



# Refinement of Abstract Specifications to Java Code

**Kurt Stenzel, Holger Grandy**

University of Augsburg

# Challenges in Java Program Verification



1. **What** to verify?
2. **How** to do it (efficiently)?

In this talk: **What to verify?**

1. Single Algorithms
2. Abstract data types
3. Applications implementing a communication protocol  
(cryptographic, E-Commerce, ...)



# Single Algorithms



```
static void arrayCopy(byte[] src, short srcOff,
    byte[] dest, short destOff, short length) {
    if (srcOff < destOff) {
        for(short i=(short)(length-1); i>=0; i--)
            dest[destOff+i] = src[srcOff+i];
    } else {
        for(short i=0; i<length; i++)
            dest[destOff+i] = src[srcOff+i]; }}
```

**Prove:**

*some preconditions...*,  $st = st_0 \vdash$

$\langle st; \text{arrayCopy}(r, sho, r_0, sho_0, sho_1); \rangle$

$st = \text{putByteArray}(\text{getByteArray}(r, s2i(sho), s2i(sho_1), st_0),$   
 $r_0, s2i(sho_0), st_0)$

# Abstract Data Types



```
/** Immutable arbitrary-precision integers. All
    operations behave as if BigIntegers were represented
    in two's-complement notation. */
public class BigInteger {

    /** The signum of this BigInteger: -1 for negative,
        0 for zero, or 1 for positive. */
    int signum;

    /** The magnitude of this BigInteger, in big-endian
        order */
    int[] mag;

    ...
}
```

# Java's BigInteger Class



- With MutableBigInteger, SignedMutableBigInteger, BitSieve: 4400 lines (+ comments), 2300 without
- 157 methods/constructors
- really nasty, optimized-for-speed algorithms
- support for RSA algorithms: generation of big primes, `x.modPow(y, m)`: computes  $x^y \bmod m$
- Probably every Java RSA implementation uses modPow: 87 methods, 1200 lines of code
- comment for an auxiliary method:  
“The algorithm is described in an unpublished manuscript . . . .”

# Properties of modPow



getInt(x, st) : number represented by BigInteger x

getInt(x, st) > 0, getInt(y, st) > 0, getInt(m, st) > 0, *more ...*

⊢ ⟨st; r = x.modPow(y, m); ⟩

getInt(r, st) = (getInt(x, st) ^ getInt(y, st)) % getInt(m, st)

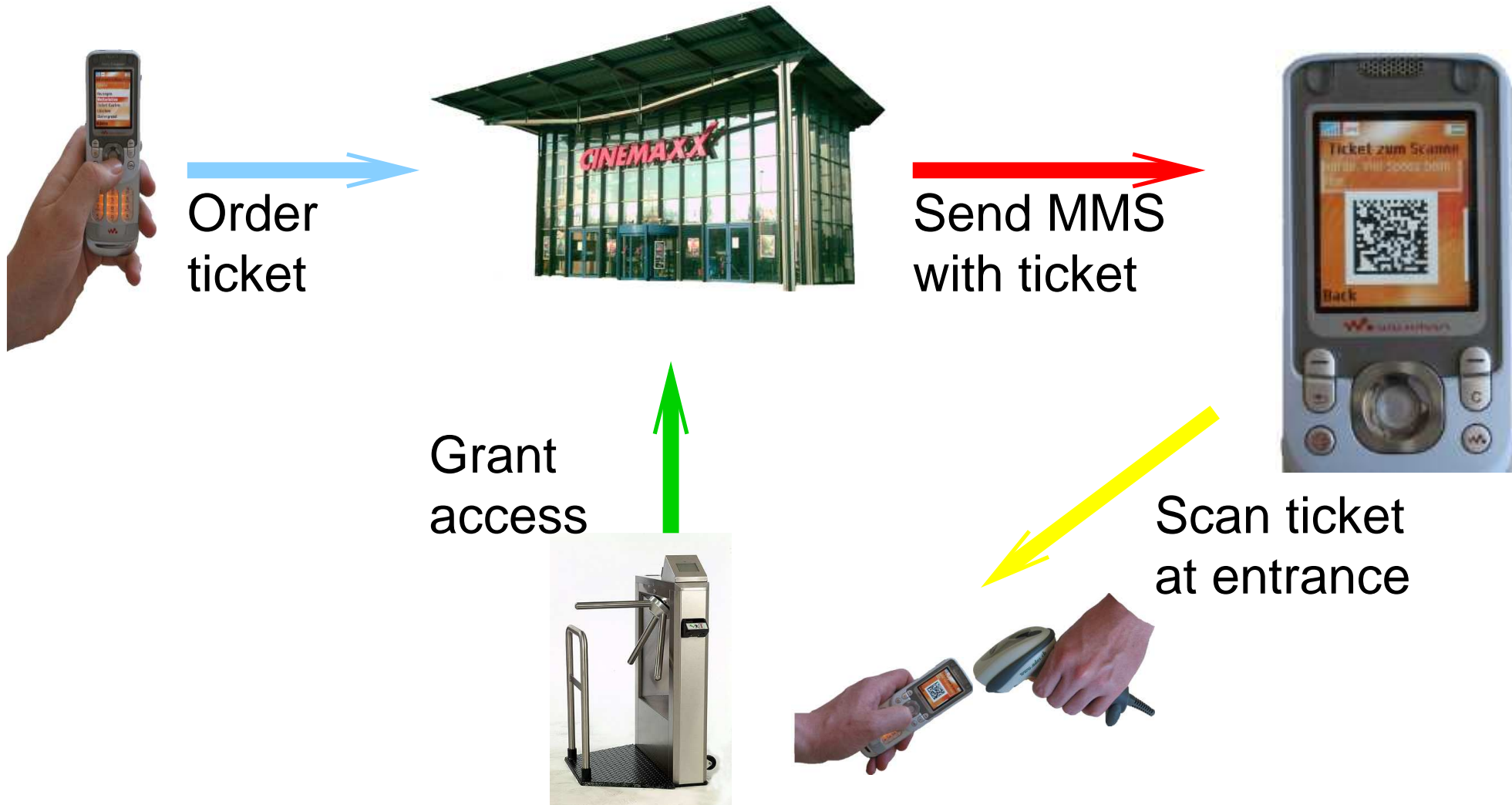
Algebraic specification of getInt: getInt : reference × store → int

