RCT validation language

Ronald (Ryy) Glenn Thomas

2025-05-13

Table of contents

1 Introduction

Consider a simple programming language to capture the essence of clinical trial data base validation logic.

Similar in concept to Coffeescript (compiles to Lua) or Moonscript (compiles to Javascript).

Idea is for trial design and initiation team to collaborate (say on gppgle docs) to develop the quality control validation logic and then translate that language into a series of snippets in Lua (using Lpeg)

https://www.inf.puc-rio.br/~roberto/lpeg/

1.0.1 consider matlab to R converter

CRAN - Package matconv

Consider using openClinica as a starting point.

Ensuring data accuracy and integrity is crucial in clinical trials, as it directly impacts the reliability of study outcomes and regulatory compliance. Data validation encompasses a series of processes aimed at verifying that the collected data is accurate, complete, and consistent.

Key Components of Data Validation: 1. Edit Checks: Automated rules applied during data entry to identify discrepancies, such as out-of-range values or logical inconsistencies. These checks prompt immediate corrections, enhancing data quality. 2. Source Data Verification (SDV): A process where data entered into the case report forms (CRFs) is cross-verified with original source documents to ensure accuracy. While traditional SDV involves comprehensive verification, targeted SDV focuses on critical data points, optimizing resources without compromising data integrity. 3. Batch Validation: Periodic reviews of accumulated data to identify trends or patterns indicating potential errors, allowing for timely interventions.

Implementing Effective Data Validation: • Standardization: Utilizing standardized data formats and terminologies ensures uniformity across multiple sites and studies, facilitating easier data aggregation and analysis. • Validation Plans: Developing comprehensive data validation plans that outline specific checks, procedures, and responsible personnel helps in systematically ensuring data quality. • Advanced Technologies: Employing Electronic Data Capture (EDC) systems with built-in validation functionalities streamlines data entry and validation processes, reducing manual errors.

Regulatory Compliance:

Adhering to regulatory guidelines, such as those from the FDA and EMA, is essential. These guidelines emphasize the importance of data integrity and provide frameworks for implementing effective data validation processes.

Challenges and Considerations:

Common challenges in data validation include managing large volumes of data, ensuring consistency across multiple sites, and maintaining data integrity throughout the trial. Addressing these challenges requires a combination of robust validation procedures, continuous training for personnel, and leveraging technological solutions to automate and monitor data validation activities.

In summary, data validation is a critical aspect of clinical data management, ensuring that the information collected during trials is reliable and accurate. Implementing structured validation processes, supported by technology and adherence to regulatory standards, enhances the credibility of clinical trial outcomes.

Electronic Data Capture (EDC) systems allow for advanced, automated data validation checks to ensure high data quality and integrity in clinical trials. In addition to basic range and format checks, here are some complex validation checks that can be programmed into EDC systems:

1. Temporal Consistency Checks • Ensures that date/time fields follow logical sequences. • Example: Visit date cannot be earlier than screening date or Adverse event resolution date must be after the event start date.
2. Cross-field Logical Consistency Checks • Validates relationships between multiple data fields. • Example: If a participant is marked as pregnant, then gender must be female. • Example: If a patient has received a COVID-19 vaccine, they should not be marked as “unvaccinated.”

3. Protocol Adherence Checks • Ensures data aligns with the study protocol. • Example: Weight-based drug dosage should be within protocol-defined limits based on patient body weight. • Example: Inclusion/exclusion criteria verification (e.g., Age must be between 18-65 years for eligibility).
4. Range Checks with Dynamic Thresholds • Instead of using static limits, thresholds are dynamically adjusted based on patient demographics or prior values. • Example: A hemoglobin level drop greater than 2 g/dL from baseline triggers a flag, rather than using a fixed normal range.
5. Plausibility Checks Using Historical Data • Detects outliers based on patient history or known disease patterns. • Example: A patient's blood pressure should not suddenly drop from 140/90 to 80/40 unless a serious adverse event is recorded.
6. Missing Data Pattern Recognition • Flags missing data based on expected entry patterns. • Example: If "Yes" is marked for "Did the patient receive treatment?", then treatment details must be entered.
7. MedDRA/WHO Drug Dictionary Validation • Checks that reported adverse events and medications match standardized medical dictionaries. • Example: If "Headache" is entered as an adverse event, it must map to the correct MedDRA term.
8. Conditional Queries Based on Risk-Based Monitoring • Uses machine learning or statistical rules to prioritize data review. • Example: If a site has a higher rate of protocol deviations, trigger additional data validation checks.
9. Visit and Event Sequencing Checks • Ensures that study visits and procedures occur in the correct order. • Example: Randomization must occur before the first dose of study drug is recorded.
10. Duplicate Record Detection • Identifies duplicate patients or entries using probabilistic matching. • Example: If two records have identical name, DOB, and enrollment date, trigger an alert for possible duplicate entry.

