Karlsruhe Institute of Technology

The KeY-verified Verified Keyserver

VerifyThis Long Term Challenge
27 April 2020

Stijn de Gouw (Open University, NL), Mattias Ulbrich, Alexander Weigl

Our program verifier KeY

Deductive verification

100% Java Card

Java Modeling
Language (JML)

Mostly automatic

Modular
Reasoning

User Interaction Concepts

collaboration with TU Darmstadt and Chalmers University, Gothenburg

Modelling HAGRID in KeY

We present two formalisations of the HAGRID framework as spec'ed and verif'ed Java implementations:

Modelling HAGRID in KeY

We present two formalisations of the HAGRID framework as spec'ed and verif'ed Java implementations:

The **array** model

- uses arrays to implement database and open requests
- specification on these arrays
- 70 loc, 90 los, 10 POs, **fully automatic**

loc/los = lines of code/spec, POs = # of proof obligations

Modelling HAGRID in KeY

We present two formalisations of the HAGRID framework as spec'ed and verif'ed Java implementations:

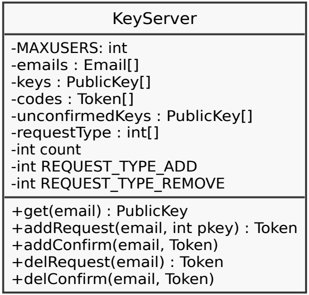
The **array** model

- uses arrays to implement database and open requests
- specification on these arrays
- 70 loc, 90 los, 10 POs, **fully automatic**

The **map** model

- uses map data structures to implement db and open requests
- specification on ADT maps
- "object singularities"
- 146 loc, 262 los, 40 POs, **89 interactions**

The array model



KeyServer

-MAXUSERS: int
-emails : Email[]
-keys : PublicKey[]
-codes : Token[]
-unconfirmedKeys : PublicKey[]
-requestType : int[]
-int count
-int REQUEST_TYPE_ADD
-int REQUEST_TYPE_REMOVE

+get(email) : PublicKey
+addRequest(email, int pkey) : Token
+addConfirm(email, Token)
+delRequest(email) : Token
+delConfirm(email, Token)

- Backend of Hagrid
  - retrieving of public keys
  - verified adding of entries
  - verified deletion of entries
- Simplifications
  - All data types are (array of) `int`'s.
  - Maps are represented by a key/value array.
- simplified/Keyserver.java

The array model: Invariants

ruling out aliasing

```
invariant emails != keys && emails != codes && emails != unconfirmedKeys;
invariant emails != requestType;
invariant keys != codes && keys != unconfirmedKeys;
invariant keys != requestType;
invariant codes != unconfirmedKeys && codes != requestType;
invariant unconfirmedKeys != requestType;
```

All arrays are non-null and have the same length (# of users)

```
invariant emails != null && keys != null && codes != null;
invariant unconfirmedKeys != null && requestType != null;
invariant emails.length == MAXUSERS && keys.length == MAXUSERS;
invariant codes.length == MAXUSERS && unconfirmedKeys.length == MAXUSERS;
invariant requestType.length == MAXUSERS;
```

number of users is bounded

```
invariant 0 <= count && count <= MAXUSERS;
```

emails are unique

```
invariant (\forall int i,j ;
                0 <= i && i < j && j < count;
```

The array model: a method contract

Informal Contract: add(Email, PublicKey)

Stores request to add the given key for the specified user. The key still needs to be confirmed with #addConfirm(Email, Token). Does nothing if the specified user does not exist.

- id the email of the user
- pkey – public key to added after confirmation
- **returns** the array index where the key will be stored

```
public int addRequest(int id, int pkey) {
    int pos = posOfId(id);  // find the entry in the current
    if(pos < 0) { pos = count++; } // not found, use an empty entry
    emails[pos] = id;
    codes[pos] = random();
    unconfirmedKeys[pos] = pkey;
    requestType[pos] = REQUESTTYPE_ADD;
    return pos;
}
```

