A principled approach to REPL interpreters applied to eFLINT

L. Thomas van Binsbergen[1]

[1]Centrum Wiskunde & Informatica
l.t.van.binsbergen@cwi.nl

April 2020

Section 1

REPL theory

Language implementations often provide an interpreter with a Read-Eval-Print Loop.

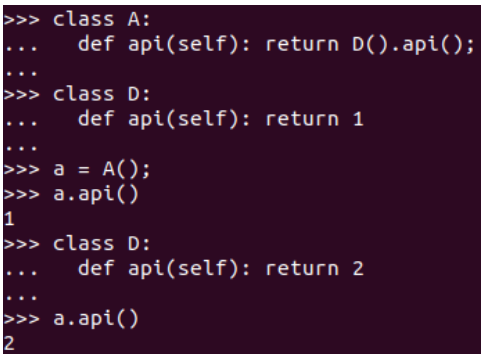


Figure: `python3`

Language implementations often provide an interpreter with a Read-Eval-Print Loop.

```
jshell> int global() { return 42; }
|  created method global()

jshell> int global(int x) { return 42 + x; }
|  created method global(int)

jshell> int global(int x, int y) { return x + y; }
|  created method global(int,int)

jshell> global();
$4 ==> 42

jshell> global(global(),global(8));
$5 ==> 92
```

Figure: `jshell`

What language are we interacting with exactly?

What language are we interacting with exactly?

REPL	syntax and semantics
`python3`	Every code fragment is a valid top-level **statement**, **expression** or **declaration** of a Python3 program
`ocaml`	Every code fragment is a valid **phrase** of an OCaml program, an OCaml program is a sequence of phrases
`ghci`	Every code fragment translates to a **statement** in the `IO()` monad, top-level **declarations** can be considered as prefixed with `let`, **expressions** can be considered as prefixed with `print`
`jshell`	More complicated to explain... In fact, I cannot find the details... there are **expressions**, **statements**, **variables**, **methods** and **classes**

Empirical analysis, conclusions

- Python3 and OCaml programs and REPL interactions are essentially the same
- `ghci` implements additional top-level constructs (syntax)
- `jshell` behaviour is hard to explain precisely in terms of Java

Empirical analysis, conclusions

- Python3 and OCaml programs and REPL interactions are essentially the same
- `ghci` implements additional top-level constructs (syntax)
- `jshell` behaviour is hard to explain precisely in terms of Java

Research questions

- What do languages like Python3 and OCaml have in common?
- How to develop a REPL for language $\mathcal{L}$ so that we can formally state:
  - which syntactic constructs are accepted as code fragments, and
  - what information is propagated between these code fragments?

Empirical analysis, conclusions

- Python3 and OCaml programs and REPL interactions are essentially the same
- `ghci` implements additional top-level constructs (syntax)
- `jshell` behaviour is hard to explain precisely in terms of Java

Research questions

- What do languages like Python3 and OCaml have in common?
- How to develop a REPL for language $\mathcal{L}$ so that we can formally state:
  - which syntactic constructs are accepted as code fragments, and
  - what information is propagated between these code fragments?

These questions are about language design...

What do Python3 and OCaml have in common?

A sequential language is a language in which $p_1; p_2$ is a (syntactically) valid program iff p_1 and p_2 are valid programs and iff $p_1; p_2$ is equivalent to 'doing' p_1 and then p_2

What do Python3 and OCaml have in common?

A sequential language is a language in which $p_1; p_2$ is a (syntactically) valid program iff p_1 and p_2 are valid programs and iff $p_1; p_2$ is equivalent to 'doing' p_1 and then p_2

Formally

A language[1] $L = \langle P, \Gamma, \gamma^0, I \rangle$ is *sequential* if there is an operator ; such that for every $p_1, p_2 \in P$ and $\gamma \in \Gamma$ it holds that $p_1; p_2 \in P$ and that $I_{p_1;p_2}(\gamma) = (I_{p_2} \circ I_{p_1})(\gamma)$

[1] This notion of language is limited to deterministic languages

What do Python3 and OCaml have in common?

