Specifying Components with Automata

for the VerifyThis Long Term Challenge

Gidon Ernst, Marieke Huisman, Raúl de Monti,
Mattias Ulbrich and Alexander Weigl

2021-02-12

Today's Agenda

Thanks for the feedback via mail!

1. **Questions from the Mail**
   In particular: Look at a few ideas for a state machine base specification.

2. **Discussion**
   Are state machines/automata a good specification language for black-box components to achieve common goals, i.e., as an interface between tools and approaches, and at what level (technical, conceptual)?

3. **Continuity**
   What would be a good follow-up challenge, with potential impact and chances for collaboration?

Mail Questions

When you think about formal methods in software development, what technique comes to your mind first?

$\Longrightarrow$ Poll

Mail Questions

Complete the following sentence: **"For me, an 'automaton' in the context of formal program specification is ..."**

- a representation of state + a transition function ($\rightarrow$ Abstr. State Machines)
- a type describing state + a family of transition functions + invariants
- (partial) specification representation as a model + guarded transitions + external events
- a state transition system with high abstraction level
- a state transition system with explicit (complex) data + data evolution
- model for both (technical) system and environment

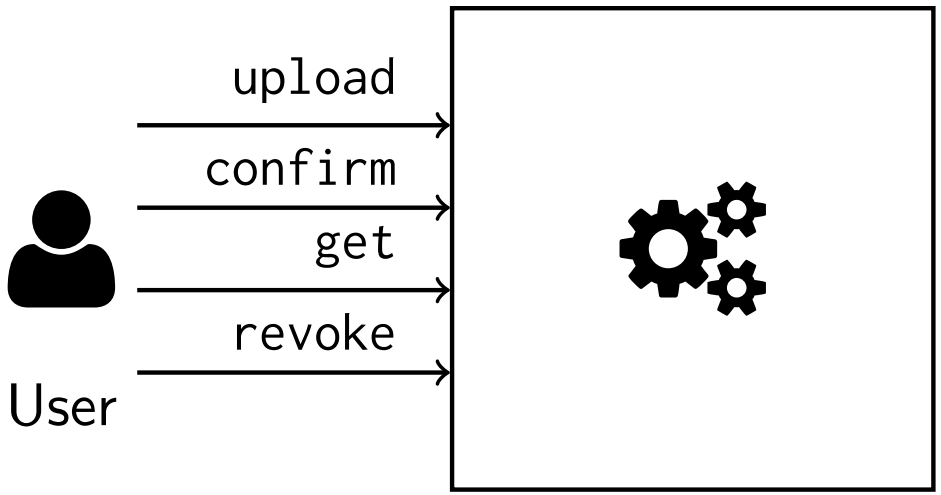
Mail Questions

If you reconsider the HAGRID system, how would the toplevel specification of the system for key registration/removal look like in a stateful specification language?

We look at a meshup of the suggested solutions.

(We received some more hints that other solutions are in the pipeline, or could be obtained from existing submissions)

