Alma Mater Studiorum · University of Bologna

SCHOOL OF SCIENCE
Bachelor's Degree in Computer Science

Stipula Workbench: An Integrated Environment to Design, Analyze, and Execute Stipula Code

Supervisor:
Prof.
Cosimo Laneve

Presented by:
Erik Dervishi

Second Session
Academic Year 2024-2025

Abstract

The Stipula language offers a powerful formal framework for defining digital legal contracts, but its syntax and concepts can represent a barrier for non-specialized users, such as legal professionals. This thesis presents the development of the Stipula Workbench, a web-based integrated development environment (IDE) designed to bridge this gap.

The Workbench abstracts the complexity of Stipula's syntax through a guided, form-based graphical interface, which allows users to build contracts by visually defining legal components like 'Parties', 'Assets', and 'Functions'. The environment provides immediate feedback by generating the corresponding Stipula code in real-time. Furthermore, it integrates static analysis tools to verify crucial properties such as liquidity and clause reachability, and it includes an interpreter to simulate contract execution.

The project's objective is to lower the entry barrier for creating secure and formally correct digital contracts, making the power of Stipula accessible to its target audience and promoting greater adoption of formal languages in the legal domain.

Contents

Chapter 1

Project Goals and Motivation

The formal methods community has made significant strides in creating languages capable of describing complex systems with mathematical precision. Stipula is one such language, offering a robust framework for defining the logic of digital legal contracts [6]. Its design, based on a state machine model and linear asset management, provides strong guarantees about a contract's behavior [12]. However, the very formalism that makes Stipula powerful also presents a significant usability challenge. Its syntax and operational semantics, while unambiguous to a computer, can be intimidating and non-intuitive for its target audience: legal professionals, academics, and students who may lack a background in software development. Prior to this project, the tools for working with Stipula were fragmented. A user would need to:

- Write Stipula code in a general-purpose text editor, without syntax highlighting or contextual help.
- Use a standalone command-line tool to run the static analyzer and check for properties like liquidity.
- Employ a separate command-line interpreter to simulate the contract's execution and test its logic.

This fragmented workflow created a high barrier to entry, hindering experimentation, learning, and adoption. The core motivation for this thesis was to address this challenge directly.

1.1 The Core Assignment: Creating a Unified Environment

The primary goal of this project was to design and implement the **Stipula Workbench**: a web-based, integrated development environment (IDE) that unifies these disparate

tools into a single, cohesive, and user-friendly application. The central idea was to move away from a code-first approach and instead offer a **concept-first** interface. Rather than expecting users to master Stipula's syntax, the Workbench would allow them to build contracts by manipulating high-level legal concepts that the language models, such as:

- The parties involved in the contract.
- The assets (e.g., money, tokens) to be managed.
- The rules and clauses (functions) that govern the contract's behavior.

By providing a graphical, form-based interface for these elements, the Workbench aims to abstract away the syntactic complexity of Stipula. The user interacts with intuitive UI components, and the corresponding formal code is generated automatically in real-time.

1.2 Key Objectives

To realize this vision, the project was guided by the following key objectives:

1. **Lower the Barrier to Entry:** Create an application that enables users with little to no programming experience to successfully design and analyze a valid Stipula contract.
2. **Provide a Seamless Workflow:** Integrate the editor, analyzer, and interpreter into a single interface. A user should be able to design, analyze, and test a contract without ever leaving the application.
3. **Offer Immediate Feedback:** Implement a live-preview model where the user's actions in the graphical editor are instantly reflected in the generated Stipula code. This tight feedback loop is crucial for learning and understanding the connection between the legal concepts and the formal syntax.
4. **Promote Correctness by Design:** By using a structured, form-based input method, the Workbench should prevent a wide range of syntax errors by construction, ensuring that any generated code is syntactically valid.
5. **Demonstrate the Power of Formal Analysis:** Make the sophisticated static analysis tools of Stipula accessible through a simple button click, allowing users to easily check their contracts for critical properties like liquidity.

Ultimately, the Stipula Workbench was conceived not just as a tool for writing code, but as an educational platform to make the power of formal methods in the legal domain accessible to the people who need it most.

Chapter 2

Workbench Architecture and Implementation

This chapter provides the technical core of the thesis. It begins with a detailed description of the high-level system architecture, its guiding principles, and the data flow between components. It then offers an in-depth analysis of the frontend's implementation, focusing on the evolution from a basic prototype to a user-centric final design. Finally, it details the backend services that power the application.

2.1 System Architecture: A Service-Oriented Approach

To achieve the goal of a seamless, scalable, and maintainable environment, the Stipula Workbench was designed using a **service-oriented architecture**. This architectural style structures the application as a collection of loosely coupled, independently deployable services. The primary driver for this choice was the principle of **separation of concerns**. This was critical for integrating the project's technologically heterogeneous components—a modern web frontend, a Python-based analysis tool, and a Java-based interpreter—into a single, cohesive system without creating a brittle, monolithic application.

2.1.1 Architectural Principles

The design was guided by several key principles common in the design of large-scale software systems:

- **Separation of Concerns:** Each component in the architecture has a single, well-defined responsibility. The frontend handles presentation, the backend orchestrates

communication, and the specialized services perform their specific computational tasks. This separation simplifies development, testing, and maintenance.

- **Loose Coupling:** Components are designed to interact with each other through well-defined, standardized interfaces (a REST API and a WebSocket protocol [10]) without needing to know the internal implementation details of other components. For instance, the frontend does not know that the analyzer is a Python script; it only knows how to communicate with the backend's '/api/analyze' endpoint. This allows any component to be modified or replaced with minimal impact on the rest of the system.
- **Componentization:** The system is broken down into logical components (Frontend, Backend Gateway, Analyzer Service, Interpreter Service) that can be developed and reasoned about independently. This modularity was essential for managing the project's complexity.