A sequential language is a language in which $p_1; p_2$ is a (syntactically) valid program iff p_1 and p_2 are valid programs and iff $p_1; p_2$ is equivalent to 'doing' p_1 and then p_2

Formally

A language[1] $L = \langle P, \Gamma, \gamma^0, I \rangle$ is *sequential* if there is an operator ; such that for every $p_1, p_2 \in P$ and $\gamma \in \Gamma$ it holds that $p_1; p_2 \in P$ and that $I_{p_1;p_2}(\gamma) = (I_{p_2} \circ I_{p_1})(\gamma)$

- The combination of P and ; informs us of the syntactically valid code fragments
- The semantics of ; informs us of the details of context (γ) propagation

[1] This notion of language is limited to deterministic languages

Back to the example languages...

REPL	syntax and semantics
`python3`	Python3 can be shown to be sequential by choosing line-breaks as ;
`ocaml`	OCaml can be shown to be sequential by choosing `;;` as ;
`ghci`	A syntactically more lenient version of $\gg=$ can be taken as ; In other words, the monad instance of `IO()` gives the semantics of ;
`jshell`	How can we formalize the syntax and semantics of JShell code fragments? This requires a choice for ; and a definition of its semantics

Back to the example languages...

REPL	syntax and semantics
`python3`	Python3 can be shown to be sequential by choosing line-breaks as ;
`ocaml`	OCaml can be shown to be sequential by choosing `;;` as ;
`ghci`	A syntactically more lenient version of $\gg=$ can be taken as ; In other words, the monad instance of `IO()` gives the semantics of ;
`jshell`	How can we formalize the syntax and semantics of JShell code fragments? This requires a choice for ; and a definition of its semantics

The paper shows how one might answer this question by giving a *formal* extension of MiniJava and building a JShell-like REPL on top of this extension

Contributions

1. The paper proposes a principled methodology for developing a REPL for $\mathcal{L}$:
   a. the methodology involves proving or extending $\mathcal{L}$ to be *sequential*, thus formalizing the syntax and semantics of code fragments
   b. the methodology lays out a generic architecture in which code fragments are run using an *exploring interpreter* that enables arbitrary backtracking
   c. the exploring interpreter is derived from a 'definitional' interpreter for $\mathcal{L}$, making minimal assumptions about the technique with which the definitional interpreter is defined
2. The benefits of the architecture are demonstrated through prototypes

The *reachability graph* for a configuration $\gamma \in \Gamma$ of a language $\langle P, \Gamma, \gamma^0, I \rangle$ contains all the configurations γ' that are reachable by executing programs $p \in P$ using I. Nodes are configurations, edges are labelled with programs

The *reachability graph* for a configuration $\gamma \in \Gamma$ of a language $\langle P, \Gamma, \gamma^0, I \rangle$ contains all the configurations γ' that are reachable by executing programs $p \in P$ using I. Nodes are configurations, edges are labelled with programs

An *exploring interpreter* for a language $\langle P, \Gamma, \gamma^0, I \rangle$ is an algorithm constructing a subgraph of the reachability graph from γ^0 by performing one of the following actions:

The *reachability graph* for a configuration $\gamma \in \Gamma$ of a language $\langle P, \Gamma, \gamma^0, I \rangle$ contains all the configurations γ' that are reachable by executing programs $p \in P$ using I. Nodes are configurations, edges are labelled with programs

An *exploring interpreter* for a language $\langle P, \Gamma, \gamma^0, I \rangle$ is an algorithm constructing a subgraph of the reachability graph from γ^0 by performing one of the following actions:

Algorithm

- **execute**(p): take $\gamma' = I_p(\gamma)$ and (p given as input, γ current context):
  - add γ' to the set of nodes (if new),
  - add $\langle \gamma, p, \gamma' \rangle$ to the set of edges (if new),
  - return the graph rooted at γ' as a response.
- **revert**(γ): take γ as the new current configuration (γ given as input, must be in the graph) and return the graph rooted at γ as a response.
- **display**: produce a structured representation of the current graph, distinguishing the current configuration in the graph from the other configurations.

