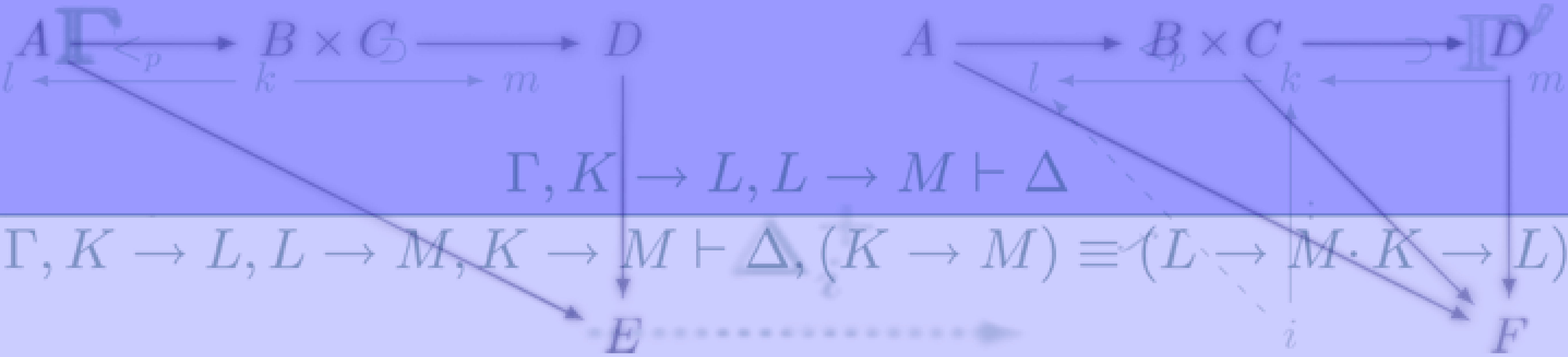


Practical Verification of Java

Joe Kiniry
KindSoftware Research Group
Systems Research Group
CASL: Complex and Adaptive Systems Laboratory
School of Computer Science and Informatics
University College Dublin, Ireland

Mobius

Acknowledgments

K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, Jim Saxe, Raymie Stata, Cormac Flanagan (while at DEC SRC); David Cok (Kodak); Carl Pully (ACME-Labs); Cees-Bart Breunesse, Arnout Engelen, Christian Haack, Ichiro Hasuo, Engelbert Hubbers, Bart Jacobs, Martijn Oostdijk, Wolter Pieters, Erik Poll, Joachim van den Berg, Martijn Warnier (RUN); Gilles Barthe, Julien Charles, Benjamin Grigore, Marieke Huisman, Clément Hurlin, and Mariela Pavlova, Gustavo Petri (INRIA); Cesare Tinelli, Jeg Hagen, and Alex Fuches (Univ. of Iowa); Aleksey Schubert (Univ. of Warsaw); Michal Moskał (Wroclaw University); David Naumann (Stevens); Patrice Chalin, Perry James, George Karabotos, Frederic Rioux (Concordia University); Torben Amtoft, Anindya Banerjee, John Hatcliff, Venkatesh Prasad Ranganath, Robby, Edwin Rodríguez, Todd Wallenstein (KSU), Yoonsik Cheon, Curtis Clifton, Gary Leaven, Todd Millstein (while at Iowa St.); Claudia Brauchli, Adam Darvas, Werner Dietl, Hermann Lehner, Ovidio Mallo, Peter Müller, Arsenii Rudich (ETHZ); Dermot Cochran, Lorcan Coyle, Steve Neely, Graeme Stevenson (UCD); and my PhD students Fintan Fairmichael, Robin Green, Radu Grigore, Mikoláš Janota, and Alan Morkan and undergraduate students Barry Denby and Conor Gallagher, and Patrick Tierney and the many students in my software engineering courses

The Java Modeling Language (JML)

- initiative of Gary Leavens [Iowa St.]
- Behavioral Interface Specification Language (BISL) for Java
  - annotations for Java programs expressing pre- and postconditions, invariants, etc.
- inspired by Eiffel's DBC and Larch
- Primary design goal: easy to learn
  - is a simple extension to Java's syntax

A JML Example

```
private int balance;
final static int MAX_BALANCE;

/*@ invariant 0 <= MAX_BALANCE &&
              balance < MAX_BALANCE;
  @*/
```

A JML Example

```
/*@ requires    0 <= amount;
    assignable balance;
    ensures    balance ==
                 \old(balance) - amount;
    signals    (PurseException)
                 balance == \old(balance);
  @*/
public void debit(int amount) {
  ...
}
```

