Verification-centric Software Development in Java with BON, JML, and ESC/Java2

Joe Kiniry
University College Dublin

Topics Covered

- example projects
- effectively using formal methods
- analysis and design with BON
- assertions and specifications
- contracts and system specifications in BON
- applying BON to Java and JML
- code standards and metrics
- static analysis for software construction
- verification with model checking and LFs

Instructors

- Dr. Joseph Kiniry (module coordinator)
- Mr. Dermot Cochran (demonstrator)
- Mr. Fintan Fairmichael (demonstrator)

Schedule

	Mon	Tues	Wed	Thurs	Fri
9:30-10:15	course intro	assertions	static analysis 1	testing	verification with LFs
10:15-10:30	break				
10:30-11:45	analysis	specifications	static analysis 2	testing	model checking
11:45-12:00	hand out afternoon assignment and discuss				
12:00-12:45	lunch				
13:00-16:00	project work in pairs				

Assessment

- short exercises in the morning (1/8)
- interaction with instructors (1/8)
- project work in afternoons in pairs (3/4)
- no exam

- all work is graded on A-F scale

Project

- electronic cash system
- deployed in the Netherlands, France, etc.
- traditionally a smartcard-based system
- architecture has three core pieces
  - Wallet, Terminal, Station
- each team will perform analysis, design, implementation, and validation (and, optionally, verification) independently

Monday: Background, BON, and Analysis

Monday 1:
Example Projects

First Year Course: One Dimensional Cellular Automaton Simulator

Cellular Automata

- a fundamental model for computation
- very simple conceptual model
- small set of concepts
- multiple complexity refinements
    - dimensionality
    - cell type

Project Dimensions

- classified as a small-sized project
  - our estimate is <<1,000 LOC
  - ~50 LOC/week/person
- our complete design has 9 classes
- some classes are optional and are only implemented by advanced students

Co-Analysis and Co-Design

- system analysis and design was conducted live, in-class, with first year students
- analysis was captured with BON
  - informal charts only, no tool support
- design was captured with JML
- *students were not told that they were doing formal analysis or design*

Implementation Process

- students implemented the resulting JML-annotated Java using design by contract
- students used Emacs & vi, not Eclipse
- a Makefile was provided that triggered javac, jml, jmlc, escjava2, javadoc, and jmldoc
- no unit testing was performed whatsoever
  - for other, larger projects tests are frequently generated with jml-junit

Results

- ~80% of the teams' programs worked correctly the first time they executed
- one team had an NPE, fixed in an hour after they ran ESC/Java2 for the first time
- another had a mysterious crash, traced and fixed using a debugger in one afternoon
- this process results in a very high-quality Java system that is very nearly "correct-by-construction", accomplished by 1st years

Second Year Course:
The C=64 Game
“Thrust”

“Thrust”

“Thrust”

The Project: The C=64 Game "Thrust"

- connection to core computing concepts via discrete event simulation
- a few major components
  - file I/O, GUI and rendering, simulation
- several key algorithms
- looks cool and is fun to play

Project Dimensions

- classified as a medium-sized project
  - our estimate is <<5,000 LOC
  - ~100-125 LOC/week/person
- our (very) complete design has 75 classes
- recall that original game written by one person in a few months in 650X assembler

Project Decomposition

- I/O: keyboard input to start and play game
- GUI: bitmaps (terrain), fonts (scores, fuel), and shapes (spaceship, bullets, stars)
- sound: music and effects
- core data structures: entities (spaceship, factory, bullet, etc.), score and high score
- discrete event simulation: main event loop, animations (barriers, explosions, factory smoke, stars, etc.), physics, collisions

What Ones Mind Wants To Do Now

- How do I open a window?
- How do I make a sound?
- How do I draw a line?
- Will I use arrays?
- Floating point numbers or integers?
- etc.

The Proper Course

- Ignore the problems of programming.
- Forget about Java.
- Step back and take a deep breath.
- Relax.

- Brainstorm about the **idea** of Thrust.

Commercial Software Development:
The KOA Tally System

Case Study: KOA Tally System

- Dutch government decided to make remote voting available in 2004 to expatriates
  - remote voting is voting by telephone or via the Internet
- a consulting firm LogicaCMG designed, developed, tested, and deployed system
- RUN participated in review of system

KOA Tally System: Background

- a primary recommendation of review was that a 3rd party should re-implement a critical part of the system from scratch
- government opened up bid on independent implementation of counting/tally component
- RUN group bid on contract and won
  - key factor in bid was proposed use of formal methods (JML) in application development

KOA Architecture

- three main components, each the responsibility of one developer
  - file and data I/O (E. Hubbers)
  - GUI (M. Oostdijk)
  - core data structures and counting algorithm (J. Kiniry)
- most of specification and verification effort was focused in the core subsystem

Code Standards

- lightweight code standards for this effort
  - basic rules about identifier naming, documentation, annotation, and spacing
  - each developer had his own idiom
  - avoid enforcement or tool use that causes merge conflicts
- coding standard checked with CheckStyle
  - http://checkstyle.sourceforge.net/

Version and Config Management

- version management via CVS
  - policies on commits and merges
    - code must build and specs must be right
  - rules are developer-enforced (not triggers)
- configuration management via Make, a single class of constants, and runtime switches
  - with more time, Java properties and bundles would be used as well

Automated Build System

- GNU make based build system
  - works on all operating systems
- single developer responsible for build architecture and major upkeep
- major targets include:
  - normal build, jmlc build, unit test generation and execution, verification, documentation generation, style checking

Unit Testing

- one developer responsible for unit test architecture and major upkeep
- each developer responsible for identifying key values of their data types
- unit test only core classes, not GUI or I/O
- automatically generate ~8,000 tests
- ensure nearly 100% coverage for core
- complements verification effort

Verification

- attempt to verify only core classes
  - focus effort on opportunities for greatest impact and lowest risk
- results of verification with ESC/Java2.0a7
  - 47% of core methods check with ESC/Java2
  - 10% fail due to Simplify issues
  - 31% of postconditions do not verify due to completeness problems
  - 12% fail due to invariant issues

Application Summary

	File I/O	GUI	Core
classes	8	13	6
methods	154	200	83
NCSS	837	1,599	395
specs	446	172	529
specs:NCSS	1:2	1:10	5:4

Monday 2: Effectively Using Formal Methods

Software Engineering Processes
incorporating Formal Specification

The Range of Software Engineering Processes

- old-school processes
  - CRC and state-chart based
- heavyweight processes
  - all up-front design, use UML or similar
- lightweight processes
  - unit test-centric (XP), design on-the-fly
- custom processes
  - use a process that works for you

Certification

- <u>Common Criteria EAL evaluation</u>
  - EAL1: Functionally Tested
  - EAL2: Structurally Tested
  - EAL3: Methodically Tested and Checked
  - EAL4: Methodically Designed, Tested, and Reviewed
  - EAL5: Semiformally Designed and Tested
  - *EAL6: Semiformally Verified Design and Tested*
  - EAL7: Formally Verified Design and Tested

Facets of Critical Software Engineering

- requires a rich environment that synthesizes all primary facets
  - code standards
  - version and configuration management
  - automated build system
  - unit tests
- requires developer investment in learning, applying, and understanding the method

Non-technical Facets

- requires social adoption
  - internal tensions caused by mandated changes in process can cause a development team to self-destruct
- requires institutional support
  - an understanding of the time, resources, and potential results of development with formal methods

Effective JML

- effectively using JML means effectively using JML tools
- development process of project (macro-scale) is realized by daily development process (micro-scale)
- rich tool support must be supported by rich process support
  - code standards and organization support

Specification in Process

- “Contract the Design”
  - one is given an architecture with no specification, little documentation and one must somehow check the system is correct
- “Design by Contract”
  - one designs and builds a system relying upon existing components and frameworks

Contract the Design

- a body of code exists and must be annotated
  - the architecture is typically ill-specified
  - the code is typically poorly documented
  - the number and quality of unit tests is typically very poor
  - the goal of annotation is typically unclear

Goals of Contract the Design

- improve understanding of architecture with high-level specifications
- improve quality of subsystems with medium-level specifications
- realize and test against critical design constraints using specification-driven code and architecture evaluation
- evaluate system quality through rigorous testing or verification of key subsystems

A Process Outline for Contract the Design

- directly translate high-level architectural constraints into invariants
  - key constraints on data models, custom data structures, and legal requirements
- express medium-level design decisions with invariants and pre-conditions
- use JML models only where appropriate
- generate unit tests for all key data values

Design by Contract

- writing specifications first is difficult but very rewarding in the long-run
  - one designs the system by *thinking* and writing contracts
- a refinement-centric process akin to early instruction in Dijkstra/Hoare approach
- ESC/Java2 works well for checking the consistency of formal designs
- resisting the urge to write code is hard

Goals of Design by Contract

- work out application design by writing contracts rather than code
- express design at multiple levels
  - BON/UML ⇨ JML ⇨ JML w/ privacy
- refine design by refining contracts
- write code once when architecture is stable

A Process Outline for Design by Contract

- outline architecture by realizing classifiers with classes
- capture system constraints with invariants
- use JML models only where appropriate
- focus on preconditions over postconditions
- develop test suite for design by writing a data generator for all interesting types

Monday 3:
BON: The Business Object Notation

BON: Meaningful UML

- BON: The Business Object Notation
- invented by Walden & Nerson around 1994
- used in Eiffel community
- describes the static and dynamic aspects of a object-oriented (software) system
- *seamless*, *reversible*, *contract-based*

BON Tools

- EiffelStudio
- the BON Visio templates
- BON-CASE
- the BON Tool Suite
- Class Skeletons, Javadoc, and JML
- the BONc Tool (new!)

Graphical and Textual

- all BON elements have a graphical *and* a textual representations
- textual representations use a concrete syntax *and* a chart-based notation

Example Graphics

PERSON

$name, address: VALUE$

$children, parents: LIST\ [PERSON]$

Invariant

$\forall\ c \in children \bullet (\exists\ p \in c.parents \bullet p = @)$

Example Chart

CLASS	*CITIZEN*	**Part:** 1/1
TYPE OF OBJECT Person born or living in a country	**INDEXING** **cluster:** *CIVIL_STATUS* **created:** 1993-03-15 jmn **revised:** 1993-05-12 kw	
Queries	Name, Sex, Age, Single, Spouse, Children, Parents, Impediment to marriage	
Commands	Marry. Divorce.	
Constraints	Each citizen has two parents. At most one spouse allowed. May not marry children or parents or person of same sex. Spouse's spouse must be this person. All children, if any, must have this person among their parents.	

Example Text

```
deferred class CITIZEN
  feature name,sex,age: VALUE
  spouse: CITIZEN --Husband or wife
  children, parents: SET[CITIZEN] --Close relatives, if any
  single: BOOLEAN --Is this citizen single?
    ensure Result <-> spouse=Void
  end
  deferred marry --Celebrate the wedding.
    ->sweetheart: CITIZEN
    require sweetheart /= Void and can_marry(sweetheart)
    ensure spouse=sweetheart
  end
  ...
  divorce --Admit mistake.
    require not single
    ensure single and (old spouse).single
  end
  invariant
    single or spouse.spouse=Current;
    parents.count=2;
    for_all c member_of children it_holds
      (exists p member_of c.parents it_holds p=Current)
end --class CITIZEN
```

Two Levels of BON Specifications

- informal charts and diagrams
  - specified primary concepts of system, scenarios of use, primary events
- formal diagrams
  - specifies contracts on type interfaces, method call sequences, architecture structure

Informal BON Charts

- the *static model*
  - system diagrams (informal charts)
  - class dictionary (a dependent chart)
- the *dynamic model*
  - object creation charts
  - scenario charts
  - event charts

Monday 4:
Dependable Systems Analysis

Concept Analysis

- purpose is to identify key concepts in an architecture and their core relationships
- in BON, each concept is represented by a *class*: *class* is short for *classifier*
- each class needs a *name* and a *description*
- negotiated, unique, and clear names provide the core *nomenclature* of a project

Syntax for Concepts

- name and description

```
class_chart ALARM_CLOCK
explanation
  "A clock with an alarm."
end
```

- relationships between concepts
  - *inheritance* (is-a) and *client* (has-a)
  - inheritance captured in class chart
  - both relations captured in static diagrams

Semantics of Inheritance

- children inherit all aspects of their parents

```
class_chart ALARM_CLOCK
inherit CLOCK
explanation
  "A clock with an alarm."
end
```

- inheritance relations form a finite lattice
- top of lattice is class ANY, bottom is NONE
  - all classes inherit from ANY
  - NONE inherits from all classes

A Class's Interface: The Contract

- you must identify
  - all questions you might ask the class
  - demands you might make of the class
  - features that differentiate this particular class from others similar to it

Queries

- questions you might ask
- an English sentence written as a question

```
What is the current time for this clock?
What is your hair color?
```

- *must not* change state of an object that realizes the class

Commands

- demands you might make of the class
- written as a (commanding) English sentence

```
Set the clock's time.
```

- often we use exclamation marks for effect

```
Your hair color is brown!
```

- *changes the state* of the object in question

Constraints

- features that differentiate this particular class from others similar to it
- written as declarative English sentences

```
The time in hours must be non-negative
and less than 24.
```

```
Hair color is either black, brown,
red, or blond.
```

Indexing Clauses

- metadata useful to process
- structured property-value pairs
- core set of properties pre-defined
  - about, author, copyright, date, organization, title, version, etc.
- written in indexing block in class chart

Rules of Thumb

- class names should be short, clear, and either in common use or domain-specific
- all descriptions are *simple* English sentences
- all queries, commands, and constraints are (respectively) *simple* English questions, commands, and declarations
- reuse core indexing clauses whenever possible

Final Class Chart

```
class_chart ALARM_CLOCK
indexing
  author: “Joe Kiniry”
explanation
  "A clock with an alarm."
inherit
  CLOCK
command
  "Set alarm time to a new time."
constraint
  "The alarm time in hours must be \
 \ non-negative and less than 24.",
  "The alarm time in minutes must be \
 \ non-negative and less than 60."
end
```

Core Concepts

- concepts of the natural world and within the foundations of computing are reused
  - COLOR, MASS, DISTANCE, TIME, etc.
  - SET, SEQUENCE, ARRAY, LIST, BAG, etc.
- a set of basic BON classes is provided

System Organization

- classes are related by inheritance and client relations
- classes reside in clusters
- cluster reside in clusters or the top-level system
- there is one unique top-level system
- clusters for a lattice

Example System Organization

```
system_chart CLOCK_SYSTEM
  cluster CLOCK_CLUSTER
    description "The cluster for all our clocks."
end

cluster_chart CLOCK_CLUSTER
  class ALARM description "An alarm."
  class ALARM_CLOCK description "A clock with an alarm."
  class CLOCK description "A settable clock storing \
    \ the time in hours, minutes and seconds."
  class LOGICAL_CLOCK description "A settable clock \
    \ storing the time in hours, minutes and seconds."
end
```

Class Dictionary

- lists all primary concepts (classifiers) in the system
  - each class's cluster(s) and description are provided
  - clusters are dependent upon the system and cluster charts
  - description is dependent upon the corresponding class chart
- the MONITORING_SYSTEM class dictionary

Example Dictionary

Object Creation Charts

- shows what classes create new instances of other classes
- serves as a link between the static and the dynamic models
- only high-level analysis classes are treated
- the MONITORING_SYSTEM creation chart

Creation Chart Example

CREATION	*CONFERENCE_SUPPORT*	**Part:** 1/1
COMMENT List of classes creating objects in the system.	**INDEXING** **created:** 1993-02-18 kw	

Class	**Creates instances of**
CONFERENCE	*PROGRAM_COMMITTEE, TECHNICAL_COMMITTEE, ORGANIZATION_COMMITTEE, TIME_TABLE*
PROGRAM_COMMITTEE	*PROGRAM, PAPER, PAPER_SESSION, PERSON*
TECHNICAL_COMMITTEE	*TUTORIAL, TUTORIAL_SESSION, PERSON*
ORGANIZATION_COMMITTEE	*MAILING, ADDRESS_LABEL, STICKY_FORM, REGISTRATION, PERSON, INVOICE, INVOICE_FORM, ATTENDEE_LIST, LIST_FORM, POSTER_SIGN, POSTER_FORM, EVALUATION_SHEET, EVALUATION_FORM, STATISTICS*
*PRESENTATION**	*STATUS, PERSON*
PAPER	*REVIEW, ACCEPTANCE_LETTER, REJECTION_LETTER, LETTER_FORM, AUTHOR_GUIDELINES*
TUTORIAL	*ACCEPTANCE_LETTER, REJECTION_LETTER, LETTER_FORM*
REGISTRATION	*CONFIRMATION_LETTER, LETTER_FORM, BADGE, BADGE_FORM*

Scenario Charts

- semi-equivalent to UML's use-case diagrams
- a scenario is a type of system usage, user or programmatic
  - focus is on important top-level scenarios that are critical to the system design
  - only natural language is used for the high-level specification

Scenarios

- the description of scenario is used as the documentation for
  - the public interface, and
  - the corresponding unit test suite
- scenarios are refined at the intermediate level of specification into object message passing descriptions

Scenario Chart Example

SCENARIOS	*CONFERENCE_SUPPORT*	**Part:** 1/1
COMMENT Set of representative scenarios to show important types of system behavior.	**INDEXING** **created:** 1993-02-16 kw	

Send out calls and invitations:
Using mailing lists and records of previous conference attendees and speakers, prepare and send out calls for papers and invitations to attend the conference.

Create sessions and chairs:
Partition the conference into sessions of suitable length; allocate session rooms and select a chairperson for each session.

Register paper and start review process:
A paper is registered and three referees are selected; the paper is sent to each referee, and the paper status is recorded.

Accept paper and notify authors:
A submitted paper is selected and an acceptance date is entered; a notification letter is created and sent to the authors.