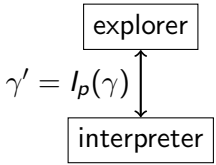


Figure: Generic architecture for REPLs

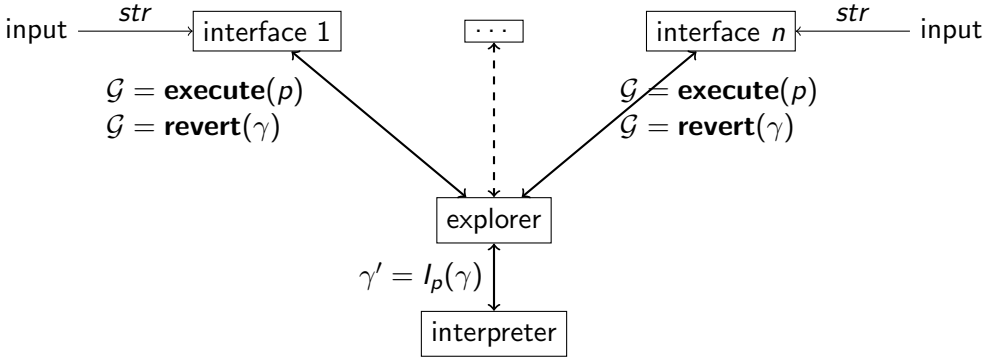


Figure: Generic architecture for REPLs

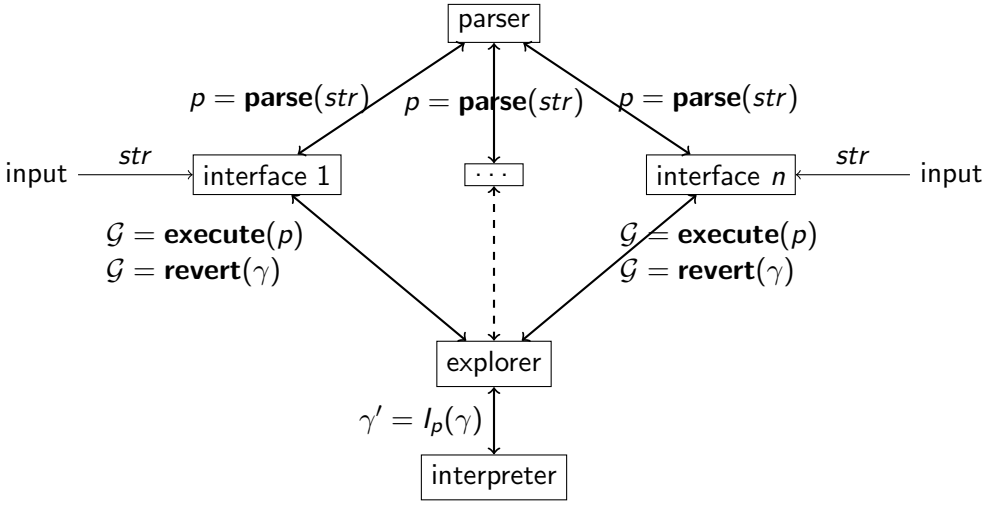


Figure: Generic architecture for REPLs

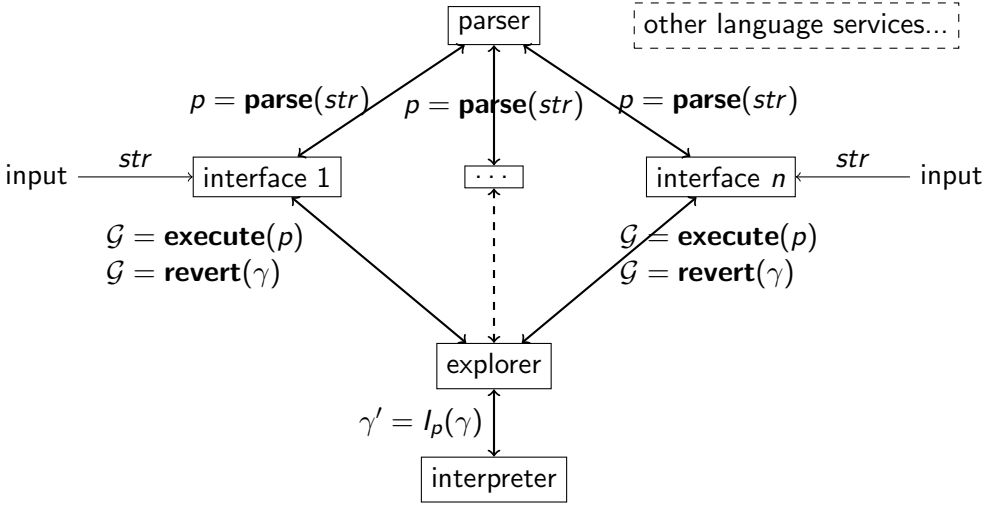


Figure: Generic architecture for REPLs

Section 2

Applications to eFLINT

1. An exploring interpreter has been defined on top of the eFLINT interpreter
    - enables backtracking for manual exploration and (programmatic) simulation