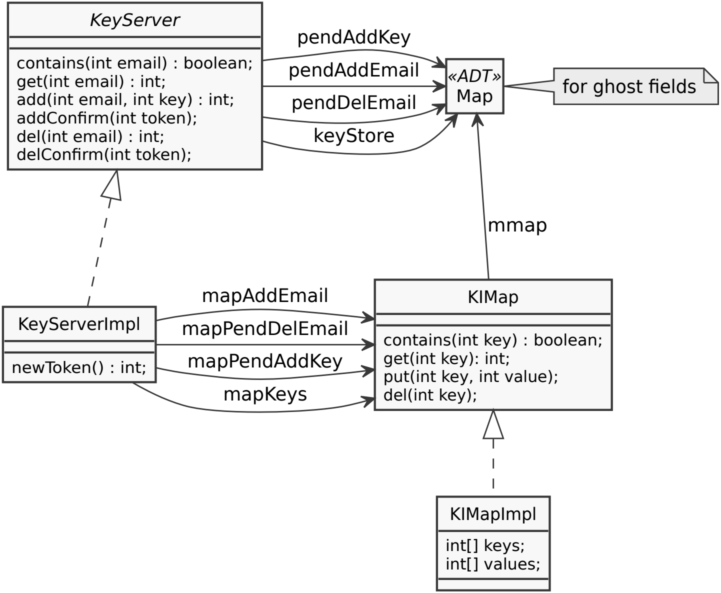
The array model: addConfirm

```
public normal_behaviour
requires count < MAXUSERS;      // internal capacity limit not reached
ensures 0 <= \result;
ensures count == \old(count) && \result < count
     ||  count == \old(count) + 1 && \result == count - 1;
ensures emails[\result] == id && unconfirmedKeys[\result] == pkey && codes[\result
      ]>0;
ensures requestType[\result] == REQUESTTYPE_ADD;

// preservation of the other entries
ensures (\forall int i; 0<=i && i<count;
             (emails[i] == (i == \result ? id : \old(emails[i])))
           && (unconfirmedKeys[i] == (i == \result ? pkey : \old(unconfirmedKeys[i]))
      )
           && (i != \result ==> (codes[i] == \old(codes[i])))
           && (i != \result ==> (requestType[i] == \old(requestType[i]))));
assignable emails[*], unconfirmedKeys[*], codes[*], requestType[*], count;

public int addRequest(int id, int pkey) { ...
```

The map model



Confirming a new key

Original syntax:

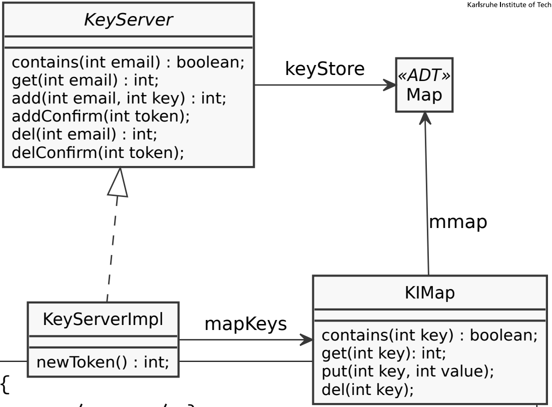
```
/*@ public normal_behavior
  @  requires \dl_inDomain(pendAddEmail, token);
  @  ensures keyStore ==
  @   \dl_mapUpdate(\old(keyStore),
  @      \dl_mapGet(\old(pendAddEmail), token),
  @      \dl_mapGet(\old(pendAddKey), token));
  @  ensures pendAddEmail ==
  @   \dl_mapRemove(\old(pendAddEmail), token);
  @  ensures pendAddKey ==
  @   \dl_mapRemove(\old(pendAddKey), token);
  @  ensures pendDelEmail == \old(pendDelEmail);
  @  assignable footprint;
  @*/
public void addConfirm(int token);
```

Confirming a new key

More mathematical syntax:

```
/*@ public normal_behavior
  @  requires token \in pendAddEmail;
  @
  @  ensures keyStore == \old(keyStore)[
  @    \old(pendAddEmail)[token] <-
  @    \old(pendAddKey)[token]];
  @
  @  ensures pendAddEmail == \old(pendAddEmail) - token;
  @
  @  ensures pendAddKey == \old(pendAddKey) - token;
  @
  @  ensures pendDelEmail == \old(pendDelEmail);
  @
  @  assignable footprint;
  @*/
public void addConfirm(int token);
```