Assign paper to session:
A session suitable for the paper is selected and the paper is entered in the list of presentations for that session.

Register attendee:
An attendee is registered with his/her address and selected tutorials are recorded.

Print conference attendee list:
All registrations are scanned and a list with attendee names, addresses and affiliations is produced and sent to a printer.

Print badge:
An attendee is selected, and the corresponding badge is printed in appropriate format.

Event Charts

- object interactions are ultimately caused by *external events*
  - external events trigger system execution
- *internal events* are high-level, important triggers within a system
  - typically an external event triggers one or more internal events

Event Identification

- external events connote the external (perhaps public) interface of a system
- internal events connote the private subcomponent interfaces *within* a system
- each event is either ingoing or outgoing
- the MONITORING_SYSTEM <u>external event diagram</u> and <u>internal event diagram</u>

Example External Event Chart

EVENTS	*CONFERENCE_SUPPORT*	**Part:** 1/2
COMMENT Selected external events triggering representative types of behavior.	**INDEXING** **created:** 1993-02-15 kw **revised:** 1993-04-07 kw	

External (incoming)	Involved object types
Request to register a submitted paper	*CONFERENCE, PROGRAM_COMMITTEE, PAPER*
Request to accept a paper	*CONFERENCE, PROGRAM_COMMITTEE, PAPER, STATUS*
Request to assign a paper to a session	*CONFERENCE, PROGRAM_COMMITTEE, PROGRAM, PAPER, PAPER_SESSION*
Selection of a session chairperson	*CONFERENCE, PROGRAM_COMMITTEE, PROGRAM, PAPER_SESSION, PERSON*
Request to register an attendee	*CONFERENCE, ORGANIZING_COMMITTEE, REGISTRATION, PERSON*
Request to print conference attendee list	*CONFERENCE, ORGANIZING_COMMITTEE, REGISTRATION, PERSON, ATTENDEE_LIST*

Example Internal Event Chart

EVENTS	*CONFERENCE_SUPPORT*	**Part:** 2/2
COMMENT Selected internal events triggering system responses leaving the system.	**INDEXING** **created:** 1993-02-15 kw **revised:** 1993-04-03 kw	

Internal (outgoing)	**Involved object types**
Call for papers is sent	*CONFERENCE, ORGANIZING_COMMITTEE, PERSON, MAILING*
Invitations are sent	*CONFERENCE, ORGANIZING_COMMITTEE, PERSON, MAILING*
A paper is sent to referees	*CONFERENCE, ORGANIZING_COMMITTEE, PAPER, STATUS, REVIEW, PERSON*
An invoice is sent	*CONFERENCE, ORGANIZING_COMMITTEE, REGISTRATION, PERSON, INVOICE, INVOICE_FORM*
Warning issued for exceeding tutorial session capacity	*CONFERENCE, REGISTRATION, TUTORIAL*
An author notification is sent	*CONFERENCE, PROGRAM_COMMITTEE, PERSON, PRINT_OUT*, LETTER_FORM*

Tuesday: Specifications

Tuesday 1: Assertions and Specifications

Assertions

- the **assert** statement is the fundamental construct used to specify the correct behavior of software
- the statement

```
assert S;
```

means

“S **must** be true at **this** point

in the program’s execution”

Assertion Syntax in Java

- **all** modern programming languages have an **assert** statement
- beginning in Java 1.4, **assert** is a keyword
- the syntax of a Java assert statement is

```
assert <boolean>[: <String>]
```

- `boolean` is the predicate that **must** be true
- `String` is an optional message that will be printed if/when the assertion fails

Examples of Assertion Use

```
assert z != 0;
x = y/z;

assert (x > MIN_WIDTH);
my_window.setWidth(x);

assert p(x) : “p failed when x=” + x;
a_method_that_depends_upon_p(x);
```

Assertions vs. Logging

- if an assertion fails, the program **halts**
- thus, assertion failures are **critical** failures
- to assert something that is not critical, then a logging message is appropriate

```
if (Debug.DEBUG && !p(x))
  System.err.println("p("+x+") fails");
a_method_that_depends_upon_p(x);
```

Logging Frameworks

- it is **always** wiser to use a logging framework than to use embedded `printlns`
- if a `println` must be used, guard it with a conditional on a constant boolean
  - setting the guard false eliminates all logging code (saves space and time)
- the premier logging frameworks are `java.util.logging`, log4J, and IDebug

Specifications

- specifications of software range in formality
  - informal - English documentation (e.g., "normal" comments)
  - semi-formal - structured English documentation (e.g., **Javadoc**)
  - formal - annotations and assertions (e.g,. **assert** statements and **contracts**)
- **contracts** are a **key concept** in robust software design and construction

Informal Specifications

```
/* Deduct some cash from this account and
   return how much money is left. */

public int debit(int amount)
```

Informal Specifications

```
/* Deduct some cash from this account and
   return how much money is left. */

public int debit(int amount)
```

- what happens when:

Informal Specifications

```
/* Deduct some cash from this account and
   return how much money is left. */

public int debit(int amount)
```

- what happens when:
  - amount is negative?

Informal Specifications

```
/* Deduct some cash from this account and
   return how much money is left. */

public int debit(int amount)
```

- what happens when:
  - amount is negative?
  - amount is bigger than the balance?

Informal Specifications

```
/* Deduct some cash from this account and
   return how much money is left. */

public int debit(int amount)
```

- what happens when:
  - amount is negative?
  - amount is bigger than the balance?
  - is the balanced changed when failure?

Semi-Formal Specifications

```
/** Debit this account.
  * @param amount the amount to debit.
  *        <code>amount</code> must be
  *        non-negative.
  * @result the balance of this account
  * after the debit successfully occurs.
  */
  public int debit(int amount)
```

Semi-Formal Specifications

```
/** Debit this account.
  * @param amount the amount to debit.
  *        <code>amount</code> must be
  *        non-negative.
  * @result the balance of this account
  * after the debit successfully occurs.
  */
  public int debit(int amount)
```

- many of the same questions arise even though the documentation is much clearer

Formal Specifications

```
/** Debit this account.
  * @param amount the amount to debit.
  * @result the resulting balance.
  */
/*@ requires amount >= 0;
  @ ensures balance == \old(balance-amount) &&
  @                    \result == balance;
  @*/
  public int debit(int amount)
```

Writing and Calling Methods Incorrectly

Writing and Calling Methods Incorrectly

```
/* Deduct some cash from this account and
   return how much money is left. */
  public int debit(int amount) {
    if (amount < 0) throw NDE(amount);
    if (balance < amount)
      throw NBE(balance);
    ...
  }
```

Writing and Calling Methods Incorrectly

```
/* Deduct some cash from this account and
   return how much money is left. */
  public int debit(int amount) {
    if (amount < 0) throw NDE(amount);
    if (balance < amount)
      throw NBE(balance);
    ...
  }
```

```
try {
  b = debit(a);
  if (b < 0) throw NBE();
} catch (Exception e) {
  System.exit(-1);
}
```

Writing and Calling Methods Incorrectly

```
/* Deduct some cash from this account and
   return how much money is left */
  public int debit(int amount) {
    if (amount < 0) throw NDE(amount);
    if (balance < amount)
      throw NBE(balance);
    ...
  }
```

HORRIBLE!

```
try {
  b = debit(a);
  if (b < 0) throw NBE();
} catch (Exception e) {
  System.exit(-1);
}
```

Calling Methods Correctly

```
/*@ requires amount >= 0;
  @ ensures balance == \old(balance-amount) &&
  @                    \result == balance;
  @*/
  public int debit(int amount) {
    ...all conditionals are gone!
    ...
  }

if (debit_amount < 0)
  handle_bad_debit(debit_amount);
else
  resulting_balance = debit(debit_amount);
```

Design by Contract

- capture architectural, class-level decisions early as **constraints**
  - e.g., all `Citizens` have two parents
- realize constraints in software as **invariants**
  - an **invariant** is an assertion that must **always** be true whenever a method is called or exits
- capture contracts at method-level in medium-level design using English
  - realize contracts in code using **requires** and **ensures** statements

An Example Use of Design by Contract

CLASS	*CITIZEN*	**Part:** 1/1
TYPE OF OBJECT Person born or living in a country	**INDEXING** **cluster:** *CIVIL_STATUS* **created:** 1993-03-15 jmn **revised:** 1993-05-12 kw	
Queries	Name, Sex, Age, Single, Spouse, Children, Parents, Impediment to marriage	
Commands	Marry. Divorce.	
Constraints	Each citizen has two parents. At most one spouse allowed. May not marry children or parents or person of same sex. Spouse s spouse must be this person. All children, if any, must have this person among their parents.	

Related Class Features

- queries
  - spouse? single?
- command
  - marry! divorce!
- constraints
  - at most one spouse is allowed
  - spouse's spouse must be this person

Class Sketch

```
Citizen my_spouse;
//@ invariant (my_spouse != null) ==>
//@             my_spouse.my_spouse == this;

Citizen spouse() { returns spouse; }
boolean single() { returns spouse == null; }
//@ requires single();
//@ ensures !single() && spouse() == new_spouse;
void marry(Citizen new_spouse)
  { my_spouse = new_spouse; }
//@ requires !single();
//@ ensures single();
void divorce() { my_spouse = null; }
```

Testing with Specifications

- specifications mean that no valid parameter testing is necessary in implementations
  - the precondition is **requiring** the client to fulfill their side of the contract for supplier
- when calling a method that has a specification, checking for errors, return values, etc. is no longer necessary
  - the supplier is **ensuring** (guaranteeing) their side of the contract to client

Unit Testing and Programming with Specs

- ~90% of your method-level unit tests are automatically generated
- ~25% **less code** is written because there is no need to test parameters values nor results of method calls for correctness
- code is not littered with `try/catch` blocks to catch exceptions

Tuesday 2: Contracts and Specifications in BON and JML

BON Assertion Elements

ASSERTION ELEMENTS		
Graphical BON	**Textual BON**	**Explanation**
Δ *name* **old** *expr*	**delta** *name* **old** *expr*	Attribute changed Old return value
Result **@** $\varnothing$	*Result* *Current* *Void*	Current query result Current object Void reference
+ – * / **^** **//** ****	**+ – * /** **^** **//** ****	Basic numeric operators Power operator Integer division Modulo
= ≠ **<** ≤ **>** ≥	**=** **/=** **<** **<=** **>** **>=**	Equal Not equal Less than Less than or equal Greater than Greater than or equal

BON Assertion Elements

$\rightarrow$	**->**	Implies (semi-strict)
$\leftrightarrow$	**<->**	Equivalent to
¬	**not**	Not
and	**and**	And (semi-strict)
or	**or**	Or (semi-strict)
xor	**xor**	Exclusive or
$\exists$	**exists**	There exists
$\forall$	**for_all**	For all
|	**such_that**	Such that
•	**it_holds**	It holds
$\in$	**member_of**	Is in set
$\notin$	**not member_of**	Is not in set
: *type*	**:** *type*	Is of type
{ }	{ }	Enumerated set
..	..	Closed range

The Person Class

PERSON

$name, address: VALUE$

$children, parents: LIST\,[PERSON]$

Invariant

$\forall\, c \in children \bullet (\exists\, p \in c.parents \bullet p = @)$

Textual Specification

```
deferred class CITIZEN
  feature name,sex,age: VALUE
  spouse: CITIZEN --Husband or wife
  children, parents: SET[CITIZEN] --Close relatives, if any
  single: BOOLEAN --Is this citizen single?
    ensure Result <-> spouse=Void
  end
  deferred marry --Celebrate the wedding.
    ->sweetheart: CITIZEN
    require sweetheart /= Void and can_marry(sweetheart)
    ensure spouse=sweetheart
  end
  ...
  divorce --Admit mistake.
    require not single
    ensure single and (old spouse).single
  end
  invariant
    single or spouse.spouse=Current;
    parents.count=2;
    for_all c member_of children it_holds
      (exists p member_of c.parents it_holds p=Current)
end --class CITIZEN
```

JML: Contracts for Java

- assertions go in special Java comments

```
/*@ assertion */
//@ assertion
/** <JML> assertion </JML> */
```

- properties are Java boolean expressions with several extra keywords

```
\result  \old  \forall  \exists
```

- JML's keywords for specifying contracts

```
requires  ensures  signals
assignable  pure  invariant
non_null  nullable
```

JML Assertion Elements

- logical operators

conjuction (AND)	disjunction (OR)	negation (NOT)	implication	equivalence
`&`	`|`	`!`	`<==` `==>`	`<==>` `<=!=>`

- logical quantifiers

`\forall` `\exists`

- generalized quantifiers

`\max` `\min` `\sum` `\product` `\num_of`

Tuesday 3:
Introduction to JML

Tuesday 4: Applying BON to Java and JML

Using Code Skeletons for BON and DBC

- rather than using a specification language, one can use a programming language for analysis and design
- code skeletons are used to sketch out concepts and define class interfaces
- language-specific tools are used to annotate higher-level ideas and lower-level contracts

Java Tools

- structured Javadoc comments are used to annotate classes and features
- the Java Modeling Language (JML) is used to annotate the Java with formal models and contracts
- the JML tool suite and ESC/Java2 are used to runtime check contracts, unit test, and statically check code against specifications

Our Running Example

- we will use the CITIZEN/NOBLEPERSON examples from the BON book
- each chart is written as a Javadoc-annotated class skeleton
- each interface specification is written as a JML-annotated class skeleton
- the implementation is written in Java

Informal Charts: CITIZEN

CLASS	*CITIZEN*	**Part:** 1/1
TYPE OF OBJECT Person born or living in a country	**INDEXING** **cluster:** *CIVIL_STATUS* **created:** 1993-03-15 jmn **revised:** 1993-05-12 kw	
Queries	Name, Sex, Age, Single, Spouse, Children, Parents, Impediment to marriage	
Commands	Marry. Divorce.	
Constraints	Each citizen has two parents. At most one spouse allowed. May not marry children or parents or person of same sex. Spouse's spouse must be this person. All children, if any, must have this person among their parents.	

Informal Charts in Java: Citizen

```
/**
 * Person born or living in a country.
 *
 * @created 1993-03-15 jmn
 * @revised 1993-05-12 kw
 *
 */
package civil_status;

class Citizen {
  /** @bon Name? */
  ...
  /** @bon Marry. */
  ...
  /** @bon Each citizen has two parents. */
}
```

Informal Charts: NOBLEPERSON

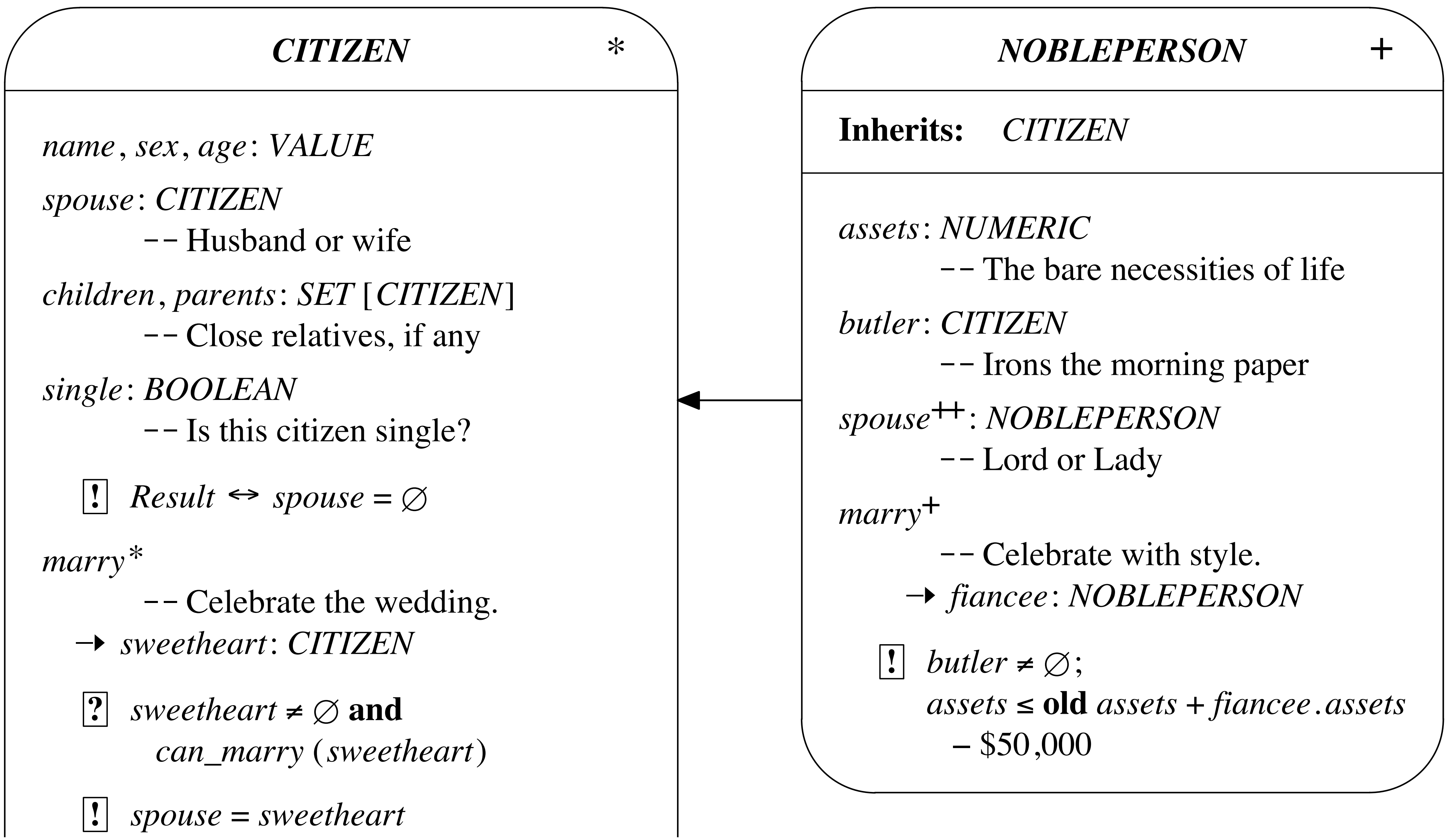
CLASS	*NOBLEPERSON*	**Part:** 1/1
TYPE OF OBJECT Person of noble rank	**INDEXING** **cluster:** *CIVIL_STATUS* **created:** 1993-03-15 jmn **revised:** 1993-05-12 kw, 1993-12-10 kw	
Inherits from	*CITIZEN*	
Queries	Assets, Butler	
Constraints	Enough property for independence. Can only marry other noble person. Wedding celebrated with style. Married nobility share their assets and must have a butler.	

