [6]
  + A post-condition violation indicates a bug on the supplier’s (method/routine) side
  + The stronger the post-condition, the heavier the burden on the supplier
* Class Invariants
  + A condition that must hold true before, during and after the execution of a routine from said class
  + “*The invariant must be preserved by every exported routine of the class. Any such routine must guarantee that the invariant is satisfied on exit if it was satisfied on entry*”
  + The invariant is added to the pre/post-conditions of every routine in the class
* Inheritance (Behavioural Subtyping)
  + All assertions from the base class are passed onto all derived classes
  + Derived classes can add in further assertions if required as long as they follow certain rules
    - Precondition cannot be strengthened
    - Postcondition cannot be weakened
    - Invariant cannot be weakened
* Exception Handling (abnormal cases)
  + Failure
    - Execution of a routine cannot fulfil its contract (specification) [6]
  + Exception
    - Occurs when a certain strategy for fulfilling a routine’s contract has not succeeded

**How to Write Specifications [10]**

* Start with foundation and library routines
* For each field:
  + Is there an invariant for this field?
* For each reference field:
  + Should it be non-null?
  + Should an owner field be set for it?
* For each method:
  + Should it be pure?
  + Should the arguments or the result be non-null?
* For each class:
  + What invariant expresses the self-consistency of the internal data
* Add pre and post conditions to limit the inputs and outputs of each method
* Add possible unchecked exceptions to throws clauses
* Separate conjunctions to get information about which conjunct is violated
* Use assert statements to find out what is going wrong
* Use assume statements that you know are correct to help the prover along

# JML

* Runtime Assertion Checking vs Static Checking

# Runtime Assertion Checking

**References**

1. Introduction to JML, David Cok, Joe Kiniry and Erok Poll, Tutorial Slides
2. Runtime assertion checking with JML, Erik Poll, Tutorial Slides

**Purpose**

* The main application of runtime assertion monitoring is debugging [6]
* Tests against what a developer thinks their software does versus what is actually does
  + Compile java file using jmlc to do JML runtime assertion checking (jmlc - a preproscessor for javac) [8]
  + Execute java file using jmlrac to perform runtime assertion checks (jmlrac - a wrapper for java) [8]
* jmlrac – test for violations of assertions during execution [7]
* jmlrac compiler created by Gary Leavens, Yoonsik Cheon, et al. translates JML assertions into runtime checks [7]
* Produces an error if any violation of an assertion occurs (**Note:** Can be an error in code or an error in specification)
  + Provides information about the cause of the problem, rather than the consequence [8]
* Cheap and easy to execute however the program runs slower and uses more memory
* Good addition to testing, however not a replacement for testing
* jmlunit tool combines jmlrac and unit testing

# ESC/Java 2

**References**

1. ESC/Java2 Use and Features, David Cok, Joe Kiniry and Erok Poll, Tutorial Slides
2. Specification Tips and Pitfalls, David Cok, Joe Kiniry and Erok Poll, Tutorial Slides

**Purpose**

* ESC = Extended Static Checking
* ESC/Java 2, an extension of ESC/Java, supports more JML functionality
* Fully automated verification
* Tries to prove correctness of specifications at compile time
* Not sound: May miss an error that is present [9]
* Not complete: May warn of errors that are impossible [9]
* However, it finds a lot of potential bugs quickly
* Good at proving absence of runtime exceptions and verifying relatively simple properties
* Verification conditions are generated using a standard weakest-precondition procedure [4]
* All aliases must be known statically at the time of verification condition generation in order to apply the Hoare-style rule for assignment without the need for heap memory
  + Consequence is that recursive data types cannot have mutable components
* WhyML does not separate interface and implementation
* Checks specs at compile time as opposed to jmlrac which is done at runtime
* Proves correctness of specifications, jmlrac only tests correctness of specs [9]
* Provides a higher degree of confidence by forcing the user to specify a contract
* Structure [10]

1. Parsing phase (syntax checking)
2. Typechecking phase (type and usage checking)
3. Static checking phase (reasoning to find potential bugs) by running a behind-the-scenes prover called Simplify

* Phases a and b produce cautions an errors
* Phase c produces warnings
* Design by Contract rules applied to classes/methods
* Enforces behavioural subtyping

**Design/Workings [11]**

* The procedure specifications are translated into assumptions and assertions interleaved with the Java code, based on the semantics of the specification language
* The code, assumptions and assertions are translated into a basic block form thag uses single-assignment labelling of variables
* The basic blocks are translated into compact verification conditions (VC’s)
* The VC’s are expressed in SMTLIBv2 format
* An SMT solver of choice is applied to VC
* If the VC is invalid, a counterexample is obtained from the SMT tool
* The logical variables of the counterexample are translated back to source code varoiables and text locations; logical variables values are expressed in programming language terms
* The counterexample values and the static execution path are displayed in the source code editor hover information and highlighting

# OpenJML

**References**

1. OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse by David R. Cok circa 2014
2. Does your software do what is should? Tutorial and user guide to specification and verification with the Java Modelling Language and OpenJML by David R. Cok circa 2018
3. Static Verification of PtolemyRely Programs Using OpenJML by Jose Sanchez and Gary T. Leavens circa 2014
4. Official Website http://www.openjml.org/

**Purpose**

* OpenJML is a program verification tool for Java programs that allows you to check the specifications of programs annotated in the Java Modelling Language [14]
* Replace ESC/Java2 with a universal JML implementation
* Provide a freely available verification tool that can be used in industry as well as academia
* Structure
  + Parsing phase (syntax checking)
  + Typechecking phase (type and usage checking)
  + Static checking phase (OpenJML translates Java code + JML specs into verification conditions that are checked by the SMT slovers) [11]
  + Runtime assertion checking (Compiles specs as assertions into the usual Java .class files)
    - OpenJML uses the OpenJDK compiler, enhancing the processing of source files to add appropriate checks that assertions and other specifications hold during execution of the program
* Integration with both Eclipse and command-line tools
* Programmatic access through an API allows access to:
  + Internal Abstract Syntax Trees (AST’s),
  + Type information
  + Compilation
  + Checking commands
  + Results of verification attempts, including counterexamples
* Design is adapted from ESC/Java2 [11]
  + Extends ESC/Java2 model to also check the generated VC for vacuity or for multiple falsified assertions
  + Constructs a single VC for a method
    - Default behaviour is to check vacuity of entire VC
    - Can also check each distinct path/set of paths to each distinct assertion or other sub-expressions of the full VC
  + Checks that there are feasible paths to the procedure exit and to each assertion
  + Parses JML specifications as well as java code (different to ESC/Java2 or just to OpenJDK?)