HALMSTAD UNIVERSITY

From Explicit to Implicit Dynamic Frames in Java Dynamic Logic and KeY

Wojciech Mostowski
Halmstad University

NWPT 2015, 21st October 2015

Overview

Projects

- VerCors:
  - Verification of Concurrent Data Structures
  - Permission-based Separation Logic for Java
  - JML with permissions on the specification layer
  - Automated tool support, Chalice/Silicon based
  - http://fmt.cs.utwente.nl/research/projects/VerCors/

Projects

- VerCors:
  - Verification of Concurrent Data Structures
  - Permission-based Separation Logic for Java
  - JML with permissions on the specification layer
  - Automated tool support, Chalice/Silicon based
  - `http://fmt.cs.utwente.nl/research/projects/VerCors/`

- KeY:
  - Deductive Verification of Object-Oriented Programs
  - Emphasis on Java, based on Dynamic Logic
  - Specification language JML with dynamic frames – JML*
  - Self-contained, automated interactive verifier
  - `http://www.key-project.org`

KeY

Projects

- VerCors:
  - Verification of Concurrent Data Structures
  - Permission-based Separation Logic for Java
  - JML with permissions on the specification layer
  - Automated tool support, Chalice/Silicon based
  - `http://fmt.cs.utwente.nl/research/projects/VerCors/`

- KeY:
  - Deductive Verification of Object-Oriented Programs
  - Emphasis on Java, based on Dynamic Logic
  - Specification language JML with dynamic frames – JML*
  - Self-contained, automated interactive verifier
  - `http://www.key-project.org`

- Both work with Design-by-Contract principles and (modified) JML
- Marriage of the two to enable interactive verification with permissions

Classical Permission-Based Reasoning

- Specifications provide permission annotations (fractions)
- Programs are verified (thread locally) w.r.t. these annotations

Classical Permission-Based Reasoning

- Specifications provide permission annotations (fractions)
- Programs are verified (thread locally) w.r.t. these annotations
- Each heap read access guarded by $p \leq 1$ (or 100%)
- Each heap write access guarded by $p = 1$

Classical Permission-Based Reasoning

- Specifications provide permission annotations (fractions)
- Programs are verified (thread locally) w.r.t. these annotations
- Each heap read access guarded by $p \leq 1$ (or 100%)
- Each heap write access guarded by $p = 1$
- Synchronisation:
  - Forking & locking
  - Permission transfers (produce/consume style)
- [Resource invariants]

Classical Permission-Based Reasoning

Example

```
class Counter {
  int c;

  //@ requires Perm(this.c, 1); ensures Perm(this.c, 1);
  void increase() { this.c++; }

  void use() { lock(); increase(); unlock(); }

  //@ requires true; ensures Perm(this.c, 1);
  native void lock();

  //@ requires Perm(this.c, 1); ensures true;
  native void unlock();
}
```

Explicit and Implicit Framing

- In Separation Logic-like reasoning framing is implicit:
  - Write permission indicates that a location might be changed
  - Read permission indicates that a location might be read
  - Both are very important for modular reasoning
  - Heap locations without permission are out of scope

Explicit and Implicit Framing

- In Separation Logic-like reasoning framing is implicit:
  - Write permission indicates that a location might be changed
  - Read permission indicates that a location might be read
  - Both are very important for modular reasoning
  - Heap locations without permission are out of scope
- JML* and Java Dynamic Logic are based on the original dynamic frames idea where framing is explicit:
  - Explicitly listed read and write frames (**`accessible`** & **`assignable`**)
  - Explicit heap (logic) variable
  - Changes specified in terms of old and new values (**`\old`**)
  - Frames can be abstract

Example

JML*

```
class Counter {
  int c; //@ model \locset fp = this.c;

  //@ ensures this.c == \old(this.c) + 1; assignable fp;
  void increase() { this.c++; }

  //@ ensures \result == this.c; accessible fp;
  int /*@ strictly_pure @*/ get() { return this.c; }
}
```

Example

JML*

```
class Counter {
  int c; //@ model \locset fp = this.c;

  //@ ensures this.c == \old(this.c) + 1; assignable fp;
  void increase() { this.c++; }

  //@ ensures \result == this.c; accessible fp;
  int /*@ strictly_pure @*/ get() { return this.c; }
}
```