Informal Charts in Java: Nobleperson

```
/**
 * Person of noble rank.
 *
 * @created 1993-03-15 jmn
 * @revised 1993-05-12 kw, 1993-12-10 kw
 */
package civil_status;

class Nobleperson extends Citizen {
  /** @bon Assets? */
  ...
  /** @bon Enough property for independence. */
}
```

Formal Specification: Graphical BON



Formal Specification: Graphical BON

can_marry: *BOOLEAN*

-- No legal hindrance?

$\rightarrow$ *other*: *CITIZEN*

$\boxed{?}$ $other \neq \emptyset$

$\boxed{!}$ $Result \rightarrow (single \textbf{ and } other.single$
$\textbf{and } other \notin children$
$\textbf{and } other \notin parents$
$\textbf{and } sex \neq other.sex)$

divorce

-- Admit mistake.

$\boxed{?}$ $\neg\, single$

$\boxed{!}$ $single \textbf{ and } (\textbf{old } spouse).single$

Invariant

$single \textbf{ or } spouse.spouse = \textbf{@};$
$parents.count = 2;$
$\forall\, c \in children \bullet (\exists\, p \in c.parents \bullet p = \textbf{@})$

Formal Specification in BON: CITIZEN

```
deferred class CITIZEN
feature
    name, sex, age: VALUE
    spouse: CITIZEN                              -- Husband or wife
    children, parents: SET [CITIZEN]             -- Close relatives, if any
    single: BOOLEAN                              -- Is this citizen single?
        ensure
            Result <-> spouse = Void
        end
    deferred marry                               -- Celebrate the wedding.
        -> sweetheart: CITIZEN
        require
            sweetheart /= Void and can_marry (sweetheart)
        ensure
            spouse = sweetheart
        end
```

Formal Specification in JML: Citizen

```
abstract class Citizen {
  private Value name,sex,age;
  /** Husband or wife */
  private Citizen spouse;
  /** Close relatives, if any */
  private Set[Citizen] children, parents;
  /** Is this citizen single? */
  //@ invariant single <==> spouse == null;
  private boolean single;

  /** Celebrate the wedding. */
  //@ requires sweetheart != null;
  //@ requires can_marry(sweetheart);
  //@ ensures spouse == sweetheart;
  abstract void marry(Citizen sweetheart);
  ...
```

Formal Specification in BON: CITIZEN

```
can_marry: BOOLEAN                              -- No legal hindrance?
    -> other: CITIZEN
    require
        other /= Void
    ensure
        Result -> (single and other.single
            and other not member_of children
            and other not member_of parents
            and sex /= other.sex)
    end
divorce                                         -- Admit mistake.
    require
        not single
    ensure
        single and (old spouse).single
    end
```

Formal Specification in JML: Citizen

```
/** No legal hinderance? */
/*@ requires other != null;
  @ ensures \result <==> (single &
  @                        other.single &
  @                        !children.has(other) &
  @                        !parents.has(other) &
  @                        sex != other.sex);
  @*/
abstract boolean can_marry(Citizen other);

/** Admit mistake. */
/*@ requires !single;
  @ ensures single & \old(spouse.single);
  @*/
abstract void divorce();
```

Formal Invariant in BON and JML

invariant
  single **or** *spouse.spouse* = *Current*;
  parents.count = 2;
  for_all *c* **member_of** *children* **it_holds**
    (**exists** *p* **member_of** *c.parents* **it_holds** *p* = *Current*)
end -- class *CITIZEN*

```
/*@ invariant single | spouse.spouse == this; */
/*@ invariant parents.count == 2; */
/*@ invariant (\forall Citizen c; children.has(c);
  @           (\exists Citizen p; parents.has(p);
  @                               p == this;)); */
```

Formal Spec in BON: NOBLEPERSON

effective class *NOBLEPERSON*
inherit
    CITIZEN
feature
    assets: *NUMERIC* -- The bare necessities of life
    butler: *CITIZEN* -- Irons the morning paper
    redefined *spouse*: *NOBLEPERSON* -- Lord or Lady
    effective *marry* -- Celebrate with style.
        -> *fiancee*: *NOBLEPERSON*
        ensure
            butler /= *Void*;
            assets **<=** **old** *assets* + *fiancee*.*assets* – \$50,000
        end
end -- class *NOBLEPERSON*

Formal Specification in JML: Nobleperson

```
class Nobleperson extends Citizen {
  /** The bare necessities of life. */
  Numeric assets;
  /** Irons the morning paper. */
  Citizen butler;
  /** Lord or Lady */
  //@ invariant \typeof(spouse) == \type(Nobleperson);

  /** Celebrate with style. */
  //@ ensures butler != null;
  //@ ensures assets <= \old(assets + fiancee.assets - 50000);
  void marry(Nobleperson fiancee) {
    //@ assert false;
  }
}
```

Wednesday: Static Analysis

- static checkers we will be using today that you must have installed:
  - CheckStyle v4 or v5 (eclipse-cs)
  - Eclipse Metrics 1.3.6 (from SourceForge)
  - FindBugs 1.3.8
  - PMD 3.2.6
  - ESC/Java2 2.0.8
  - AutoGrader 0.1.0

Wednesday 1: Code Standards and Metrics

Code Standards

- the “look and feel” of development artifacts
- includes program code, docs, scripts, etc.
- primary focus is on improving team *communication* and *comprehension*
- team members focus their attention and spend time on *important* things—*not* code formatting or trivial design decisions
- helps with merging and maintenance
- standard are automatically <u>checked</u>

Structural Standards

- small-scale structure
  - code indentation
  - block placement
  - identifier naming
  - method ordering
- large-scale structure
  - package and module structuring
  - design patterns and anti-patterns

Example Use of Standard

```
class Citizen
{
  /** The spouse of this Citizen; if null, this citizen
      is single. */
  Citizen my_spouse = null;
  //@ invariant (my_spouse != null) ==>
  //@           my_spouse.my_spouse == this;

  /** Constructs a new Citizen object who is single. */
  //@ ensures single();
  Citizen() {
    my_spouse = null;
  }
  ...
```

Some Basic Rules of Good Programming

- simple (even trivial!) constructors
- focus on data abstraction
  - appropriate levels of visibility
  - work from tight (private) to loose (public)
- short method signatures
- no globals and few static or class variables
- avoid concurrency at all costs

The KindSoftware Coding Standard

- the “gold standard” of coding standards
- used in dozens of companies and groups around the world
  - e.g,. influenced coding standard at Sun
- written as generic rules with specific application to Java and Eiffel
- http://kind.ucd.ie/documents/whitepapers/code_standards/

Metrics

- provide *quantitative* (but “fuzzy”) analysis of software artifacts
- generated numbers mean *absolutely nothing* in almost all cases
    - *they are only valuable in a relative context*
- dozens (hundreds?) of metrics have been invented but *very* few are seriously used
- usually *the worst* metrics are the ones heard about most often (e.g., KLOC)

Standard Metrics

- lines of code (LOC, KLOC, MLOC)
  - effectively means "count the semicolons," *not* the curly braces
  - counts *real* statements, declarations, etc.
- lines of comments/docs (LOD, KLOD, etc.)
  - counts lines of *real* comments
  - count clauses or measure *information complexity* of documentation

Standard Non-Trivial Metrics

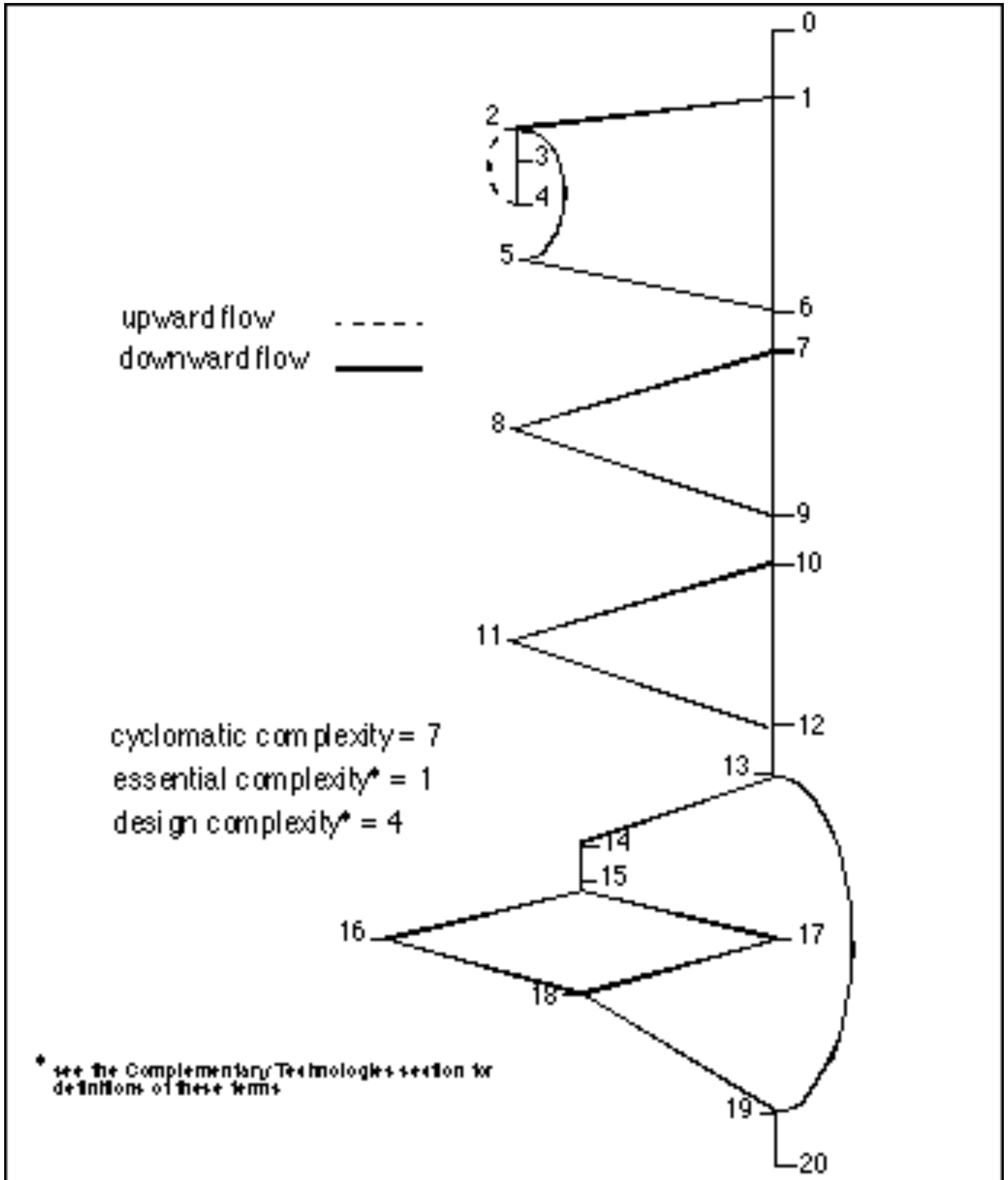
- <u>cyclomatic code complexity</u>
  - roughly counts the number of execution paths through code
  - CC = E - N + 2p, where

    E = the number of edges of the graph

    N = the number of nodes of the graph

    p = the number of connected components

CC Example


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
upward flow
downward flow
cyclomatic complexity = 7
essential complexity* = 1
design complexity* = 4
* see the Complementary Technologies section for definitions of these terms

CC Evaluation

Cyclomatic Complexity	Risk Evaluation for Expert Programmers
1-10	a simple program, low risk
11-20	more complex, moderate risk
21-50	complex, high risk
>50	untestable, very high risk

Other Popular Metrics

Complexity Measure	Primary Measure of
Halstead	Algorithmic complexity, measured by counting operators and operands
Henry and Kafura	Coupling between modules (parameters, global variables, calls)
Bowles	Module and system complexity; coupling via parameters and global variables
Troy and Zweben	Modularity or coupling; complexity of structure (maximum depth of structure chart); calls-to and called-by
Ligier	Modularity of the structure chart

Doc and Spec Coverage

- documentation coverage
  - ensure all modules, methods, and attributes are documented appropriately
    - i.e., *no* Javadoc warnings whatsoever
- specification coverage—at least one...
  - invariant per attribute/field
  - precondition per method parameter
  - postcondition per method
  - assertion per branch in body

Unit Testing
Code Coverage

- desire that tests exercise all execution paths in your code
  - every branch, try/catch, switch case, etc.
- tools exist that measure code coverage while the program runs its unit tests
  - 100% coverage is ideal but rarely met
  - 80-90% coverage is realistic with effort

Popular Java Code Coverage Tools

- Emma - scalable bytecode instrumentation
  - included with Eclipse installed on server
- Quilt - extended classloader; optimized for JUnit, Ant, and Maven
- Hansel - extended classloader
- Gretel - bytecode recompilation
- GroboUtils - extended classloader

Simple Assessment of Software Quality

- ensure assessment in all programming-related assignments is *directly* coupled with these three forms of simple (sometimes static) checking
- system's code, docs, and specs *must* conform to the provided coding standard and metrics and coverage guidelines
- concrete guidelines are built-in to the environment and/or provided

Wednesday 2: Static Analysis for Software Construction

Static Analysis

- static and dynamic are duals
- dynamic analysis means examining an artifact as it changes
  - e.g., watch a program as it executes
- static analysis means examining an artifact when it does not change, *in the context of its meaning and purpose*

Common Kinds of Static Analysis

- *typechecking*
- source code programming standards
- documentation standards
- metrics guidelines
- unit test coverage guidelines
- null pointer analysis
- checking for good programming idioms/ patterns and poor use of anti-patterns

Code Standard Example

Code Standard Example

```
class Citizen {
  /** The spouse of this Citizen; if null, this citizen
      is single. */
  Citizen my_spouse;

  /** Returns a new citizen who is single. */
  Citizen();
  ...
```

Code Standard Example

```
class Citizen
{
  /** The spouse of this Citizen; if null, this citizen
      is single. */
  Citizen my_spouse;

  /** Returns a new citizen who is single. */
  Citizen();
  ...
```

Code Standard Example

Documentation Example

```
/** The spouse of this Citizen; if null, this citizen
    is single. */
Citizen my_spouse;

/** Returns a new citizen who is single. */
Citizen();

/** @return this citizen’s name. */
String name();

/** Sets this citizen’s age.
 *  @param new_age the new age of this citizen.
 */
void age(byte new_age);
...
```

Specification Example

```
class Citizen
{
  /** The spouse of this Citizen; if null, this citizen
      is single. */
  /*@ nullable @*/ Citizen my_spouse = null;
  //@ invariant (my_spouse != null) ==>
  //@           my_spouse.my_spouse == this;

  /** Returns a new citizen who is single. */
  //@ ensures single();
  Citizen() {
    my_spouse = null;
  }
  ...
```

Trivial Static Checking

- lexical analysis only
- scan/lex source code
- typically keep only a small amount of contextual information
- check each construct on the fly
  - e.g., pattern match on strings

Syntactic Static Analysis

- scan and parse (parts of) a program
- generate AST for structures of interest
- walk over AST, pattern matching on interesting structures
- analyze each match for properties of interest, usually with a simple algorithm
- report results to user

Semantic Static Analysis

- scan, parse, and generate AST as before
- transform AST into an intermediate representation amendable to analysis
  - e.g., reduced language, guarded command language, static single assignment form
- analyze this representation semantically, generate verification conditions that logically express properties of interest
- give VCs to a theorem prover for checking
- interpret prover response for programmer

Static Checkers Included in CSI Eclipse

- CheckStyle - source and docs style checker
- Metrics - source-based metrics analysis
- PMD - source-based good/bad patterns
- FindBugs - bytecode-based patterns
- EclEmma - unit test code coverage
- ESC/Java2 - common programming errors

Grading with Checkers

- project's are partially graded based upon how well documentation, specifications, and code pass static checkers
- essentially, always try to ensure that there are no errors or warnings
  - code conforms to specified style
  - metrics guidelines are followed
  - no PMD or FindBugs markers
  - no typechecking errors from JML checker
  - no warnings from ESC/Java2

Tools for Thursday

- EclEmma
- JML 5.5

Thursday: Validation

Thursday 1:
Validation = Testing

Testing Topics

- why test?
- when to test?
- where to test?
- who tests?
- how to test?

Testing Topics

- why test?
- when to test?
- where to test?
- who tests?
- how to test?

Testing Topics

- why test?
- when to test?
- where to test?
- who tests?
- how to test?

Quality

Testing Topics

- why test?
- when to test?
- where to test?
- who tests?
- how to test?

Testing Topics

- why test?
- when to test?
- where to test?
- who tests?
- how to test?

Early

Testing Topics

- why test?
- when to test?
- where to test?
- who tests?
- how to test?

Early

Frequently

Testing Topics

- why test?
- when to test?
- where to test?
- who tests?
- how to test?

Testing Topics

Appropriately

- why test?
- when to test?
- where to test?
- who tests?
- how to test?

Testing Topics

- why test?
- when to test?
- where to test?
- who tests?
- how to test?

Appropriately

At Multiple Granularities

Testing Topics

- why test?
- when to test?
- where to test?
- who tests?
- how to test?