2.1.2 High-Level Architectural Diagram and Data Flow

The overall architecture is depicted in Figure 2.1. It illustrates the four primary components and the communication pathways between them.

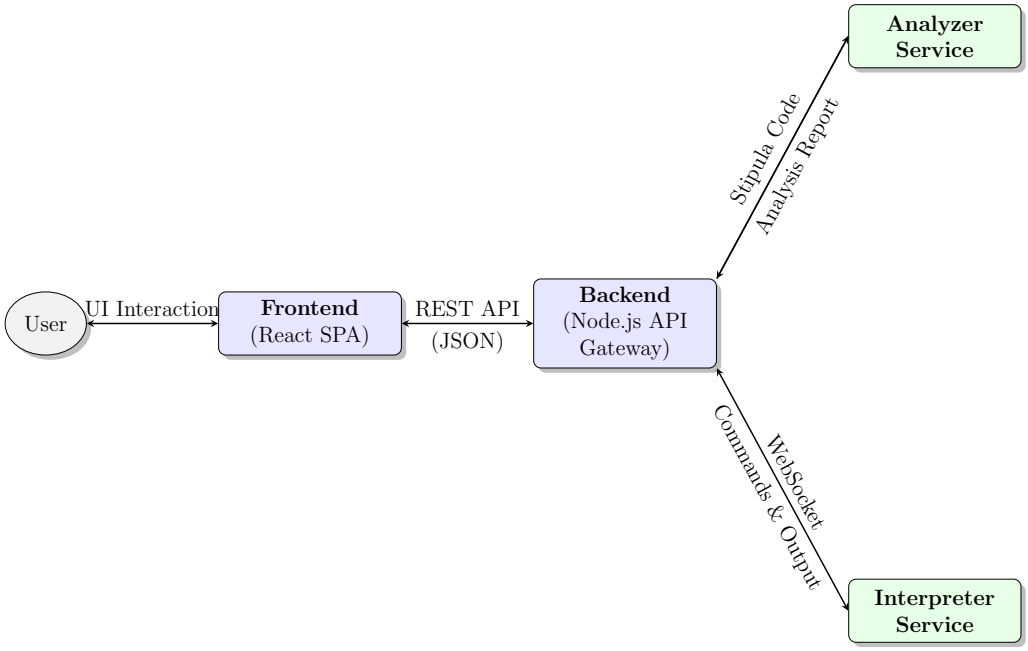

Figure 2.1: High-level architecture of the Stipula Workbench. The Frontend communicates with a central Backend Gateway, which orchestrates requests to the specialized Analyzer and Interpreter services. Source: Author.

The data flow follows a clear client-server model. The user interacts exclusively with the ‘Frontend‘. All subsequent actions, such as performing an analysis or running the interpreter, are initiated by the ‘Frontend‘, which sends requests to the ‘Backend‘. The ‘Backend‘, in turn, acts as a gateway, delegating these requests to the appropriate specialized ‘Service‘ and returning the result to the ‘Frontend‘ for display.

2.1.3 Component Deep Dive

The Frontend: The Presentation Layer

The ‘frontend‘ is the system's presentation layer, responsible for the entire user experience.

- **Role and Technology:** It is engineered as a **React** [1] Single-Page Application. This technology was chosen for its component-based architecture, which allows for the creation of a modular and maintainable UI, and its use of a virtual DOM, which ensures a highly responsive and performant user experience—critical for the real-time code generation feature.
- **Responsibilities:** Its primary duties include rendering all UI elements, managing the client-side application state (the contract object being built), handling all user interactions, and initiating communication with the backend. It is the "smart" client in the architecture.

The Backend: The API Gateway and Orchestration Layer

The ‘backend‘ is the central nervous system of the application, designed following the **API Gateway pattern**.

- **Role and Technology:** Implemented in **Node.js** [2] with the **Express** [17] framework, its role is not to perform business logic but to act as a single, unified entry point for the frontend. It is an orchestrator that routes incoming requests, manages service processes, and aggregates responses. Node.js was selected for its non-blocking, event-driven I/O model, making it highly efficient for managing concurrent connections and I/O-bound tasks like forwarding requests and managing child processes.
- **Responsibilities:**
  1. **Request Routing:** It exposes the application's API to the frontend and routes incoming requests to the appropriate downstream service. For example, a ‘POST‘ request to ‘/api/analyze‘ is routed to the Analyzer Service.

2. **Process Management:** A critical function of the gateway is to manage the lifecycles of the external command-line tools. It uses Node.js's `child_process` module to spawn, monitor, and terminate the Python analyzer and Java interpreter processes on behalf of the user.
3. **Protocol Translation:** It acts as a bridge between the web and the command-line worlds. It translates HTTP requests from the client into file-based inputs for the services and translates the standard output of those services back into JSON or WebSocket responses for the client.

The Analyzer Service: A Stateless Microservice

The 'analyzer' is a specialized, single-purpose microservice.

- **Role and Architectural Style:** Its sole responsibility is to perform static analysis on a given piece of Stipula code. It is designed as a **stateless service**, meaning each request is an atomic, independent transaction that contains all the necessary information for its completion. The service retains no memory of past requests, which makes it highly robust, easy to scale horizontally, and simple to reason about.
- **Implementation:** It is a thin wrapper around the pre-existing Python-based command-line analyzer. The backend gateway manages the orchestration, such as writing the code to a temporary file that the script can access.

The Interpreter Service: A Stateful Microservice

In contrast to the analyzer, the 'interpreter' is a stateful service.