Connecting ghosts and implementation



KeyServer

contains(int email) : boolean;
get(int email) : int;
add(int email, int key) : int;
addConfirm(int token);
del(int email) : int;
delConfirm(int token);

keyStore

«ADT»
Map

mmap

KeyServerImpl

newToken() : int;

mapKeys

KIMap

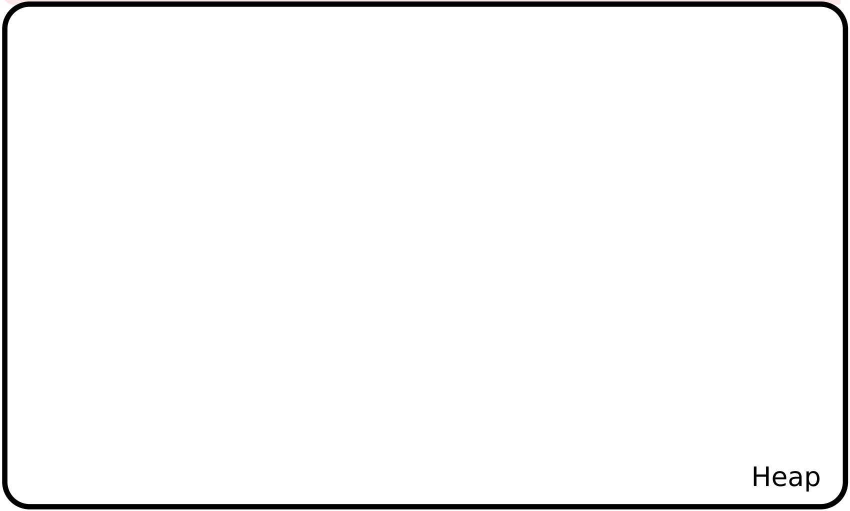
contains(int key) : boolean;
get(int key): int;
put(int key, int value);
del(int key);

```
interface KeyServer {
    ghost \map keyStore; /*...*/ }

class KeyServerImpl implements KeyServer {
    KIMap mapKeys = KIMap.newMap();
    invariant mapKeys.<inv>;
    invariant keyStore == mapKeys.mmap; /*...*/}
```

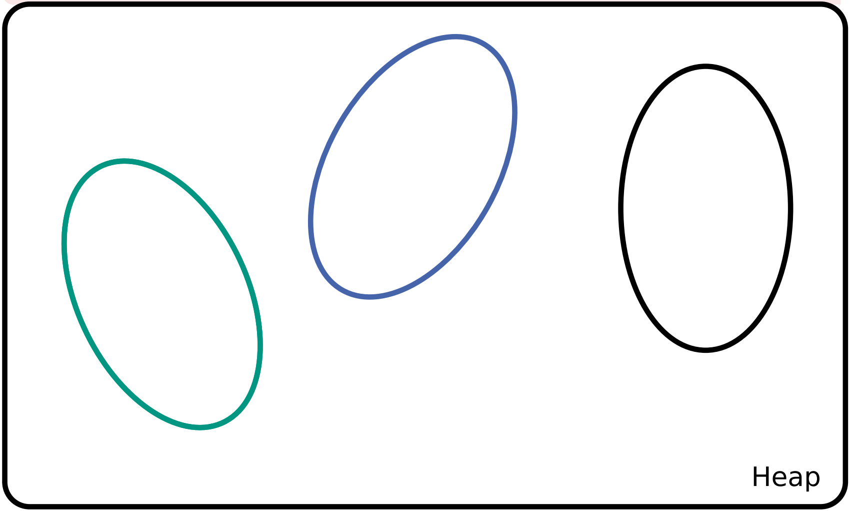
Dynamic Frames – “Singularities”

Proofs cannot be conducted due to framing problems.