Testing Topics

- why test?
- when to test?
- where to test?
- who tests?
- how to test?

All Developers

Testing Topics

- why test?
- when to test?
- where to test?
- who tests?
- how to test?

All Developers

Q/A Team

Testing Topics

- why test?
- when to test?
- where to test?
- who tests?
- how to test?

Testing Topics

- why test?
- when to test?
- where to test?
- who tests?
- how to test?

Automatically

Testing Topics

- why test?
- when to test?
- where to test?
- who tests?
- how to test?

Why Test?

- if you do not care about bugs, don't bother
- many companies, and most developers, do little to no testing
  - product with bugs ship every day, but this is not always an *ethical* decision
- but if you want your software to be of moderate quality, you **must** test
- and if you want high quality, you must test *well*, *frequently*, and *rigorously*

When to Test?

- developers should run tests frequently, automatically, and locally
  - typically, one should rerun tests as frequently as one commits changes
- teams should run tests semi-frequently, automatically, and globally
  - system-wide builds, tests, and reports typically run every night automatically

Where to Test?

- testing must occur on all subcomponents of a software product
- focus most/first on the critical parts, then less/later on other parts
- tests should be written as early in the development process as is *reasonable*
  - writing tests too early means rewriting tests as requirements or design change

Who Tests?

- companies that need to deliver quality software products must test a great deal
- the more critical the product, the (radically) more testing is necessary
- some people really enjoy testing (!)
  - quality assurance (Q/A) is key role in many groups, but is often under-appreciated
  - manually writing tests requires a special frame of mind, a real analytical diligence

How to Test?

- write tests that
  - run automatically
  - exercise expected and unexpected behavior
  - focus on the smallest to the largest units
  - check all input values (within reason) that characterize the types involved

Automatize Everything

- avoid interactively testing either from the console or from a GUI
- add targets in your build system for lightweight and full functional testing
- always try to generate
  - low-level (feature-level) unit tests
  - GUI interaction robustness tests

Standard Targets

- include *quicktest* targets in your build system that run the quick and local tests
  - these tests should run in about the same amount of time it takes to compile
- write *fulltest* targets in your build system that run all tests of all components
  - these tests should be set up to run overnight and report results

Thursday 2:
Types of Testing

Unit Testing

- the most "popular" form of testing
  - (it only takes one great framework or one rockstar to make us love something we used to hate)
- lots of misunderstandings about
  - definition, applicability, use

Unit Testing Granularity

- variable-level
- method-level
- class-level
- component-level
- feature-level
- product-level

Unit Testing Granularity

- variable-level
- method-level
- class-level
- component-level
- feature-level
- product-level

```
public int i;

protected byte b;

Object o;

private String s;
```

Unit Testing Granularity

- variable-level
- method-level
- class-level
- component-level
- feature-level
- product-level

Unit Testing Granularity

- variable-level
- method-level
- class-level
- component-level
- feature-level
- product-level

```
public int m();

protected byte n();

Object o();

private long l();
```

Unit Testing Granularity

- variable-level
- method-level
- class-level
- component-level
- feature-level
- product-level

Unit Testing Granularity

- variable-level
- method-level
- class-level
- component-level
- feature-level
- product-level

```
C my_c = new C();
my_c.init();
my_c.do_someting();
my_c.finish_up();
```

Unit Testing Granularity

- variable-level
- method-level
- class-level
- component-level
- feature-level
- product-level

Architecture Testing

- does your implementation and its specification (informal and formal) correspond to your architecture?
- if there is no architecture description, there is no way to check this property
- using a high-level specification language like BON, or an architecture description language like ADL or ArchJava, this property becomes checkable

Integration Testing

- do separately developed subsystems play well with each other?
- do interfaces match/agree?
- does communication across module boundaries perform as expected?
  - communication is via APIs, RMI/RPC, message passing, shared variables

System Testing

- does the whole gosh, darn thing work?
- view the program as a single, enormous, compositional function
- send input in, test its output
- "easy" to do for command-line/STDIN-STDOUT tools
- **much** harder to accomplish for open systems

Regression Testing

- ensure that we learn from our mistakes
- each time a new bug is discovered and fixed in the system, introduce a test
- re-run all such tests each release to ensure that no old bugs crop up again in new releases
- such tests are often written by hand unless one uses formal specifications

Platform Testing

- ensure that your system works in a variety of environments
  - different operating systems, runtime scenarios, in the presence of other "competing" programs, etc.
- extremely difficult to do unless one has a system description that describes the rely/guarantee specification of the deployment
  - component descriptions, datafile and library dependencies, etc.

Deployment Testing

- does your system work as expected "in the wild"?
- typically used in server-side development, patch distribution, or large-scale network deployment (e.g., smartcards, mobile phones, etc.)
- alpha/beta testers are useful
- physical and virtual testbenches are critical
  - use virtualization and freely available resources like SourceForge's and Apple's machine farms

Testing Levels

- whitebox testing
  - see all the dirty laundry
- greybox testing
  - partial peek into implementation details
- blackbox testing
  - external client API only

Thursday 3:
Unit Testing

Defining “Unit”

- unit = function/method for 90+% of testers
- other granularities are appropriate, but harder to test
  - class = unit => testing sequences of calls
  - module = unit => testing semantics of aggregate component

XP's Perspective

- write tests before you write code
- tests define and exercise system interfaces
- if requirements change, so do the tests, and thence the APIs and implementations
- in practice, extremely rare that a team actually eats their own dog food

A Realistic Model

- tests are written
  - before, during, and after implementation
  - for stable APIs only
  - to match stable requirements

Thursday 4:
Testing in Practice

Code Coverage

- measure of the quality of a test suite
- should focus on critical subsystems
- aim is not 100% coverage, but an appropriate amount for your application and resources (people, budget, customer)
- dynamic coverage analysis is most straightforward means by which to reflect on your testing strategy and practices

Types of Coverage

- statement coverage
  - is each statement executed?
- branch coverage
  - is each branch in the program executed?
- value coverage
  - is every value of every type seen?

Coverage Rigor

- most test suites are ad hoc
- ad hoc testing = lack of rigor in strategy
- rigorous testing means having a justification for each test
- if a test has no justification, it is superfluous
- easiest path to rigor is the use of specifications and test generation

Manual Test Writing

- the manual test writing methodology:
  - determine critical values of all types
  - test all combinations of:
    - all standard boundary values
    - all critical values
    - all adjacent-to-critical values

Explosion of Values

method signature	# of necessary tests
`void m()`	1
`void n(boolean b)`	2
`void o(byte b)`	256
`void p(long l)`	2^64
`void q(String s)`	nearly ∞
`void r(byte[] b)`	2^40
`void s(String[] s)`	nearly ∞
`void t(List<T> l)`	∞

Return Types

method signature	# conditions in test harness
`void m()`	1
`boolean m()`	2
`byte m()`	256
`Object m()`	∞

Formal Parameters

method signature	# tests
`void m()`	1
`void m(boolean b)`	2
`void m(boolean b, byte c)`	2 * 256
`void m(boolean b, byte c, Object o)`	2 * 256 * ∞

Cyclomatic Complexity

```
boolean m(byte b) {
  while (true) {
    switch (b) {
      case 0: break;
      case -1: continue;
      case 1: b++;
      case 2: return true;
      default: b--;
    }
  return false;
}
```

Using Critical Values

- each type in your system has “interesting” values, as defined by your architecture
- identifying these values, and their equivalence classes, drives test set size reduction and increases code coverage
- interesting values are also related to method bodies’ structure via their control flow graph

Testing Frameworks

- jLog/log4j
  - designed for logging simple applications
- IDebug/KindDebug
  - designed for logging and testing complex concurrent and distributed systems
- jUnit/nUnit
  - automated unit testing for Java
  - test glue code generated automatically

Thursday 5: Testing with Specifications

Specifications as Oracles

- preconditions and invariant stipulate what values are legal, and thus “interesting”
- postcondition dictate what a method must accomplish
- pre/post pairs are thus oracles for behavior

Exhaustive Testing

- generate a test framework that exercises every possible value for every possible execution
- possible in extremely limited scenarioes
- necessary for certification in very limited agencies (e.g., NASA, biomedial, etc.)

Fuzz-Testing

- use existing test framework as a foundation for automated “tweaking” of values
- adjacent value generation is automatic
- fuzz-generation system “squints” at your values and guesses at appropriate similar-but-different values for testing

Mutation Testing

- like fuzz-testing, but for *program code*
- consumes method bodies and converts implementation into equivalent, simpler format
- mutates program to induce interesting, but slightly different structures to ensure that test coverage is complete

Your Testing Requirements

- write test code only for scenarios and events
- otherwise...
  - use JML-junit tool suite
  - identify critical values of your architecture
  - customize test drivers
  - execute early and often

Friday: Verification

Friday 1:
Kinds of Verification

Verification in a Nutshell

- rigorously, usually mathematically, analyze a program for specific properties
- analysis is often, but not always, *expensive*, *static*, *sound*, and *conservative*, but is rarely *complete*
- one can verify source or object code
- verification often, but not always, requires significant domain and tool expertise

Some Common Types of Verification

- abstract interpretation
- symbolic interpretation
- model checking
- push-button shallow analysis via automatic theorem proving
- deep analysis via interactive theorem proving

Not a Substitute

- verification fulfills EAL7, but is not a substitute for rigorous testing, code reviews, and other quality software engineering practices
- verification tools rarely fess-up to their failings
  - soundness of a mathematical formalism says nothing about the quality of a tool

Rules of Thumb

- who: verification should be performed by domain and tool experts, not necessarily the developer who wrote the code
- what: verify core, key subsystems
- when: verify after all other analysis
- why: verify when it is mandated by customers, government, certification bodies, agencies, law, or business need
- how: use the appropriate, complementary verification techniques in tandem

Friday 2:
Details of a
Verification Tool

ESC/Java2

- ESC/Java2 is an *extended static checker*
  - based upon DEC/Compaq SRC ESC/Java
  - operates on JML-annotated Java code
  - behaves like a compiler
    - error messages similar to javac & gcc
    - completely automated
    - hides enormous complexity from user

What is Extended Static Checking?

- type systems
  - Error: wrong number of arguments to call
- lint & modern compilers
  - Error: unreachable code
- full program verification
  - Error: qsort does not yield a sorted array

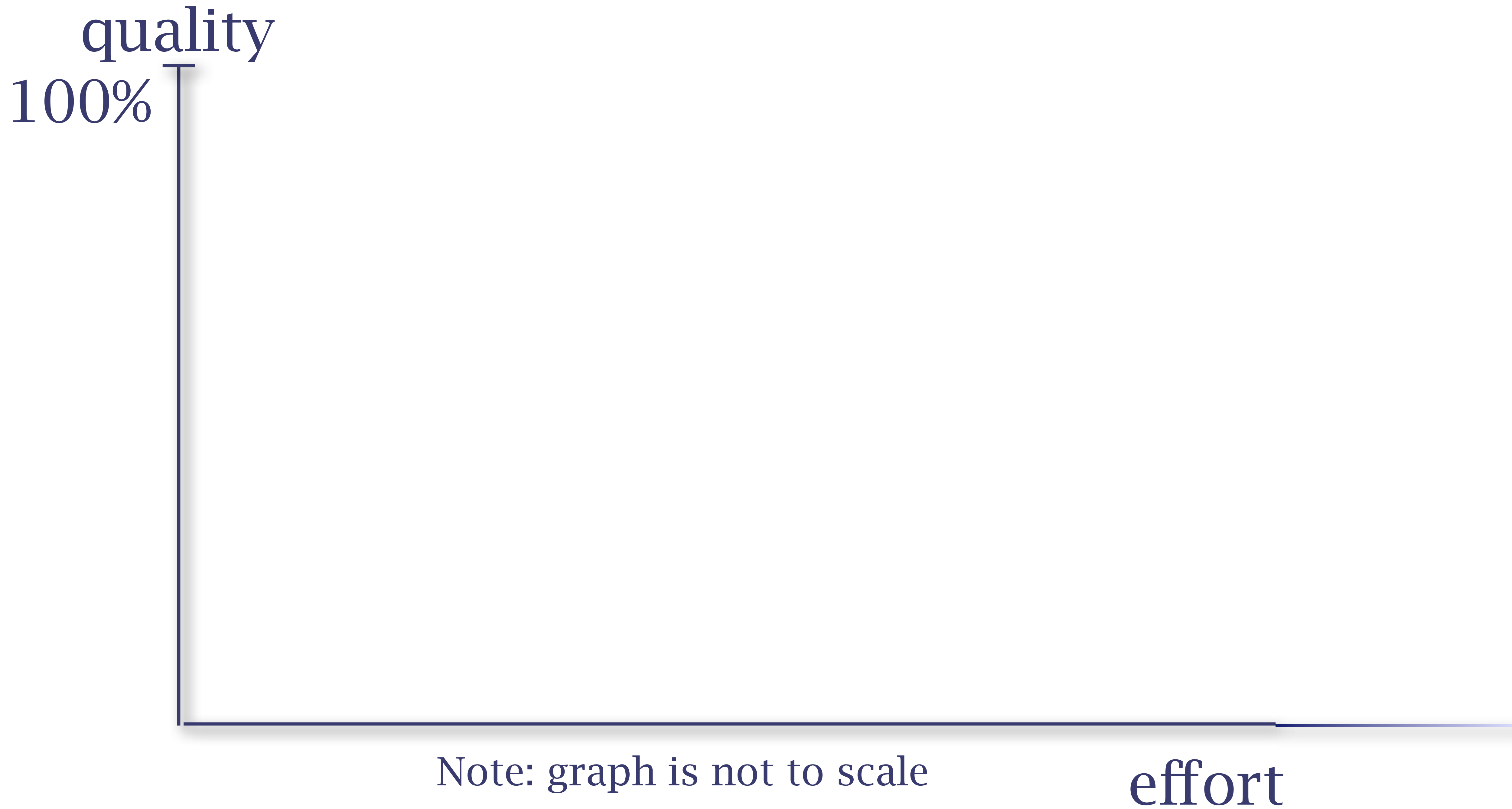
Why Not Just Test?

- testing is essential, but
  - expensive to write and *maintain*
  - finds errors *late*
  - *misses* many errors

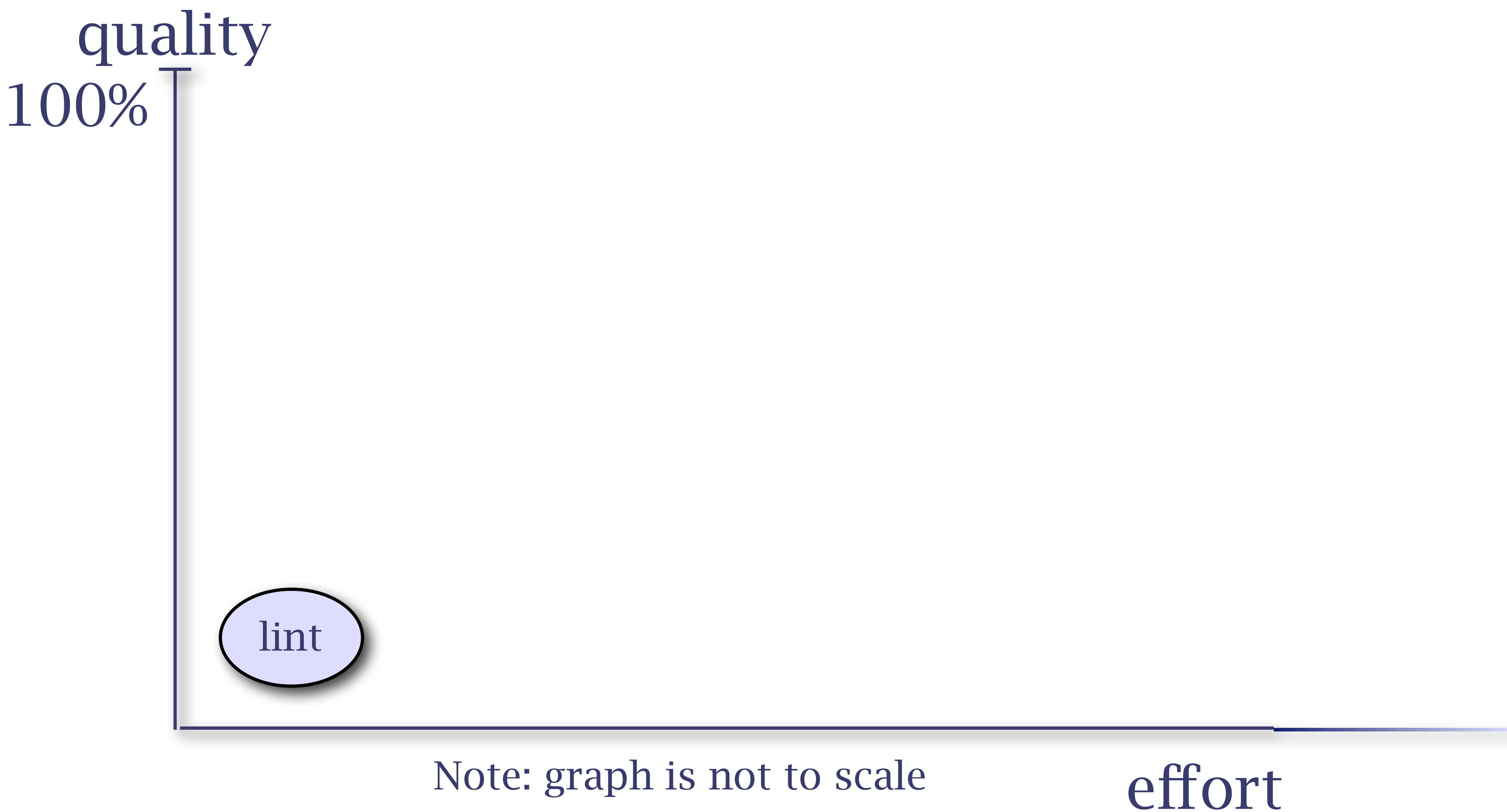
- static checking and testing are *complementary techniques*