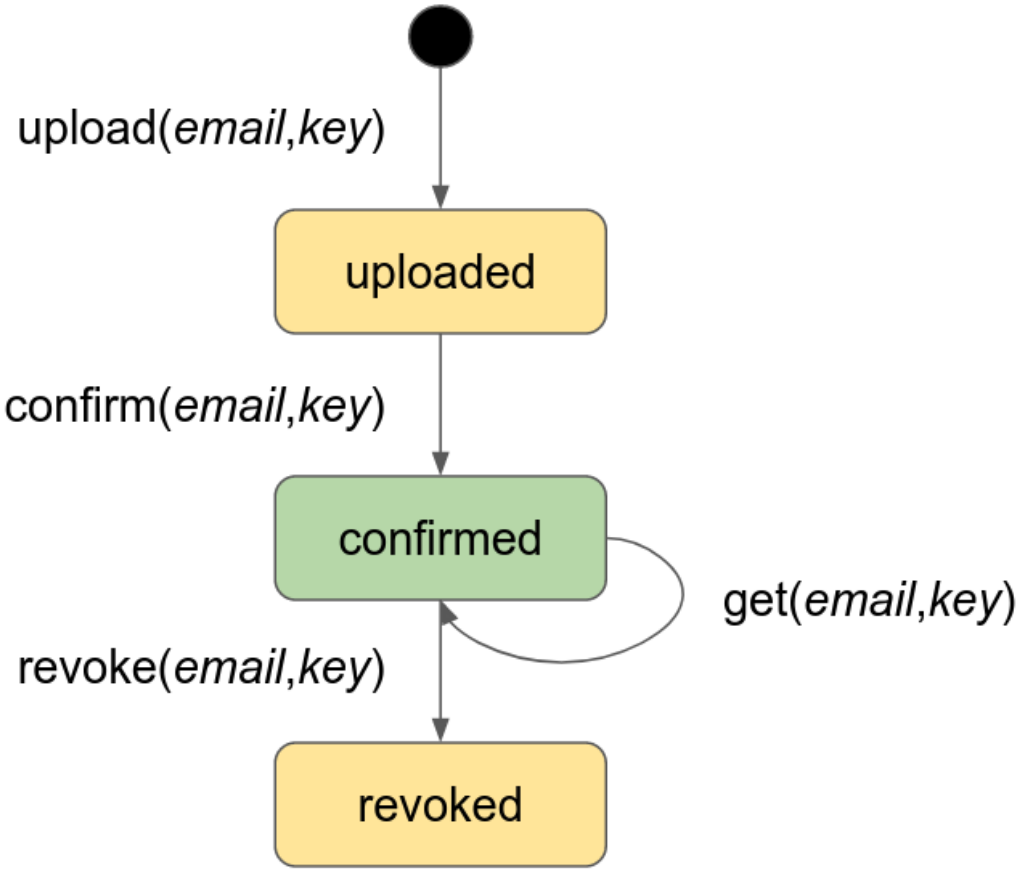
Hagrid Seen as a Blackbox Component



- operations receive and return immutable data
- operations modify a self-contained state