1. An exploring interpreter has been defined on top of the eFLINT interpreter
   - enables backtracking for manual exploration and (programmatic) simulation
2. eFLINT has been extended to a more general, sequential variant
   - type-declarations as phrases enable dynamic policy construction

1. An exploring interpreter has been defined on top of the eFLINT interpreter
   - enables backtracking for manual exploration and (programmatic) simulation
2. eFLINT has been extended to a more general, sequential variant
   - type-declarations as phrases enable dynamic policy construction
3. Two interfaces have been defined on top of the exploring interpreter
   - `eflint-repl`: command line tool for interacting with the explorer
   - `eflint-server`: server that listens on a port for incoming phrases

1. An exploring interpreter has been defined on top of the eFLINT interpreter
   - enables backtracking for manual exploration and (programmatic) simulation
2. eFLINT has been extended to a more general, sequential variant
   - type-declarations as phrases enable dynamic policy construction
3. Two interfaces have been defined on top of the exploring interpreter
   - `eflint-repl`: command line tool for interacting with the explorer
   - `eflint-server`: server that listens on a port for incoming phrases

Valid phrases: type-declarations, initialization, action/event invocation, queries

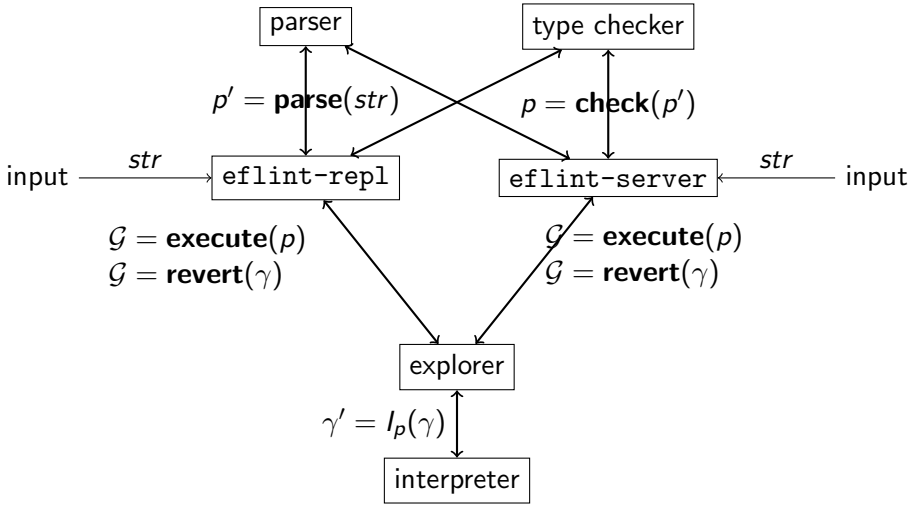


Figure: eFLINT REPL architecture

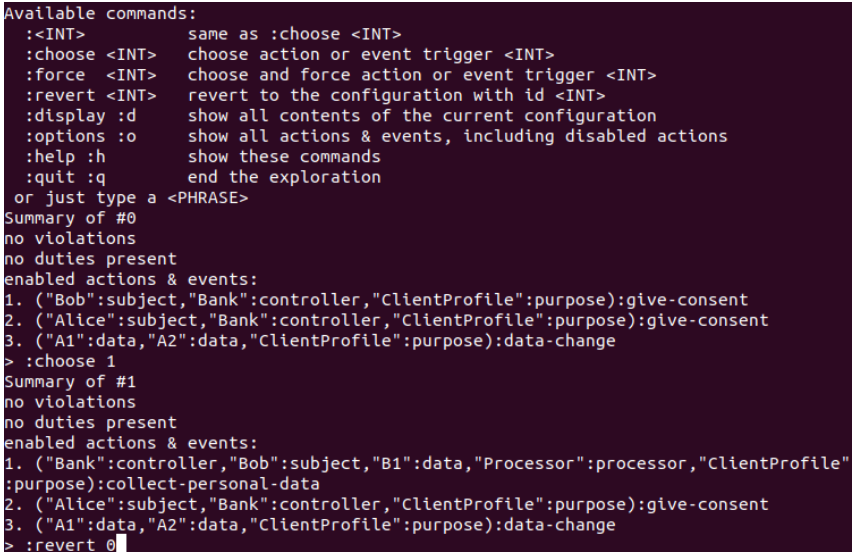
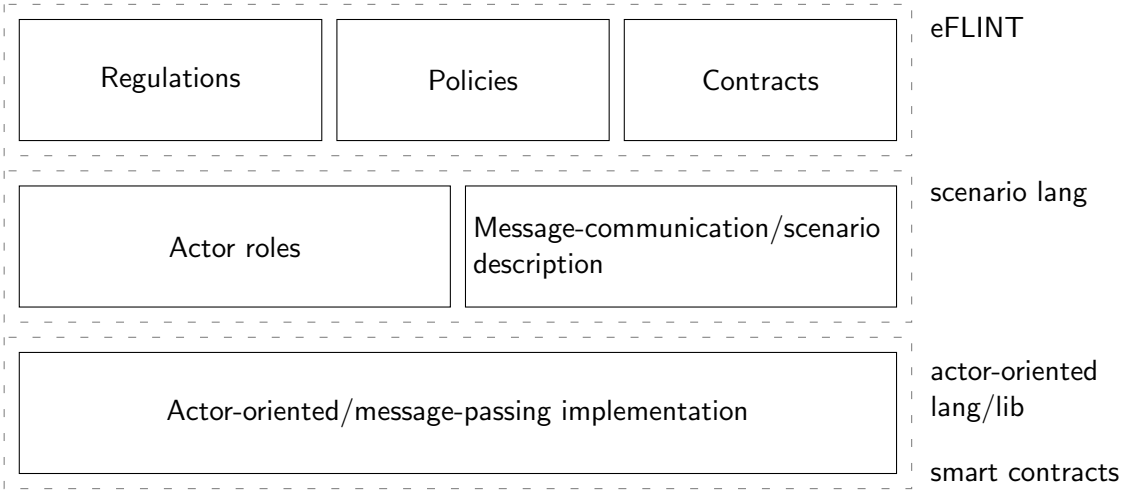