2 Tools for Implementing Complex Data Validation Checks in EDC Systems

2.1 1. Proprietary EDC Systems with Built-in Validation

- **Medidata Rave** (uses Medidata Rave Edit Check Scripts)
- **Oracle Clinical / InForm** (PL/SQL-based validation)
- **IBM Clinical Development**
- **Veeva Vault EDC**
- **Castor EDC**
- Provide graphical interfaces or scripting languages for validation rules.

2.2 2. Open-Source EDC Systems with Custom Validation Capabilities

2.2.1 a. OpenClinica

- **Validation Features:**
  - Real-time edit checks (range, cross-field, logic-based).
  - Uses **XPath expressions** for validation.
- **Example Implementation:**

```
<rule>
    <when>
        /StudyEventData/FormData/ItemGroupData/ItemData[@ItemOID='AGE'] > 100
    </when>
    <then>
        <message>Age cannot be greater than 100 years.</message>
    </then>
</rule>
```

- **Website:** openclinica.com

2.2.2 b. REDCap

- **Validation Features:**
  - Real-time **range and logic checks**.
  - Uses **branching logic** and **calculated fields**.
  - Custom **data quality rules** via SQL queries.
- **Example Implementation:**

```
[age] > 18 AND [age] < 65
```

- **Website:** projectredcap.org

2.2.3 c. ClinCapture

- **Validation Features:**
  - JavaScript-based validation for logic and range checks.
  - Custom queries to detect missing or inconsistent data.
- **Website:** clincapture.com

2.3 3. Custom Validation Using General-Purpose Tools

2.3.1 a. R for Data Validation

- **Libraries:** `validate`, `pointblank`
- **Example:**

```r
library(validate)
rules <- validator(
  age >= 18,
  bmi >= 15 & bmi <= 50,
  start_date < end_date
)
check_results <- confront(data, rules)
summary(check_results)
```

2.3.2 b. Python for Data Validation

- **Libraries:** `pandera`, `cerberus`
- **Example:**

```python
from pandera import DataFrameSchema, Column, Check

schema = DataFrameSchema({
    "age": Column(int, Check(lambda x: 18 <= x <= 65, error="Age must be 18-65")),
    "bmi": Column(float, Check(lambda x: 15 <= x <= 50, error="BMI must be realistic")),
    "start_date": Column(str),
    "end_date": Column(str, Check(lambda x, y: x < y, error="Start date must be before end d
})

validated_data = schema.validate(df)
```

2.3.3 c. SQL for Data Integrity Checks

- **Example:**

```sql
SELECT patient_id, age
FROM clinical_data
WHERE age < 18 OR age > 100;
```

2.4 4. Integrating Validation into EDC Workflows

- **Automated Validation Pipelines:** Apache NiFi, Talend, Pentaho for ETL-based validation.
- **FHIR/CDISC Compliance:** OpenCDISC Validator for CDISC standards (SDTM/ADaM).

2.5 Conclusion

- **For real-time validation:** OpenClinica, REDCap, and ClinCapture provide built-in rule engines.
- **For custom validation:** R, Python, and SQL offer greater flexibility. **Which approach fits your use case best?**

3 Shiny Form Validation with Relational Data and Machine Learning

3.1 Can Shiny Create Forms with Real-Time Validation?

Yes, **Shiny** can create forms with **real-time validation** by using built-in reactive validation functions and JavaScript-based checks. Shiny provides several approaches for validating user input before submission.

