A Java Reference Model of Transacted Memory for Smart Cards

Erik Poll University of Nijmegen

Joint work with

Pieter Hartel University of Twente

Eduard de Jong Sun Microsystems

Overview

- Case study in specifying and testing (i.e. debugging) a piece of smart card OS software that provides transactions
- using the formal specification language JML
- using the runtime assertion checking tool for JML

Transactions

- **Possible power loss due to card tear at any moment**

Transactions

- **Possible power loss due to card tear at any moment**
- **Therefore: smartcard OS supports transactions, atomic writes consisting of several EEPROM writes**

Transactions

- **Possible power loss due to card tear at any moment**
- **Therefore: smartcard OS supports transactions, atomic writes consisting of several EEPROM writes**
- **On power-up: OS cleans up any unfinished transaction**

Transactions

- **Possible power loss due to card tear at any moment**
- **Therefore: smartcard OS supports transactions, atomic writes consisting of several EEPROM writes**
- **On power-up: OS cleans up any unfinished transaction**
- **This clean-up can again be interrupted by a card tear . . .**

Transacted Memory

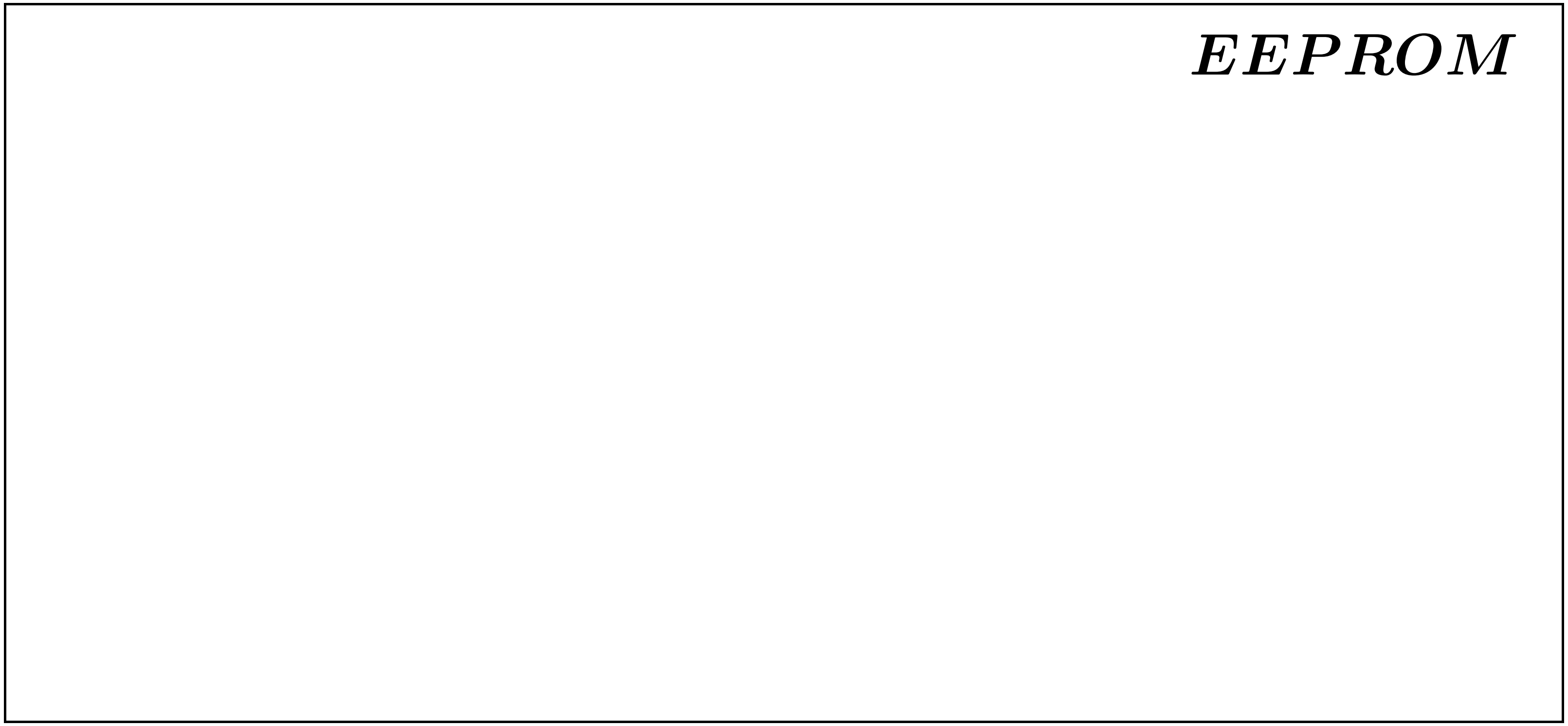
Implementation idea by Bos & de Jong:

```
Tag        NewTag(length)
InfoSeq    Read(tag)
void       Write(tag, infoSeq)
void       Commit(tag)
void       Tidy()
```

NB not as implemented in the current JavaCard API.

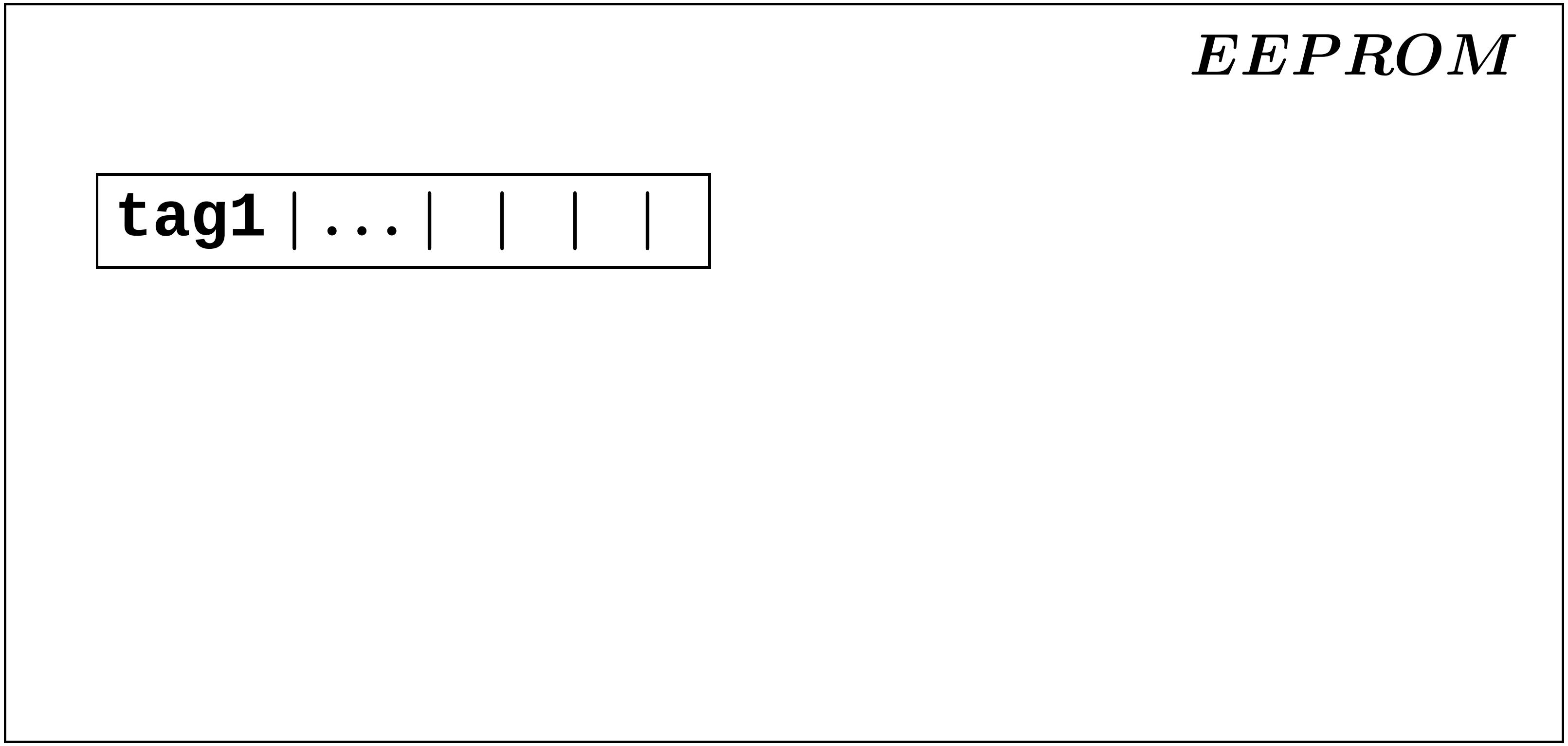
Provides multiple, concurrent, transactions and logging.