- **Role and Architectural Style:** Its purpose is to simulate the execution of a Stipula contract over time. This is an inherently **stateful** process, as the service must maintain the contract's current state (e.g., asset distribution, active clauses) throughout a user's interactive session.
- **Implementation and Lifecycle:** It is a wrapper around the pre-existing Java-based command-line interpreter. The backend gateway is responsible for managing the lifecycle of these stateful instances. For each new user session initiated from the frontend, the gateway spawns a new Java process. It maintains a persistent WebSocket connection to this process, piping data back and forth. When the user's session ends (e.g., the WebSocket connection is closed), the gateway terminates the associated Java process and cleans up any resources.

2.2 Development Process and Workflow

The development of the Workbench followed a logical, incremental progression that leveraged the modularity of its architecture.

1. **Frontend First:** The initial focus was on building the React-based frontend and perfecting the user interface for the contract editor. This allowed for rapid iteration on the user experience with mock data, without the dependency of a fully functional backend.

2. **Backend Integration:** Once the UI was established, the Node.js backend gateway was developed. Initial API endpoints were created to bridge the frontend to dummy services, establishing the communication contracts.

3. **Service Connection:** Finally, the standalone analyzer and interpreter tools were encapsulated as services and connected to the backend gateway. This final step completed the end-to-end workflow, enabling the full user journey within the integrated environment.

This structured approach, enabled by the service-oriented architecture, was instrumental in managing the project's complexity and resulted in a robust and scalable final product.

Chapter 3

Implementation Details

This chapter provides a detailed technical overview of the implementation of the Stipula Workbench, expanding on the high-level architecture discussed previously. We will examine the specific technologies and code patterns used in each of the major components, illustrating how they work together to create a functional and responsive application.

3.1 User Interface and User Experience Evolution

While the core functionalities of the backend services and the frontend's state management provide the engine for the Stipula Workbench, the user interface (UI) is the vehicle through which the user interacts with that power. The initial prototype, while functional, was created primarily as a proof-of-concept. This section details the significant redesign and reimplementation of the UI, a transition from that basic prototype to a sophisticated and user-centric Integrated Development Environment (IDE) [16]. The primary goal was to create a professional and intuitive tool for legal contract programming, guided by several key objectives: modernizing the aesthetics, improving the information architecture, enhancing interactivity with real-time feedback, streamlining the management of complex entities like functions, and empowering advanced users with direct code manipulation capabilities.

3.1.1 The Initial Prototype: A Functional Foundation

The original UI, branded "Stipula Editor," served as a functional first iteration that successfully validated the core concepts of the application. However, it presented significant limitations in terms of user experience (UX) and advanced functionality that hindered a fluid and efficient workflow. Its structure was a monolithic, single-page application with a vertical-scrolling layout, where all components—Assets, Fields, Parties, Agreement, Functions, and the Code Output—were stacked sequentially. As shown in Figure

4.2, this design forced the user to scroll extensively to see the effect of their changes on the generated code. The code itself was rendered in a non-interactive <p> HTML tag, with the only available action being a "Copy to clipboard" button. The most critical limitations identified in this initial version were:

- **Lack of Concurrency:** The user could not view the contract's graphical representation and the resulting code simultaneously, making the iterative process of writing and reviewing code inefficient.
- **Static Code Preview:** The code output was a passive display. There was no integrated way to edit the code directly, perform static analysis, or execute it within the editor, forcing users into an external, disconnected workflow.
- **Cluttered Function Editing:** As detailed in Section 3, functions were edited "inline" directly on the main page. This approach resulted in a cluttered and confusing user experience, especially when defining functions with numerous attributes like callers, parameters, and state transitions.

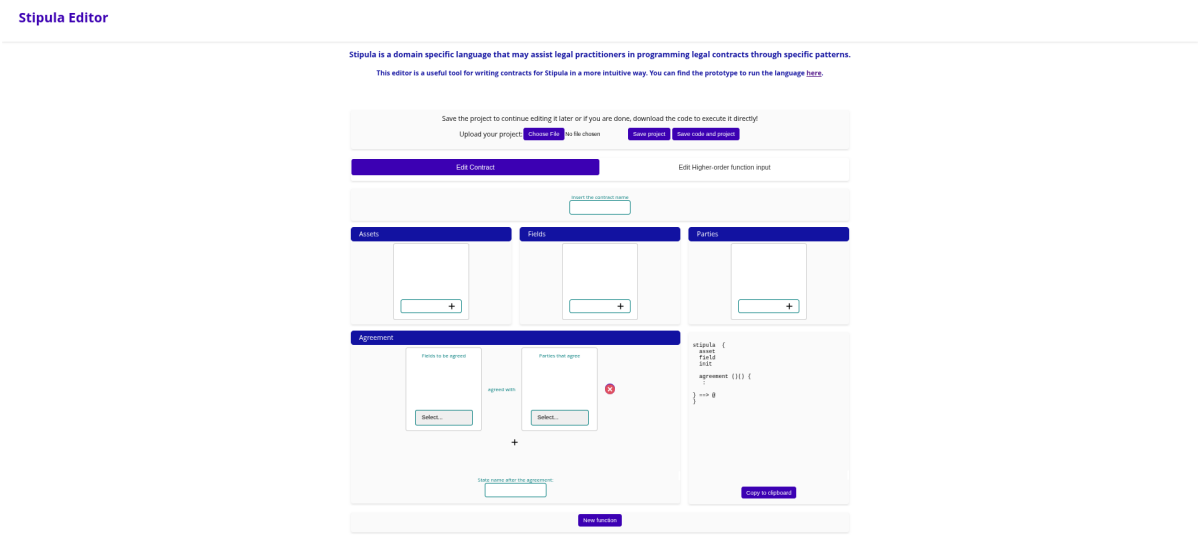

Figure 3.1: The initial prototype of the Stipula editor, featuring a monolithic, top-to-bottom layout. All sections, from contract definition to code output, are contained in a single scrolling view.