Java Dynamic Logic

$$\forall_{o:Object,f:Field}\ (o,f) \in fp \vee o.f@\mathsf{heap} = o.f@\mathsf{heapAtPre} \qquad \text{(assignable)}$$

$$\mathsf{get}() = \{\mathsf{heap} := anon(\mathsf{heap}, allLocs \setminus fp, \mathsf{anonHeap})\}\mathsf{get}() \qquad \text{(accessible)}$$

Permissions in JML*

- Permission system that allows for the $new = modified\ old$ specification style
  - Symbolic permissions
  - Additional flexibility for complex permission flows

Permissions in JML*

1. Permission system that allows for the $new = modified\ old$ specification style
   - Symbolic permissions
   - Additional flexibility for complex permission flows
2. Second heap to store permissions
   - Parallel to the regular heap
   - Separate framing
   - Heaps named explicitly
   - Can be switched-off – sequential reasoning

Permissions in JML*

1. Permission system that allows for the $new = modified\ old$ specification style
   - Symbolic permissions
   - Additional flexibility for complex permission flows
2. Second heap to store permissions
   - Parallel to the regular heap
   - Separate framing
   - Heaps named explicitly
   - Can be switched-off – sequential reasoning
3. Method to show self-framing of specifications w.r.t. permissions
   - Self-framing is not automatic like in Separation Logic

Permissions in JML*

1. Permission system that allows for the $new = modified\ old$ specification style
   - Symbolic permissions
   - Additional flexibility for complex permission flows
2. Second heap to store permissions
   - Parallel to the regular heap
   - Separate framing
   - Heaps named explicitly
   - Can be switched-off – sequential reasoning
3. Method to show self-framing of specifications w.r.t. permissions
   - Self-framing is not automatic like in Separation Logic
4. Modular specifications with abstractions – synchronisation through Java API

Permissions in JML*

Example

```
public class ArrayList {
  Object[] cnt; int s; //@ model \locset fp = s, cnt, cnt[*];

  //@ requires \readPerm(\perm(s));
  //@ ensures \result == s;
  //@ accessible<heap> fp; accessible<permissions> \nothing;
  /*@ pure @*/ int size() { return s; }








}
```

Permissions in JML*

Example

```
public class ArrayList {
  Object[] cnt; int s; //@ model \locset fp = s, cnt, cnt[*];

  //@ requires \readPerm(\perm(s));
  //@ ensures \result == s;
  //@ accessible<heap> fp; accessible<permissions> \nothing;
  /*@ pure @*/ int size() { return s; }

  //@ requires \readPerm(\perm(cnt));
  //@ requires \writePerm(\perm(s)) && \writePerm(\perm(cnt[s]));
  //@ ensures size() == \old(size()) + 1;
  //@ assignable<heap> fp; assignable<permissions> \strictly_nothing;
  void add(Object o) { cnt[s++] = o; }
}
```

Specification Self-Framing

Sound

```
//@ requires \writePerm(\perm(this.f)); ensures this.f == v;
//@ assignable this.f; assignable<permissions> \nothing;
void setF(int v) { this.f = v; }
```

Specification Self-Framing

Sound

```
//@ requires \writePerm(\perm(this.f)); ensures this.f == v;
//@ assignable this.f; assignable<permissions> \nothing;
void setF(int v) { this.f = v; }
```

Unsound

```
//@ requires \writePerm(\perm(this.f));
//@ ensures this.f == v;
//@ assignable this.f; assignable<permissions> this.f;
void setFandUnlock(int v) { this.f = v; l.unlock(); }
```

Specification Self-Framing

Sound

```
//@ requires \writePerm(\perm(this.f)); ensures this.f == v;
//@ assignable this.f; assignable<permissions> \nothing;
void setF(int v) { this.f = v; }
```

Corrected

```
//@ requires \writePerm(\perm(this.f));
//@ ensures \readPerm(\perm(this.f)) && this.f == v;
//@ assignable this.f; assignable<permissions> this.f;
void setFandUnlock(int v) { this.f = v; l.unlock(); }
```