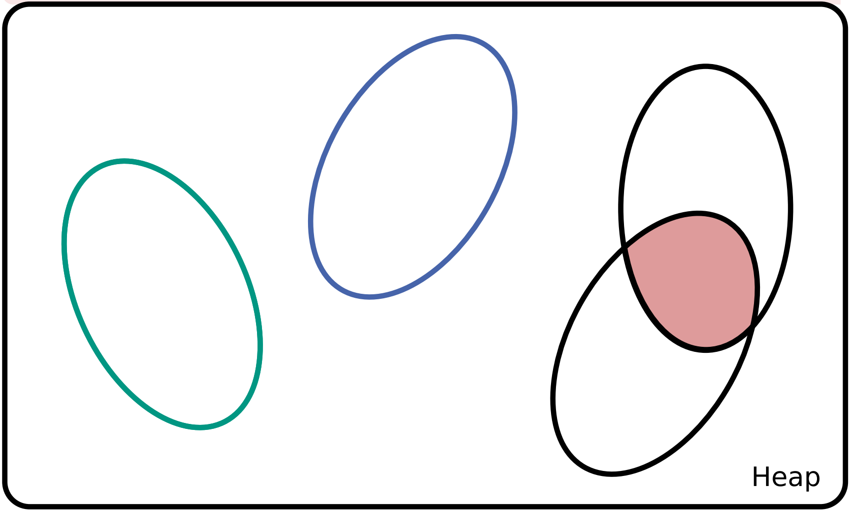
Dynamic Frames – “Singularities”

Proofs cannot be conducted due to framing problems.



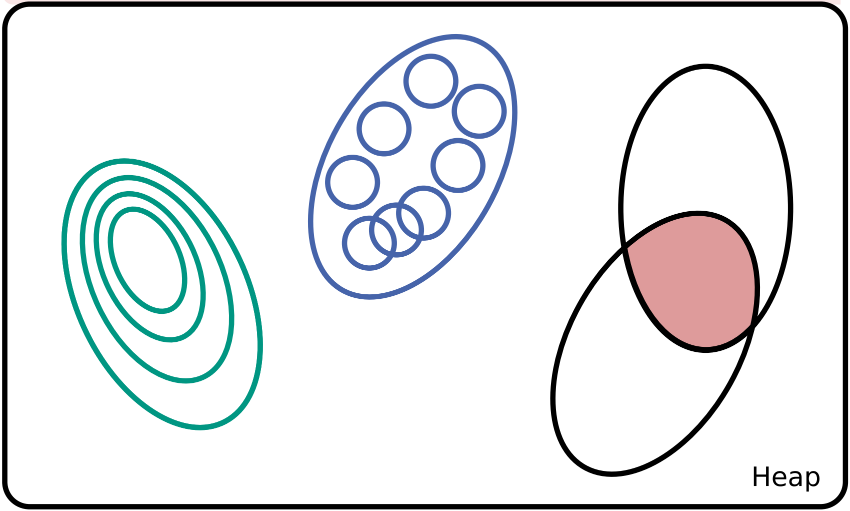
Dynamic Frames – “Singularities”

Proofs cannot be conducted due to framing problems.



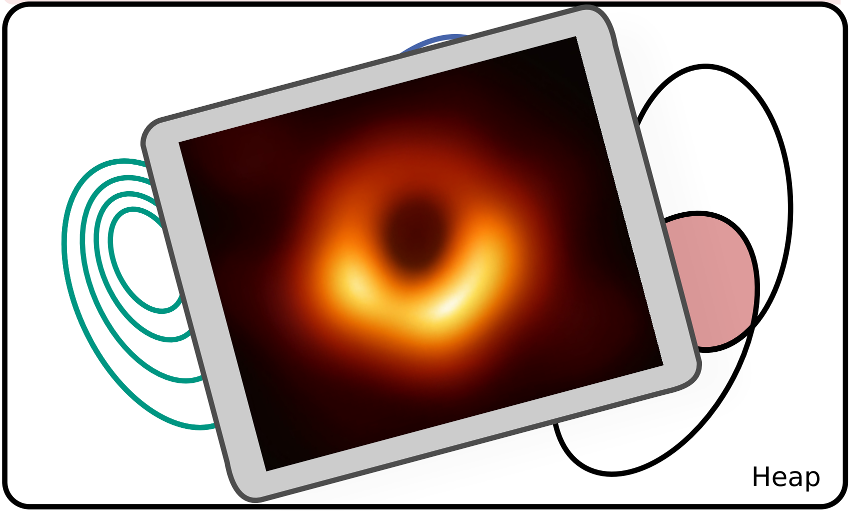
Dynamic Frames – “Singularities”

Proofs cannot be conducted due to framing problems.