A JML Example

```
/*@ requires   0 <= amount;
    assignable balance;
    ensures    balance ==
                 \old(balance) - amount;
    signals    (PurseException)
                 balance == \old(balance);
  @*/
public void debit(int amount);
```

A JML Example

```
/*@ requires    0 <= amount;
    assignable balance;
    ensures    balance ==
                 \old(balance) - amount;
    signals    (PurseException)
                 balance == \old(balance);
  @*/
public void debit(int amount);
```

A JML Example

```
/*@ requires   0 <= amount;
    assignable balance;
    ensures    balance ==
                 \old(balance) - amount;
    signals    (PurseException)
                 balance == \old(balance);
  @*/
public void debit(int amount);
```

A JML Example

```
/*@ requires   0 <= amount;
    assignable balance;
    ensures    balance ==
                 \old(balance) - amount;
    signals    (PurseException)
                 balance == \old(balance);
  @*/
public void debit(int amount);
```

A JML Example

```
/*@ requires    0 <= amount;
    assignable balance;
    ensures    balance ==
                 \old(balance) - amount;
    signals    (PurseException)
                 balance == \old(balance);
  @*/
public void debit(int amount);
```

A JML Example

```
/*@ requires    0 <= amount;
    assignable balance;
    ensures    balance ==
                 \old(balance - amount);
    signals    (PurseException)
                 balance == \old(balance);
  @*/
public void debit(int amount);
```

A JML Example

```
private byte[] pin;
private byte   appletState;

/*@ invariant
      appletState == PERSONALIZED
        ==>
      pin != null &&
      pin.length == 4 &&
      (\forall int i; 0 <= i && i < 4;
                      0 <= pin[i] && pin[i] <= 9);
  @*/
```

A JML Example

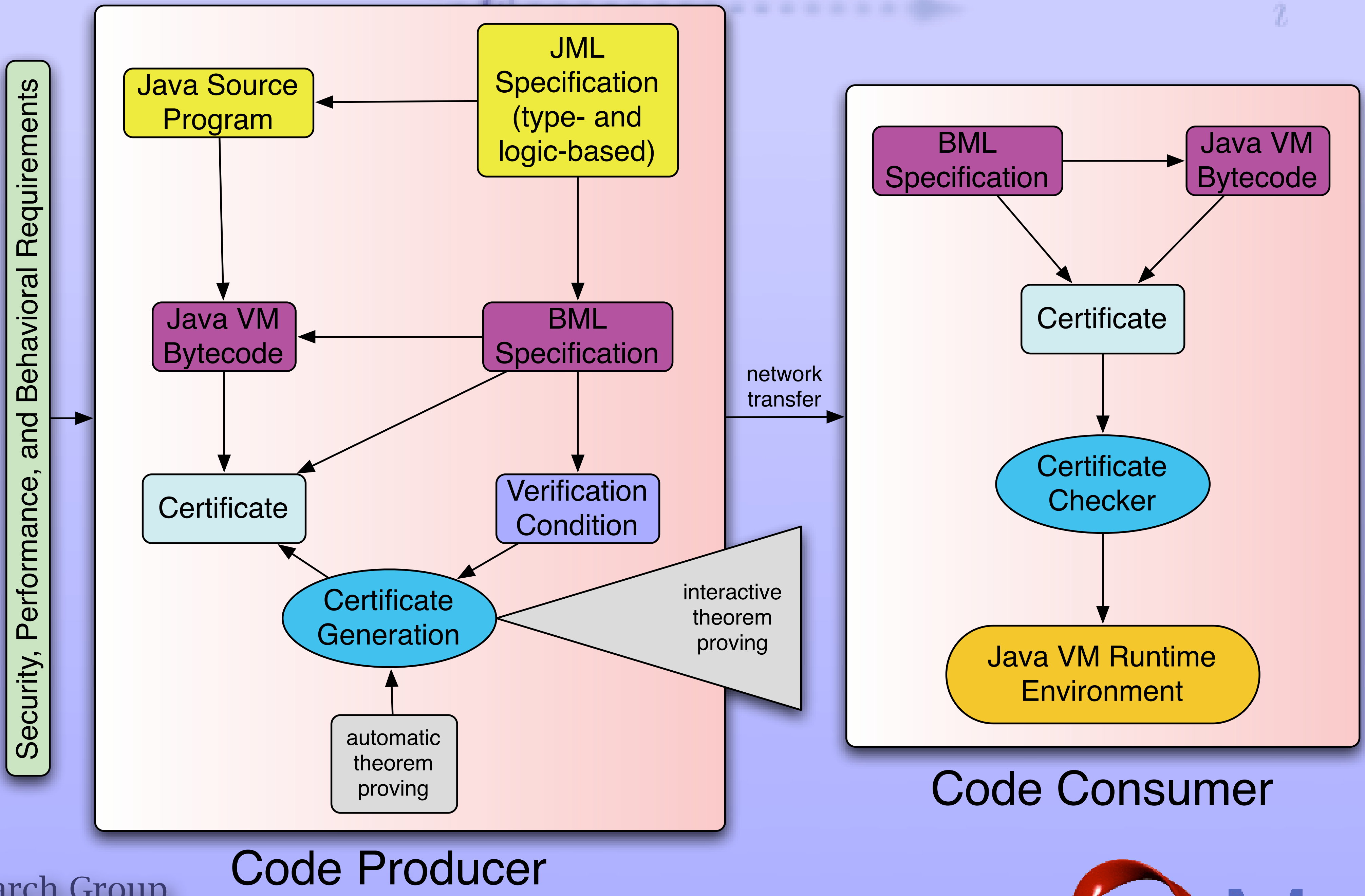
```
private byte[] pin;
private byte   appletState;

/*@ invariant
      (appletState == PERSONALIZED)
        ==>
      (pin != null) &&
      (pin.length == 4) &&
      (\forall int i; ((0 <= i) && (i < 4));
                      ((0 <= pin[i]) && (pin[i] <= 9));
  @*/
```

