

# Verifying JAVA CARD Programs

API Calls vs. Language Semantics

Official Specs vs. Reality

Wojciech Mostowski

woj@cs.ru.nl

Radboud University Nijmegen, the Netherlands

# The Setting

## Smart Cards

Small computers with very constrained resources

## JAVA CARD

A technology allowing to program smart cards with a **subset** of JAVA language

## Technological Race...

From the point of view of the programmer/formal verification seems to be somewhat slower than in the big world but this is about to change dramatically some time in near future

(When it comes to hardware/silicon race it seems it is as crazy as everything else...)

# An Example

```
public class TryCounter {  
  
    private byte tryCounter;  
    private byte max;  
  
    public TryCounter(byte max) {  
        this.max = max;  
        tryCounter = max;  
    }  
  
    private void setValue(byte value) throws IllegalArgumentException {  
        if(value < 0 || value > max)  
            throw new IllegalArgumentException();  
        tryCounter = value;  
    }  
  
    public void decrease() throws IllegalArgumentException {  
        if(tryCounter > 0)  
            setValue((byte)(tryCounter - 1));  
    }  
  
    public void max() throws IllegalArgumentException { setValue(max); }  
  
    public byte getTryCounter() { return tryCounter; }  
}
```

# An Example

```
public class TryCounter {  
  
    private byte tryCounter;  
    private byte max;  
  
    public TryCounter(byte max) {  
        this.max = max;  
        tryCounter = max;  
    }  
  
    private void setValue(byte value) throws IllegalArgumentException {  
        if(value < 0 || value > max)  
            throw new IllegalArgumentException();  
        tryCounter = value;  
    }  
  
    public void decrease() throws IllegalArgumentException {  
        if(tryCounter > 0)  
            setValue((byte)(tryCounter - 1));  
    }  
  
    public void max() throws IllegalArgumentException { setValue(max); }  
  
    public byte getTryCounter() { return tryCounter; }  
}
```

# An Example

```
public class TryCounter {  
  
    private byte tryCounter;  
    private byte max;  
  
    public TryCounter(byte max) {  
        this.max = max;  
        tryCounter = max;  
    }  
  
    private void setValue(byte value) throws IllegalArgumentException {  
        if(value < 0 || value > max)  
            throw new IllegalArgumentException();  
        tryCounter = value;  
    }  
  
    public void decrease() throws IllegalArgumentException {  
        if(tryCounter > 0)  
            setValue((byte)(tryCounter - 1));  
    }  
  
    public void max() throws IllegalArgumentException { setValue(max); }  
  
    public byte getTryCounter() { return tryCounter; }  
}
```

# Verification of the Example

- ▶ Take some common JML behavioural + exceptional specification
- ▶ The correctness should be provable even by hand and probably by all the verification tools on the market:
  - ▶ full behavioural specification
  - ▶ absence of exceptions for strong enough preconditions
  - ▶ in general – piece of cake. . .

# Verification of the Example

- ▶ Take some common JML behavioural + exceptional specification
- ▶ The correctness should be provable even by hand and probably by all the verification tools on the market:
  - ▶ full behavioural specification
  - ▶ absence of exceptions for strong enough preconditions
  - ▶ in general – piece of cake. . .

# Approaching Reality

- ▶ Can we use this try counter to e.g. implement a JAVA CARD library PIN object?

- ▶ Let's see what the JAVA CARD specification says:

*Even if a transaction is in progress, update of internal state [of the PIN object] – the try counter, the validated flag, and the blocking state, shall not participate in the transaction.*

- ▶ So, does our try counter adhere to this requirement? Probably not, have to check what those transactions are first...
- ▶ (Side effect: PhD topic)

# Approaching Reality

- ▶ Can we use this try counter to e.g. implement a JAVA CARD library PIN object?
- ▶ Let's see what the JAVA CARD specification says:

*Even if a transaction is in progress, update of internal state [of the PIN object] – the try counter, the validated flag, and the blocking state, shall not participate in the transaction.*

- ▶ So, does our try counter adhere to this requirement? Probably not, have to check what those transactions are first...
- ▶ (Side effect: PhD topic)