3.1.2 Stipula Workbench 2.0: A Paradigm Shift in Design

The redesigned UI, "Stipula Workbench 2.0," represents a paradigm shift in both design and functionality. It addresses the limitations of the prototype by fundamentally re-

structuring the information architecture and integrating a suite of powerful development tools.

A Modern, Two-Panel Layout

The most significant architectural change is the move to a responsive, two-panel layout, as seen in Figure 3.2. This design separates the graphical contract editor from the code output and execution tools.

- **The Editor Panel (Left)** is dedicated exclusively to the visual construction of the contract. It houses all the graphical components in a clean, organized, and collapsible accordion-style interface.
- **The Output Panel (Right)** consolidates all code-related functionalities. It acts as an interactive hub for code viewing, direct editing, analysis, and interpretation, creating a "cause-and-effect" workflow where changes in the Editor are instantly reflected.

This layout is also flexible; the panels are resizable and collapsible, managed by the `expandedPanel` state in `ContractView.jsx`, allowing the user to focus on either the graphical editor or the code output as needed.

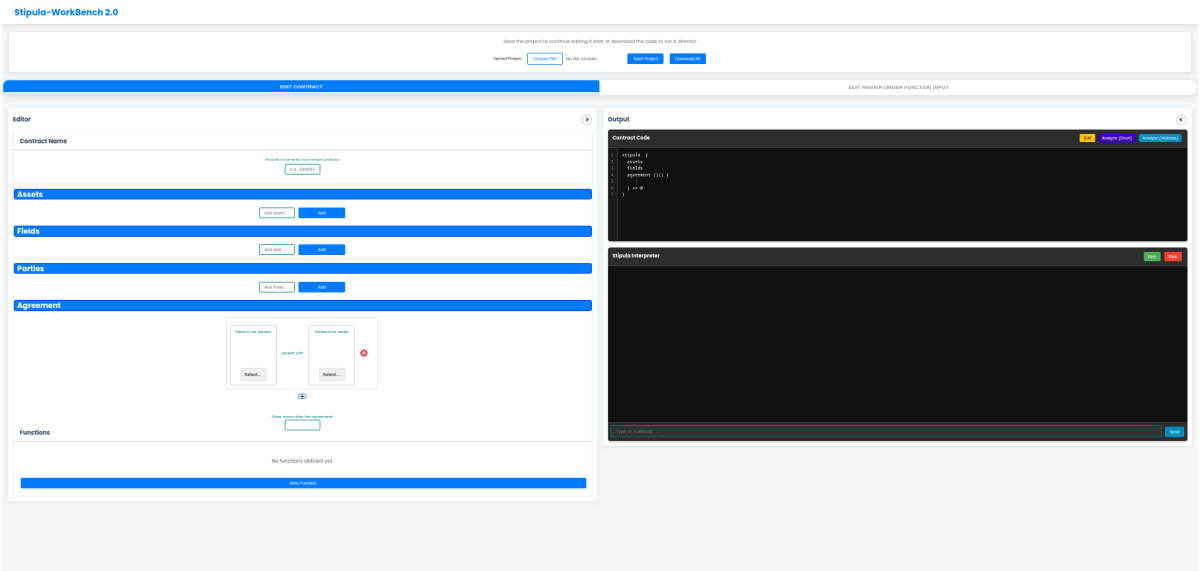

Figure 3.2: The redesigned two-panel layout of Stipula Workbench 2.0. The left 'Editor' panel manages the graphical contract construction, while the right 'Output' panel provides an interactive environment for the generated code.

The Interactive Output Panel: A Live Development Environment

The Output Panel transforms the static text of the prototype into a dynamic and interactive workspace. This was achieved through several key feature implementations.

Editable Code Editor with Manual Override: The passive code display was replaced with a proper code editor featuring line numbers and syntax awareness. Crucially, an "escape hatch" for advanced users was implemented. By clicking the "Edit" button, the user can disconnect the graphical editor (via the `isEditorDisconnected` state flag) and modify the code directly within a `<textarea>`. This feature provides flexibility without compromising the stability of the primary graphical editing experience, as a "Reset & Re-sync" option is provided to revert changes. **Integrated Static Analysis:** The Workbench now seamlessly integrates with the backend analyzer service. "Unreachability Analyzer" and "Unreachability Analyzer (verbose output)" buttons trigger the `handleAnalyze` async function, which makes a REST API call to the backend. The results of the analysis are then displayed directly within the UI, providing immediate feedback on the code's correctness without ever leaving the application. **The Stipula Interpreter:** A fully integrated, terminal-like interpreter console has been added. It features "Run" and "Stop" controls, a real-time output display, and a command input field. This interactivity is powered by a persistent WebSocket connection managed by functions like `connectWebSocket` and `sendInterpreterInput`. This brings the entire contract lifecycle—from creation and analysis to interactive execution and testing—into a single, unified environment.

Streamlined Function Management via Modal Editing

To address the confusing "in-line" editing of the prototype, the new UI adopts a modal-based workflow for managing functions. The main interface now displays a clean list of existing functions. When a user chooses to create a new function or edit an existing one, a modal window appears. This is managed by the `editingFunction` state, which conditionally renders the `FunctionEditor.jsx` component as an overlay. This pattern, as described in Section 3, isolates the complexity of function definition into a "focused, distraction-free environment," dramatically improving clarity and usability.