getInt(x, st) = st[x.signum] \* ints2uint(getintarray(st[x.mag], st))

ints2unit(ints) = bits2uint(ints2bits(ints))

ints2bits([]) = [], ints2bits(i + ints) = fix(32, int2bits(i)) + ints2bits(ints)

# The Cindy Application



# The Phone Program



- Java code (J2ME)
- main protocol: 220 lines of code ( $\Rightarrow$  verify)
- abstract data types (**Document**): 900 lines ( $\Rightarrow$  verify)
- data encoding: 500 lines ( $\Rightarrow$  verify)
- GUI/MMS/storage interface: 3000 lines (do not verify)
- runs on Sony Ericsson W550i and Nokia 3250

**What should this program do?**

**It should implement the communication protocol correctly!**



# The Phone Program



```
public void step() {
    if(comm.available(USERPORT)) {
        Document inmsg = comm.receive(USERPORT);
        userStep(inmsg); }
    else if(comm.available(PHONEPORT)) {
        Document inmsg = comm.receive(PHONEPORT);
        phoneStep(inmsg); } }
private void userStep(Document indoc) {
    indoc = indoc.getPart(2);
    //PRESENT
    int index = getPresentIndex(indoc);
    if (0 <= index) {
        comm.send(nonce(tickets.getPart(index+1)), CINEMA,
            DISPLAYPORT, DIRECTSEND);
    }
    return; } //PASSON ...
```

# The KIV Approach



1. Model E-Commerce application with UML
2. Model protocols, and attacker
3. Use generic *documents* for exchanged messages
4. Transform into Abstract State Machine (ASM)
5. Prove security properties (e.g. valid ticket allows access, no double entry possible)
6. Refine some agents (in this case the phone) by a [Java program](#)
7. Prove correctness of refinement

# Refinement



ABSTRACT LEVEL

AOP(non-refined-agent)

AOP(java-agent)

AINIT

AFIN

R

R

R

R

R

R

new Protocol()

CINIT

CFIN

COP(non-refined-agent)

TOSTORE

FROMSTORE

CONCRETE LEVEL

step()

COP(java-agent)

# Documents



algebraic specification:

```
document = encdoc(key, document) | noncedoc (nonce)
          | doclist(documentlist) | ...
```

Java:

```
abstract class Document { ... }
class EncDoc extends Document { private byte[] encrypted; ... }
class NonceDoc extends Document { private Nonce nonce; ... }
class Doclist extends Document { private Document[] docs; ... }
...