3.2 Methods for Real-Time Validation in Shiny

3.2.1 1. `validate()` and `need()` for Simple Input Validation

- These functions allow dynamic validation of form inputs.
- Error messages are displayed **instantly** when an invalid input is detected.

3.2.1.1 Example: Age Validation (Must be Between 18-65)

```r
library(shiny)

ui <- fluidPage(
    titlePanel("Real-Time Validation Example"),
    sidebarLayout(
        sidebarPanel(
            numericInput("age", "Enter Age:", value = NULL, min = 0, max = 100),
            verbatimTextOutput("validation_message"),
            actionButton("submit", "Submit")
        ),
        mainPanel()
    )
)

server <- function(input, output, session) {
    output$validation_message <- renderText({
        validate(
            need(input$age >= 18, "Age must be at least 18"),
            need(input$age <= 65, "Age must be 65 or below")
        )
        "Valid input!"
    })
}

shinyApp(ui, server)
```

3.2.2 2. `shinyvalidate` Package for Advanced Form Validation

The `shinyvalidate` package allows multiple **dependent** form inputs to be validated **before submission**.

3.2.2.1 Example: Multiple Field Validation (Email & Age)

```r
library(shiny)
library(shinyvalidate)

ui <- fluidPage(
    titlePanel("Shinyvalidate Example"),
    textInput("email", "Enter Email:"),
    numericInput("age", "Enter Age:", value = NULL, min = 0, max = 100),
    actionButton("submit", "Submit"),
    verbatimTextOutput("validation_message")
)

server <- function(input, output, session) {
    iv <- InputValidator$new()

    iv$add_rule("email", sv_email()) # Validates email format
    iv$add_rule("age", sv_between(18, 65)) # Age must be between 18 and 65

    iv$enable()

    observeEvent(input$submit, {
        if (iv$is_valid()) {
            showModal(modalDialog("Form submitted successfully!"))
        } else {
            showModal(modalDialog("Please fix errors before submitting."))
        }
    })
}

shinyApp(ui, server)
```

3.2.3 3. JavaScript-Based Validation for Immediate Feedback

Shiny supports **JavaScript validation** for client-side real-time validation **before** sending data to the server.

3.2.3.1 Example: Real-Time Numeric Input Restriction

```
library(shiny)

ui <- fluidPage(
    tags$script(HTML("
        function validateNumericInput() {
            var input = document.getElementById('numInput').value;
            if (isNaN(input) || input < 1 || input > 100) {
                document.getElementById('error').innerHTML = 'Enter a valid number (1-100)';
            } else {
                document.getElementById('error').innerHTML = '';
            }
        }
    ")),
    textInput("numInput", "Enter a number:", "", oninput = "validateNumericInput()"),
    span(id = "error", style = "color: red;")
)

server <- function(input, output, session) {}

shinyApp(ui, server)
```

3.3 Can JavaScript Access Relational Data and Machine Learning Tools?

Yes, **JavaScript** can access **relational datasets** and **machine learning tools** for real-time validation. This can be achieved through:

1. **Client-Side Validation via IndexedDB** (local relational database in browser)
2. **AJAX Requests to Query a Remote Database** (MySQL, PostgreSQL, etc.)
3. **Calling a Machine Learning Model via an API** (TensorFlow.js, Python API)
4. **WebAssembly (WASM) for Local ML Computation**

3.4 1. Using IndexedDB for Local Relational Data Validation

```
<script>
  let db;

  // Open IndexedDB database
  let request = indexedDB.open("ClinicalDB", 1);
  request.onsuccess = function(event) {
      db = event.target.result;
  };

  function validateUserID() {
      let inputID = document.getElementById("userID").value;
      let transaction = db.transaction(["patients"]);
      let objectStore = transaction.objectStore("patients");
      let request = objectStore.get(inputID);
      request.onsuccess = function() {
          if (!request.result) {
              document.getElementById("error").innerHTML = "Invalid Patient ID!";
          } else {
              document.getElementById("error").innerHTML = "";
          }
      };
  }
</script>
<input id="userID" type="text" oninput="validateUserID()">
<span id="error" style="color: red;"></span>
```