Transacted Memory



$EEPROM$

Transacted Memory

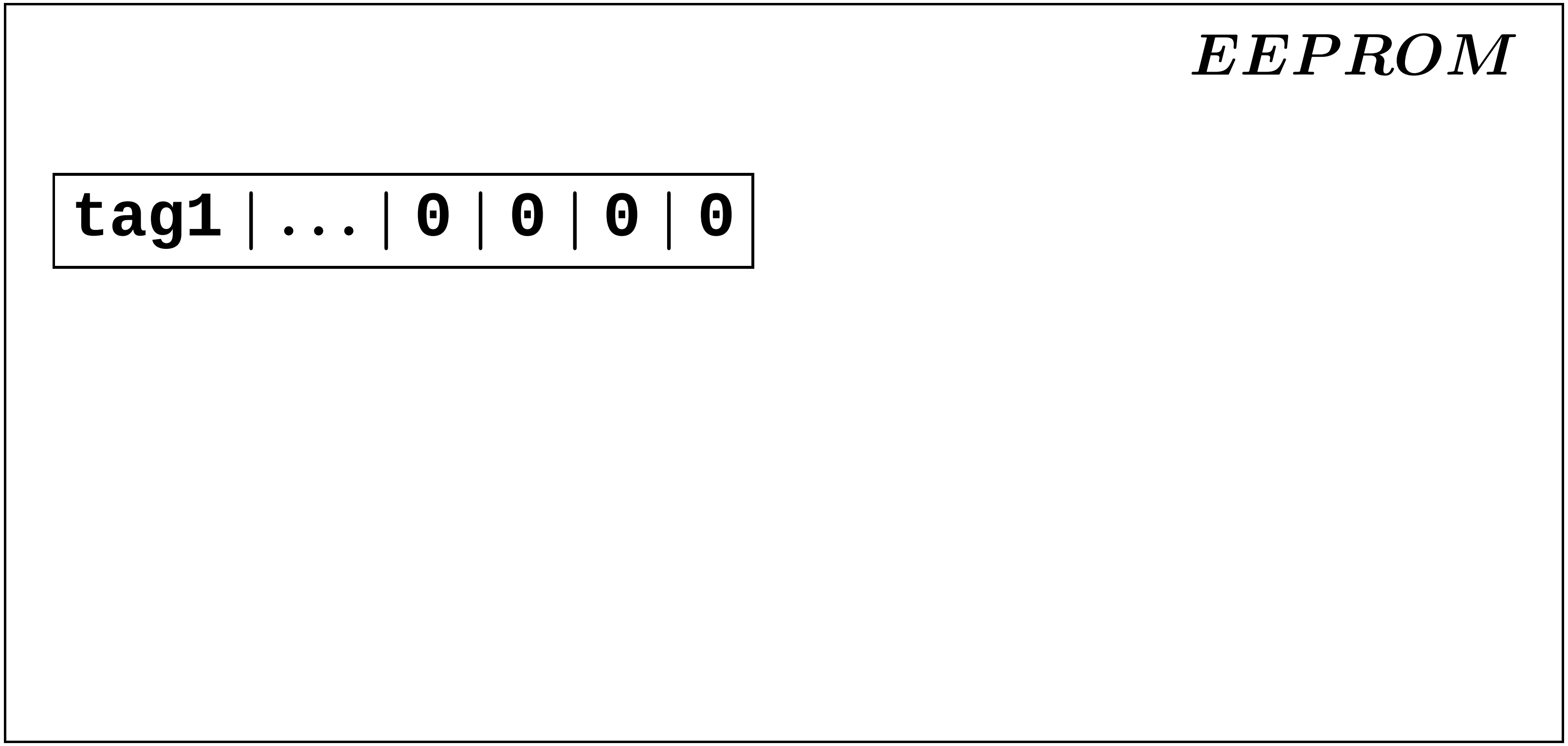


EEPROM

tag1 | ... | | | |

NewTag(4) returns tag1 with length 4

Transacted Memory

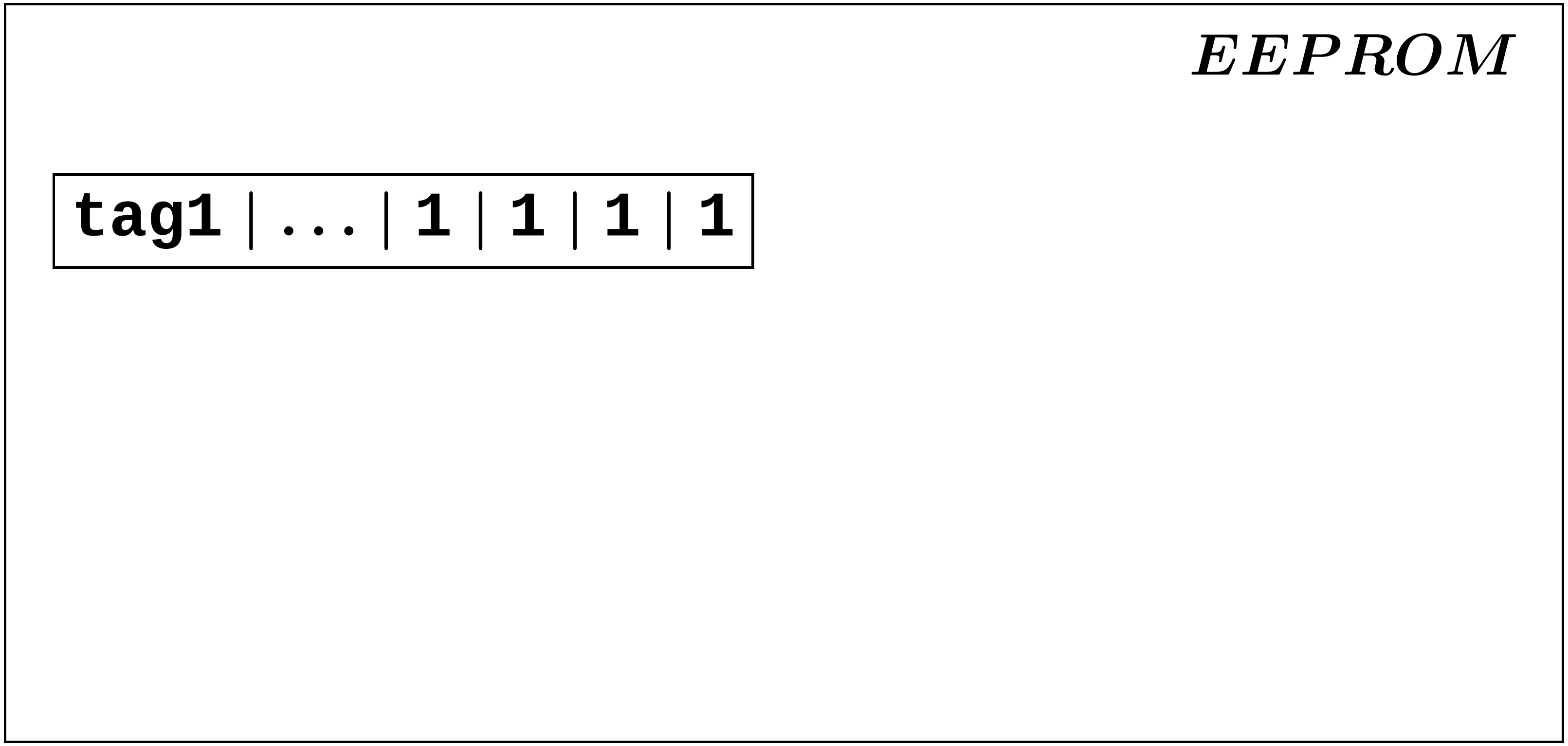


EEPROM

tag1	...	0	0	0	0

`Write(tag1,[0,0,0,0])` possibly in several EEPROM writes

Transacted Memory

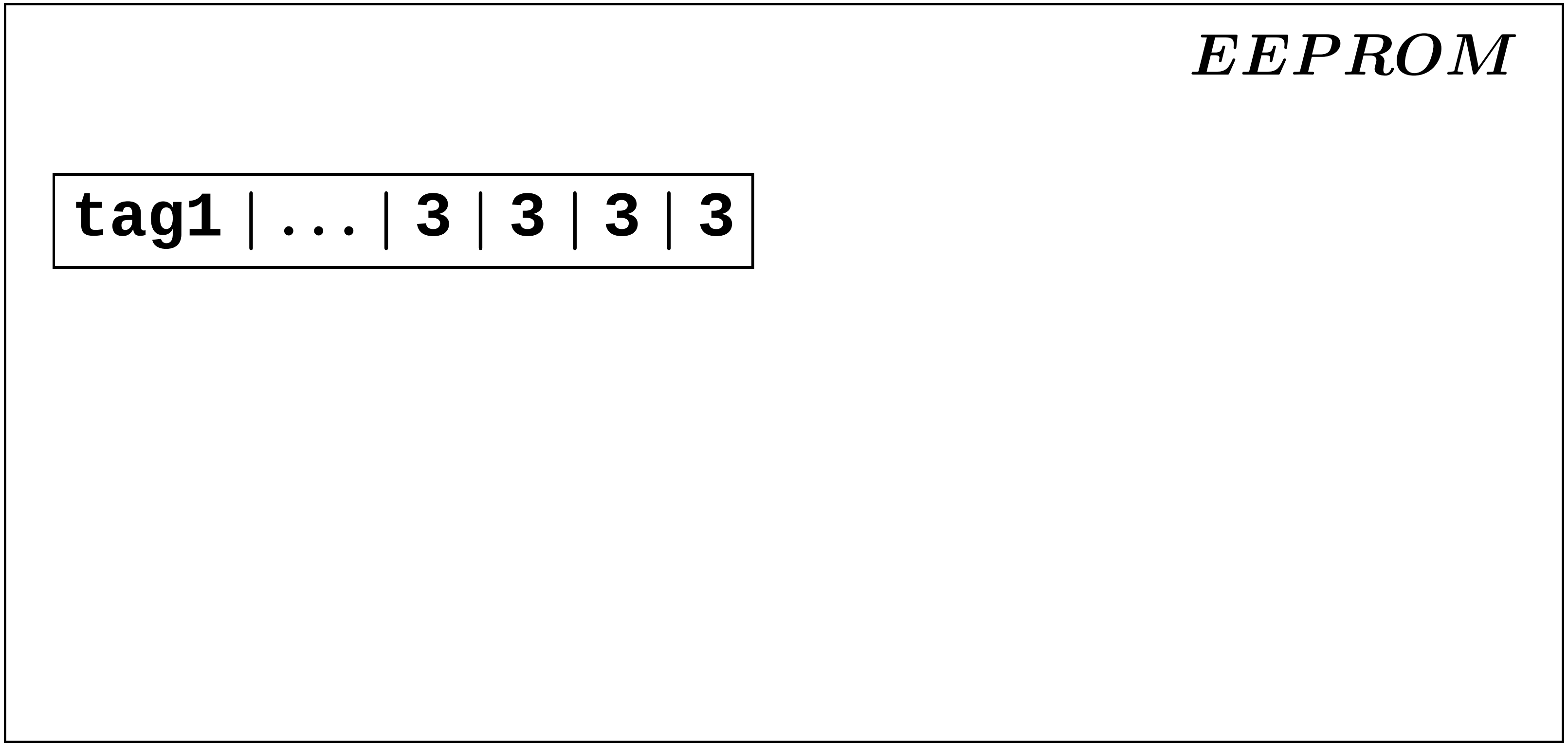


$EEPROM$

tag1	...	1	1	1	1

```
Write(tag1,[1,1,1,1])
```