Figure: `eflint-repl`

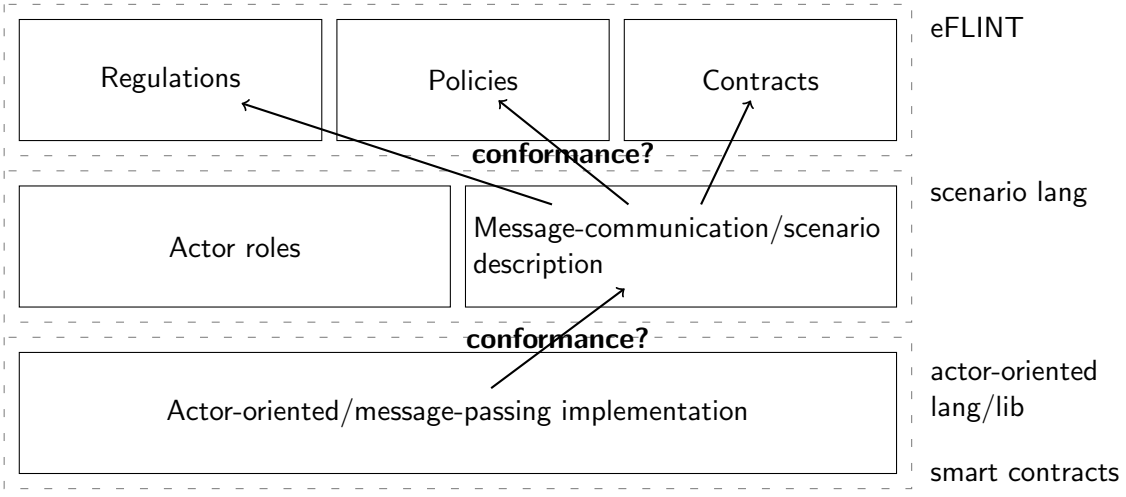
Section 3

`eflint-server` – KYC example

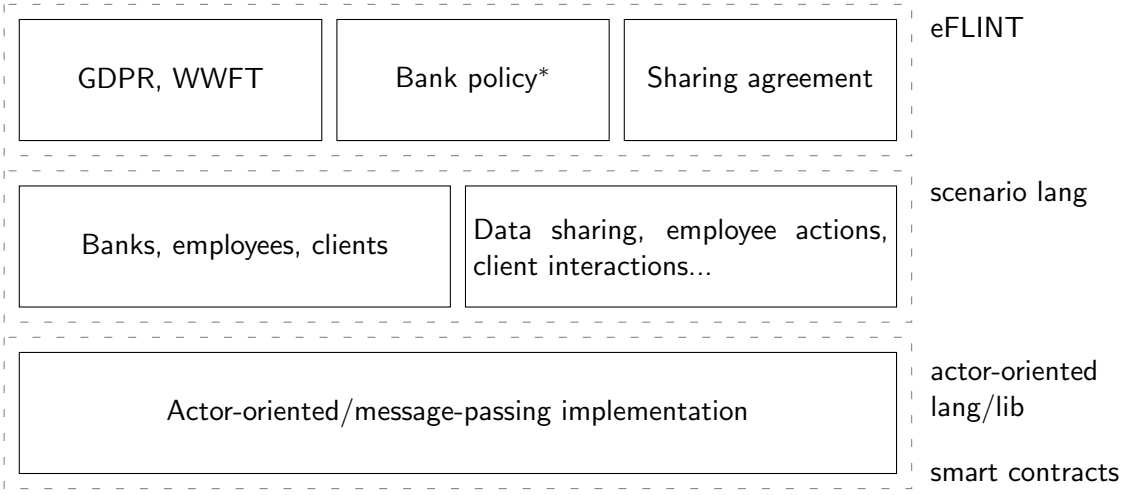
General approach



General approach



KYC case study



Employees' risk assessment experiment

The experiment involves

- one bank policy, one bank behaviour
- multiple employee and client behaviours
- implementation: eFLINT (policy, using `eflint-server`) and Java (behaviours)

Employees' risk assessment experiment

The experiment involves

- one bank policy, one bank behaviour
- multiple employee and client behaviours
- implementation: eFLINT (policy, using `eflint-server`) and Java (behaviours)

Bank policy

- The bank assigns a particular risk (low, medium, high) to SIB and country values
- When computing the risk of a client, an employee must assign a risk at least as high as the SIB- and country-risk for that client

```
1  Act assign-risk
2    Actor         employee
3    Recipient     client
4    Related to    risk
5    Conditioned by  country-of(country = country)        && sib-of(sib = sib)
6                 &&  country-risk(risk = country_risk)   && sib-risk(risk = sib_risk)
7                 &&  risk >= country_risk                && risk >= sib_risk
8    Terminates    risk-of(risk = risk') When risk' != risk
9    Creates       risk-of()
```

Actor roles

```
bank {
  client   : receive_application()