The Mobius Program Verification Environment

- Mobius Program Verification Environment [Mobius consortium, led by UCD]
  - combines existing best-of-breed tools
  - new, single semantic foundation
  - reasons about Java source and bytecode
  - produces proof-carrying code certificates
  - includes a PCC infrastructure for Java-like languages (i.e., OO + bytecode, Spec#-like)

Mobius

Mobius Architecture



Systems Research Group
School of Computer Science and Informatics
University College Dublin

Mobius

Mobius PVE *User* Features

- Java program code features
  - writing new code
  - type-aware completion
  - compiling, debugging, refactoring, folding code
  - generate Javadoc documentation
  - analyze code complexity
  - analyze coding standard conformance
  - detecting common programming errors

Mobius PVE *User* Features

- Java Modeling Language features
  - writing new specifications
  - tracking refinement
  - compiling specifications to runtime tests
  - generate Javadoc documentation
- Bytecode Modeling Language features
  - display BML-annotated Java VM bytecode
  - edit BML
  - edit Java VM bytecode

Mobius PVE *User* Features

- JML-annotated programs features
    - unit test generation
    - specification generation
        - class and loop invariant generation, precondition propagation, requirement-driven generation
    - translation to guarded commands
    - indicate program parts that correspond to the current VC or proof fragment
    - annotation tracing
        - creation, justification, relationship to requirements, features, and bugs, etc.

Mobius PVE *User* Features

- theorem prover features
  - use in a natural way interactive provers
  - choose which automatic provers to use
- application requirements features
  - authoring new requirements
  - refine requirements into specifications
  - tracing requirements over time, through architecture, and into proof and certificate
- certificate features
  - generate Mobius PCC certificates

Verification Bus Features

- Java, JML, and BML lexer, parser, type checker, and transformation subsystem
  - generates, visualizes, and manipulates Java VM bytecode, JML annotations, Javadocs, BML-annotated bytecode, and DOT files
- FreeBoogie subsystem
  - FreeBoogiePL = structured and unstructured BoogiePL + explicit heap + separation logic
  - FreeBoogie VC generation targets Mobius VC back-end

Verification Bus Features

- Mobius VC back-end
  - unsorted and sorted VC representation
  - logic-aware syntax generation to several automatic and interactive theorem provers
    - generation of Mobius VCs in Base Logic in Coq
- Mobius ESC VC back-end
  - generation of ESC VCs in ESC Logics
  - generate ESC VCs for several automatic and interactive theorem provers
  - extended static checking of ESC VCs with rich in-editor feedback

Verification Bus Features

- Mobius Prover back-end
  - generic interaction with a variety of automatic and interactive theorem provers
    - automatic provers supported
      - Simplify, SMT, CVC3, Yices, Fx7
    - interactive provers supported
      - Coq and PVS
  - proof status maintenance
  - proof unit/smoke testing
  - automatic and seamless proof sharing amongst distributed collaborators