Comparison of Static Checkers

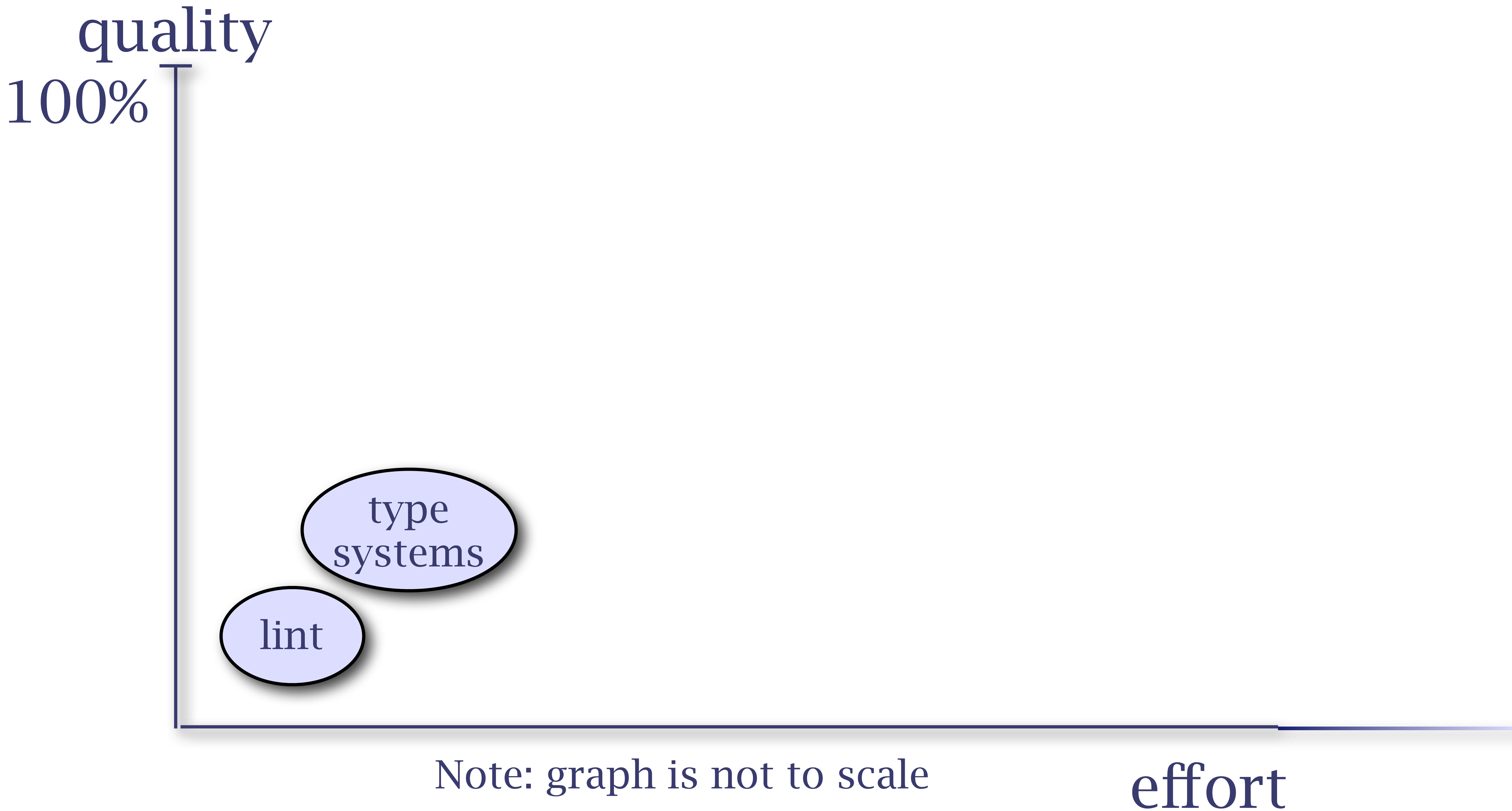
Comparison of Static Checkers



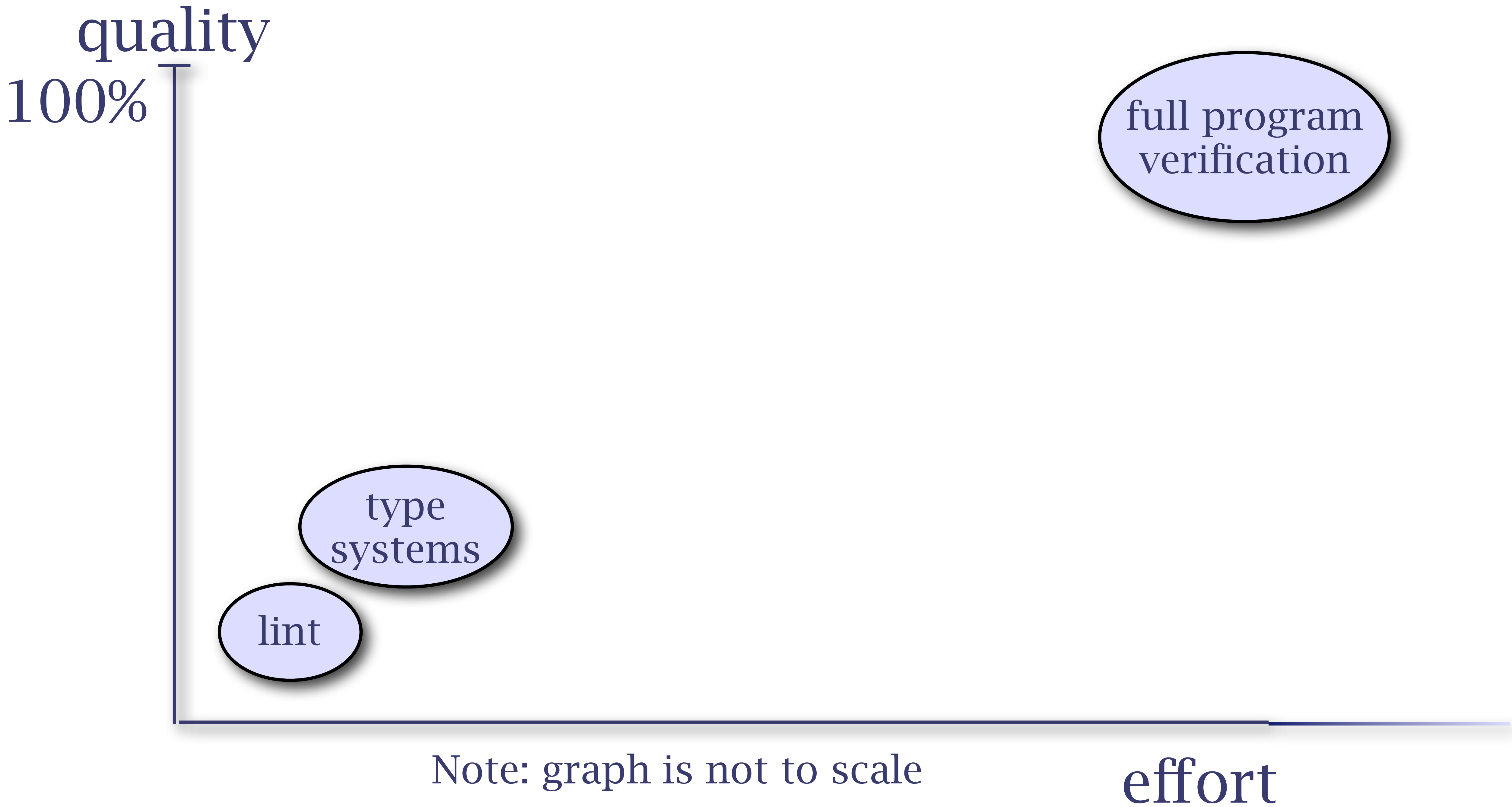
Comparison of Static Checkers



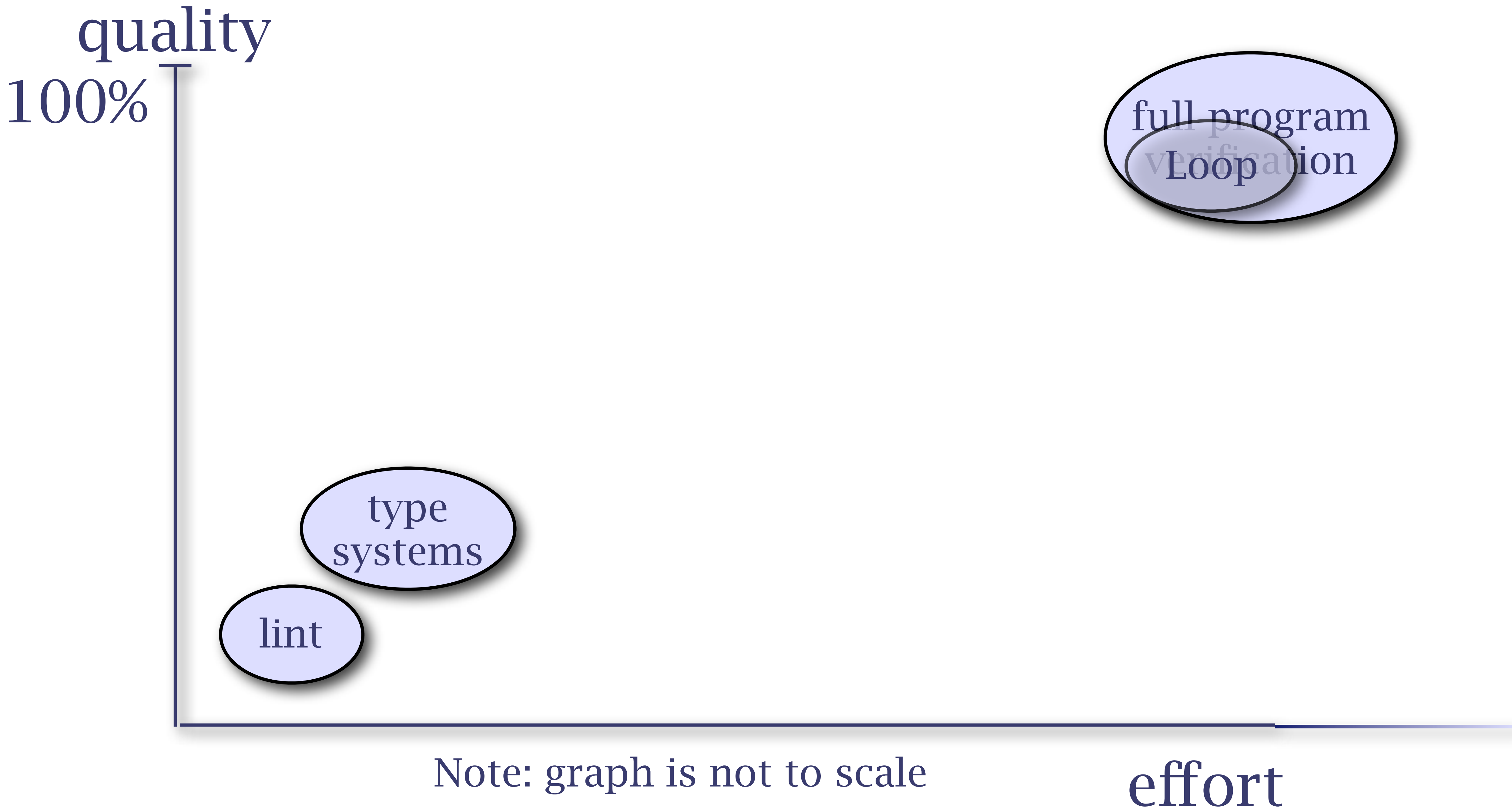
Comparison of Static Checkers



Comparison of Static Checkers

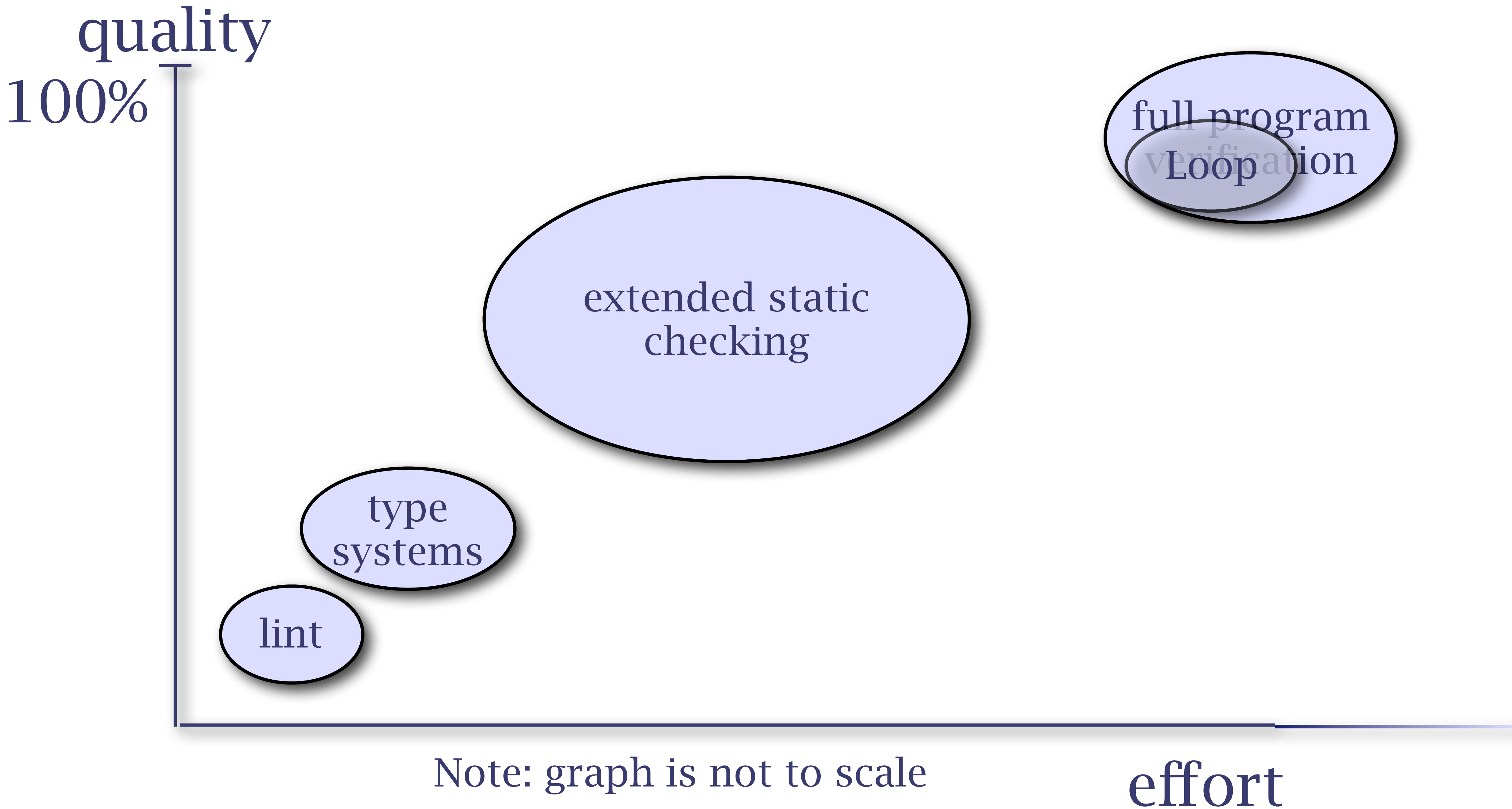


Comparison of Static Checkers

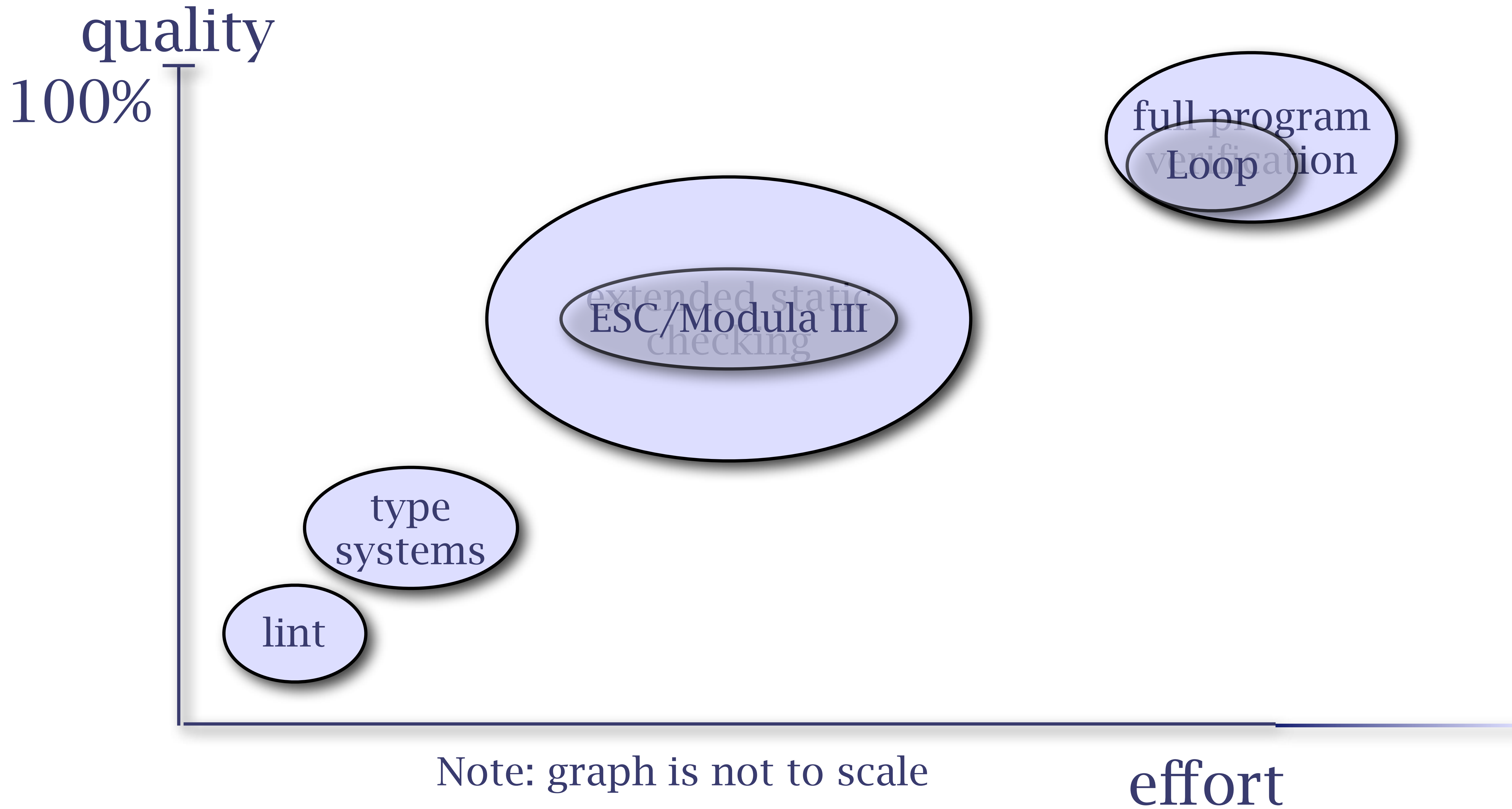


Friday, 1 May, 2009

Comparison of Static Checkers

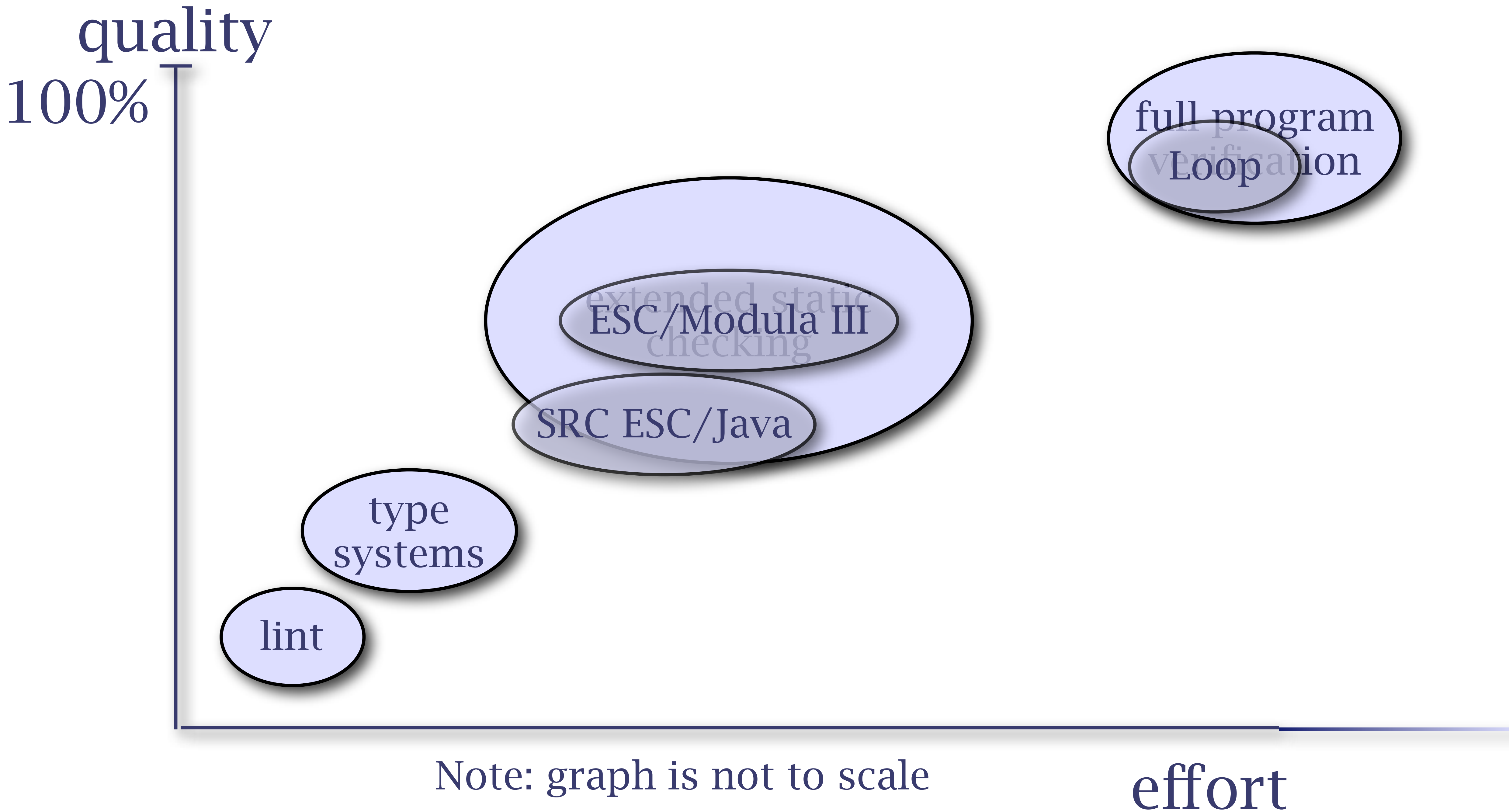


Comparison of Static Checkers

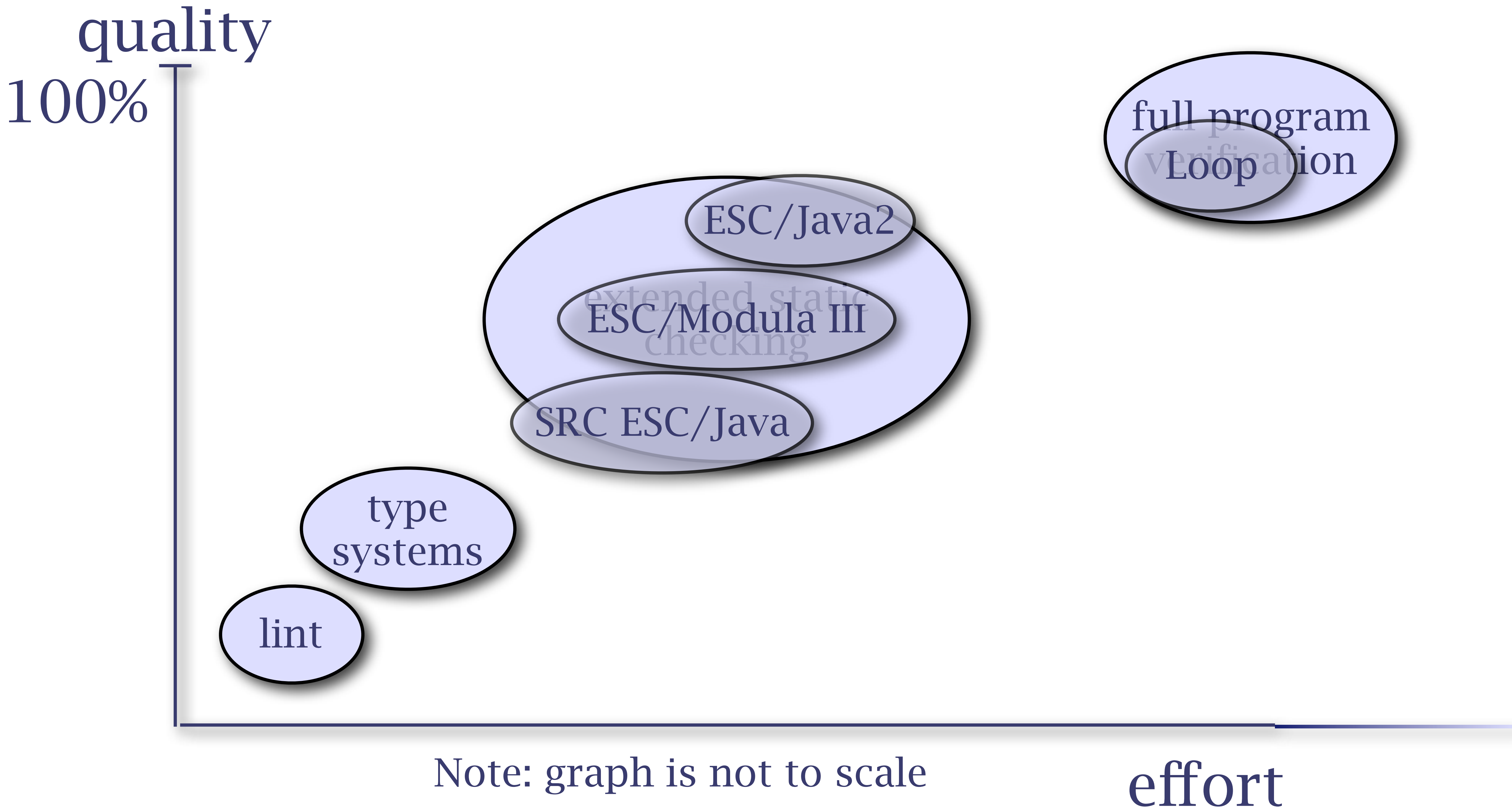


Note: graph is not to scale

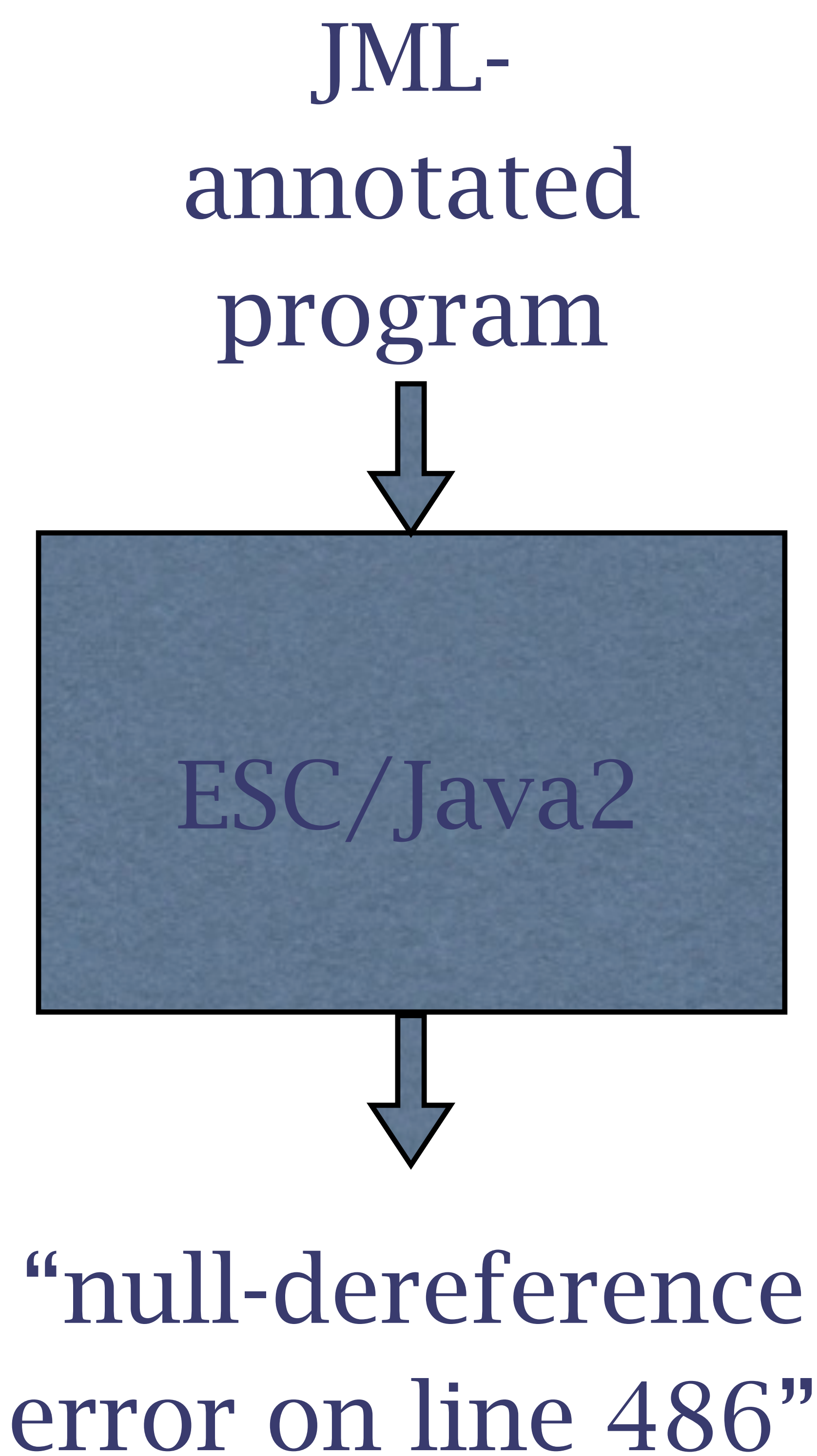
Comparison of Static Checkers