# Approaching Reality

- ▶ Can we use this try counter to e.g. implement a JAVA CARD library PIN object?
- ▶ Let's see what the JAVA CARD specification says:

*Even if a transaction is in progress, update of internal state [of the PIN object] – the try counter, the validated flag, and the blocking state, shall not participate in the transaction.*

- ▶ So, does our try counter adhere to this requirement? Probably not, have to check what those transactions are first. . .
- ▶ (Side effect: PhD topic)

# Approaching Reality

- ▶ Can we use this try counter to e.g. implement a JAVA CARD library PIN object?
- ▶ Let's see what the JAVA CARD specification says:

*Even if a transaction is in progress, update of internal state [of the PIN object] – the try counter, the validated flag, and the blocking state, shall not participate in the transaction.*

- ▶ So, does our try counter adhere to this requirement? Probably not, have to check what those transactions are first. . .
- ▶ (Side effect: PhD topic)

# Transactions

## JAVA CARD transaction mechanism

Looks very innocent:

- ▶ makes blocks of assignments (persistent memory) atomic w.r.t. unforeseen program interruptions (e.g. power loss)
- ▶ only three statements (static methods): `beginTransaction`, `commitTransaction`, `abortTransaction`
- ▶ the **abort** is normally triggered by the JAVA CARD VM, but can also be called explicitly

## Formalisation complications

- ▶ Two types of JAVA CARD memory: persistent & transient...
- ▶ Only persistent memory participates in transactions, transient is “transparent” w.r.t. transactions
- ▶ explicit abort: semi-abrupt termination – the program still continues execution in “somewhat slightly abrupt” state

# Transactions

## JAVA CARD transaction mechanism

Looks very innocent:

- ▶ makes blocks of assignments (persistent memory) atomic w.r.t. unforeseen program interruptions (e.g. power loss)
- ▶ only three statements (static methods): `beginTransaction`, `commitTransaction`, `abortTransaction`
- ▶ the **abort** is normally triggered by the JAVA CARD VM, but can also be called explicitly

## Formalisation complications

- ▶ Two types of JAVA CARD memory: persistent & transient...
- ▶ Only persistent memory participates in transactions, transient is “transparent” w.r.t. transactions
- ▶ explicit abort: semi-abrupt termination – the program still continues execution in “somewhat slightly abrupt” state

# Transactions

## JAVA CARD transaction mechanism

Looks very innocent:

- ▶ makes blocks of assignments (persistent memory) atomic w.r.t. unforeseen program interruptions (e.g. power loss)
- ▶ only three statements (static methods): `beginTransaction`, `commitTransaction`, `abortTransaction`
- ▶ the **abort** is normally triggered by the JAVA CARD VM, but can also be called explicitly

## Formalisation complications

- ▶ Two types of JAVA CARD memory: persistent & transient...
- ▶ Only persistent memory participates in transactions, transient is “transparent” w.r.t. transactions
- ▶ explicit abort: semi-abrupt termination – the program still continues execution in “somewhat slightly abrupt” state

# Semantics Hidden behind the API

Semantics of an assignment:

- ▶ `this.a = x;`  
*memory(this, a) = x* (choose your favourite formalisation...)
- ▶ `JCSYSTEM.beginTransaction();`  
`this.a = x;`  
??? not able to say until we know how the transaction ends...
- ▶ ends with a commit: *memory(this, a) = x*
- ▶ ends with an abort: *skip*;
- ▶ on the other hand assignments to transient locations (e.g. local variables) have always the same semantics (well, almost...)

Stepping across the `beginTransaction` in the proof destroys the whole “one small step at a time” program analysis paradigm

# Semantics Hidden behind the API

Semantics of an assignment:

- ▶ `this.a = x;`  
 $memory(this, a) = x$  (choose your favourite formalisation...)
- ▶ `JCSYSTEM.beginTransaction();`  
`this.a = x;`  
??? not able to say until we know how the transaction ends...
- ▶ ends with a commit:  $memory(this, a) = x$
- ▶ ends with an abort: *skip*;
- ▶ on the other hand assignments to transient locations (e.g. local variables) have always the same semantics (well, almost...)