Verification Bus Features

- Mobius proof-transforming compiler
  - VC generation from Java source
  - compilation of source code-level proofs to bytecode-level proofs
- integration of several support tools
  - e.g., CheckStyle and FindBugs
  - the Race Condition Checker (RCC)
- help system and process management
  - task and feature tracking
  - online hypertext architecture docs and help

Mobius PVE: Status

- full support available for:
  - all Java and nearly all JML features
    - editing, compilation, doc generation, etc.
  - code complexity and style checking
  - partial BML support
    - no editing of BML or bytecode
  - Mobius VC back-end
  - Mobius Prover back-end
  - interactive proof support for Coq

Mobius PVE: Next Steps

- next version to integrate the subsystems:
  - full BML support
  - Universe type inference
  - FreeBoogie and the Race Condition Checker
  - user feedback of proof state in JML/Java
  - proof status and unit/smoke testing
  - Mobius VC generator (in Coq)
  - interactive proof support for PVS
  - Coq PCC certificate generation
  - basic help system and process management

Use in Industry

- JavaCard
  - subset of a superset of Java for programming smart cards
    - the subset: no floats, no threads, limited API, optional GC
    - the superset: support for allocation in EEPROM or RAM, transactions
  - ideal target for formal methods

Use in Industry

- MIDP (Mobile Information Device Profile)
  - subset of a superset of Java for programming small devices
    - primarily mobiles and PDAs
    - includes networking capability, limited concurrency, small API, persistent store
    - is the primary target of the Mobius PVE

Use in Industry

- isolated cases in specific settings
- mostly independent from JML team
- sometimes the result of interactions with members of the JML team (e.g., consulting)
- most popular uses:
  - runtime checking of preconditions
  - static checking to eliminate NPEs

Use in Academia

- JML's use in undergraduate and graduate instruction
  - dozens of universities use JML and these tools for many different kinds of courses
    - introductory programming, programing languages, software engineering, problem- and project-based learning, applied formal methods, semantics, etc.
  - primary tools in use are the core JML tool suite and ESC/Java2

Use in The Netherlands

- OOTI course at Eindhoven [jointly with Oostdijk, Hubbers, and Poll]
  - graduate course for mature international students, some with industry experience
  - focused on applied use of JML and its tools and model checking of protocols
  - students focused one of a set of JavaCard applets (e.g., electronic purse)
  - applets installed and run on real cards

Use in Ireland

- all of my courses include the use of JML
  - first year students learn to *read* (basic) JML and work in independent small teams on a small project, identical across all teams
  - second year students learn to *write* (basic) JML and work on a full-class project in medium-sized teams
  - third year students use JML and other techniques and tools as necessary on projects of their own design

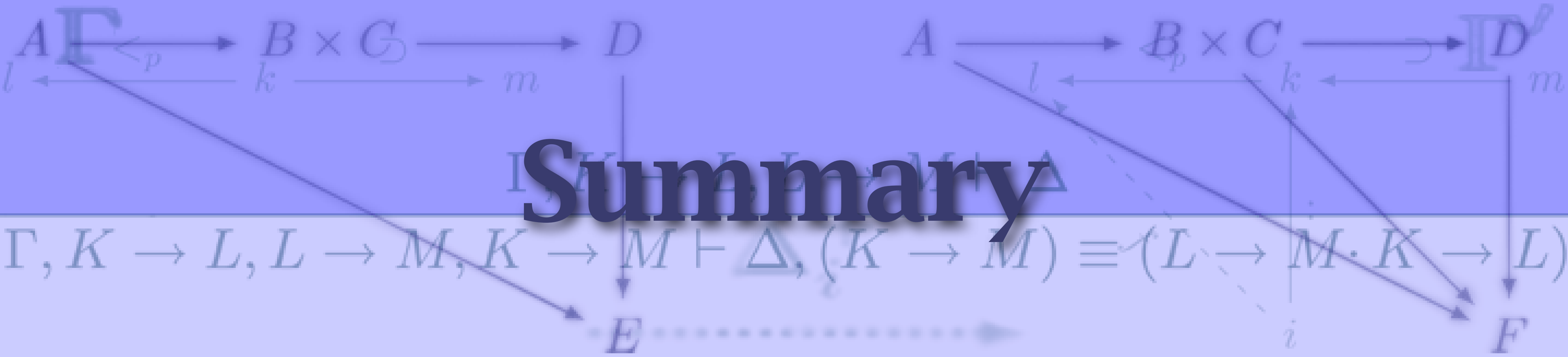
Research Challenges

- challenges of OO program verification
  - aliasing, callbacks, open systems, concurrency, modular & sound reasoning
- (deeper) theoretical and tool integration
  - several semantics of VM, Java, and JML
- incorporation of leading languages, frameworks, tools, and theory
  - e.g., B, VDM, Z, CSP, etc.