Comparison of Static Checkers



ESC/Java2 Use

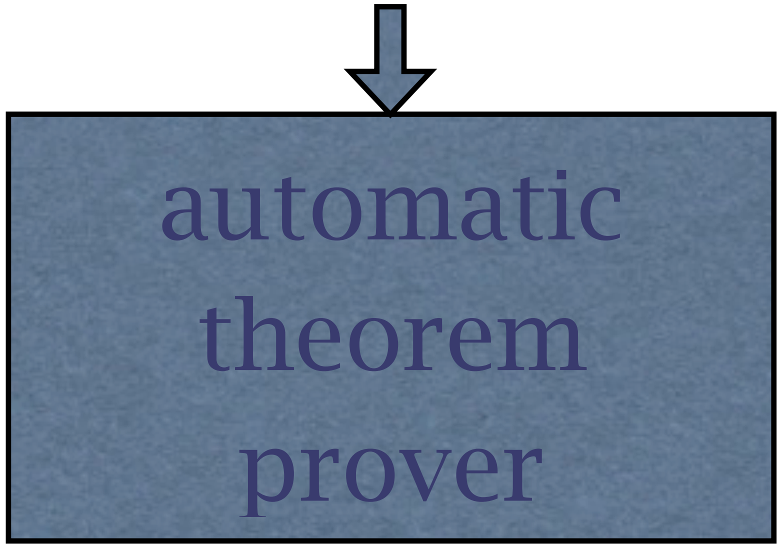


- Modularly checks for:
  - null-dereference errors
  - array bounds errors
  - type cast errors
  - specification violations
  - race conditions & deadlocks
  - ... dozens of other errors

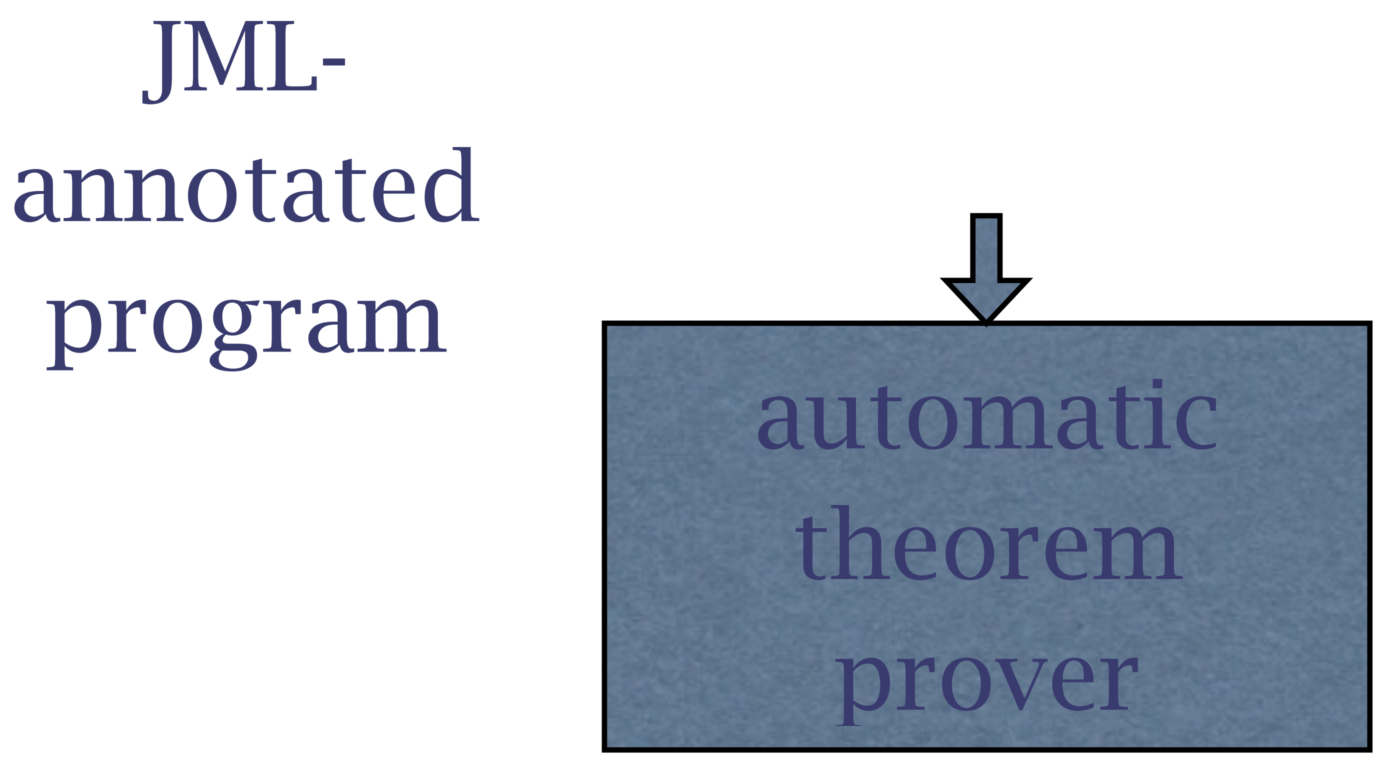
Soundness and Completeness

- a sound and complete prover is non-automated, very complex, and expensive
  - modular checking
  - properties of arithmetic and floats
  - complex invariants and data structures
- instead, design and build an unsound and incomplete verification tool
  - trade soundness and completeness for automation and usability