Transacted Memory

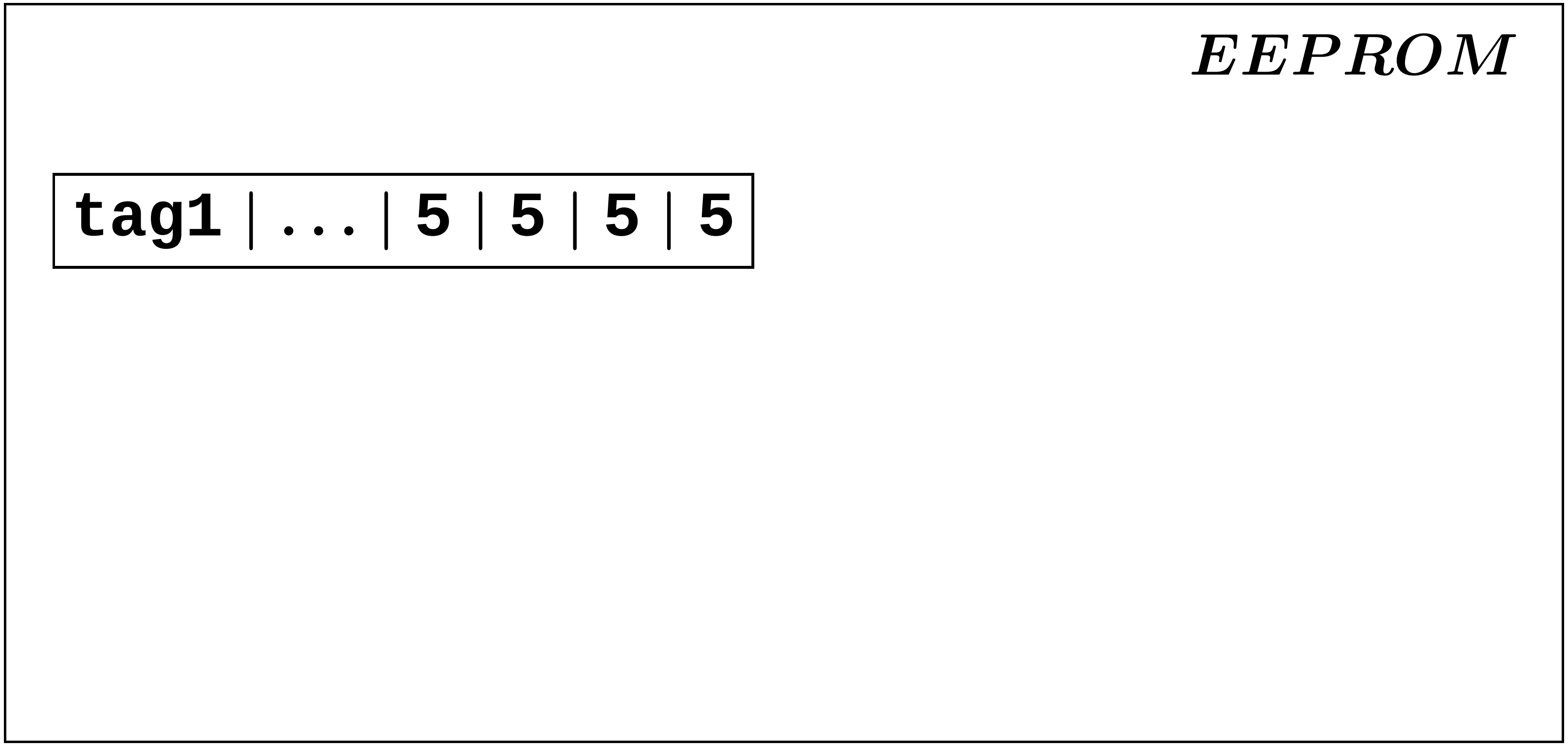


EEPROM

tag1	...	3	3	3	3

```
Write(tag1,[3,3,3,3])
```

Transacted Memory

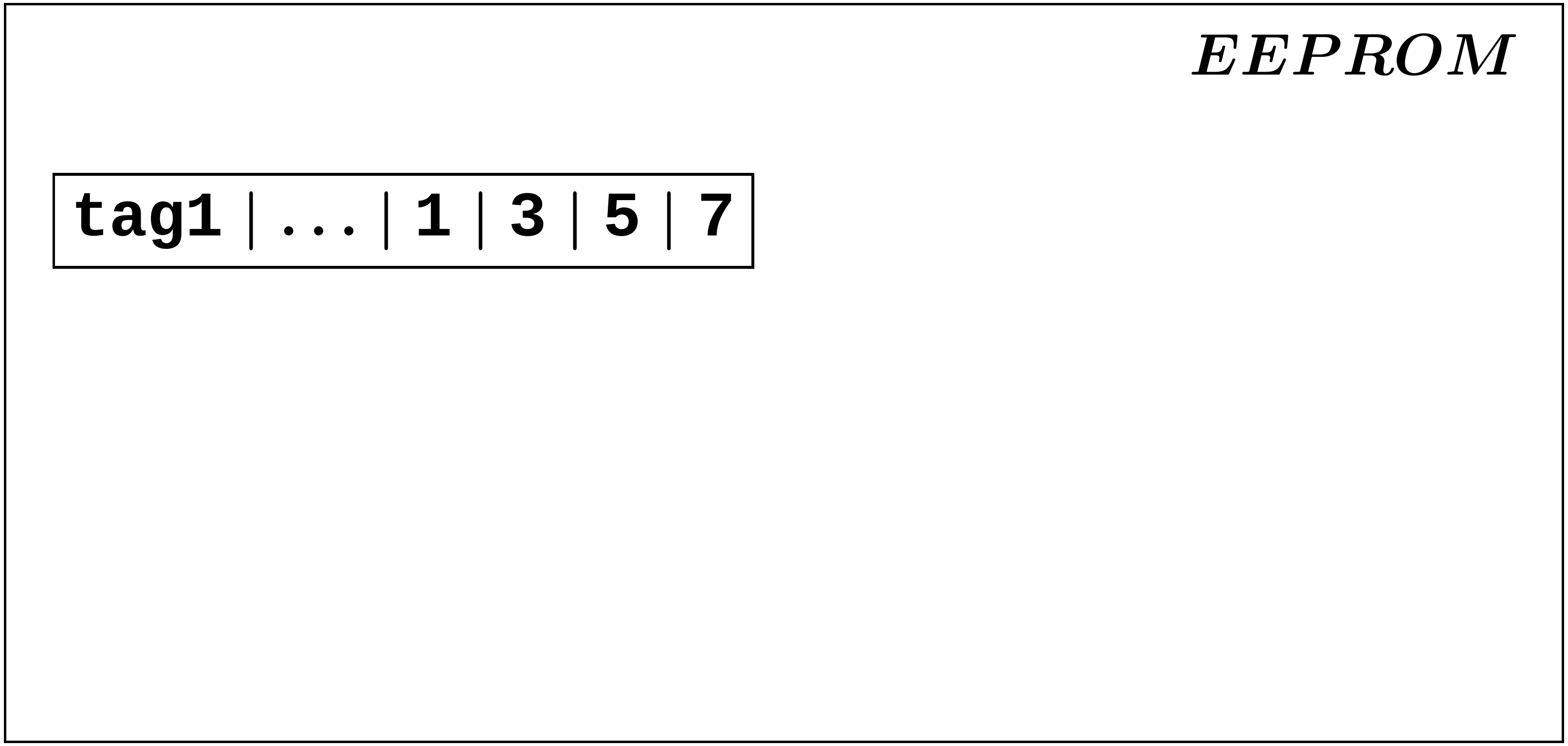


EEPROM

tag1	...	5	5	5	5

```
Write(tag1,[5,5,5,5])
```

Transacted Memory

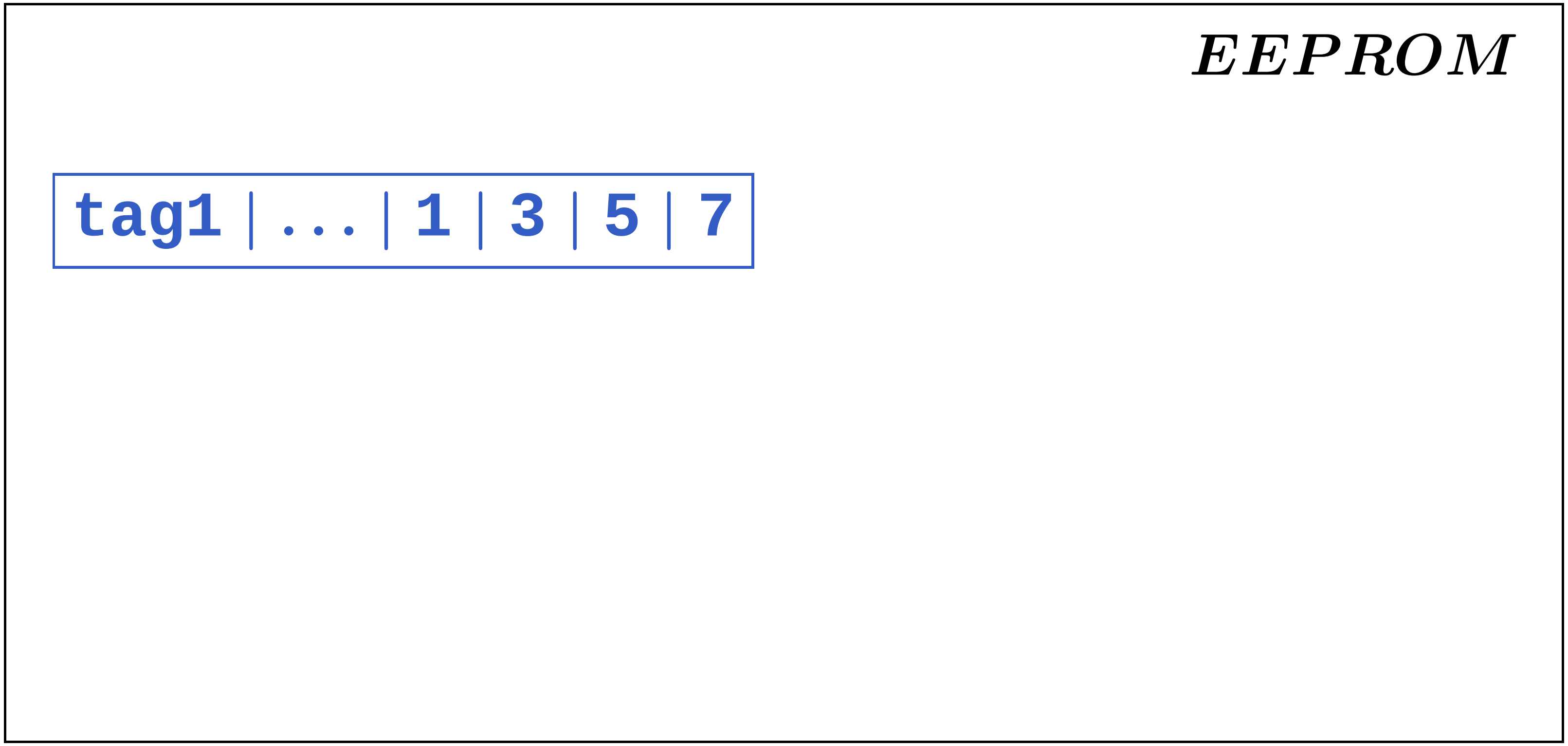


EEPROM

`tag1`	`...`	`1`	`3`	`5`	`7`

```
Write(tag1,[1,3,5,7])
```

Transacted Memory

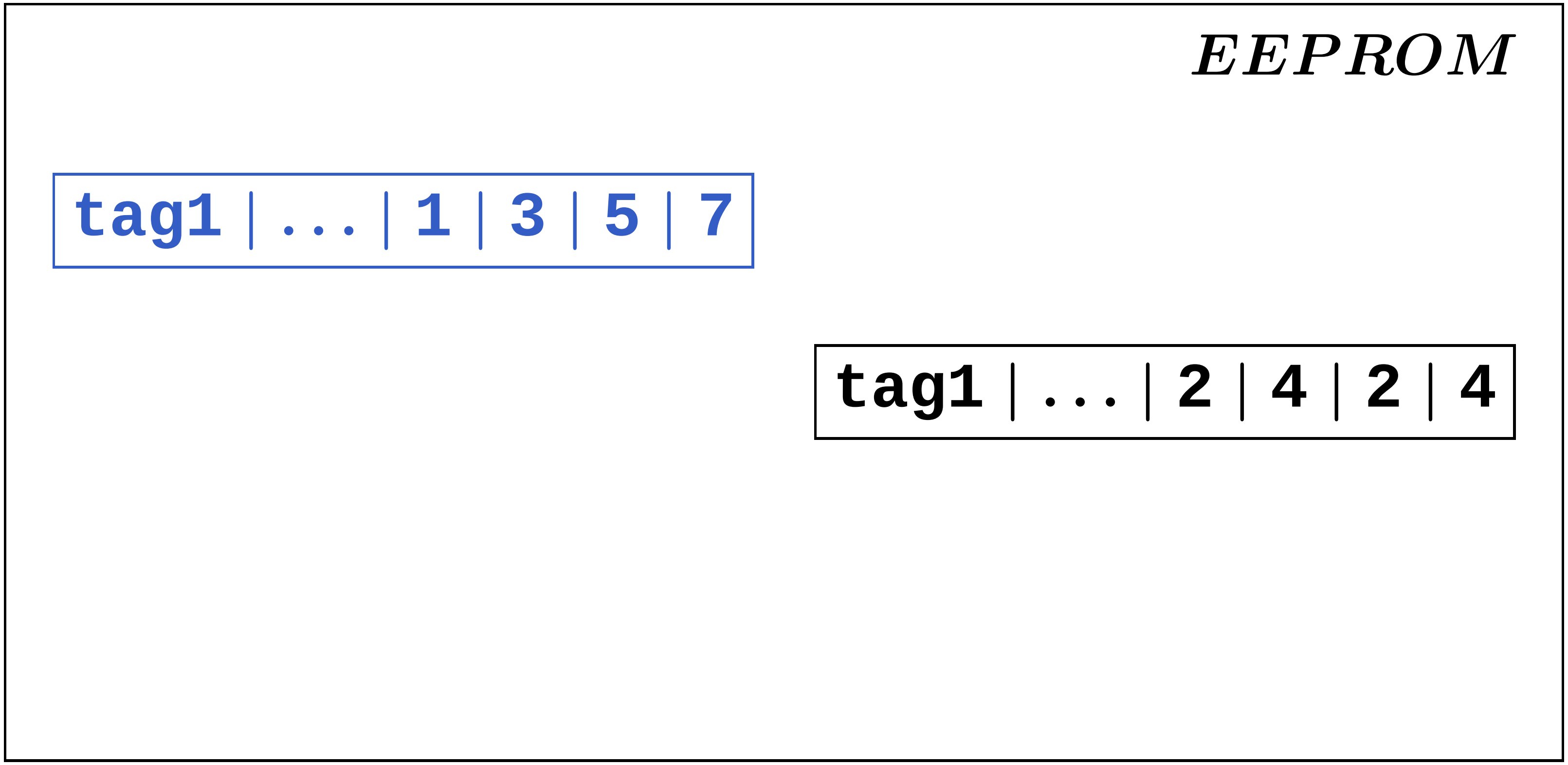