3.1.3 Conclusion: From Prototype to Professional Tool

The evolution of the Stipula Workbench UI is a story of deliberate transformation. By moving from the linear, static prototype to a dynamic, two-panel, interactive environment, the application has made a significant leap in usability, efficiency, and power. The new design does not merely offer a better user experience; it integrates essential development tools directly into the workflow. This establishes the Stipula Workbench as a comprehensive and mature platform, effectively equipped to support the complexities of

legal contract programming.

3.1.4 Managing Complexity: The Modal Flow for Functions and Actions

A Stipula function is the most complex entity in a contract, containing not just metadata but also a body of actions which can be deeply nested (e.g., an ‘if‘ statement within another ‘if‘ statement). The initial prototype's attempt to edit this "in-line" created significant UI clutter and cognitive overload. To solve this, a **modal editing flow** was implemented as a deliberate design choice. The intent was to create a focused, distraction-free environment for the complex task of defining a function's logic.

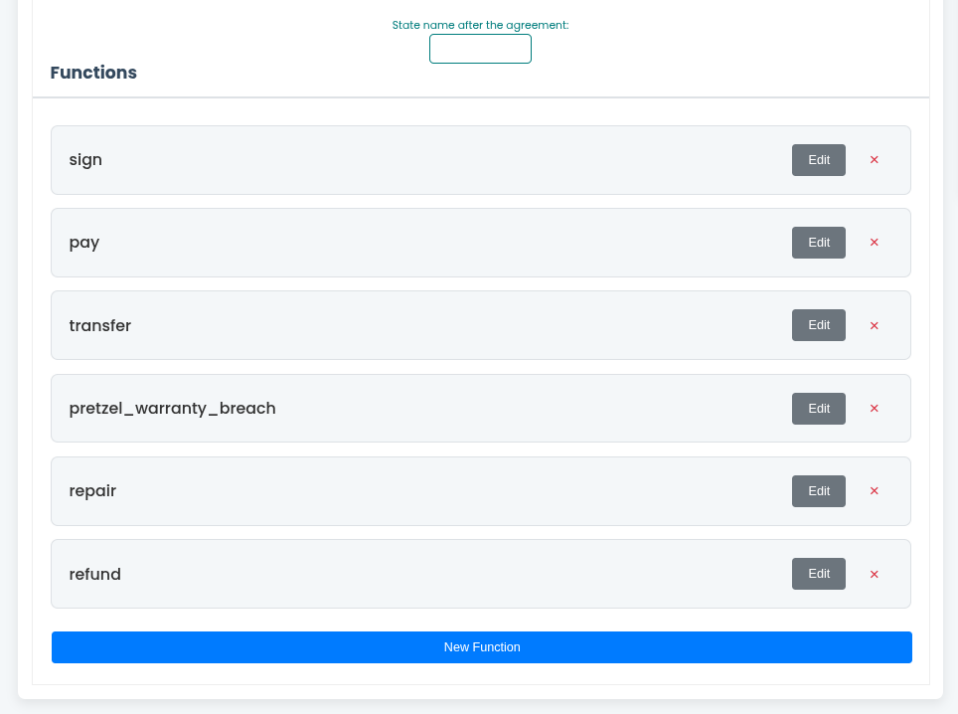

Figure 3.3: The main editor panel, showing the clean list of functions. This uncluttered view serves as the entry point to the focused modal editing workflow.

When a user clicks "New Function" or "Edit," the ‘FunctionEditor.jsx‘ component is rendered as a modal overlay. This pattern was chosen for two key reasons:

1. **Contextual Focus:** The modal isolates the task of function editing. By overlaying the main UI, it forces the user to concentrate only on the properties of the function they are currently defining.

2. **Efficient State Management:** The ‘FunctionEditor‘ modal manages its own internal state during editing. The central contract state is only updated once when the user clicks ”Save,” making the data flow predictable and performant.

Representing Recursive Logic: The Actions Tree

The greatest challenge within the ‘FunctionEditor‘ was representing the function’s body—a potentially recursive tree of actions. The chosen solution was to implement a **recursive React component**, ‘ActionsList.jsx‘. This component is designed to render a list of actions. If an action is a container (like ‘if-then-else‘), the component renders the UI for the condition and then recursively renders another instance of ‘ActionsList.jsx‘ within the ”then” and ”else” blocks. This implementation directly mirrors the recursive nature of the data structure, making the logical hierarchy immediately apparent from the UI.