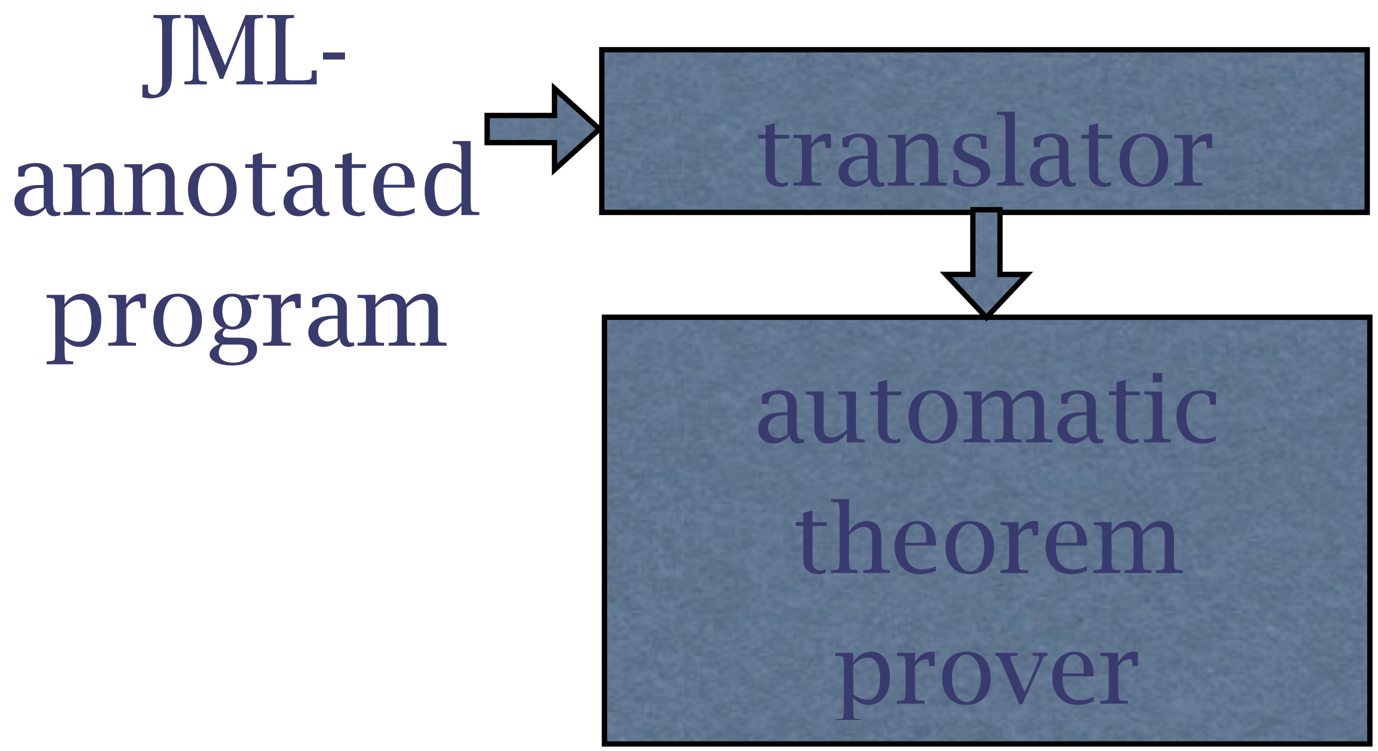
ESC/Java2 Architecture



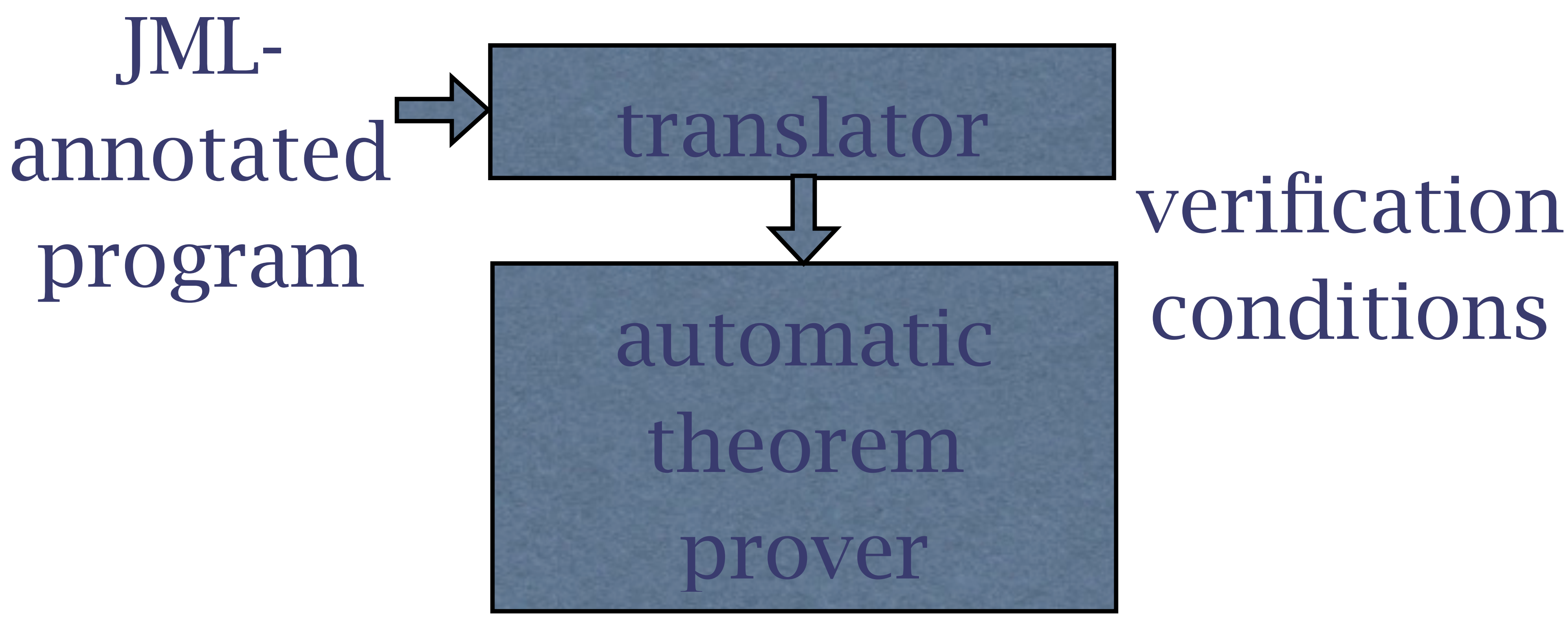
ESC/Java2 Architecture



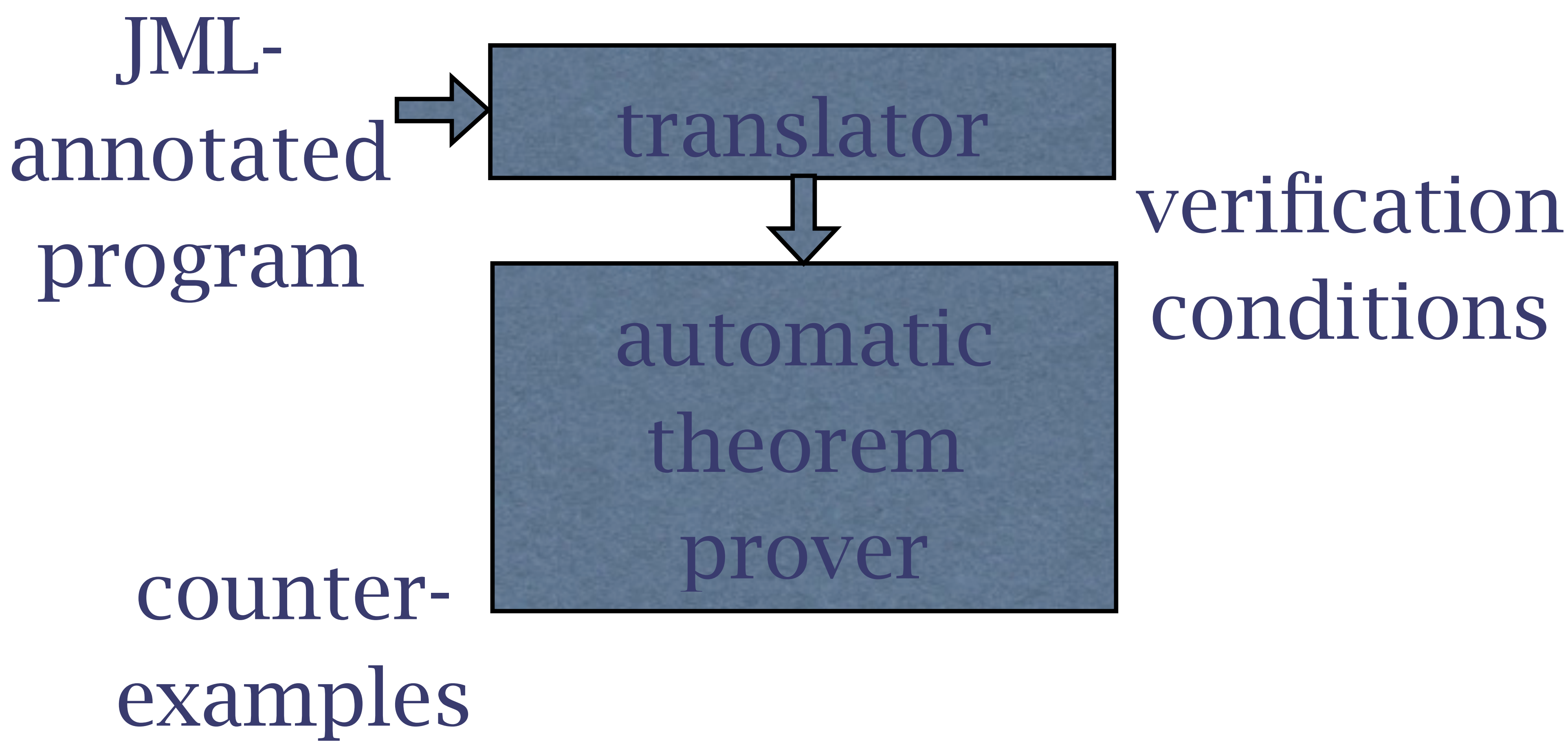
ESC/Java2 Architecture



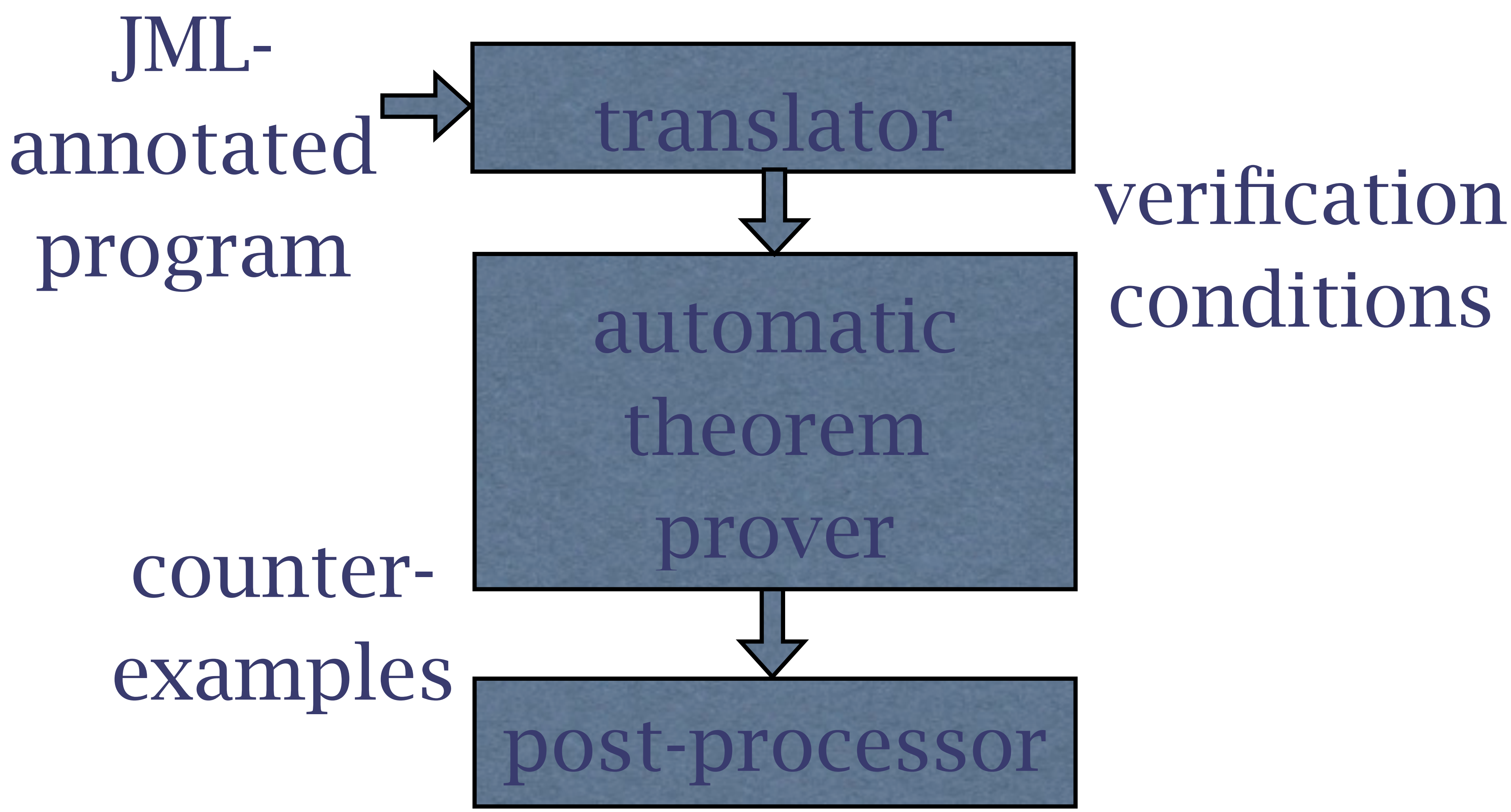
ESC/Java2 Architecture



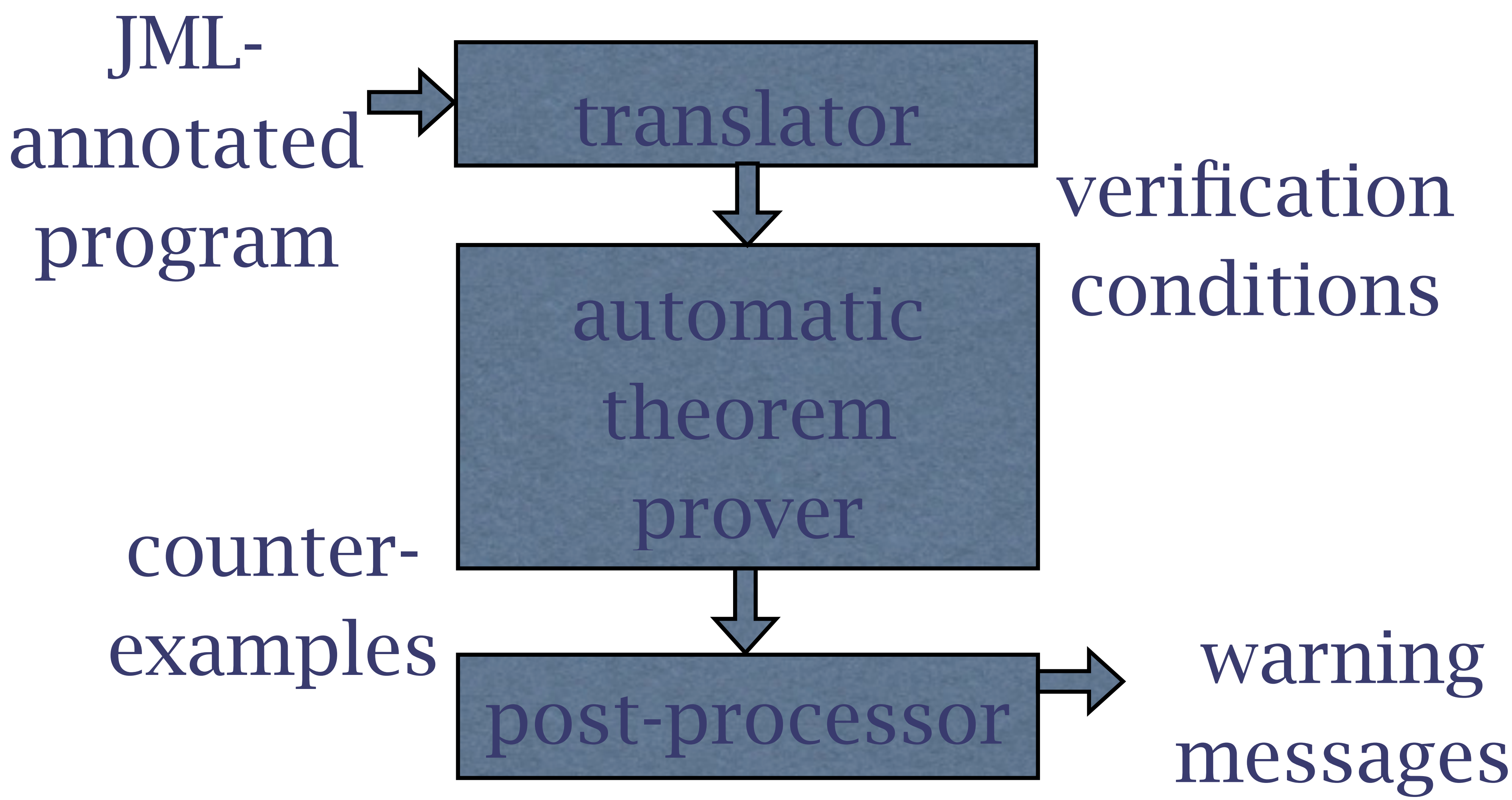
ESC/Java2 Architecture



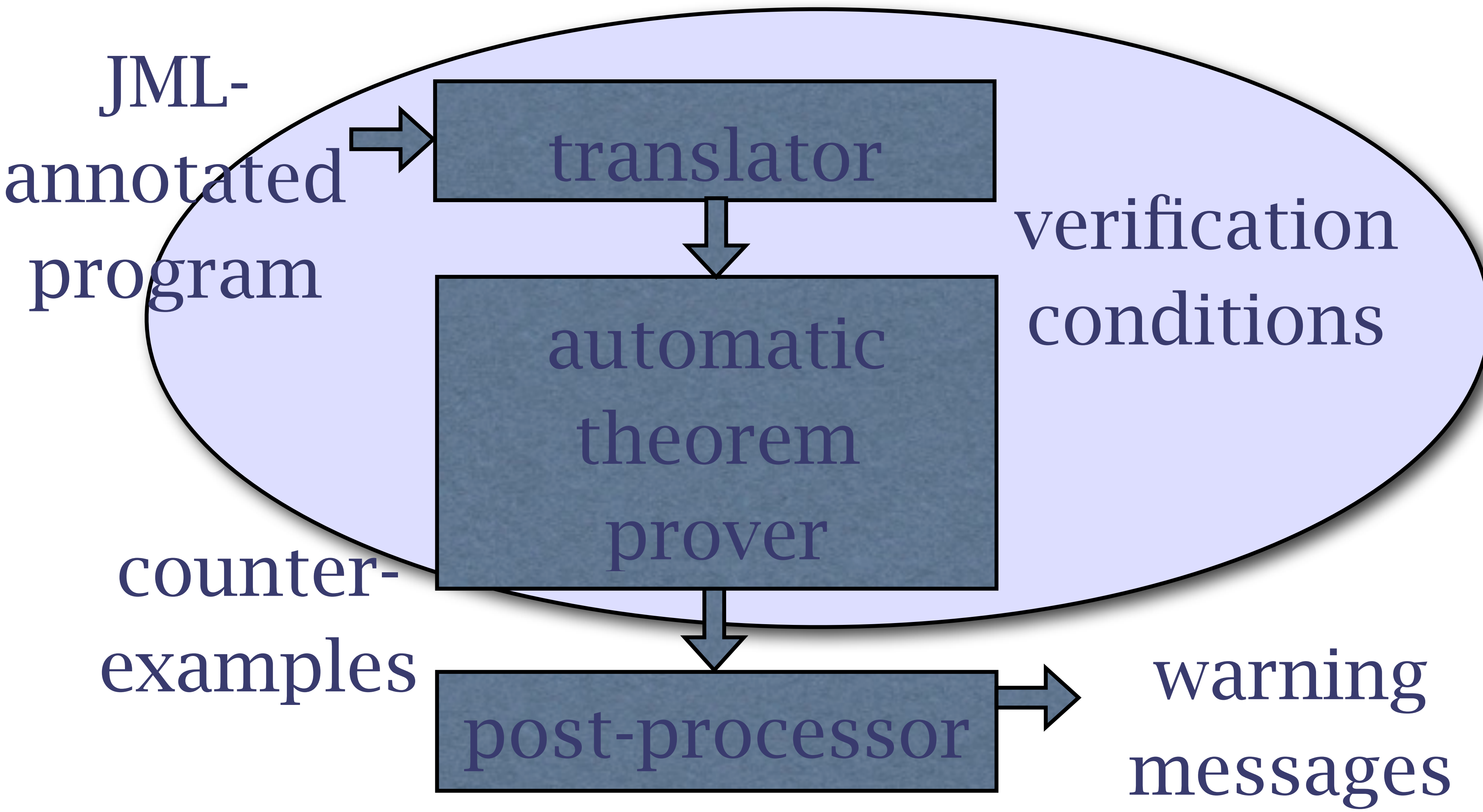
ESC/Java2 Architecture



ESC/Java2 Architecture



ESC/Java2 Architecture



The ESC/Java2 Object Logic

- (very) partial semantics for Java and JML
- written in unsorted first-order logic
- highly tuned to current theorem prover's capabilities and quirks
  - Nelson's Simplify prover circa mid-80s
- originally consisted of 81 axioms
- extended by 20 axioms in ESC/Java2

Example Java Type Axioms

```
(DEFPRED (<: t0 t1))

(BG_PUSH (<: |T_java.lang.Object|
             |T_java.lang.Object|))

; <: reflexive
(BG_PUSH
  (FORALL (t)
    (<: t t)))


; <: transitive
(BG_PUSH
  (FORALL (t0 t1 t2)
    (IMPLIES (AND (<: t0 t1) (<: t1 t2))
      (<: t0 t2))))
```

```
;anti-symmetry
(BG_PUSH
 (FORALL
  (t0 t1)
  (IMPLIES (AND (<: t0 t1) (<: t1 t0))
           (EQ t0 t1))))


; primitive types are final
(BG_PUSH (FORALL (t)
        (IMPLIES (<: t |T_boolean|)
                 (EQ t |
T_boolean|))))
```

Examples Showing Java Incompleteness

```
(BG_PUSH (FORALL (x)
          (IFF (is x |T_char|) (AND (<= 0 x) (<= x 65535)))))
(BG_PUSH (FORALL (x)
          (IFF (is x |T_byte|) (AND (<= -128 x) (<= x 127)))))
(BG_PUSH (FORALL (x)
          (IFF (is x |T_short|) (AND (<= -32768 x) (<= x 32767)))))
(BG_PUSH (FORALL (x)
          (IFF (is x |T_int|) (AND (<= intFirst x) (<= x intLast)))))
(BG_PUSH (FORALL (x)
          (IFF (is x |T_long|) (AND (<= longFirst x) (<= x longLast)))))

(BG_PUSH (< longFirst intFirst))
(BG_PUSH (< intFirst -1000000))
(BG_PUSH (< 1000000 intLast))
(BG_PUSH (< intLast longLast))
```

Examples of Java & JML Semantics

```
(DEFPRED (is x t))

(BG_PUSH (FORALL (x t)
                (is (cast x t) t)))

(BG_PUSH (FORALL (x t)
                (IMPLIES (is x t) (EQ (cast x t) x))))

(BG_PUSH
 (FORALL (e a i)
        (is (select (select (asElems e) a) i)
            (elemtype (typeof a)))))

(DEFPRED (nonnullelements x e)
   (AND (NEQ x null)
        (FORALL (i)
                (IMPLIES (AND (<= 0 i) (< i (arrayLength x)))
                         (NEQ (select (select e x) i) null)))))
```

ESC/Java2 Calculi

- used for verification condition generation in Dijkstra wp/wlp style
- easy for small/research languages
- much harder for "real world" languages
  - typed concurrent object-oriented language
  - dynamic memory allocation and GC
  - exceptions
  - aliasing

VC Generation for Java

annotated source

↓

guarded commands

↓

verification condition

```
x = a[ i++ ];

assume preconditions
assume invariants
...
i0 = i;
i = i + 1;
assert (LABEL null@218: a != null);
assert (LABEL IndexNeg@218: 0 <= i0);
assert (LABEL IndexTooBig@218: i0 < a.length);
x = elems[a][i0];
...
assert postconditions
assert invariants
```

$$\forall i_0.(i_0 = i \implies \ldots)$$

Verification Condition

- formula in unsorted, first-order predicate calculus
  - equality and function symbols
  - quantifiers
  - arithmetic operators
  - select and store operations
  - e.g., $\forall x. \forall y. \exists z. (x > y \implies z \times 2 == \ldots$

Example Verification Condition

- verification condition large & unstructured

```
(EXPLIES (LBLNEG |vc.Bag.isEmpty.11.2| (IMPLIES (AND (EQ |n@pre:3.6| |n:
3.6|) (EQ |n:3.6| (asField |n:3.6| T_int)) (EQ |a@pre:2.8| |a:2.8|) (EQ |a:
2.8| (asField |a:2.8| (array T_int))) (< (fClosedTime |a:2.8|) alloc) (EQ |
MAX_VALUE@pre:10..| |MAX_VALUE:10..|) (EQ |@true| (is |MAX_VALUE:10..|
T_int)) (EQ |length@pre:unknown| |length:unknown|) (EQ |length:unknown|
(asField |length:unknown| T_int)) (EQ |elems@pre| elems) (EQ elems (asElems
elems)) (< (eClosedTime elems) alloc) (EQ LS (asLockSet LS)) (EQ |
alloc@pre| alloc) (EQ |state@pre| state)) (NOT (AND (EQ |@true| (is this
T_Bag)) (EQ |@true| (isAllocated this alloc)) (NEQ this null) (EQ RES
(integralEQ (select |n:3.6| this) 0)) (LBLPOS |trace.Return^0,12.4| (EQ |
@true| |@true|)) (NOT (LBLNEG |Exception@13.2| (EQ |ecReturn| |
ecReturn|))))))) (AND (DISTINCT |ecReturn|) (< 1000000 pos2147483647)))
```

Friday 3:
Model Checking

What is a Model?

- a model is a *conservative* abstraction of another artifact
- software model checking means defining or building a model of your software system and checking properties of that model
  - if the model has a problem, then your software has a problem
  - but not all problems of your software are problems in the model

Manual Model Construction

- you are doing it right now, via JML
- alternative means by which to express models include languages like BON, set theory, theories like those in ESC/Java2, etc.
- automatic model construction means model extraction from program source
    - course, difficult, conservative

Model Properties

- you are expressing model properties now
- alternative means by which one expresses model properties includes
  - predicate calculus
  - first order logic
  - temporal logics

Kinds of Model Checkers

- explicit state
  - explore the entire state space of the model explicitly (sound, complete, and very, very expensive)
- finite bound
  - explore only to a particular depth
- symbolic
  - explore a symbolic representation

State Space Size

- even a trivial program will have a state space whose size is on the order of 2^32
- most programs >> 2^100
- exploring this state space explicitly using all of Google's hardware concurrently would take >> 15 billion years (age of the universe)