Stepping across the `beginTransaction` in the proof destroys the whole “one small step at a time” program analysis paradigm

# Semantics Hidden behind the API

Semantics of an assignment:

- ▶ `this.a = x;`  
*memory(this, a) = x* (choose your favourite formalisation...)
- ▶ `JCSYSTEM.beginTransaction();`  
`this.a = x;`  
??? not able to say until we know how the transaction ends...
- ▶ ends with a commit: *memory(this, a) = x*
- ▶ ends with an abort: *skip*;
- ▶ on the other hand assignments to transient locations (e.g. local variables) have always the same semantics (well, almost...)

Stepping across the `beginTransaction` in the proof destroys the whole “one small step at a time” program analysis paradigm

# Semantics Hidden behind the API

Semantics of an assignment:

- ▶ `this.a = x;`  
 $memory(this, a) = x$  (choose your favourite formalisation...)
- ▶ `JCSYSTEM.beginTransaction();`  
`this.a = x;`  
??? not able to say until we know how the transaction ends...
- ▶ ends with a commit:  $memory(this, a) = x$
- ▶ ends with an abort: *skip*;
- ▶ on the other hand assignments to transient locations (e.g. local variables) have always the same semantics (well, almost...)

Stepping across the `beginTransaction` in the proof destroys the whole “one small step at a time” program analysis paradigm

# Semantics Hidden behind the API

Semantics of an assignment:

- ▶ `this.a = x;`  
 $memory(this, a) = x$  (choose your favourite formalisation...)
- ▶ `JCSYSTEM.beginTransaction();`  
`this.a = x;`  
??? not able to say until we know how the transaction ends...
- ▶ ends with a commit:  $memory(this, a) = x$
- ▶ ends with an abort: *skip*;
- ▶ on the other hand assignments to transient locations (e.g. local variables) have always the same semantics (well, almost...)

Stepping across the `beginTransaction` in the proof destroys the whole “one small step at a time” program analysis paradigm

# Semantics Hidden behind the API

Semantics of an assignment:

- ▶ `this.a = x;`  
 $memory(this, a) = x$  (choose your favourite formalisation...)
- ▶ `JCSYSTEM.beginTransaction();`  
`this.a = x;`  
??? not able to say until we know how the transaction ends...
- ▶ ends with a commit:  $memory(this, a) = x$
- ▶ ends with an abort: *skip*;
- ▶ on the other hand assignments to transient locations (e.g. local variables) have always the same semantics (well, almost...)

Stepping across the `beginTransaction` in the proof destroys the whole “one small step at a time” program analysis paradigm

## Where is the problem?

- ▶ Naively one would say that since `beginTransaction` is an API call it should be fully specifiable in e.g. JML
- ▶ But transaction methods are `native` – as native as it gets:
  - ▶ they do not change the state of the program at all
  - ▶ they do trigger a state change in the JVM (and the underlying operating system) that runs the program
  - ▶ and that affects the semantics of subsequent statements

NB: ideally, transactions should not be part of the API, a separate language construct should be devoted to this (like `synchronized`) – would make our life easier, but card/compiler developers' life difficult. . .

# Where is the problem?

- ▶ Naively one would say that since `beginTransaction` is an API call it should be fully specifiable in e.g. JML
- ▶ But transaction methods are **native** – as native as it gets:
  - ▶ they do not change the state of the program at all
  - ▶ they do trigger a state change in the JVM (and the underlying operating system) that runs the program
  - ▶ and that affects the semantics of subsequent statements

NB: ideally, transactions should not be part of the API, a separate language construct should be devoted to this (like `synchronized`) – would make our life easier, but card/compiler developers' life difficult. . .

## Where is the problem?

- ▶ Naively one would say that since `beginTransaction` is an API call it should be fully specifiable in e.g. JML
- ▶ But transaction methods are **native** – as native as it gets:
  - ▶ they do not change the state of the program at all
  - ▶ they do trigger a state change in the JVM (and the underlying operating system) that runs the program
  - ▶ and that affects the semantics of subsequent statements

NB: ideally, transactions should not be part of the API, a separate language construct should be devoted to this (like `synchronized`) – would make our life easier, but card/compiler developers' life difficult. . .