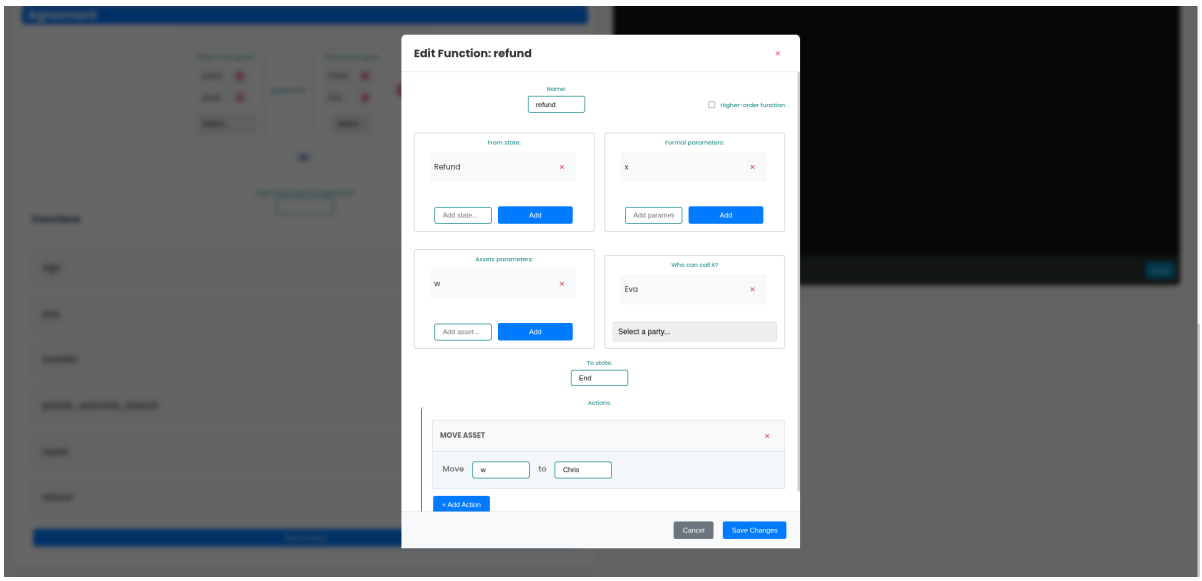

Figure 3.4: The Function Editor modal with the recursive Actions List. The nested and indented structure allows users to visually build and understand complex logic trees.

This modal-based, recursive component approach successfully tames the complexity inherent in defining Stipula functions, transforming it into a structured and intuitive process.

3.2 Frontend: A Reactive and Modular User Interface

The frontend is the core of the user's experience, engineered as a Single-Page Application (SPA) using the **React** [1] JavaScript library. The choice of React was strategic, driven by its component-based architecture and declarative rendering model. This paradigm is ideally suited for building a complex, stateful interface like the Workbench, allowing for a logical separation of concerns and a predictable data flow.

3.2.1 The Core: Centralized State Management in 'ContractView.jsx'

The most critical architectural decision in the frontend is the centralization of the application's state. The entire digital contract, with all its constituent parts (parties, assets, functions, etc.), is managed within a single state object in the main 'ContractView.jsx' component. This is achieved with a single 'useState' hook:

```
1 const [cont, setCont] = useState(new Contract());
```

This approach was chosen for several key reasons:

- **Single Source of Truth:** It ensures that there is only one authoritative representation of the contract at any given time. This eliminates the possibility of data inconsistencies between different UI components.
- **Simplified Data Flow:** The data flow is unidirectional and easy to trace. The parent component, 'ContractView', holds the state and passes down the necessary data and callback functions to its children as props. When a change is needed, a child component invokes a callback, which updates the state in the parent.
- **Enabling Real-Time Feedback:** This centralized model is the cornerstone of the live-preview feature. Any update to the 'cont' state object triggers a re-render of 'ContractView' and its children. The code output panel, which calls the 'getCode(cont)' function, is therefore re-evaluated on every change, instantly reflecting the user's modifications in the generated Stipula code.

3.2.2 A Declarative UI: The Presentational Component Pattern

The application's UI is decomposed into a hierarchy of specialized components. This structure follows a common React pattern where "container" components manage logic and state, while "presentational" (or "dumb") components are responsible only for rendering UI based on the props they receive. 'ContractView.jsx' is the primary container, while components like 'Parties.jsx', 'Assets.jsx', and 'Name.jsx' are presentational. Let's

examine ‘Parties.jsx‘ as a case study. It is responsible for displaying the list of parties and providing a form to add a new one.

```jsx
import React, { useState } from "react";
import { cleanStr } from "./Contract";
function Parties(props) {
  // Local state for the controlled input field
  const [input, setInput] = useState("");
  function handleSubmit(e) {
    props.handleAdd(input); // Call the parent's handler
    e.preventDefault();
    setInput("");
    // Reset local state
  }

  return (
    <div>
      <div className="title">Parties</div>
      <ul>
        {/* Render the list passed down as a prop */}
        {props.value.map((element) => (
          <li key={element}>
            <span>{element}</span>
            <button
              className="btn-delete"
              onClick={() => props.deleteParty(element)}
            >
              &times;
            </button>
          </li>
        ))}
      </ul>
      <form onSubmit={handleSubmit}>
        <input
          type="text"
          value={input}
          onChange={(e) => setInput(cleanStr(e.target.value))}
          placeholder="Add Party..."
          required
        />
        <button type="submit">Add</button>
      </form>
    </div>
  );
}
export default Parties;
```

Listing 3.1: The ‘Parties.jsx‘ presentational component.

This component does not know how to add or delete a party from the main contract

state. It only knows how to render the list it is given (‘props.value‘) and to call the functions (‘props.handleAdd‘, ‘props.deleteParty‘) that its parent has provided. This separation of concerns makes the component highly reusable, predictable, and easy to test in isolation.

3.2.3 Managing Complexity: The Modal Flow for Function Editing

A Stipula function is a complex entity with multiple attributes: a name, callers, parameters, assets, state transitions, and a body of actions. Implementing an inline editor for such an object directly within a list would lead to a cluttered and confusing user experience. To solve this, a modal editing flow was implemented using the ‘FunctionEditor.jsx‘ component. This choice provides a focused, distraction-free environment for the complex task of defining a function's logic.

1. The ‘FunctionList.jsx‘ component renders the list of existing functions and provides "Edit" and "New Function" buttons.

2. Clicking either of these buttons invokes a callback in ‘ContractView‘ (‘handleOpenFunctionEditor‘ or ‘handleOpenNewFunction‘).

3. ‘ContractView‘ then updates its ‘editingFunction‘ state. This state holds the function's data and a flag indicating if it is a new or existing function.

4. The ‘FunctionEditor‘ modal is conditionally rendered based on this state. It appears as an overlay, receiving the function data and callback functions (‘onSave‘, ‘onClose‘) as props.

5. All edits within the modal are managed by its internal state. Only when the user clicks "Save" is the ‘onSave‘ callback invoked, which updates the central contract state in ‘ContractView‘ and closes the modal.

This pattern isolates the complexity of function editing, keeping the main UI clean while providing a powerful and intuitive interface for the user.