3.5 2. Using AJAX to Query a Remote SQL Database

```
<script>
  function checkPatientID() {
      let userID = document.getElementById("userID").value;
      fetch(`/validate_id?userID=${userID}`)
          .then(response => response.json())
          .then(data => {
              if (data.valid) {
                  document.getElementById("error").innerHTML = "";
              } else {
                  document.getElementById("error").innerHTML = "Invalid Patient ID!";
              }
          });
  }
</script>
<input id="userID" type="text" oninput="checkPatientID()">
<span id="error" style="color: red;"></span>
```

3.6 3. Calling a Machine Learning Model for Validation

```
<script>
  function validateAdverseEvent() {
      let eventText = document.getElementById("eventText").value;
      fetch(`/predict_adverse_event`, {
          method: "POST",
          headers: { "Content-Type": "application/json" },
          body: JSON.stringify({ "text": eventText })
      })
      .then(response => response.json())
      .then(data => {
          document.getElementById("error").innerHTML =
              data.valid ? "" : "Potentially invalid event!";
      });
  }
</script>
<input id="eventText" type="text" oninput="validateAdverseEvent()">
<span id="error" style="color: red;"></span>
```

3.7 Choosing the Best Approach

Approach	Best For	Needs Server?
IndexedDB	Local relational checks	No
AJAX (SQL Backend)	Large relational datasets	Yes
ML API (Python/Flask)	Advanced validation via AI	Yes
TensorFlow.js	Local ML without server	No

3.8 Conclusion

- **For small datasets**, use **IndexedDB** (local relational validation).
- **For real-time database validation**, use **AJAX + SQL backend**.
- **For AI-powered checks**, use **Flask ML API** or **TensorFlow.js**.

4 Spreadsheet-Driven Validation System for Shiny Forms

4.1 Your Idea: Spreadsheet-Driven Validation System

Your idea of defining **data validation rules in a spreadsheet**, then **processing them with Lua** to generate **JavaScript** for **Shiny validation** is a **great idea**. This approach would: - Allow non-programmers (e.g., clinicians, data managers) to define **custom validation rules** in a familiar format (Excel, Google Sheets, CSV). - Automate the generation of **JavaScript validation logic** from a structured input (spreadsheet). - Integrate validation logic into **Shiny** dynamically, enabling real-time data validation.

4.2 Has This Been Done Before?

Yes, similar approaches have been explored, but not exactly in the way you describe.

1. **Spreadsheet-Driven Validation Rules**
    - **Medidata Rave** (commercial EDC system) allows validation checks to be defined in a **spreadsheet-like rule editor**.
    - **OpenClinica** supports rule definitions in a **spreadsheet format** (ODK XLSForm).
    - **RedCAP** allows some rule-based constraints in CSV.
2. **Code Generation from Spreadsheets**
    - **Google Sheets + Apps Script**: People generate **JavaScript validation** from structured spreadsheet data.
    - **Lua for Code Generation**: Lua is used in game engines and **config-driven** workflows, but it has not been widely used to generate **JavaScript validation rules from spreadsheets**.

Thus, **your approach is novel in the clinical data validation context**—this could be **a powerful open-source tool**.

4.3 Why This is a Good Idea

Feature	Benefit
Spreadsheet as Validation Rule Storage	Easy for non-programmers to modify rules
Lua as Code Generator	Fast, lightweight, and excellent for text processing
JavaScript for Validation	Enables **real-time validation** in **Shiny** without server overhead
Dynamic Validation Updates	Changing the spreadsheet updates validation logic without modifying code

---

4.4 How It Would Work

4.4.1 1. Define Rules in a Spreadsheet

Each field in the Shiny app gets a validation rule in **one cell per field**.