# Formalisation of Transactions

Several approaches possible to solve the problem:

- ▶ global program transformation for verification systems that allow it (which KeY does not)
- ▶ solutions that involve proof splitting (the transaction will eventually either commit or abort) and small logic “hackery”
- ▶ all in all the problem is solvable:
  - ▶ KeY
  - ▶ LOOP (not implemented though)
  - ▶ Krakatoa

# Formalisation of Transactions

Several approaches possible to solve the problem:

- ▶ global program transformation for verification systems that allow it (which KeY does not)
- ▶ solutions that involve proof splitting (the transaction will eventually either commit or abort) and small logic “hackery”
- ▶ all in all the problem is solvable:
  - ▶ KeY
  - ▶ LOOP (not implemented though)
  - ▶ Krakatoa

## How to Bypass a Transaction?

Now we know about transactions (i.e. we think we know), still... what about:

*Even if a transaction is in progress, update of internal state [of the PIN object] – the try counter, the validated flag, and the blocking state, shall not participate in the transaction.*

- ▶ Reading JAVA CARD VM specs – nothing really
- ▶ Reading JAVA CARD API specs – turns out later the information is there, but it is very well hidden
- ▶ The saviour: JAVA CARD API reference implementation from SUN
- ▶ There, the PIN try counter is decremented this way:

```
temps[0] = (byte)(triesLeft[0]-1);
Util.arrayCopyNonAtomic(temps, (short)0, triesLeft,
    (short)0, (short)1);
```
- ▶ Hmm...

## How to Bypass a Transaction?

Now we know about transactions (i.e. we think we know), still... what about:

*Even if a transaction is in progress, update of internal state [of the PIN object] – the try counter, the validated flag, and the blocking state, shall not participate in the transaction.*

- ▶ Reading JAVA CARD VM specs – nothing really
- ▶ Reading JAVA CARD API specs – turns out later the information is there, but it is very well hidden
- ▶ The saviour: JAVA CARD API reference implementation from SUN
- ▶ There, the PIN try counter is decremented this way:

```
temps[0] = (byte)(triesLeft[0]-1);
Util.arrayCopyNonAtomic(temps, (short)0, triesLeft,
    (short)0, (short)1);
```
- ▶ Hmm...

## How to Bypass a Transaction?

Now we know about transactions (i.e. we think we know), still... what about:

*Even if a transaction is in progress, update of internal state [of the PIN object] – the try counter, the validated flag, and the blocking state, shall not participate in the transaction.*

- ▶ Reading JAVA CARD VM specs – nothing really
- ▶ Reading JAVA CARD API specs – turns out later the information is there, but it is very well hidden
- ▶ The saviour: JAVA CARD API reference implementation from SUN
- ▶ There, the PIN try counter is decremented this way:

```
temps[0] = (byte)(triesLeft[0]-1);
Util.arrayCopyNonAtomic(temps, (short)0, triesLeft,
    (short)0, (short)1);
```
- ▶ Hmm...

## How to Bypass a Transaction?

Now we know about transactions (i.e. we think we know), still... what about:

*Even if a transaction is in progress, update of internal state [of the PIN object] – the try counter, the validated flag, and the blocking state, shall not participate in the transaction.*

- ▶ Reading JAVA CARD VM specs – nothing really
- ▶ Reading JAVA CARD API specs – turns out later the information is there, but it is very well hidden
- ▶ The saviour: JAVA CARD API reference implementation from SUN
- ▶ There, the PIN try counter is decremented this way:

```
temps[0] = (byte)(triesLeft[0]-1);
Util.arrayCopyNonAtomic(temps, (short)0, triesLeft,
    (short)0, (short)1);
```
- ▶ Hmm...

## How to Bypass a Transaction?

Now we know about transactions (i.e. we think we know), still... what about:

*Even if a transaction is in progress, update of internal state [of the PIN object] – the try counter, the validated flag, and the blocking state, shall not participate in the transaction.*

- ▶ Reading JAVA CARD VM specs – nothing really
- ▶ Reading JAVA CARD API specs – turns out later the information is there, but it is very well hidden
- ▶ The saviour: JAVA CARD API reference implementation from SUN
- ▶ There, the PIN try counter is decremented this way:

```
temps[0] = (byte)(triesLeft[0]-1);
Util.arrayCopyNonAtomic(temps, (short)0, triesLeft,
    (short)0, (short)1);
```
- ▶ Hmm...

