Verification of Safety Properties in Presence of Transactions

Reiner Hähnle   Wojciech Mostowski

KeY Workshop 2004

Königswinter, June 2004

Overview

- Stripped down version of Reiner's talk at CASSIS'04
- Some repetition
- Demo (a.k.a. CASSIS demo)
- *Demoney* Case Study
- Design for Verification
- Lessons for KeY

Java Class Requirement Specification

- **Class Invariant**

  Restrict legal attribute values in each stable execution state

- **Method Contract**

  For initial states satisfying precondition,
  implementation must guarantee postcondition after execution

Java Class Requirement Specification

- Class Invariant

  Restrict legal attribute values in each stable execution state

- Method Contract

  For initial states satisfying precondition,
  implementation must guarantee postcondition after execution

Additional challenges in Java Card

- Incomplete termination (card rip-out)
- Intentional non-termination (controllers)

Require finer granularity than stable state semantics

Java Card Safety Properties

Safety

Nothing bad will happen during execution (eg, when card is ripped out)

Property (Example)

Sensitive data must be in consistent state at all times

Strong Invariant

Holds throughout program execution (in all intermediate states):
$[\![\cdot]\!]$ (throughout) modality

Transaction

Statements in scope of transaction executed completely or not at all

Throughout Modality

Semantics

$[\![p]\!]\,F$: F holds in all states during the execution of p

- including the initial and the final state
- excluding states while a transaction is in progress

Throughout Modality

Semantics

$[\![p]\!]F$: F holds in all states during the execution of p

- including the initial and the final state
- excluding states while a transaction is in progress

Remarks

- Related to always $\Box$ in temporal logic
- Program p may contain statements that form transactions
- Sequent calculus for $[\![\cdot]\!]$ (with Bernhard, KeY 2002 & FASE 2003)

Proof Obligations

Typical Proof Obligation involving throughout

In KeY attach strong invariant to classes

Let $TOut$ be strong invariant of C
Let Inv be (weak) invariant of C, Pre precondition of $C{::}m()$

Activating context-sensitive menu of method $C{::}m()$ in KeY

`(KeYExtension | Throughout Correctness | PreservesThroughout)`

Proof Obligations

Typical Proof Obligation involving throughout

In KeY attach strong invariant to classes

Let $TOut$ be strong invariant of C
Let Inv be (weak) invariant of C, Pre precondition of C::m()

Activating context-sensitive menu of method C::m() in KeY

`(KeYExtension | Throughout Correctness | PreservesThroughout)`

Starts proof of

$$(TOut \wedge Inv \wedge Pre) \rightarrow [\![m();]\!]\, TOut$$

The good old .key files

Observation

Java Card specifications are usually packed with ugly stuff:

- **low-level data types (byte arrays, arrays of byte arrays, etc.)**
- **going deep into Java Card API (JRE!), e.g. transaction depth**
- **lots of typing information (dynamic resource allocation)**

The good old .key files

Observation

Java Card specifications are usually packed with ugly stuff:

- **low-level data types (byte arrays, arrays of byte arrays, etc.)**
- **going deep into Java Card API (JRE!), e.g. transaction depth**
- **lots of typing information (dynamic resource allocation)**

Conclusion

OCL not so good. Java Card DL is a way to go:

- **enough (all) the expressive power**
- **no altering of the source code – *Post Mortem* verification**

Demo

- Can prove strong invariant with proper initialisation sequence
- Cannot prove strong invariant with buggy initialisation sequence

Demo

`key/myprojects/cassisdemo/LogRecord.java::setRecord()`

Transactions

Transaction mechanism

Allows the programmer to guarantee data consistency

`JCSystem.beginTransaction();`

Assignments to persistent locations (only) are done preliminarily

`JCSystem.commitTransaction();`

All preliminary assignments are finalised (in one atomic step)

`JCSystem.abortTransaction();`

All preliminary updates are forgotten

Transactions: Example

```
this.a = 0;
int i  = 0;
JCSystem.beginTransaction();
    this.a = 1;
    i      = this.a;
JCSystem.abortTransaction();
```

Final State: $\texttt{this.a} \doteq 0$

$\texttt{i} \doteq 1$

Transactions affect semantics of $\langle\cdot\rangle$, $[\cdot]$: influence final state

Demo

Demo

`key/myprojects/cassisdemo/Purse.java::processSale()`

Typical data consistency property:

`balance` in current log entry and `balance` in application are in sync

How Realistic is the Example?