```

Correspondence: getDoc (similar to getInt), addDoc



# From abstract state to Java store



- abstract agents have an (abstract) state:  
phone: list of received tickets (a documentlist)
- Java implementation: `Field private Doclist tickets;`
- Refinement relation R defines correspondence:  
 $R \equiv \text{phone.tickets} = \text{getDoc}(\text{st}[\text{Protocol.tickets}], \text{st})$   
 $\wedge \dots \textit{more refinement relations} \dots$
- Cryptography: Axiomatize Java's **doFinal**:  
*... preconditions ...*  
 $\vdash \langle \text{st}; \text{res} = \text{cip.doFinal}(\text{input}); \rangle \text{getByteArray}(\text{res}, \text{st}) =$   
 $\text{encrypt}(\text{getKey}(\text{cip}, \text{st}), \text{getByteArray}(\text{input}, \text{st}))$





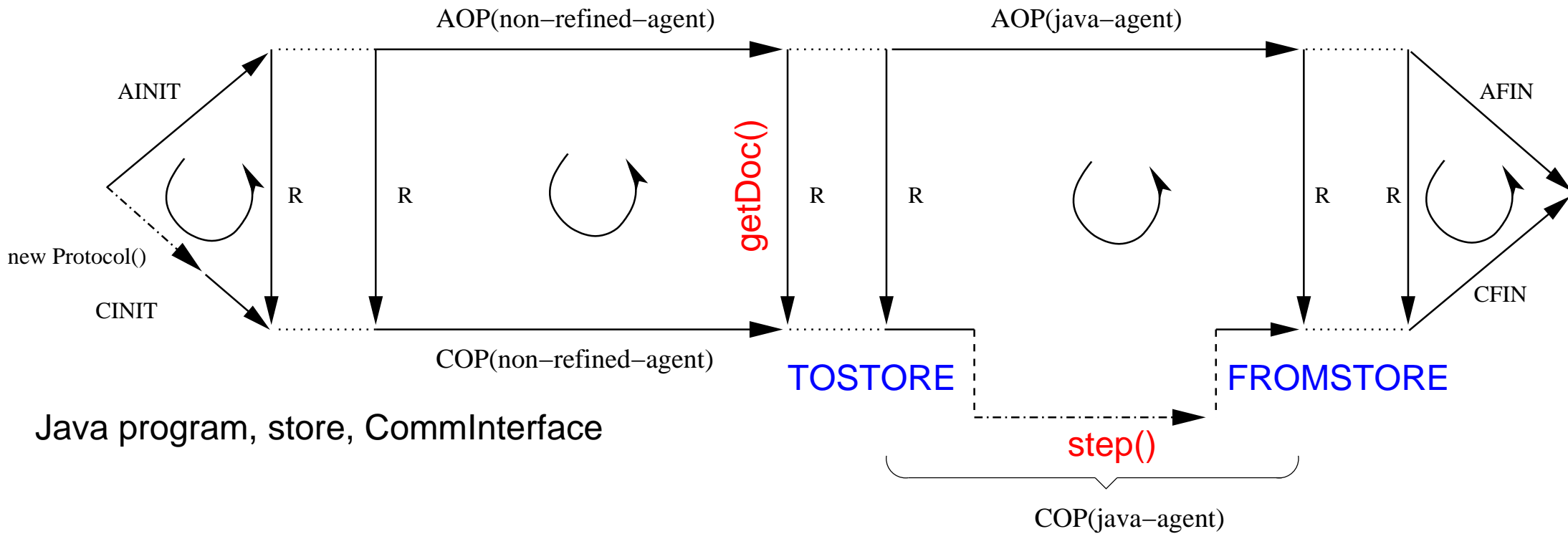
- abstract level: Sender 'puts' *doc* into mailbox of receiver.  
⇒ TOSTORE: place *doc* into store:  $st' \times r = \text{addDoc}(\text{doc}, st)$
- Java program: method `CommInterface.receive(port)` returns *r* (reference to *doc*)
- Communication interface:  
convert documents to something suitable for sending (ASN.1 encoding, XML, Java's serialize, ...)  
... and do the real sending ...  
(create MMS and send via J2ME API methods)
- FROMSTORE: retrieve document sent by Java with `send(...)`



# Refinement



algebraic specifications, agents, documents



# Results



Java program correct w.r.t. to protocol

- ⇒ security properties hold (agents viewed as black boxes, no malicious code, access only through communication interface)
- ⇒ seamless formal treatment from abstract protocol spec to Java program using established refinement concepts

Modular approach: application/documents/encoding

Complications: Cryptography + attacker can send junk

**work in progress . . .**



# Conclusion



- One challenge in Java program verification: What to prove?
- single algorithms ✓, implementation of abstract data types (✓),
- to do: (E-Commerce) applications with cryptography, input/output
- KIV approach: ASM refinement, algebraic specifications for mapping ADT/Java store
- Some case studies done, several under way