  employee: interview_completed(profile, boolean)
          , receive_risk_result(profile, risk)
}

employee {
  bank   : interview_client(profile)
         , perform_risk_analysis(profile)
  client: receive_sib(string)
        , receive_country(string)
        , receive_email_address(string)
}

client {
  employee: send_sib()
          , send_country()
          , send_email_address()
  bank     : receive_accept()
          , receive_reject()
}
```

Scenario fragment

```
1  ...
2  IF received_sib || received_country
3   policy.phrase("+assign-risk(employee = <employee.id>,client = <client.id>)")
4   bank.perform_risk_analysis(profile) -> employee
5
6   EITHER
7    ASSERT policy.phrase("?Enabled(assign-risk(<employee.id>, <client.id>, low))")
8    policy.phrase("assign-risk(<employee.id>, <client.id>, low)")
9    employee.receive_risk_analysis(profile,low) -> bank
10  OR
11   ASSERT policy.phrase("?Enabled(assign-risk(<employee.id>, <client.id>, medium))")
12   policy.phrase("assign-risk(<employee.id>, <client.id>, medium)")
13   employee.receive_risk_analysis(profile,medium) -> bank
14  OR
15   ASSERT policy.phrase("?Enabled(assign-risk(<employee.id>, <client.id>, high))")
16   policy.phrase("assign-risk(<employee.id>, <client.id>, high)")
17   employee.receive_risk_analysis(profile,high) -> bank
18  END
19 END
20 ...
```