Demoney:

Realistic Java Card purse (demo) application (Trusted Logic)

Our case study is similar to *Demoney* in several respects:

- **Stores transaction log records (*Demoney* Card Specification p. 17)**
- **Stipulates consistency of persistent data (p. 18)**

How Realistic is the Example?

Demoney:

Realistic Java Card purse (demo) application (Trusted Logic)

Our case study is similar to *Demoney* in several respects:

- **Stores transaction log records (*Demoney* Card Specification p. 17)**
- **Stipulates consistency of persistent data (p. 18)**

Major difference:

- **In Demoney, one log record is single array of bytes**

  For example, `short balance`: two fields within log record array

- **Log file is array of log records**

  Java Card does not allow 2-dimensional arrays, thus:

  ```
  Object[] logFile = new Object[logFileSize];
  ```

Design for Verification

Storage optimisation problematic for verification

Record type encoded into homogeneous array, consequences:

- **Comparison of values requires wrapping/unwrapping**
- **(Un)wrapping involves non-trivial Java modulo arithmetics**
- **Need to add explicit type assumptions for `Object`**

Design for verification

- **Represent data in object-oriented fashion, use type system**
- **Serialise objects only when necessary (for I/O)**
- **Decouple application from communication model**

  Loosely coupled design likely to enable decomposable verification

Demoney Verification

Difficult:

- (not our fault) due to the way it's designed and coded
- (our fault) some of seemingly simple specification parts are quite difficult to specify (syntax!) and occasionally impossible to prove with KeY (bugs)
- (our fault) parser limitations, remaining bugs, . . .

Demoney Verification

Difficult:

- **(not our fault) due to the way it's designed and coded**
- **(our fault) some of seemingly simple specification parts are quite difficult to specify (syntax!) and occasionally impossible to prove with KeY (bugs)**
- **(our fault) parser limitations, remaining bugs, . . .**

But. . .

Proved total correctness of two simple methods from *Demoney*

Problems Summarised

- Use of byte arrays (TLV standard) – different representations of the same data type, e.g. `balance` can be a short in one place and a pair of bytes in another
- no static type information, e.g. `Object[] logFile;`
- coding conventions, overuse of modulo operator:

```
currentRecord = (currentRecord + 1) % logFileSize ;
```

```
currentRecord++;
if(currentRecord == logFileSize) currentRecord = 0;
```

Specification Patterns

Data consistency is standard requirement

Now have to write

```
logFile.log.get(logFile.currentRecord).balance = balance
```

Specification Patterns

Data consistency is standard requirement

Now have to write

```
logFile.log.get(logFile.currentRecord).balance = balance
```

Instead would like to mark occurrences of balance in class diagram and say “prove data consistency”

Specification Patterns

Data consistency is standard requirement

Now have to write

```
logFile.log.get(logFile.currentRecord).balance = balance
```

Instead would like to mark occurrences of balance in class diagram and say “prove data consistency”

Develop and implement library of specification patterns

Good starting point (for security relevant properties):

R. Marlet & D. Le Metayer. Security Properties and Java Card Specificities to be Studied in the SecSafe Project, 2001. SECSAFE-TL-006

Performance

`setRecord` – 4 LoC

`processSale` – nested method calls to 5 classes, $<$30 LoC, transaction

	Time (sec)	Steps	Branches
$[\![$`setRecord`$]\!]$	2.0	234	20
$\langle$`setRecord`$\rangle$	1.5	129	6
$[\![$`processSale`$]\!]$	101.9	6861	329
$\langle$`keyNum2tag`$\rangle^{D}$	3.1	396	18
$\langle$`keyNum2keySet`$\rangle^{D,1}$	5.2	567	33

^{D} Methods from *Demoney* (full pre/post behavioural specification)

^{1} Hacks in KeY required (static instanceof evaluation, parser, etc.)

Summary

- **Safety properties of non-trivial Java Card programs verified automatically (!)**
- **Full Java Card coverage, but still small problems exist (bugs, the almighty parser, . . .)**
- **Speed could still be improved**
- **Loops require non-trivial interaction**

  But: most loops e.g. in *Demoney* used for initialisation
- **Design with verification in mind makes big difference**

  Design patterns for to-be-verified code
- **Specification patterns help to create formal requirements**
- **Mostly automatic verification of software like *Demoney* possible**