EEPROM

tag1	...	1	3	5	7

`Commit(tag1)`

Transacted Memory



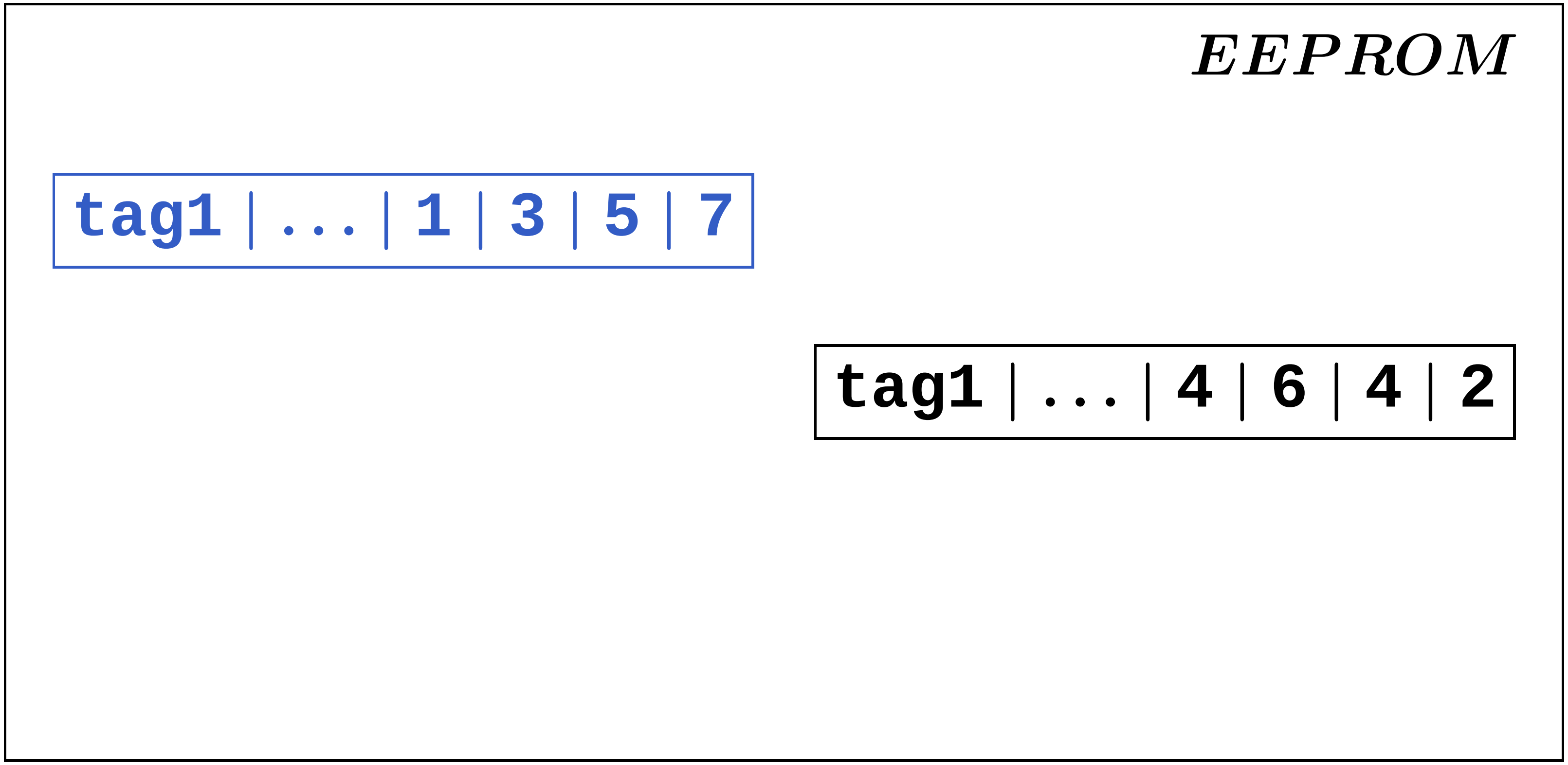
EEPROM

tag1 | ... | 1 | 3 | 5 | 7

tag1 | ... | 2 | 4 | 2 | 4

`Write(tag1,[2,4,2,4])`, undone in case of card tear

Transacted Memory



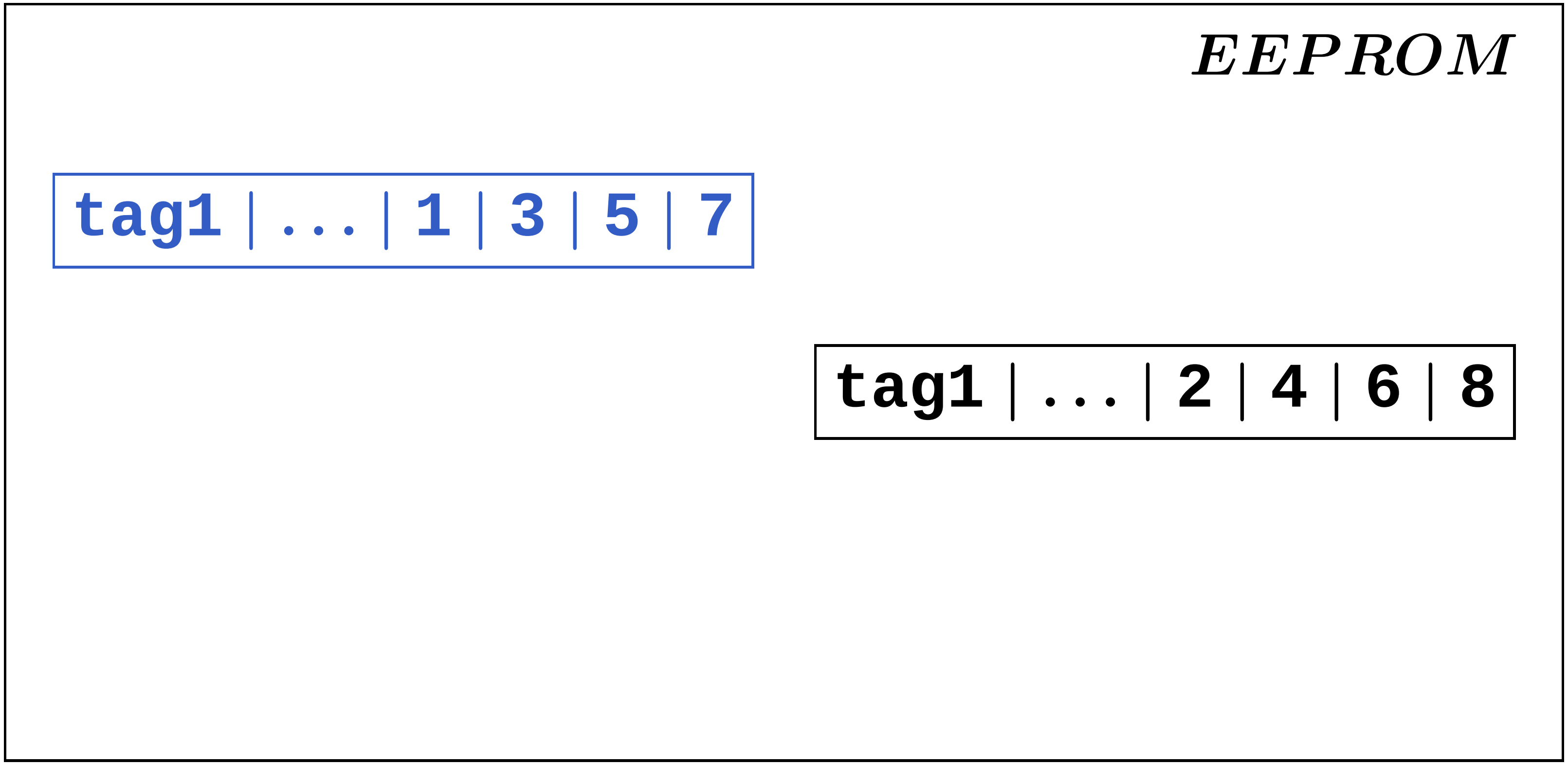
$EEPROM$

tag1 | ... | 1 | 3 | 5 | 7

tag1 | ... | 4 | 6 | 4 | 2

```
Write(tag1,[4,6,4,2])
```

Transacted Memory



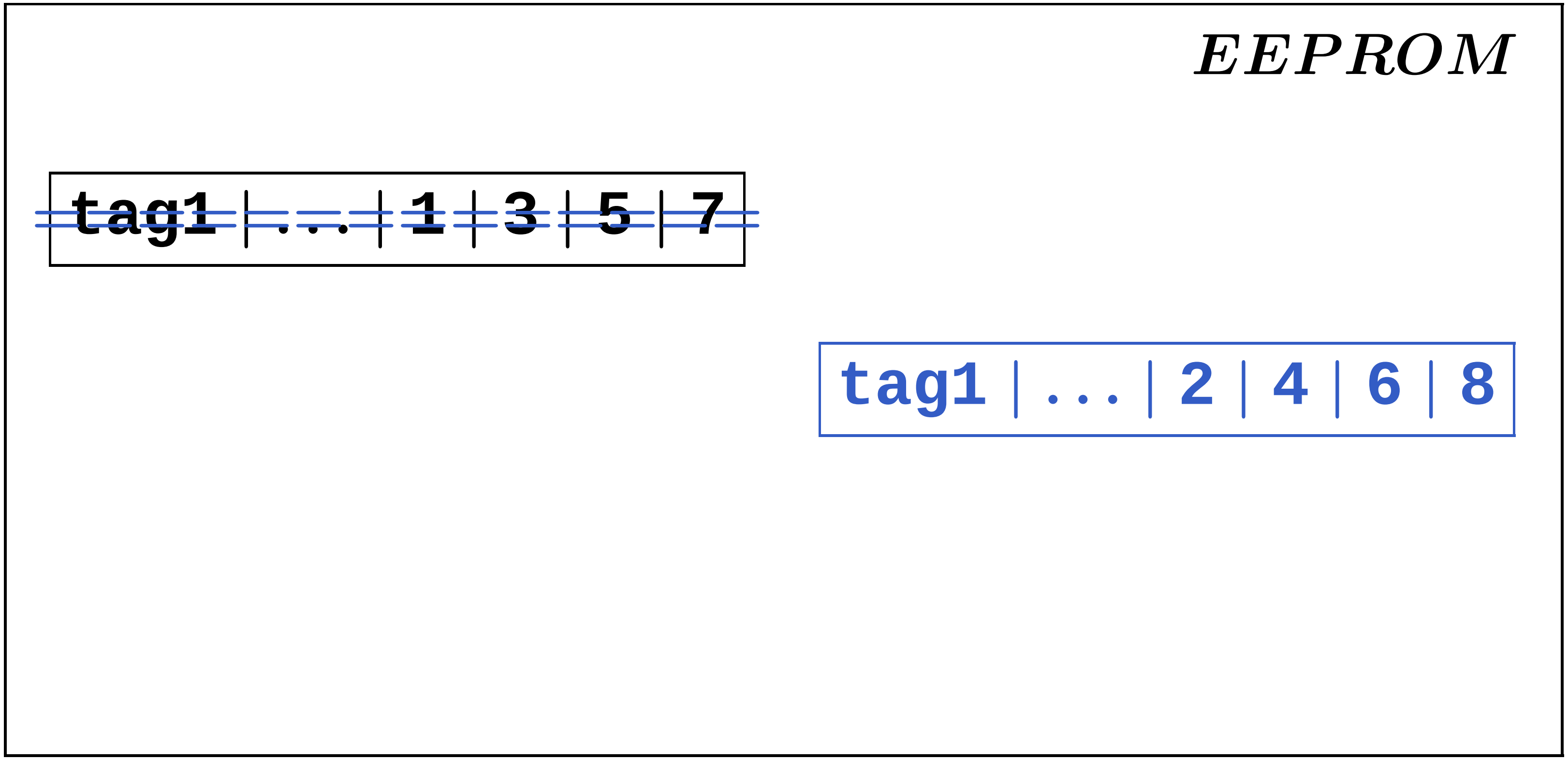