Specification Self-Framing

Sound

```
//@ requires \writePerm(\perm(this.f)); ensures this.f == v;
//@ assignable this.f; assignable<permissions> \nothing;
void setF(int v) { this.f = v; }
```

Corrected

```
//@ requires \writePerm(\perm(this.f));
//@ ensures \readPerm(\perm(this.f)) && this.f == v;
//@ assignable this.f; assignable<permissions> this.f;
void setFandUnlock(int v) { this.f = v; l.unlock(); }
```

Additional Proof Obligation in Java DL

Involves on-the-fly building of frame – Implicit Dynamic Frames

Removing Explicit Frames

- **`assignable`** & **`accessible`** clauses are redundant

Removing Explicit Frames

- **`assignable`** & **`accessible`** clauses are redundant
- Whatever the method reads or writes requires a permission in the precondition
- These permissions determine both frames and they are verified

Removing Explicit Frames

- **`assignable`** & **`accessible`** clauses are redundant
- Whatever the method reads or writes requires a permission in the precondition
- These permissions determine both frames and they are verified
- [Can also be easily over-approximated!]

Removing Explicit Frames

- **`assignable`** & **`accessible`** clauses are redundant
- Whatever the method reads or writes requires a permission in the precondition
- These permissions determine both frames and they are verified
- [Can also be easily over-approximated!]
- But, this works well for the regular heap, permission heap is usually untouched
- In particular, a write frame indicates that a location is possibly modified

Removing Explicit Frames

- **`assignable`** & **`accessible`** clauses are redundant
- Whatever the method reads or writes requires a permission in the precondition
- These permissions determine both frames and they are verified
- [Can also be easily over-approximated!]
- But, this works well for the regular heap, permission heap is usually untouched
- In particular, a write frame indicates that a location is possibly modified
- Imposing the permission-based frame on the permission heap means that corresponding permissions might be modified, in particular lost

Removing Explicit Frames

- **assignable** & **accessible** clauses are redundant
- Whatever the method reads or writes requires a permission in the precondition
- These permissions determine both frames and they are verified
- [Can also be easily over-approximated!]
- But, this works well for the regular heap, permission heap is usually untouched
- In particular, a write frame indicates that a location is possibly modified
- Imposing the permission-based frame on the permission heap means that corresponding permissions might be modified, in particular lost
- Not a problem with a dedicated explicit frame

```
assignable<permissions> \strictly_nothing;
```

Removing Explicit Frames

- **assignable** & **accessible** clauses are redundant
- Whatever the method reads or writes requires a permission in the precondition
- These permissions determine both frames and they are verified
- [Can also be easily over-approximated!]
- But, this works well for the regular heap, permission heap is usually untouched
- In particular, a write frame indicates that a location is possibly modified
- Imposing the permission-based frame on the permission heap means that corresponding permissions might be modified, in particular lost
- Not a problem with a dedicated explicit frame
  `assignable<permissions> \strictly_nothing;`
- Untouched permissions have to be repeated in postconditions
  (like in Separation Logic)
- New keyword – `\samePerm`

Repeating Permissions

Example

```
public class ArrayList {
  Object[] cnt; int s;

  //@ requires \readPerm(\perm(s));
  //@ ensures \result == s;
  //@ ensures \samePerm(\perm(s));
  /*@ pure @*/ int size() { return s; }







}
```

Repeating Permissions

Example

```
public class ArrayList {
  Object[] cnt; int s;

  //@ requires \readPerm(\perm(s));
  //@ ensures \result == s;
  //@ ensures \samePerm(\perm(s));
  /*@ pure @*/ int size() { return s; }

  //@ requires \readPerm(\perm(cnt));
  //@ requires \writePerm(\perm(s)) && \writePerm(\perm(cnt[s]));
  //@ ensures size() == \old(size()) + 1;
  //@ ensures \samePerm(\perm(cnt));
  //@ ensures \samePerm(\perm(s)) && \samePerm(\perm(cnt[s]));
  void add(Object o) { cnt[s++] = o; }
}
```

Dynamic Frame Construction

- Anonymisation (havocing) function to prove the **accessible** frame:

$$\texttt{get}() = \{\texttt{heap} := anon(\texttt{heap}, allLocs \setminus fp, \texttt{anonHeap})\}\,\texttt{get}()$$