3.2.4 Advanced Feature: Manual Code Override

To accommodate expert users who may wish to write or modify Stipula code directly, an "escape hatch" feature was implemented. Acknowledging that synchronizing manual code changes back to the graphical editor is a highly complex problem, a pragmatic one-way override was chosen. When the user clicks the "Edit" button in the code panel:

- A confirmation dialog warns them that this action will disconnect the graphical editor.

- If they proceed, two state flags are set in ‘ContractView‘: ‘isCodeEditable‘ and ‘isEditorDisconnected‘.
- These flags have two effects:
    1. The entire graphical editor panel becomes disabled, preventing further changes. A warning message is displayed.
    2. The code display, which is normally a ‘¡pre¿‘ element, is swapped for a ‘¡textarea¿‘, allowing direct text input. The ‘editedCode‘ state is initialized with the current contract code.

This provides a power-user feature without compromising the stability of the primary graphical editing experience. A ”Reset & Re-sync” button allows the user to discard their manual changes at any time, re-enabling the graphical interface and restoring the connection to the underlying state model. This demonstrates a thoughtful approach to balancing user freedom with application integrity.

3.2.5 Communication with the Backend

The frontend communicates with the backend services via standard web protocols.

- **REST API for Analysis:** For stateless requests like analysis, the frontend uses the ‘fetch‘ API to make a POST request to the backend’s ‘/api/analyze‘ endpoint. The body of the request contains the Stipula code as a JSON payload.
- **WebSockets for Interpretation:** For the interactive interpreter, a persistent, bidirectional communication channel is required. The frontend uses the native ‘WebSocket‘ API [10] to connect to the interpreter_server. This allows for real-time, terminal-like interaction, where user commands are sent to the server and output from the interpreter is streamed back to the client immediately.

3.3 Backend: Orchestration and Service Integration

The backend infrastructure is composed of two main Node.js servers: a primary API gateway (‘server.js‘) and a dedicated WebSocket server for the interpreter (‘server_interpreter.js‘). Both act as orchestrators, managing communication and executing external command-line tools.

3.3.1 The API Gateway and Analyzer Service

The main backend, defined in ‘server.js‘, handles requests for static analysis. Its implementation follows a clear pattern: receive a request, delegate the task to an external

process, and return the result. This is achieved using the ‘child_process‘ module in Node.js.

When a POST request is made to ‘/api/analyze‘, the server performs the following steps, as shown in the code snippet below:

1. It receives the Stipula code from the request body.
2. It writes this code to a temporary ‘.stipula‘ file on the server. This is necessary because the analyzer is a command-line tool that operates on files.
3. It uses ‘spawn‘ to execute the Python-based analyzer (‘analyzer.py‘), passing the path to the temporary file as an argument.
4. It captures the ‘stdout‘ (standard output) and ‘stderr‘ (standard error) streams from the Python process.
5. Once the process completes, it deletes the temporary file and sends the captured output back to the frontend as a JSON response.

```
const { spawn } = require('child_process');
const fs = require('fs/promises');
app.post('/api/analyze', async (req, res) => {
  const { code } = req.body;
  const tempFilePath = path.join(__dirname, '..', 'analyzer', `
   contract_${Date.now()}.stipula`);

  try {
    // 1. Save the code to a temporary file
    await fs.writeFile(tempFilePath, code);

    // 2. Spawn the Python analyzer process
    const pythonProcess = spawn('python', ['analyzer.py', tempFilePath
   ]);

    let stdoutData = '';
    pythonProcess.stdout.on('data', (data) => {
      stdoutData += data.toString();
    });

    // 3. On process completion, clean up and send response
    pythonProcess.on('close', async () => {
      await fs.unlink(tempFilePath);
      res.status(200).json({ output: stdoutData });
    });

  } catch (error) {
    // Error handling...
```

```
27   }
28 });
```

Listing 3.2: Executing the analyzer from ‘server.js‘

This approach effectively wraps the existing command-line tool in a web service without modifying the tool itself.

3.3.2 The Interpreter WebSocket Server

For the interpreter, a more stateful and interactive communication model is needed. This is handled by ‘server_interpreter.js‘, which leverages the ‘ws‘ library to create a WebSocket server. This server manages long-lived Java processes that run the Stipula interpreter. When a client connects via WebSocket and sends a ‘START_INTERPRETER‘ message, the server performs a similar sequence of actions:

1. It writes the received Stipula contract and any higher-order input files to temporary files.
2. It uses ‘spawn‘ to start the Java-based interpreter (‘HOstipula_lan.jar‘) [15], passing the paths to the temporary files.
3. It establishes a connection between the WebSocket client and the Java process. The server listens for ‘stdout‘ from the Java process and forwards any output directly to the client through the WebSocket connection.
4. It listens for incoming messages on the WebSocket (e.g., ‘SEND_INPUT‘) and writes them to the ‘stdin‘ (standard input) of the Java process, allowing the user to interact with the running contract.
5. When the connection is closed or a ‘STOP_INTERPRETER‘ message is received, it terminates the Java process and cleans up the temporary files.

```
1  const WebSocket = require('ws');
2  const { spawn } = require('child_process');
3  wss.on('connection', ws => {
4    let javaProcess;
5
6    ws.on('message', message => {
7      const data = JSON.parse(message);
8
9      if (data.type === 'START_INTERPRETER') {
10       const contractFilePath = // ... create temp file ...
11
12       // Spawn the Java interpreter process
```

```
13        javaProcess = spawn('java', ['-jar', 'HOstipula_lan.jar',
    contractFilePath]);
14
15        // Pipe interpreter output back to the client
16        javaProcess.stdout.on('data', output => {
17          ws.send(JSON.stringify({ type: 'INTERPRETER_OUTPUT', output:
    output.toString() }));
18        });
19
20        // ... other event handlers ...
21      } else if (data.type === 'SEND_INPUT') {
22        // Write user input to the Java process's standard input
23        if (javaProcess && javaProcess.stdin.writable) {
24          javaProcess.stdin.write(data.payload.input + '\n');
25        }
26      }
27    });
28 });
```

Listing 3.3: Spawning the Java interpreter from ‘server_interpreter.js‘

This implementation successfully integrates the interactive, command-line Java interpreter into the web interface, providing a seamless simulation experience for the user.