EEPROM

`tag1 | ... | 1 | 3 | 5 | 7`

`tag1 | ... | 2 | 4 | 6 | 8`

```
Write(tag1,[2,4,6,8])
```

Transacted Memory



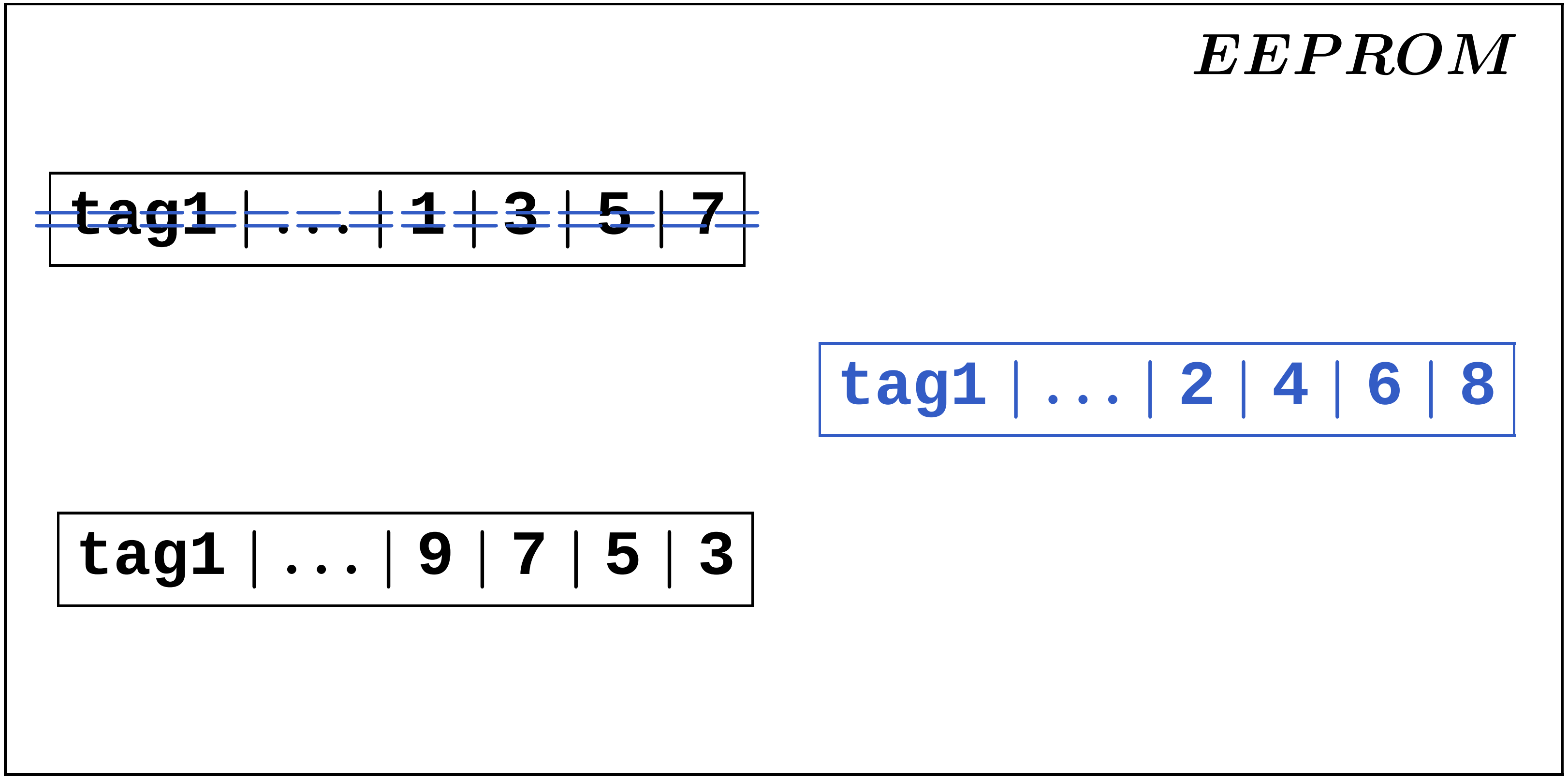
EEPROM

~~tag1 | ... | 1 | 3 | 5 | 7~~

tag1 | ... | 2 | 4 | 6 | 8

`Commit(tag1)` **, removing previous committed generation**

Transacted Memory



EEPROM

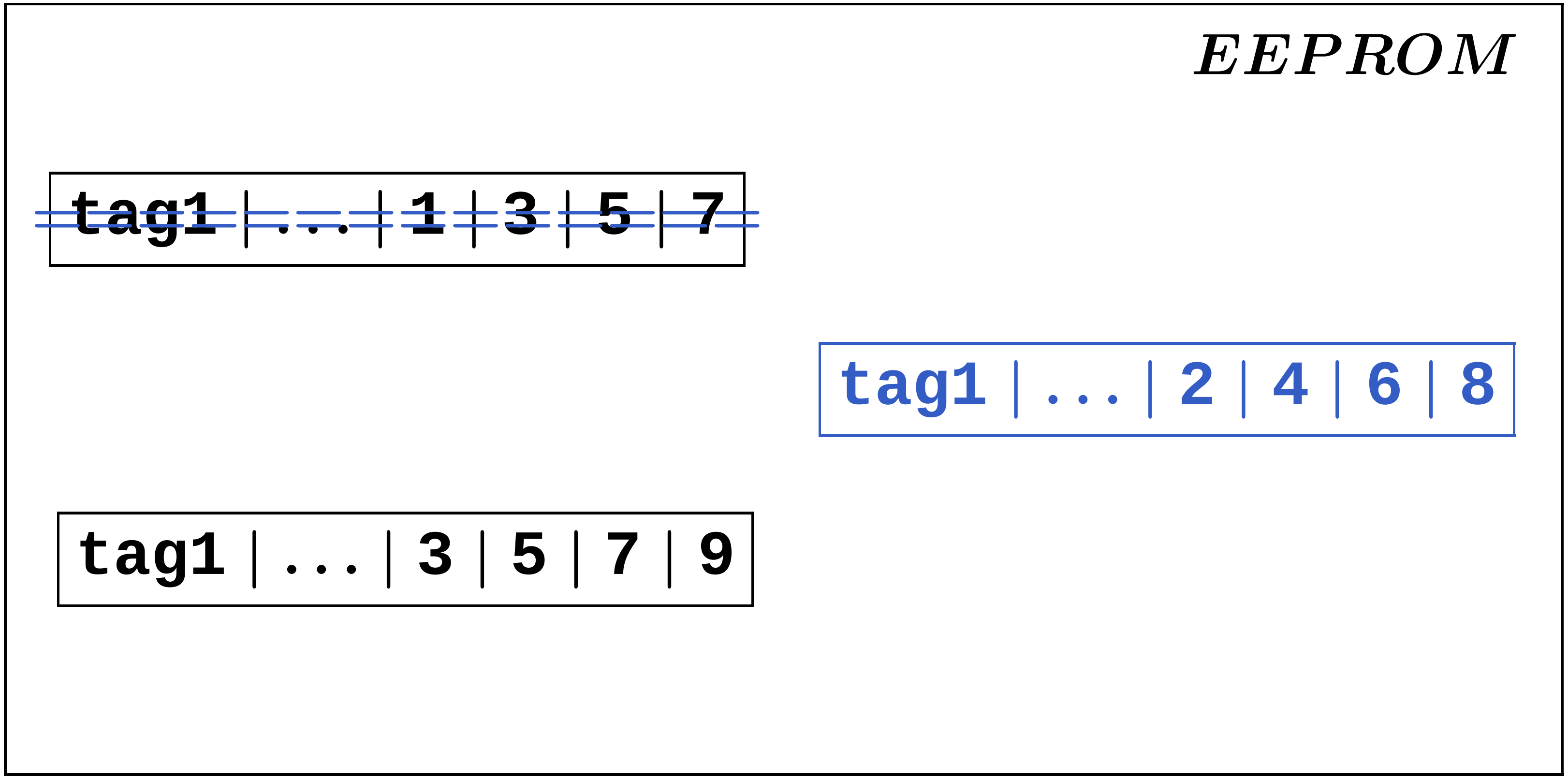
~~tag1 | ... | 1 | 3 | 5 | 7~~

tag1 | ... | 2 | 4 | 6 | 8