Dynamic Frame Construction

- Anonymisation (havocing) function to prove the **accessible** frame:

$$\mathtt{get}() = \{\mathtt{heap} := anon(\mathtt{heap}, allLocs \setminus fp, \mathtt{anonHeap})\}\,\mathtt{get}()$$

- To prove self-framing – collect the frame from the specification:

$$\mathtt{pre} \land \forall_{o:Object, f:Field}\; readPerm(o.f@\mathtt{permissions}) \rightarrow (o,f) \in readLocs \\ \rightarrow \mathtt{pre} = \{\mathtt{heap} := anon(\mathtt{heap}, allLocs \setminus readLocs, \mathtt{anonHeap})\}\,\mathtt{pre}$$

Dynamic Frame Construction

- Anonymisation (havocing) function to prove the **accessible** frame:

$$\texttt{get}() = \{\texttt{heap} := \mathit{anon}(\texttt{heap}, \mathit{allLocs} \setminus \mathit{fp}, \texttt{anonHeap})\}\,\texttt{get}()$$

- To prove self-framing – collect the frame from the specification:

$$\texttt{pre} \wedge \forall_{o:Object, f:Field}\, \mathit{readPerm}(o.f@\texttt{permissions}) \rightarrow (o, f) \in \mathit{readLocs}$$
$$\rightarrow \texttt{pre} = \{\texttt{heap} := \mathit{anon}(\texttt{heap}, \mathit{allLocs} \setminus \mathit{readLocs}, \texttt{anonHeap})\}\,\texttt{pre}$$

Read frame is constructed on-the-fly!

Dynamic Frame Construction

- Anonymisation (havocing) function to prove the **accessible** frame:

$$\texttt{get}() = \{\texttt{heap} := anon(\texttt{heap}, allLocs \setminus fp, \texttt{anonHeap})\}\,\texttt{get}()$$

- To prove self-framing – collect the frame from the specification:

$$\begin{array}{l}\texttt{pre} \wedge \forall_{o:Object,f:Field}\, readPerm(o.f@\texttt{permissions}) \rightarrow (o,f) \in readLocs \\ \quad \rightarrow \texttt{pre} = \{\texttt{heap} := anon(\texttt{heap}, allLocs \setminus readLocs, \texttt{anonHeap})\}\,\texttt{pre}\end{array}$$

Read frame is constructed on-the-fly!

- A write frame is dynamically constructed with:

$$\texttt{pre} \wedge \forall_{o:Object,f:Field}\, writePerm(o.f@\texttt{permissions}) \rightarrow (o,f) \in writeLocs$$

To Separation Logic

- Very fine grained separation of heaps – single locations

To Separation Logic

- Very fine grained separation of heaps – single locations
- In practice needed only for whole footprints of expressions
  e.g. state of `obj1` does not interfere with state of `obj2`

To Separation Logic

- Very fine grained separation of heaps – single locations
- In practice needed only for whole footprints of expressions
  e.g. state of `obj1` does not interfere with state of `obj2`
- KeY and Java Dynamic Logic have facilities for that

To Separation Logic

- Very fine grained separation of heaps – single locations
- In practice needed only for whole footprints of expressions
  e.g. state of `obj1` does not interfere with state of `obj2`
- KeY and Java Dynamic Logic have facilities for that
- But treatment of magic wand operator **`-*`** unclear (yet)

Conclusions

- Work in progress (even the explicit solution not yet fully implemented)
- Not discussed – modular specifications for API-based synchronisation
  Scales up from the explicit frames solution
- KeY implementation very flexible, but going fully implicit is a big step
  Need to keep the implementation modular in this respect
- Unknown interactions with other KeY developments,
  e.g. information flow calculus & extension

Conclusions

- Work in progress (even the explicit solution not yet fully implemented)
- Not discussed – modular specifications for API-based synchronisation
  Scales up from the explicit frames solution
- KeY implementation very flexible, but going fully implicit is a big step
  Need to keep the implementation modular in this respect
- Unknown interactions with other KeY developments,
  e.g. information flow calculus & extension
- Not working yet, but can show explicit frames version working 🙂

The End

Thank You!