## How to Bypass a Transaction?

Now we know about transactions (i.e. we think we know), still... what about:

*Even if a transaction is in progress, update of internal state [of the PIN object] – the try counter, the validated flag, and the blocking state, shall not participate in the transaction.*

- ▶ Reading JAVA CARD VM specs – nothing really
- ▶ Reading JAVA CARD API specs – turns out later the information is there, but it is very well hidden
- ▶ The saviour: JAVA CARD API reference implementation from SUN
- ▶ There, the PIN try counter is decremented this way:

```
temps[0] = (byte)(triesLeft[0]-1);
Util.arrayCopyNonAtomic(temps, (short)0, triesLeft,
    (short)0, (short)1);
```
- ▶ Hmm...

## Non-atomic Methods

```
static native short arrayCopyNonAtomic(  
    byte[] src, short srcOff,  
    byte[] dest, short destOff,  
    short length);
```

*This method **does not use the transaction facility** during the copy operation even if a transaction is in progress. Thus, this method is suitable for use only when the contents of the destination array can be left in a **partially modified state** in the event of a power loss in the middle of the copy operation.*

- ▶ Bypassing not mentioned explicitly
- ▶ Yet another special method
- ▶ The associated code is executed in some very mysterious way known only to the card's operating system

## Non-atomic Methods

```
static native short arrayCopyNonAtomic(  
    byte[] src, short srcOff,  
    byte[] dest, short destOff,  
    short length);
```

*This method **does not use the transaction facility** during the copy operation even if a transaction is in progress. Thus, this method is suitable for use only when the contents of the destination array can be left in a **partially modified state** in the event of a power loss in the middle of the copy operation.*

- ▶ Bypassing not mentioned explicitly
- ▶ Yet another special method
- ▶ The associated code is executed in some very mysterious way known only to the card's operating system

## New TryCounter Class

Let's assume for a minute that a non-atomic method is a correct way to bypass a transaction:

```
public class TryCounter {

    private byte[] temp;
    private byte[] tryCounter;

    public TryCounter(byte max) throws IllegalArgumentException {
        temp = JCSysyem.makeTransientByteArray((short)1, JCSysyem.CLEAR_ON_RESET);
        tryCounter = new byte[1];
        this.max = max;
        setValue(max);
    }

    private void setValue(byte value) throws IllegalArgumentException {
        if(value < 0 || value > max)
            throw new IllegalArgumentException();
        temp[0] = value;
        Util.arrayCopyNonAtomic(temp, (short)0, tryCounter, (short)0, (short)1);
    }

    public byte getTryCounter() { return tryCounter[0]; }
    ...
}
```

# Formal Verification of the TryCounter

- ▶ Let's assume there are some JML specs for this class. . .
- ▶ To verify the code we need to know the exact semantics of non-atomic method
- ▶ We already know some bits (transaction bypassing), but the specs are vague – perhaps there is more to non-atomic methods that initially thought
  - ▶ Run experiments on real cards and. . .
  - ▶ Really surprising results

When card tears occur during a non-atomic method call the memory can be left in an undefined state! But that depends on the actual card. . .

# Formal Verification of the TryCounter

- ▶ Let's assume there are some JML specs for this class. . .
- ▶ To verify the code we need to know the exact semantics of non-atomic method
- ▶ We already know some bits (transaction bypassing), but the specs are vague – perhaps there is more to non-atomic methods that initially thought
- ▶ Run experiments on real cards and. . .
  - ▶ Really surprising results

When card tears occur during a non-atomic method call the memory can be left in an undefined state! But that depends on the actual card. . .

# Formal Verification of the TryCounter

- ▶ Let's assume there are some JML specs for this class. . .
- ▶ To verify the code we need to know the exact semantics of non-atomic method
- ▶ We already know some bits (transaction bypassing), but the specs are vague – perhaps there is more to non-atomic methods that initially thought
- ▶ Run experiments on real cards and. . .
- ▶ Really surprising results

When card tears occur during a non-atomic method call the memory can be left in an undefined state! But that depends on the actual card. . .