Preventing or detecting incorrect risk assignments

```
1  class Bank {
2    ...
3    public void receive_risk_result(Employee employee, ClientProfile profile, Risk risk) {
4      Risk old_value = kyc_score.get(profile.getName());
5      if (old_value == null || !old_value.equals(risk)) {
6        try { // see if this risk assignment is compliant with the bank's policy
7          eflint.action(new Value("assign-risk"
8                        ,new Value("employee", employee.getName())
9                        ,new Value("client", profile.getName())
10                       ,new Value("risk", risk.ordinal())));
11         kyc_score.put(profile.getName(), risk);
12       } catch (InvalidEFlintTransitionAttempt e) {
13         System.out.println(employee.getName() + " gave " + profile.getName() + " a risk value of " +
       risk + " which is too low");
14       }
15     }
16   }
17   ...
18 }
```

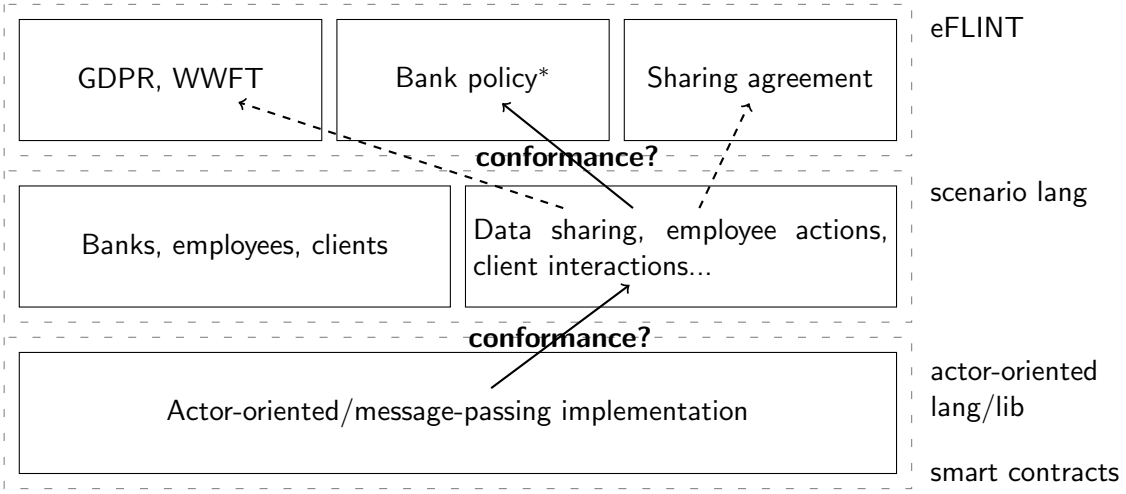
- The bank class starts and kills an `eflint-server`; queries and action are sent to the server as 'phrases' for prevention and detection

Compliant by construction

```
1  class DiligentEmployee {
2    ...
3    public void perform_risk_analysis(Bank bank, ClientProfile profile) {
4     Set<Risk> options = new HashSet<Risk>();
5     for (Risk risk : Risk.values()) {
6      if(bank.eFLINT().enabled(new Value("assign-risk"
7                                ,new Value("employee", this.getName())
8                                ,new Value("client", profile.getName())
9                                ,new Value("risk", risk.ordinal()))))) {
10       options.add(risk);
11     }
12    }
13    bank.receive_risk_result(this, profile, choose_risk(options));
14   }
15
16   public Risk choose_risk(Set<Risk> options){
17     List<Risk> risk_list = new ArrayList<Risk>(options);
18     Collections.sort(risk_list);
19     return risk_list.get(0);
20   }
21   ...
22 }
```

- By construction, the employee only considers valid risk assignments

KYC case study



Notes on the implementation

- The bank loads its own SIB and country risk assignments from a `.csv` file
- All client input is loaded from an input file, making it possible to try different scenarios without changing code (to some extent)
- A similar experiment has been done based on a simple sharing agreement and different bank behaviours
- Extensions needed: GDPR, richer bank policies, realistic sharing agreement, etc.
- Java implementation not concurrent, although adaptable to the AKKA library