tag1 | ... | 9 | 7 | 5 | 3

`Write(tag1,[9,7,5,3])`, undone in case of card tear

Transacted Memory



EEPROM

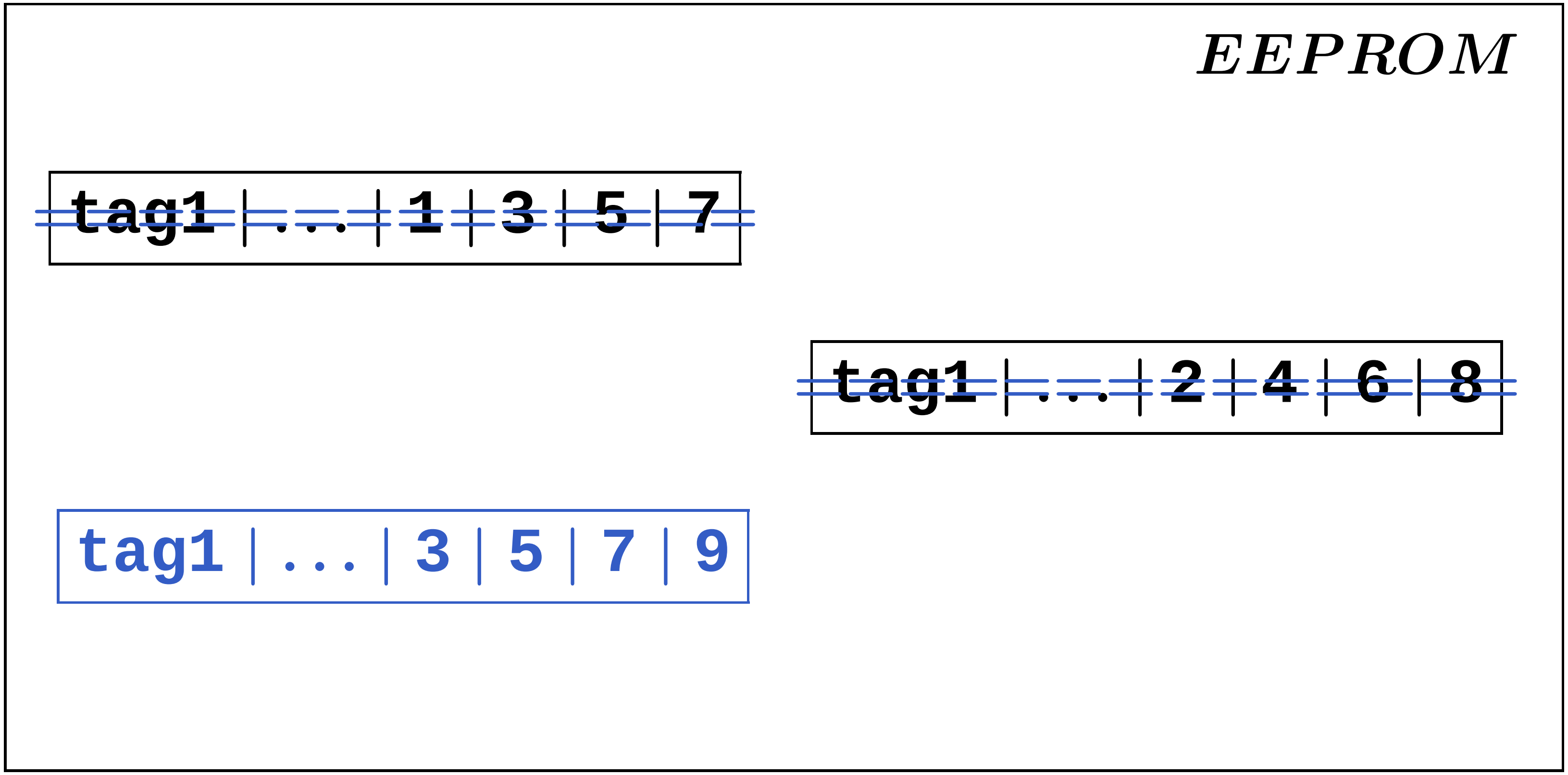
~~tag1 | ... | 1 | 3 | 5 | 7~~

tag1 | ... | 2 | 4 | 6 | 8

tag1 | ... | 3 | 5 | 7 | 9

```
Write(tag1,[3,5,7,9])
```

Transacted Memory



$EEPROM$

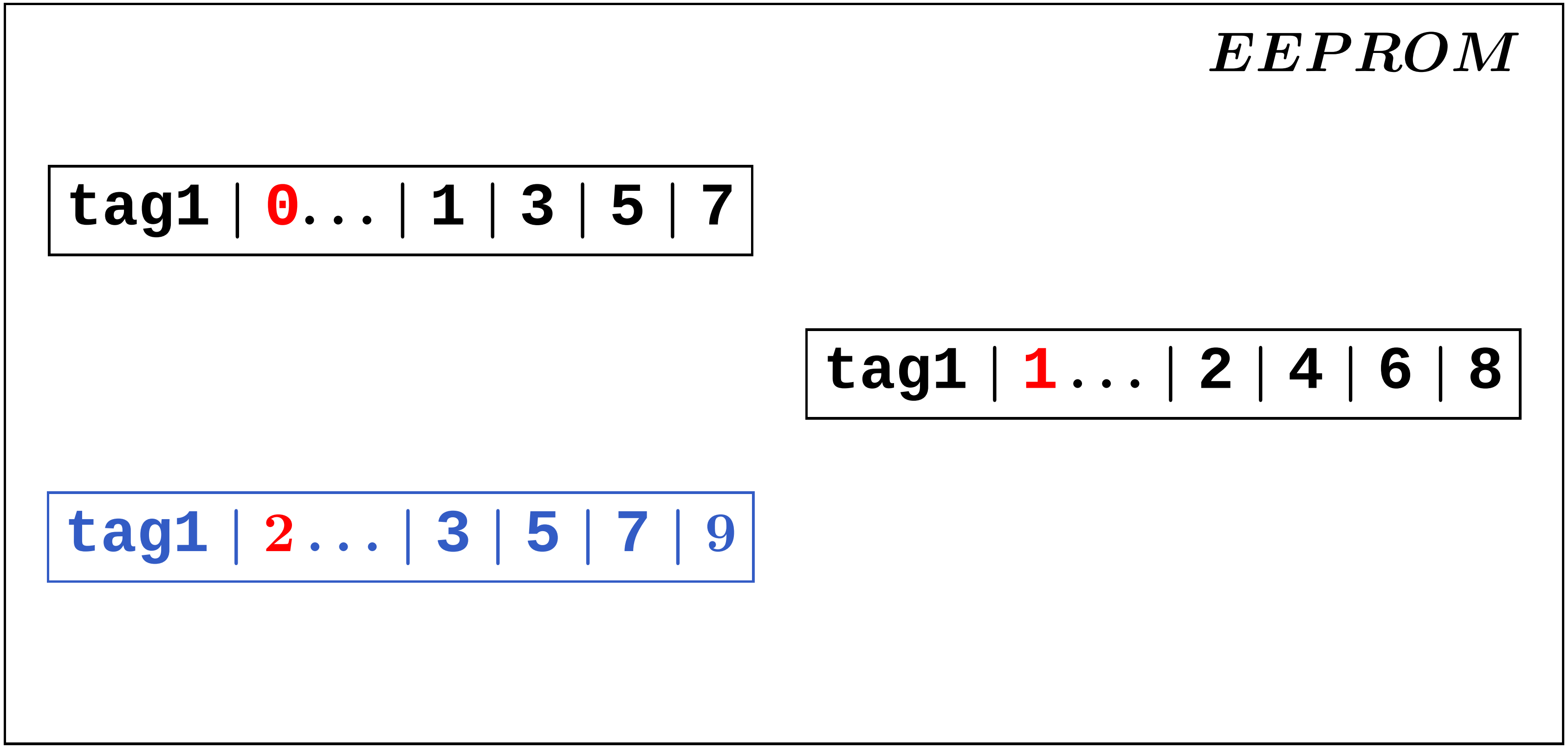
~~tag1 | ... | 1 | 3 | 5 | 7~~

~~tag1 | ... | 2 | 4 | 6 | 8~~

tag1 | ... | 3 | 5 | 7 | 9