# Formalising Non-Atomic Methods

- ▶ All cards behave differently, but we want to have one verification system for all cards
- ▶ Extract a common “non-tear” behaviour of non-atomic methods and formalise this
- ▶ Now possible to reason about JAVA CARD programs with transactions and non-atomic methods with KeY, but not in the presence of card tears during a non-atomic call

# Back to Reality

- ▶ Nevertheless, cards exhibit problems and are subject to (cheap) fault injection attacks:

```
private void setValue(byte value) throws IllegalArgumentException {  
    if(value < 0 || value > max)  
        throw new IllegalArgumentException();  
    temp[0] = value;  
    Util.arrayCopyNonAtomic(temp, (short)0, tryCounter,  
        (short)0, (short)1);  
}
```

- ▶ This leads to a very insecure implementation of a PIN object
- ▶ We can demonstrate this attack
- ▶ NB: we are talking about a reference implementation of PINs, not the ones that actually sit on the card, don't panic...

# Back to Reality

- ▶ Nevertheless, cards exhibit problems and are subject to (cheap) fault injection attacks:

```
private void setValue(byte value) throws IllegalArgumentException {  
    if(value < 0 || value > max)  
        throw new IllegalArgumentException();  
    temp[0] = value;  
    Util.arrayCopyNonAtomic(temp, (short)0, tryCounter,  
        (short)0, (short)1);  
}
```

- ▶ This leads to a very insecure implementation of a PIN object
- ▶ We can demonstrate this attack
- ▶ NB: we are talking about a reference implementation of PINs, not the ones that actually sit on the card, don't panic...

# Back to Reality

- ▶ Nevertheless, cards exhibit problems and are subject to (cheap) fault injection attacks:

```
private void setValue(byte value) throws IllegalArgumentException {  
    if(value < 0 || value > max)  
        throw new IllegalArgumentException();  
    temp[0] = value;  
    Util.arrayCopyNonAtomic(temp, (short)0, tryCounter,  
        (short)0, (short)1);  
}
```

- ▶ This leads to a very insecure implementation of a PIN object
- ▶ We can demonstrate this attack
- ▶ NB: we are talking about a reference implementation of PINs, not the ones that actually sit on the card, don't panic...

# Neutralising Faults

## Defensive Programming

Instead of one counter register use three: trace possible faults and repair them when they occur:

```
public class TryCounter {  
  
    private /*@spec_public@*/ byte[] _temp;  
    /*@ invariant _temp != null &&  
        _temp.length == 1 &&  
        JCSysyem.isTransient(_temp) == JCSysyem.CLEAR_ON_RESET;@*/  
  
    private /*@spec_public@*/ byte _max; /*@invariant _max > 0;@*/  
  
    private /*@spec_public@*/ byte[] _c1;  
    /*@ invariant _c1 != null &&  
        _c1.length == 1 &&  
        JCSysyem.isTransient(_c1) == JCSysyem.NOT_A_TRANSIENT_OBJECT;@*/  
  
    private /*@spec_public@*/ byte[] _c2;  
    /*@ invariant _c2 != null &&  
        _c2.length == 1 &&  
        JCSysyem.isTransient(_c2) == JCSysyem.NOT_A_TRANSIENT_OBJECT;@*/  
}
```

## Code cont'd

```
private /*@spec_public@*/ byte[] _c3;
/*@ invariant _c3 != null &&
    _c3.length == 1 &&
    JCSYSTEM.isTransient(_c3) == JCSYSTEM.NOT_A_TRANSIENT_OBJECT; @*/

/*@ invariant _c1[0] == _c2[0] && _c2[0] == _c3[0]; @*/
/*@ invariant _c1[0] >= 0 && _c1[0] <= _max; @*/

/*@ invariant _c1 != _c2 && _c2 != _c3 && _c1 != _c3
    && _temp != _c1 && _temp != _c2 && _temp != _c3; @*/

/*@ normal_behavior
    requires max > 0;
    ensures _max == max && _c1[0] == max &&
        _c2[0] == max && _c3[0] == max;
    assignable _c1, _c1[0], _c2, _c2[0], _c3, _c3[0], _max, _temp, _temp[0];
@*/
public TryCounter(byte max) {
    _temp = JCSYSTEM.makeTransientByteArray((short)1, JCSYSTEM.CLEAR_ON_RESET);
    _c1 = new byte[1]; _c2 = new byte[1]; _c3 = new byte[1];
    _max = max; setNA(_max);
}
```