Conclusions

- The REPL theory developed at CWI gives a new perspective on eFLINT,
- resulting in a better `eflint-repl` and a new `eflint-server`,
- and a powerful mechanism for other software to interact with eFLINT

Next steps

- Design, implementation and application of the scenario language
- Exploring ways of ensuring/confirming the conformance of implementations with respect to the sets of behaviours encoded in a scenario description
- Further development of the KYC case study (need all of your help here!):
  - actor-oriented and smart contract based implementations of behaviour
  - inclusion of eFLINT specifications for GDPR and sharing agreements
  - scenario descriptions of data-sharing activities

Section 4

`eflint-server` demo – voting example

eFLINT (1)

A duty indicate that its holder ought to perform some action:

```
1 Duty cast-vote-duty
2   Holder citizen
3   Claimant administrator
4   Related to weeknr'
5   Violated when weeknr' < weeknr
```

- A duty-type declaration is a fact-type declaration with a record-type

```
1 Act enable-vote
2   Actor administrator
3   Recipient citizen
4   Conditioned by !voter(citizen) && !vote-concluded()
5   Creates voter(citizen),
6           cast-vote-duty(citizen,administrator,weeknr),
7           (Foreach candidate : cast-vote(citizen,administrator,candidate))
```

- An act-type declaration is a fact-type declaration with a record-type

eFLINT (2)

```
Act cast-vote
  Actor citizen
  Recipient administrator
  Related to candidate
  Conditioned by voter(citizen) && !has-voted(citizen)
  Creates vote(citizen,candidate)
  Terminates cast-vote-duty When cast-vote-duty.citizen == citizen
```

```
Act declare-winner
  Actor administrator
  Recipient candidate
  Conditioned by
      !vote-concluded()
   && most-votes-on(candidate)
  Creates winner(candidate)
```

Java (1)

```
1  public Voting() {
2    this.monitor = new EFLINT("src/voting/voting.eflint");
3  }
```

```
1  public void register_voter(String name) {
2    registered_voters.add(name);
3    try {
4      monitor.action("enable-vote", "Admin", name).print_violations();
5    }catch(InvalidEFlintTransitionAttempt e) {}
6  }
```

```
1  public void vote(String voter, String candidate) {
2    try {
3      monitor.action("cast-vote", voter, "Admin", new Value("candidate",
       candidate)).print_violations();
4      vote_count.put(candidate, vote_count.getOrDefault(candidate,0) + 1);
5    }catch(InvalidEFlintTransitionAttempt e) {
6      System.out.println("no voting power for " + voter);
7    }
8  }
```

Java (2)

```
1   public String find_winner() {
2     String winner = null;
3     int winning_votes = 0;
4     for (Entry<String,Integer> entry : vote_count.entrySet()) {
5       if(entry.getValue() > winning_votes) {
6         winner = entry.getKey();
7         winning_votes = entry.getValue();
8       }
9     }
10    if (winner != null) {
11      try {
12        monitor.action("declare-winner", "Admin", winner).print_violations();
13        return winner;
14      }catch(InvalidEFlintTransitionAttempt e) {
15        System.out.println(winner + " cannot be declared winner");
16      }
17    }
18    return null;
19  }
```

Java (3)

```
1 public static ServerState weeks_later(EFLINT m, int weeks) {
2   for (int i = 0; i < weeks; i++) {
3     if (m.test_present(new Value("weeknr", i))) {
4       m.terminate(new Value("weeknr", i));
5       break;
6     }
7   }
8   return m.create(new Value("weeknr", weeks));
9 }
```

Java (4)

```
1  public static void find_winner(Voting v) {
2    for (String candidate : v.vote_count.keySet()) {
3      if (v.eFLINT().test_present(new Value("most-votes-on", candidate))) {
4        System.out.println(candidate); return;
5      }
6    }
7    System.out.println("no possible winner yet");
8  }
```

A principled approach to REPL interpreters applied to eFLINT

L. Thomas van Binsbergen[1]

[1]Centrum Wiskunde & Informatica
l.t.van.binsbergen@cwi.nl

April 2020