`Commit(tag1)` , removing previous committed generation

Transacted Memory



EEPROM

`tag1 | 0... | 1 | 3 | 5 | 7`

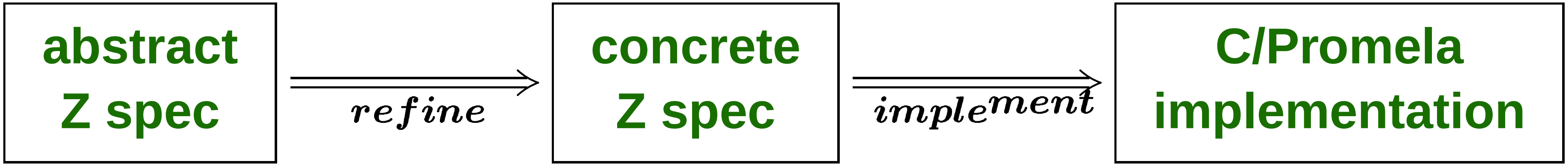
`tag1 | 1... | 2 | 4 | 6 | 8`

`tag1 | 2... | 3 | 5 | 7 | 9`

Logging for free, by numbering committed generations

Earlier work on Transacted Memory

Formal methods – Z and Promela – used for specification & implementation [Butler, Hartel, de Jong, Longley]:


abstract Z spec

$refine$

concrete Z spec

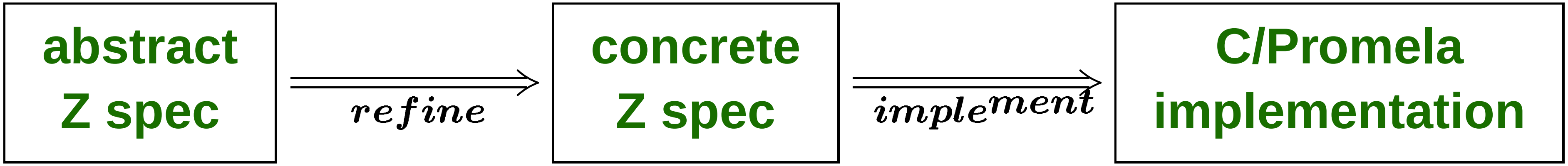
$implement$

C/Promela implementation

Model checked in SPIN.

Earlier work on Transacted Memory

Formal methods – Z and Promela – used for specification & implementation [Butler, Hartel, de Jong, Longley]:



abstract Z spec

refine

concrete Z spec

implement

C/Promela implementation

Model checked in SPIN.

But:

big gap between Z specs and C implementation

no formal relation between them

Idea behind this paper

The idea was to

- **translate C implementation to Java**
- **translate Z specs to JML**

Idea behind this paper

The idea was to

- **translate C implementation to Java**
- **translate Z specs to JML**

so that

- **spec and code are in comparable languages,**
- **tools can be used to check implementation against spec,**
- **we could ultimately prove that implementation is correct.**

Translating C implementation to Java

Translating C implementation to Java

Done by hand - doable for a program of this size.

Only real differences between implementations:

- **more type-safety in the Java implementation; e.g. for**

  ```
  #define Gen  byte  /* 0 .. maxgen  */
  ```

  we introduce a Java class `Gen`.

- **exceptions used in Java to model card tears**

modeling card tears in Java

A card tear is a form of abrupt control flow:

- **card tear is like an exception**
- **clean up after power-on is like the exception handler**
- **card tear is uncatchable exception, caught only in the main repetition of the OS**

A card tear can be faithfully modelled in Java by an exception, that may be thrown just before or after every EEPROM write.

Java implementation

```
public void Write (Tag t, InfoSeq is)
                 throws CardTearException

public void Commit (Tag t)
                 throws CardTearException
...
```

When testing, we randomly throw `CardTearException`’s to simulate card tears.

Specifying the Java implementation using JML

Java Modeling Language JML

Formal specification language tailored to Java

JML can be used to annotate Java programs with

- **pre- and postconditions**
- **invariants**
- **. . .**

Similar to Eiffel (‘Design by Contract’) but more powerful.

Several tools available, incl. runtime assertion checker by Gary Leavens et al (from `www.jmlspecs.org`)

JML spec for Write

```
 public void Write (Tag t, InfoSeq is)
                     throws CardTearException
/*@ requires inUse(t);
  @ ensures
  @    Read(t).equals(is)
  @*/
```

This gives a pre- and postcondition for Write.

JML spec for Commit (1)

```
 public void Commit (Tag t)
                    throws CardTearException
/*@ requires inUse(t);
  @ ensures
  @    Read(t).equals(\old(Read(t)))
  @*/
```

Here `\old` is used to refer to the value that `Read(t)` had in the pre-state.

Of course, this spec is far from complete . . .

JML spec for Commit (2)

```
 public void Commit (Tag t)
                    throws CardTearException
/*@ requires inUse(t);
  @ ensures
  @    Read(t).equals(\old(Read(t)))
  @ && ReadCommitted(t).equals(\old(Read(t)));
  @*/
```

where `ReadCommitted(t)` returns most recent committed generation for `t`.

This spec is still not complete: what if a card tear happens during Commit . . . ?

JML spec for Commit (3)

```
 public void Commit (Tag t)
                    throws CardTearException
/*@ requires inUse(t);
  @ ensures
  @     Read(t).equals(\old(Read(t)))
  @ && ReadCommitted(t).equals(\old(Read(t)));
  @ signals (CardTearException)
  @     ReadCommitted(t).equals(\old(ReadCommitted
  @ || ReadCommitted(t).equals(\old(Read(t)));
  @*/
```

Exceptional postcondition expresses atomicity of Commit

JML spec for Tidy (1)

```
 public void Tidy() throws CardTearException
/*@ ensures (\forall Tag t; 0 <= t && t < MAXTAG;
  @           Read(t).equals(
  @                   \old(CommittedRead(t)));
  @*/
```

Postcondition says that after Tidy-ing all tags are restored to their old committed values.

Here `\forall` is used to quantify over all tags.

JML spec for Tidy (2)

```
 public void Tidy() throws CardTearException
/*@ ensures (\forall Tag t; 0 <= t && t < MAXTAG;
  @           Read(t).equals(
  @                   \old(CommittedRead(t))));
  @ signals (CardTearException)
  @         (\forall Tag t; 0 <= t && t < MAXTAG;
  @           CommittedRead(t).equals(
  @                   \old(CommittedRead(t))));
  @*/
```

Exceptional postcondition says that if Tidy is interrupted none of the committed values change.

Runtime assertion checking

We have translated

- **C implementation to Java**
- **(parts of) the Z spec to JML**

The runtime assertion checker can now be used to test the Java implementation against the JML specs.

Runtime assertion checking

We have translated

- **C implementation to Java**
- **(parts of) the Z spec to JML**

The runtime assertion checker can now be used to test the Java implementation against the JML specs.

The runtime assertion checker checks pre-, post-, and exceptional post-conditions, including uses of `\old` and `\forall`, if the domain of quantification is finite.

Results

Bugs found:

Results

Bugs found:

- one typo - giving ‘version number’ instead of ‘generation number’
  Found during typechecking Java code

Results

Bugs found:

- **one typo - giving ‘version number’ instead of ‘generation number’**
  Found during typechecking Java code
- **one serious error - card tear at certain point is fatal**
  Found using runtime assertion testing
  Repairing this bug was non-trivial!

Results

Bugs found:

- **one typo - giving ‘version number’ instead of ‘generation number’**
  Found during typechecking Java code
- **one serious error - card tear at certain point is fatal**
  Found using runtime assertion testing
  Repairing this bug was non-trivial!

Improvements made to code:

Results

Bugs found:

- one typo - giving ‘version number’ instead of ‘generation number’
  Found during typechecking Java code
- one serious error - card tear at certain point is fatal
  Found using runtime assertion testing
  Repairing this bug was non-trivial!

Improvements made to code:

- throwing an exception when no unused EEPROM is available

Results

Bugs found:

- one typo - giving ‘version number’ instead of ‘generation number’
  Found during typechecking Java code
- one serious error - card tear at certain point is fatal
  Found using runtime assertion testing
  Repairing this bug was non-trivial!

Improvements made to code:

- throwing an exception when no unused EEPROM is available
- throwing an exception when no fresh tags are available

Future/Ongoing work

- VHDL implementation
- fine-tuning the implementation: storing some data in RAM rather than EEPROM
- more detailed specs: translating the complete functional specification from Z to JML
- going beyond testing:
  verification using theorem prover PVS & LOOP tool

Conclusions

Conclusions

Modeling of card tears as Java exceptions allows

- **realistic testing**
- **precise specification in JML**

Conclusions

Modeling of card tears as Java exceptions allows

- **realistic testing**
- **precise specification in JML**

Benefit of formal JML specs (& runtime assertion checking)

- **detailed & precise interface spec**
- **reduced effort for writing test code**
- **improved feedback when testing**

Conclusions

Modeling of card tears as Java exceptions allows

- **realistic testing**
- **precise specification in JML**

Benefit of formal JML specs (& runtime assertion checking)

- **detailed & precise interface spec**
- **reduced effort for writing test code**
- **improved feedback when testing**

JML-annotated Java code is a very accessible formal spec; spec and code together in same file, in similar languages