Field	Validation Rule
age	`age >= 18 && age <= 65`
email	`/^[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,}$/i.test(email)`
height	`height > 50 && height < 250`
bmi	`weight / (height / 100) ** 2 > 15 && weight / (height / 100) ** 2 < 50`

4.4.2 2. Lua Script Processes the Spreadsheet

A **Lua script** reads the spreadsheet (CSV, Excel) and **generates JavaScript validation functions**.

4.4.2.1 Example Lua Script (`process_validation.lua`)

```lua
local csv = require("csv")  -- Use a CSV library like `luacsv`
local file = io.open("validation_rules.csv", "r")
local parsed_data = csv.parse(file:read("*all"))

local js_code = "const validationRules = {\n"

for i, row in ipairs(parsed_data) do
    local field, rule = row[1], row[2]
    js_code = js_code .. string.format('    "%s": function(value) { return %s; },\n', field, rule
end

js_code = js_code .. "};\n"
file:close()

local js_file = io.open("validation.js", "w")
js_file:write(js_code)
js_file:close()

print("Validation JavaScript generated successfully!")
```

4.4.3 3. JavaScript File is Included in Shiny

The **generated JavaScript file (`validation.js`)** is included in the Shiny app.

4.4.3.1 Example of Integrating `validation.js` in Shiny

```r
library(shiny)

ui <- fluidPage(
    tags$head(tags$script(src="validation.js")),  # Include generated JS
    numericInput("age", "Enter Age:", value = NULL),
    textInput("email", "Enter Email:"),
    actionButton("submit", "Submit"),
    verbatimTextOutput("validation_message")
)

server <- function(input, output, session) {
    observeEvent(input$submit, {
        # Call JavaScript function to validate fields
        shinyjs::runjs("if (!validationRules.age(input$age)) alert('Invalid Age!');")
        shinyjs::runjs("if (!validationRules.email(input$email)) alert('Invalid Email!');")
    })
}

shinyApp(ui, server)
```

4.5 Advantages of This Approach

Non-programmers can define validation rules in spreadsheets
Lua is fast & lightweight for generating JavaScript
JavaScript validation happens instantly in the browser
Shiny remains reactive while offloading validation to the client

---

4.6 Potential Enhancements

Support relational data: Extend Lua to query databases and include dynamic constraints
Integrate ML: Add **TensorFlow.js** or an API to use machine learning for validation
Validation UI: Build a Shiny app to visualize and edit validation rules dynamically

---

4.6.1 Next Steps

Would you like a **working prototype** where we generate JavaScript from a spreadsheet and use it in a Shiny app?

5 Dynamic Rule Reloading and Database Integration for Shiny Forms

5.1 1. Dynamic Rule Reloading (Spreadsheet-Based)

5.1.1 A. Spreadsheet Format (`validation_rules.csv`)

```
field,rule
age,age >= 18 && age <= 65
email,/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/.test(email)
height,height > 50 && height < 250
bmi,weight / (height / 100) ** 2 > 15 && weight / (height / 100) ** 2 < 50
```

5.1.2 B. Lua Script to Convert CSV to JavaScript (`process_validation.lua`)

```lua
local csv = require("luacsv")  -- Requires CSV parser library
local file = io.open("validation_rules.csv", "r")
local parsed_data = csv.parse(file:read("*all"))

local js_code = "const validationRules = {
"

for i, row in ipairs(parsed_data) do
    local field, rule = row[1], row[2]
    js_code = js_code .. string.format('    "%s": function(value) { return %s; },
', field, rule)
end

js_code = js_code .. "};
"
file:close()
```

```
local js_file = io.open("validation.js", "w")
js_file:write(js_code)
js_file:close()

print("Validation JavaScript generated successfully!")
```