Mobius

Opportunities

- large, friendly, open community
  - finding collaborators is easy
- wide use in academia and growing use in industry
  - course materials available for reuse
  - if you build a good tool, they will come
  - many open and interesting problems



Summary

- assertion-based languages are a promising way to leverage applied formal methods
  - familiar syntax and semantics
  - no need for a formal model for users
  - easy to introduce and use incrementally
  - JML as a de facto standard and a vehicle for research
  - join us! http://www.jmlspecs.org/

Tools for JML

- tools for reading and writing specifications
- tools for generating specifications
- tools for checking an implementation against specifications

Tools for Reading and Writing Specifications

- parsing and typechecking (primarily as a part of other tools)
  - two best tools for these purposes are ESC/Java2 and the JML typechecker (“jml”)
- jmldoc: javadoc for JML
  - extends Javadoc to include specs in generated documentation

Tools for Reading and Writing Specifications

- the JML Eclipse plugin
  [KSU, David Cok, Nijmegen, UCD]
- the ESC/Java2 Eclipse plugin
  [Cok, Nijmegen, Warsaw, UCD]
- the ESC/Java2 specification consistency checker [Kiniry et al, UCD]
  - detect when specifications are unsound and identify problematic specs

Tools for Generating Specifications

- invariant detection using Daikon [Michael Ernst, MIT]
  - Daikon observes execution of code to detect likely invariants
- specification guessing using Houdini [Rustan Leino, DEC SRC then Poll et al, Nijmegen]
  - Houdini statically analyzes program structure to guess likely specifications

Tools for Generating Specifications

- loop invariant derivation using ESC/Java [Cormac Flanagan et al, SRC then Kiniry et al, UCD]

  ESC/Java2 uses heuristics, abstract interpretations, and wp calculi to guess then check loop invariants and variants

Tools for Checking Specifications

- the Runtime Assertion Checker (RAC) [Gary Leavens et al, Iowa St. and many others]

  tests if specs are violated at runtime

  - not terrifically exciting to academia, but very appealing to industry
  - well-specified code is easy to test
    - RAC handles \old, \forall, \exists, etc.

Tools for Checking Specifications

- jmlunit (JML and jUnit combined)
  [Gary Leavens et al, Iowa St.
  and many others]

  use specifications as test oracles

  - automatically generate and compose a large number of unit tests into a test suite
  - again, useful for students and industry
  - no more hand-written unit tests

Tools for Checking Specifications

- the Extended Static Checker for Java (ESC/Java) [Rustan Leino et al, SRC]

  automatic verification of simple properties

  - not sound or complete, but finds many bugs quickly
  - e.g., can statically “prove” the absence of many kinds of runtime exceptions

Tools for Checking Specifications

- the Chase tool [Nestor Cataño, INRIA]

  frame condition checker

  - remedies one important source of unsoundness in static checking

- the Race-Condition Checker (RCC) [Flanagan et al, SRC then Kiniry et al, UCD]

  statically detects field race conditions in concurrent Java programs

Mobius

Tools for Checking Specifications

- ESC/Java2
  [Cok and Kiniry et al, Nijmegen then UCD]
  builds upon ESC/Java in *many* ways to improve static checking capabilities
  - reasons about all of core JML
  - frame condition checking
  - soundness and completeness warnings
  - specification-aware dead code detection
  - support for new provers and logics

Mobius

Tools for Checking Specifications

- “real” program verification
  - JACK tool [Lilian Burdy, Gemplus then many others at INRIA]
    - inspired by ESC/Java, integrated in Eclipse
    - targets several provers, but mainly supports the B prover and Simplify
  - LOOP tool [Bart Jacobs et al, Nijmegen]
    - automatic and interactive (A&I) program verification in PVS

Tools for Checking Specifications

- the KeY tool [Chalmers and Karlsruhe]
  - A&I program verification using a CASE tool and a dynamic logic prover
- Krakatoa tool [INRIA/Orsay]
  - A&I program verification in Coq using Why
- the Bandera, Bogor, and Kiasan tools [John Hatcliff and Robby, KSU]
  - model checking and symbolic execution

Tools for Checking Specifications

- there is a large range of tools offering differing levels of assurance at different costs (i.e., time and effort)
  - runtime assertion checking
  - specification-based unit testing
  - extended static checking
  - automated and interactive full program verification

Mobius