## Code cont'd

```
/*@ normal_behavior
    requires value >= 0 && value <= _max;
    ensures _c1[0] == value && _c2[0] == value && _c3[0] == value;
    assignable _temp[0], _c1[0], _c2[0], _c3[0];
*/
private void setNA(byte value) {
    _temp[0] = value;
    Util.arrayCopyNonAtomic(_temp, (short)0, _c1, (short)0, (short)1);
    Util.arrayCopyNonAtomic(_temp, (short)0, _c2, (short)0, (short)1);
    Util.arrayCopyNonAtomic(_temp, (short)0, _c3, (short)0, (short)1);
}

/*@ normal_behavior
    requires true;
    ensures true;
    assignable _temp[0], _c1[0], _c2[0], _c3[0];
*/
private void restore() {
    if(_c1[0] == _c2[0] && _c2[0] == _c3[0]) {
        return;
    }
}
```

## Code cont'd

```
if(_c2[0] == _c3[0]) {
    _temp[0] = _c3[0];
    Util.arrayCopyNonAtomic(_temp, (short)0, _c1, (short)0, (short)1);
    return;
}

if(_c1[0] == _c2[0]) {
    _temp[0] = _c1[0];
    Util.arrayCopyNonAtomic(_temp, (short)0, _c3, (short)0, (short)1);
    return;
}

_temp[0] = _c3[0];
Util.arrayCopyNonAtomic(_temp, (short)0, _c2, (short)0, (short)1);
Util.arrayCopyNonAtomic(_temp, (short)0, _c1, (short)0, (short)1);
}
```

## Code cont'd

```
/*@ normal_behavior
    requires true;
    ensures \result == _c1[0];
    assignable _temp[0], _c1[0], _c2[0], _c3[0];
*/
public byte get() {
    restore();
    if (_c1[0] == _c2[0] && _c2[0] == _c3[0])
        return _c1[0];
    return (byte)0;
}

/*@ normal_behavior
    requires true;
    ensures (\old(_c1[0]) > 0 ==> _c1[0] == \old(_c1[0]) - 1) &&
           (\old(_c1[0]) == 0 ==> _c1[0] == 0);
    assignable _temp[0], _c1[0], _c2[0], _c3[0];
*/
public void decrease() {
    restore();
    if(_c1[0] == (byte)0) { return; }
    setNA((byte)(_c1[0]-1));
}
```

## Code cont'd

```
/*@ normal_behavior
    requires true;
    ensures _c1[0] == _max;
    assignable _temp[0], _c1[0], _c2[0], _c3[0];
    @*/
public void max() {
    restore();
    setNA(_max);
}
}
```

## Verifying TryCounter – Final Approach

Verification now substantially more difficult:

- ▶ (more complex code)
- ▶ need to look at the whole life of an object/class and not a single method call: possibly multiple/intermittent card tears during the life time of a try counter
- ▶ You shout: **that's what invariants are for!!!**
- ▶ Not here: faults explicitly break the invariant (whatever it is) and this is allowed! Recovery routines are responsible for sanitising the object state, not the (strong) invariant
- ▶ In other words: suitable invariant/assertion is probably possible, but really complicated
- ▶ Or rather: in this setting so called stable states of a class are not the ones usually understood (as e.g. in JML)

## Verifying TryCounter – Final Approach

Verification now substantially more difficult:

- ▶ (more complex code)
- ▶ need to look at the whole life of an object/class and not a single method call: possibly multiple/intermittent card tears during the life time of a try counter
- ▶ You shout: **that's what invariants are for!!!**
- ▶ Not here: faults explicitly break the invariant (whatever it is) and this is allowed! Recovery routines are responsible for sanitising the object state, not the (strong) invariant
- ▶ In other words: suitable invariant/assertion is probably possible, but really complicated
- ▶ Or rather: in this setting so called stable states of a class are not the ones usually understood (as e.g. in JML)