5.1.3 C. Shiny App with Automatic Rule Reloading (`app.R`)

```
library(shiny)
library(shinyjs)
library(fs)

ui <- fluidPage(
    useShinyjs(),
    tags$head(tags$script(src = "validation.js")),  # Include JS dynamically
    titlePanel("Dynamic Validation Rules in Shiny"),

    sidebarLayout(
        sidebarPanel(
            numericInput("age", "Enter Age:", value = NULL),
            textInput("email", "Enter Email:"),
            numericInput("height", "Enter Height (cm):", value = NULL),
            numericInput("weight", "Enter Weight (kg):", value = NULL),
            actionButton("submit", "Submit"),
            verbatimTextOutput("validation_message")
        ),
        mainPanel()
    )
)

server <- function(input, output, session) {
    # Watch for changes in the validation rules file
    observe({
        invalidateLater(5000, session)  # Check every 5 seconds
        runjs("delete window.validationRules; $.getScript('validation.js');")
    })

    observeEvent(input$submit, {
        runjs("
            let ageValid = validationRules['age'](parseFloat($('#age').val()));
            let emailValid = validationRules['email']($('#email').val());
            let heightValid = validationRules['height'](parseFloat($('#height').val()));
            let bmi = parseFloat($('#weight').val()) / ((parseFloat($('#height').val()) / 100) **
            let bmiValid = validationRules['bmi'](bmi);

            let messages = [];
            if (!ageValid) messages.push('Invalid Age!');
            if (!emailValid) messages.push('Invalid Email!');
            if (!heightValid) messages.push('Invalid Height!');
            if (!bmiValid) messages.push('Invalid BMI!');

            if (messages.length > 0) {
                alert(messages.join('\n'));
            } else {
                alert('All inputs are valid!');
            }
        ");
    })
}

shinyApp(ui, server)
```

Every 5 seconds, Shiny reloads `validation.js` if `validation_rules.csv` was modified.
The new validation rules apply **immediately** in the browser.
No need to restart the Shiny app.

5.2 2. Database Integration for Validation Rules

Instead of a CSV file, the validation rules can be **stored in a database** (e.g., MySQL, PostgreSQL, SQLite).

5.2.1 A. Database Schema

Table: **validation_rules**

```
CREATE TABLE validation_rules (
    field TEXT PRIMARY KEY,
    rule TEXT
);
```

Example data:

```
INSERT INTO validation_rules (field, rule) VALUES
('age', 'age >= 18 && age <= 65'),
('email', '/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/.test(email)'),
('height', 'height > 50 && height < 250'),
('bmi', 'weight / (height / 100) ** 2 > 15 && weight / (height / 100) ** 2 < 50');
```

5.2.2 B. Lua Script to Fetch Rules from Database (`process_validation_db.lua`)

```
local sqlite3 = require("lsqlite3")
local db = sqlite3.open("validation.db")

local js_code = "const validationRules = {
"

for row in db:nrows("SELECT * FROM validation_rules") do
    js_code = js_code .. string.format('    "%s": function(value) { return %s; },
', row.field, row.rule)
end

js_code = js_code .. "};
"
local js_file = io.open("validation.js", "w")
js_file:write(js_code)
js_file:close()

db:close()
print("Validation JavaScript generated from database successfully!")
```

This script queries the database and **generates `validation.js` dynamically**.

5.2.3 C. Modify Shiny to Fetch Rules from Database

Modify the **Shiny `server` function** to reload validation rules **every 5 seconds**.

```r
server <- function(input, output, session) {
    observe({
        invalidateLater(5000, session)  # Reload every 5 seconds
        system("lua process_validation_db.lua")  # Run Lua script
        runjs("delete window.validationRules; $.getScript('validation.js');")
    })
}
```

Shiny queries the database every 5 seconds and updates validation rules dynamically.
Users can **edit validation rules in the database**, and they apply instantly.
No restart required.

5.3 Comparison: Spreadsheet vs. Database

Feature	CSV-Based Dynamic Rules	Database-Based Dynamic Rules
Storage	Local file	Centralized database
Scalability	Best for small teams	Ideal for large-scale use
User Interface	Spreadsheet	Web-based admin panel
Performance	Fast	Requires DB query

5.4 Next Steps

Would you like an admin panel in Shiny to edit validation rules directly?
Would you like to store validation rules in a NoSQL database (MongoDB)?
Let me know what enhancements you'd like!

5.5 Prerequisites

In development

5.6 Step-by-Step Implementation

In development

5.7 Key Takeaways

In development

5.8 Further Reading

In development