- not every operation may be invoked at all times

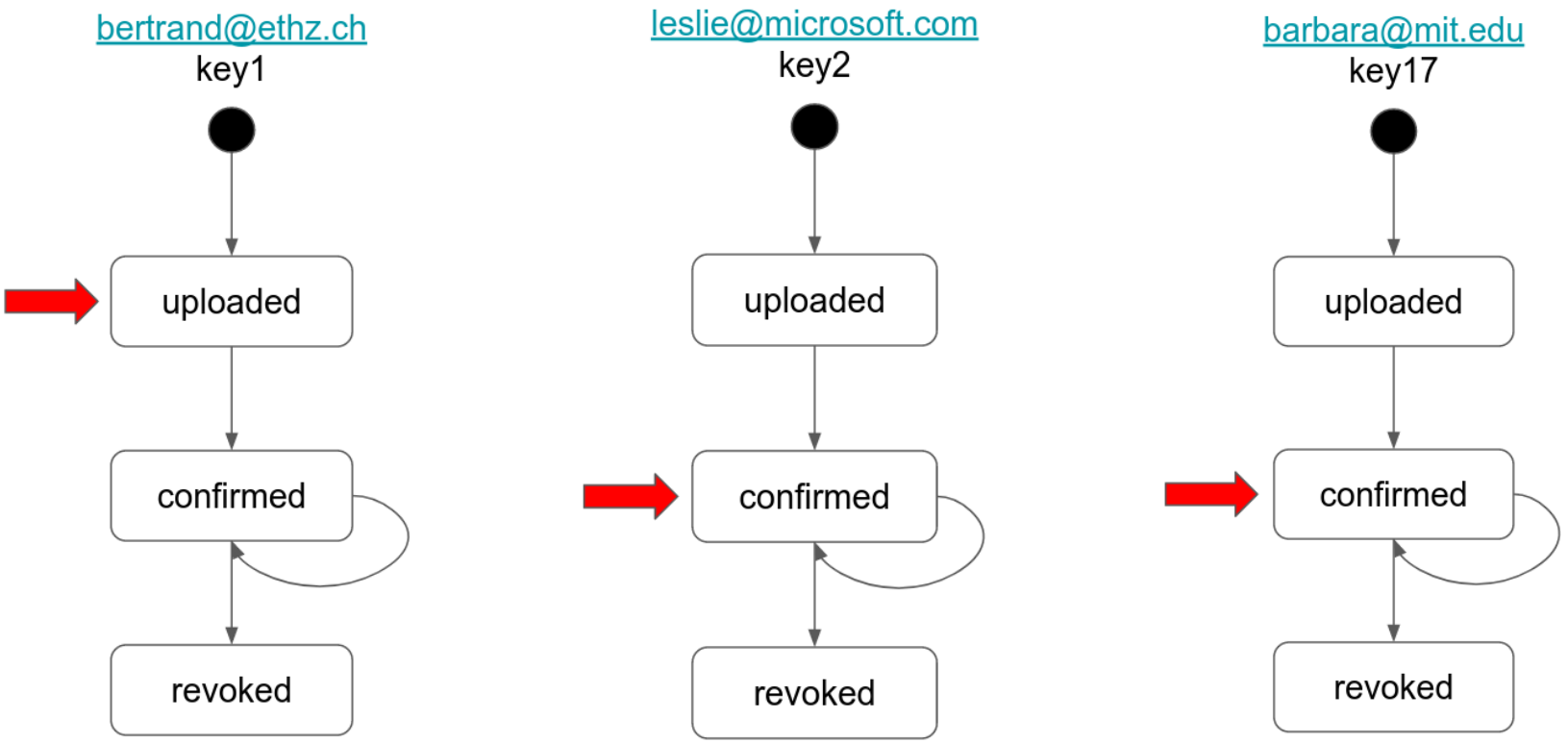
Modeling (*email*, *key*) State Transitions