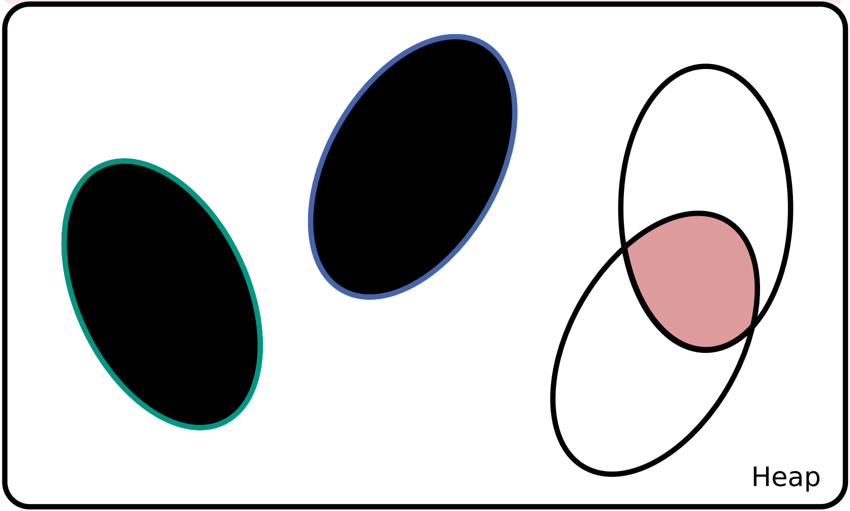
Dynamic Frames – “Singularities”

Proofs cannot be conducted due to framing problems.



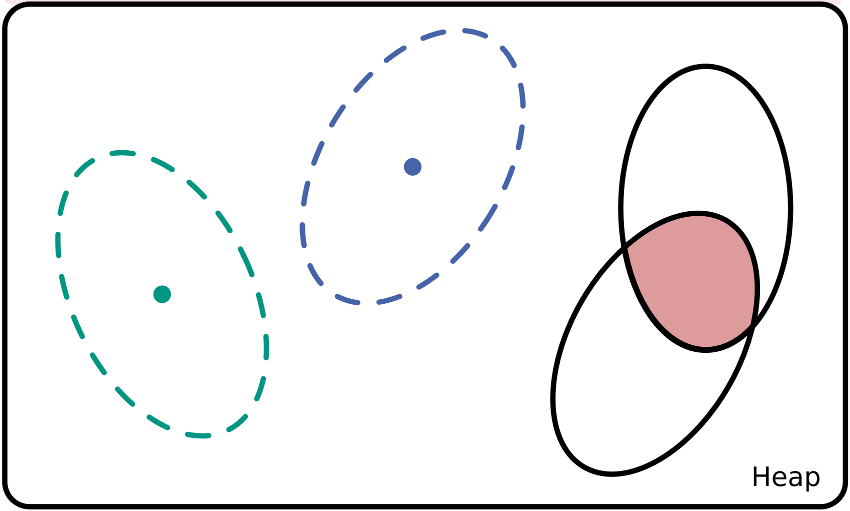
Dynamic Frames – "Singularities"

Proofs cannot be conducted due to framing problems.



Dynamic Frames – “Singularities”

Proofs cannot be conducted due to framing problems.



Singularities

Original class

```
interface Map {
 //@ ghost \locset footprint;

 //@ model \map mmap;

 /*@ ensures \result == mmap[k];
   @ accessible footprint; */
 int get(int k) {...}

 /*@ ensures mmap == \old(mmap)[k<-v];
   @ assignable footprint; */
  int get(int k, int v) {...}
}
```

Singularities

Original class

```
interface Map {
 //@ ghost \locset footprint;

 //@ model \map mmap;

 /*@ ensures \result == mmap[k];
   @ accessible footprint; */
 int get(int k) {...}

 /*@ ensures mmap == \old(mmap)[k<-v];
   @ assignable footprint; */
  int get(int k, int v) {...}
}
```

Singularity replacement

```
interface Map {
 //@ ghost \free footprint;

  ... copy the rest
```

\free is uninterpreted sort
“footprint” captures the “state”

Summary

- *we presented two models:*
  one automatic, one pretty interactive

- Limitations and open challenges:
  - integers instead of strings (→ thesis @ KIT)
  - linear maps, not hash maps (→ thesis @ OU)
  - framing, singularities (→ thesis @ KIT)

- Long-term goals:
  - Specify and verify secure information flow (using KeY)