## Verifying TryCounter – Final Approach

Verification now substantially more difficult:

- ▶ (more complex code)
- ▶ need to look at the whole life of an object/class and not a single method call: possibly multiple/intermittent card tears during the life time of a try counter
- ▶ You shout: **that's what invariants are for!!!**
- ▶ Not here: faults explicitly break the invariant (whatever it is) and this is allowed! Recovery routines are responsible for sanitising the object state, not the (strong) invariant
- ▶ In other words: suitable invariant/assertion is probably possible, but really complicated
- ▶ Or rather: in this setting so called stable states of a class are not the ones usually understood (as e.g. in JML)

## Verifying TryCounter – Final Approach

Verification now substantially more difficult:

- ▶ (more complex code)
- ▶ need to look at the whole life of an object/class and not a single method call: possibly multiple/intermittent card tears during the life time of a try counter
- ▶ You shout: **that's what invariants are for!!!**
- ▶ Not here: faults explicitly break the invariant (whatever it is) and this is allowed! Recovery routines are responsible for sanitising the object state, not the (strong) invariant
- ▶ In other words: suitable invariant/assertion is probably possible, but really complicated
- ▶ Or rather: in this setting so called stable states of a class are not the ones usually understood (as e.g. in JML)

## Verifying TryCounter – Final Approach

Verification now substantially more difficult:

- ▶ (more complex code)
- ▶ need to look at the whole life of an object/class and not a single method call: possibly multiple/intermittent card tears during the life time of a try counter
- ▶ You shout: **that's what invariants are for!!!**
- ▶ Not here: faults explicitly break the invariant (whatever it is) and this is allowed! Recovery routines are responsible for sanitising the object state, not the (strong) invariant
- ▶ In other words: suitable invariant/assertion is probably possible, but really complicated
- ▶ Or rather: in this setting so called stable states of a class are not the ones usually understood (as e.g. in JML)

## Verifying TryCounter – Final Approach

Verification now substantially more difficult:

- ▶ (more complex code)
- ▶ need to look at the whole life of an object/class and not a single method call: possibly multiple/intermittent card tears during the life time of a try counter
- ▶ You shout: **that's what invariants are for!!!**
- ▶ Not here: faults explicitly break the invariant (whatever it is) and this is allowed! Recovery routines are responsible for sanitising the object state, not the (strong) invariant
- ▶ In other words: suitable invariant/assertion is probably possible, but really complicated
- ▶ Or rather: in this setting so called stable states of a class are not the ones usually understood (as e.g. in JML)

# Verifying TryCounter – Final Approach

Use model checking:

- ▶ model = life state of the try counter
- ▶ transitions = byte code instructions
- ▶ each instruction in the scope of a non-atomic method can cause a fault
- ▶ mark stable states in the model
- ▶ prove that in all stable states the try counter is in a well defined state (has either the old value or the new/asked for one, but nothing else)
- ▶ this implementation verifies easily (Uppaal)

# Wrap-up

It will get worse. . .

- ▶ This is **only** a try counter – abstraction of this is really simple
- ▶ The code is only resistant to one type of fault injection, other types of faults will blow up the code (more redundancy)
- ▶ This is only JAVA CARD, many people are trying to have a bite at J2ME/midlets. . .

## Conclusions

- ▶ Loose and hidden JAVA CARD semantics
- ▶ Implementation bugs
- ▶ “It’s a small platform” – don’t be fooled!
- ▶ Nevertheless, verification possible, with KeY and model checking

# Wrap-up

It will get worse. . .

- ▶ This is **only** a try counter – abstraction of this is really simple
- ▶ The code is only resistant to one type of fault injection, other types of faults will blow up the code (more redundancy)
- ▶ This is only JAVA CARD, many people are trying to have a bite at J2ME/midlets. . .

## Conclusions

- ▶ Loose and hidden JAVA CARD semantics
- ▶ Implementation bugs
- ▶ “It’s a small platform” – don’t be fooled!
- ▶ Nevertheless, verification possible, with KeY and model checking