Per email/key pair: $M(email, key)$



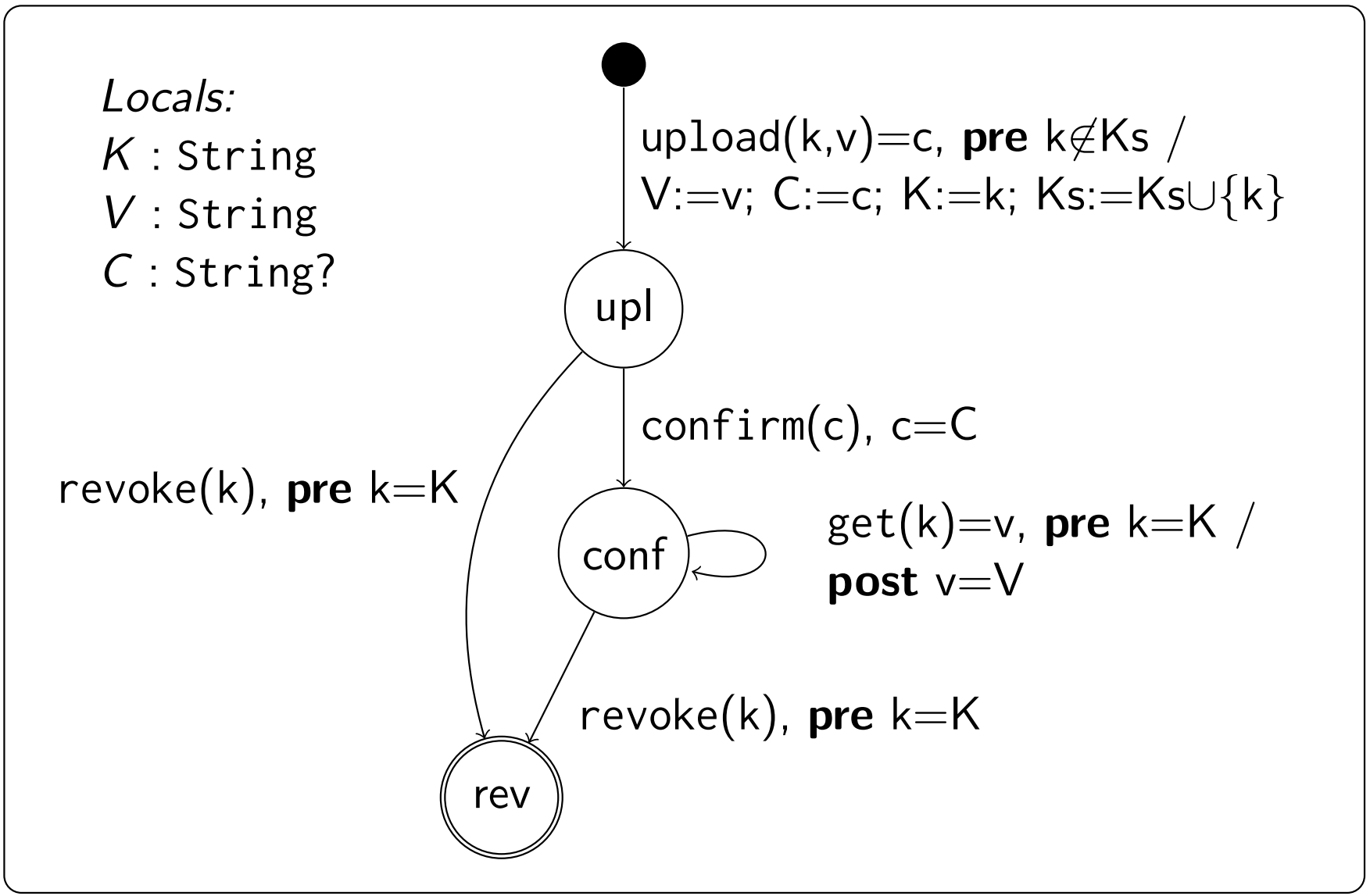
Entire System: $M(email_1, key_1) \parallel \cdots \parallel M(email_n, key_n)$