Model Checking Magic

- a state space of a hardware or software system is regular and contains symmetries
- model checkers exploit these properties to decompose state space into equivalent pieces, or find parts of the space that are unreachable, and collapse the space
- properties also are sometimes simplified

Model Checking in Industry

- VLSI design and verification
- Windows Microsoft-certified drivers
- NASA software for space missions
- concurrent hardware and software
- protocol analysis

Friday 4: Using Verification Effectively

Using ESC/Java2 Effectively

- basic familiarity with ESC/Java2 is easy
  - it is automatic and behaves like a compiler
- but any non-trivial use quickly becomes *very* difficult and time-consuming
  - complexity of Java and JML semantics
  - limitations of logic
  - designed limitations of tool
  - limitations of Simplify theorem prover

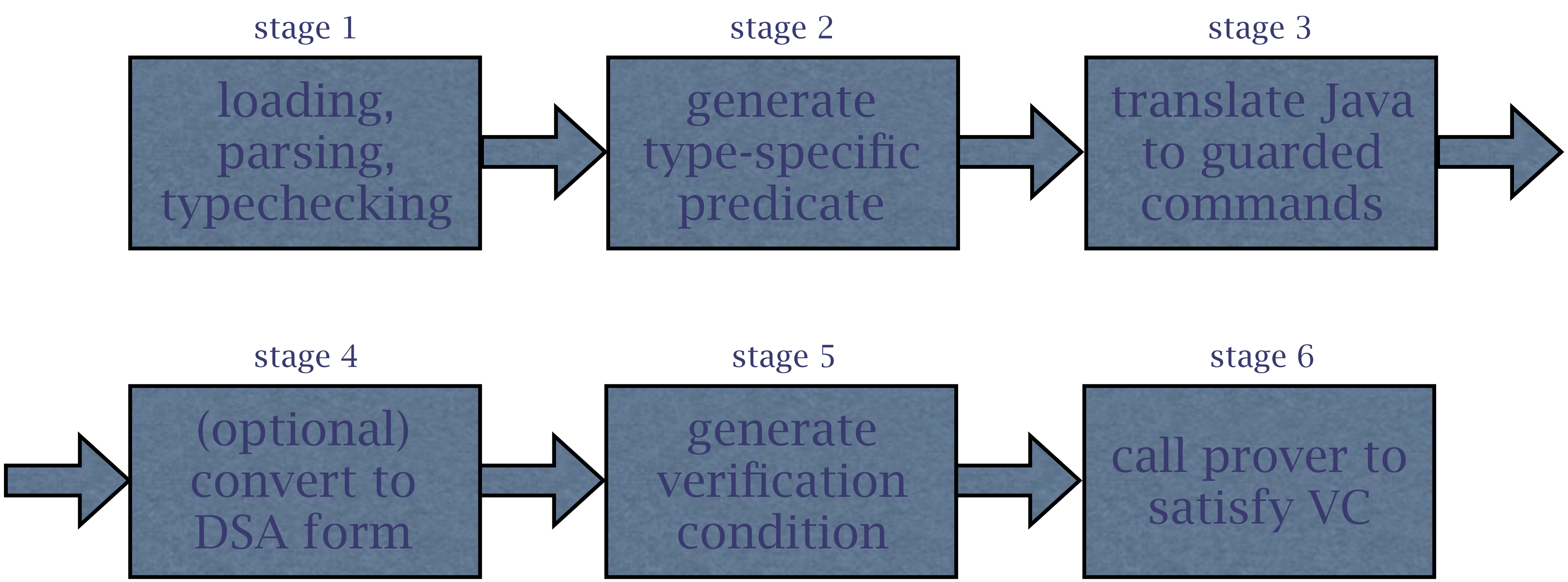
Thinking, not Hacking

- successful application of tool requires hard *thought* and very little *labor*
- recognizing that *specific misbehavior* implies *particular errors* in specifications or program code is key to effective use
- understanding *theoretical underpinnings* of extended static checking is very helpful
- a problem solving process for verification is needed for successful adoption

Verification Process

- the *key aspects* of the verification process
  - *small steps* in specification refinement and program development
  - *iterative* and *continuous* application of ESC/Java2 to method or class of focus
- use a standardized problem-solving technique for resolving warnings
  - *think* before you *type*

ESC/Java2 Architecture



How ESC/Java2 Works

- find, load, parse, and typecheck all relevant files and dependencies
  - this includes all refinements, models, etc.
- for each class being checked, generate a type-specific background predicate
  - type and subtype information about classes and fields
  - non_null-ness of references
  - size of constants

How ESC/Java2 Works (2)

- translate each routine to be check into a verification condition (VC)
  - intermediate step in this translation is to translate Java into a (Dijkstra-like) guarded-command language
  - translation is accomplished by generating strongest-postconditions or weakest-preconditions for method body

How ESC/Java2 Works (3)

- ask theorem prover to prove VC
  - background predicate for Java expressed as a set of axioms
  - type-specific background predicate generated in second step is assumed true
  - assert VC is true
- if proof fails and prover finds counterexample, translate result back to warning message and Java, if possible

Examining the Results of Each Stage

- -v alone to print information on loading, parsing, refinement, etc.
- -showDesugaredSpecs to see heavyweight specifications desugared to lightweight ones (will also be -sds in next release)
- -pgc to print guarded command
- -ppvc to pretty-print verification condition
- -pxLog to print predicate sent to prover

A Stage-Driven Process

- ensure that the proper source and bytecode files are being loaded
  - this is particularly important when initially setting up a verification problem and when using refinement
- make sure that your specs mean what you think they do be examining the desugared specs
  - multiple heavyweight specs sometimes have unintuitive meaning for the beginner

A Stage-Driven Process (2)

- check size of "local contributors"
  - e.g., 35 types 99 invariants 62 fields
- examine the generated VC
  - it must has a reasonable structure
    - type-specific background predicate, followed by translated specification and program code
  - it is reasonably sized
    - ~1MB is ok, multiple MB is a problem

Dealing with Complexity

- specification and code complexity are the primary factors in verification complexity
- if performing "Design by Contract" then one can "Design for Verification" also
- if performing "Contract the Design" then verification is sometimes only possible with refinement if code modification is not permitted

Managing Spec Complexity

- write and verify specs *iteratively* using *very small steps*
- use *independent heavyweight specification blocks* to specify *independent behaviors*
- ensure your specs are *sound*
  - assert a false predicate to check
  - eliminate suspect predicates iteratively to determine source of unsoundness

Managing Spec Complexity (2)

- use ghost variables or model fields to factor out complex specification subexpressions
  - helps with comprehension, not verification
- avoid universally quantified expressions
- use the *objectState* datagroup as much as reasonable for your frame axioms
- use the *owner* field to disambiguate objects

Managing Code Complexity

- track cyclic complexity of method bodies
  - each branch, switch case, loop, and exception block doubles complexity
- decompose methods into smallest reasonable units
  - Smalltalk and Eiffel method size rule-of-thumb applies (e.g., all methods <15 LOC)
- avoid constructors that make calls

Managing Code Complexity (2)

- focus on methods that make no calls first
- work from low to high cyclic complexity
- use assertions to check
- recognize sources of incompleteness of Java semantics
  - complex arithmetic
  - bit-level operations
  - String manipulations

Refinement for Complex Verification

- if you have a method with high cyclic complexity that you cannot refactor
  - inherit and override
  - implement and verify separate private methods for each branch of original method
  - implement overridden version as composition of verified new methods