Example State



Automaton for Key Lifecycle Contract

Globals: $Ks : set(\texttt{String})$



Single Automaton (Event-B)

MACHINE KeyServer

SEES Datatypes

VARIABLES database openUpls openRevokes

INVARIANTS

databaseType: $database \in EMAIL \leftrightarrow KEY$

noSpuriousDels: $ran(openRevokes) \subseteq database$

disjointConfirms: $dom(openRevokes) \cap dom(openUpls) = \varnothing$

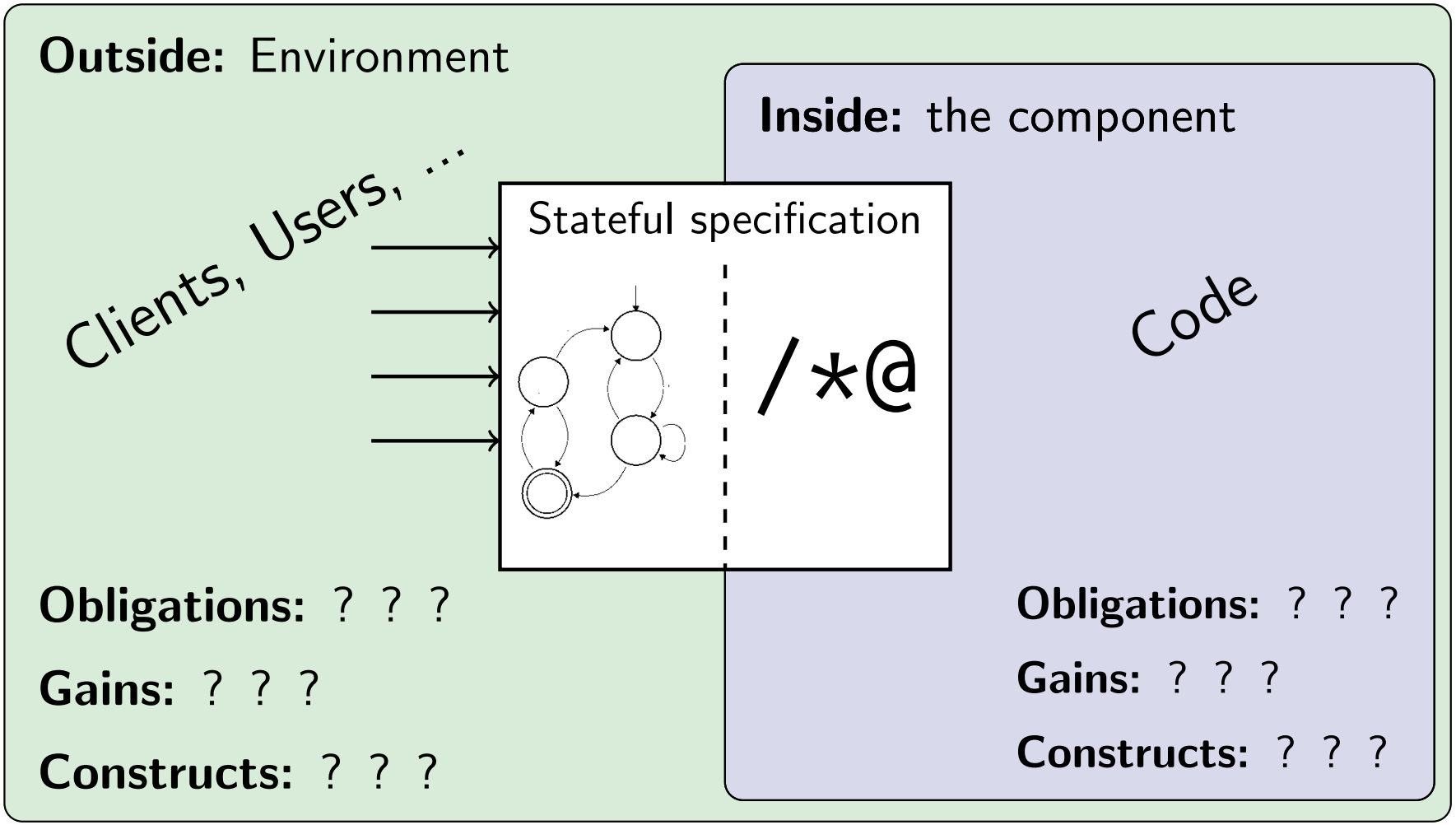
...

EVENTS

- initialize ...
- upload ...
- confirm ...
- revoke ...

[see Event-B-Model on Homepage]

The Two Faces of the Specification



In JML

```
1  interface Hagrid {
2    //@ ghost \seq state;       // sequence of States
3    //@ ghost \set allKeys;     // set of Strings
4
5    /*@ forall int i; 0 <= i < state.length;
6      @ requires state[i].key.equals(key);
7      @ ensures \result == state[i].value;
8      @ assignable \strictly_nothing;
9      @*/
10   String get(String key);
11
12   /*@ requires !(key \in allKeys);
13     @ ensures state == \old(state) + [(key, value, \result)];
14     @ ensures allKeys == \old(allKeys) + {key};
15     @ assignable this.footprint;
16     @*/
17   String upload(String key, String value);
18 }
```

In Why3 [by JC Filiâtre]

```
type state = { ghost mutable keys: email -> key; ... }
    invariant { ... }
```

The type state contains one or several ghost mutable fields, that describe the contents of the state. Invariants are attached to this type. The global state itself is then declared as a global variable of that type:

```
val global_state: state
```

Finally, operations are declared as follows:

```
val add_key (e: email) (k: key) : token
writes { global_state } requires { ... } ensures { ... }
val confirm_add_key: ...
  writes { global_state } ...
val find_key: ...
  reads  { global_state } ...
```

Potential Discussion Points

Interesting questions include:

- What is the right formalism?
- How does it relate to ghost code?
- How does it relate to design-by-contract?
- What is the specified entity? (a "component"?)
- Can this bring model checking and deduction into one integrated verification approach?
- Is this suited for lightweight and/or heavyweight specification?
- Safety properties only? Security too?
- Can be used for *separation of concerns*?
- ...

And now?

- Integrated approach from abstract model to code?
- Which formalisms?
- Tool-driven or theory-only?
- Remain with HAGRID or move on? Where?
- Next steps. Further meeting with a more concrete agenda?