Chapter 4

Analysis of Inspirations

The design and development of the Stipula Workbench were informed by established principles and user interface (UI) patterns from leading applications in the web development and blockchain sectors. This chapter analyzes the primary sources of inspiration, focusing on Remix IDE and CodePen. For each, we will discuss the conceptual similarities that guided the project's direction and the key logical differences that arise from Stipula's unique requirements.

4.1 Remix IDE: The Smart Contract Benchmark

Remix IDE [8] is one of the most popular web-based Integrated Development Environments (IDEs) for writing, testing, and deploying smart contracts on the Ethereum blockchain. As a tool designed for a similar domain—contract programming—it served as a fundamental model for the core workflow of the Stipula Workbench.

4.1.1 Conceptual Similarities

The parallels between the Stipula Workbench and Remix IDE are most evident in their shared purpose and structural layout.

- **Domain-Specific Workflow:** Both platforms are purpose-built for contract development. The core user journey is identical: write the contract, compile it to view its representation, and interact with it in a simulated environment. The Stipula Workbench adopts this three-stage process with its 'Editor', 'Contract Code', and 'Stipula Interpreter' panels.
- **Multi-Panel Layout:** The visual architecture of the Workbench, which separates user input from system output, is directly inspired by Remix. Remix's standard

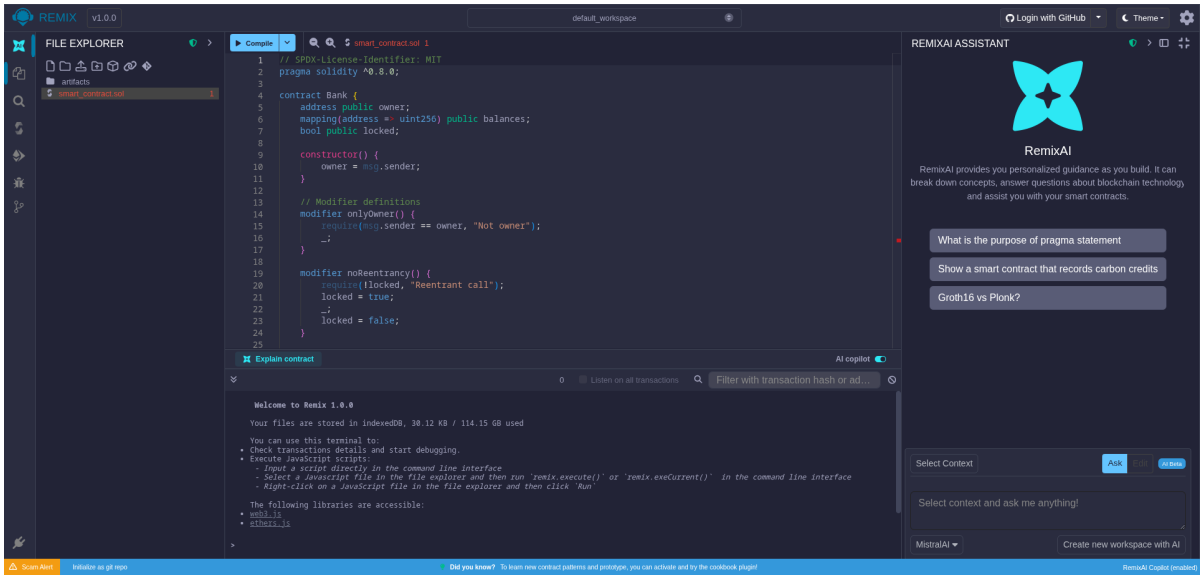

Figure 4.1: Remix Ide UI

layout features an editor panel and a utility panel for compilation and deployment. The Workbench simplifies this into a two-panel design to maintain focus on Stipula's specific features, providing a clean and functional workspace.

- **Focus on Interaction:** A key feature of Remix is the ability to interact with deployed contract functions through an auto-generated UI. The Stipula Interpreter panel serves an analogous role, providing a terminal-like interface to simulate how parties would call the functions defined in the editor.

4.1.2 Logical Differences

Despite the structural similarities, the fundamental nature of Stipula necessitates significant logical divergences from Remix.

- **Abstraction Level (High-Level vs. Code-First):** The most critical difference lies in the method of contract creation. Remix is a code-first environment where developers write raw Solidity code. In contrast, the Stipula Workbench employs a higher level of abstraction. It uses a guided, form-based interface to construct the contract's components (Parties, Assets, Functions). The Stipula code is then automatically generated from these inputs. This design choice makes the Workbench accessible to users without programming expertise, such as legal professionals, which is a primary goal of the Stipula project.

- **Target User and Language:** Remix is designed for blockchain developers proficient in languages like Solidity. The Stipula Workbench is created for a broader audience, including those with a legal background. Consequently, its entire UI is built around legal concepts (e.g., 'Agreement', 'Parties') rather than programming primitives.

4.2 CodePen: The Live Editor Paradigm

CodePen [3] is a popular online development environment for front-end web developers, known for its live-editing capabilities where users can see the results of their code in real-time. It served as a major inspiration for the interactive and responsive feel of the Stipula Workbench.

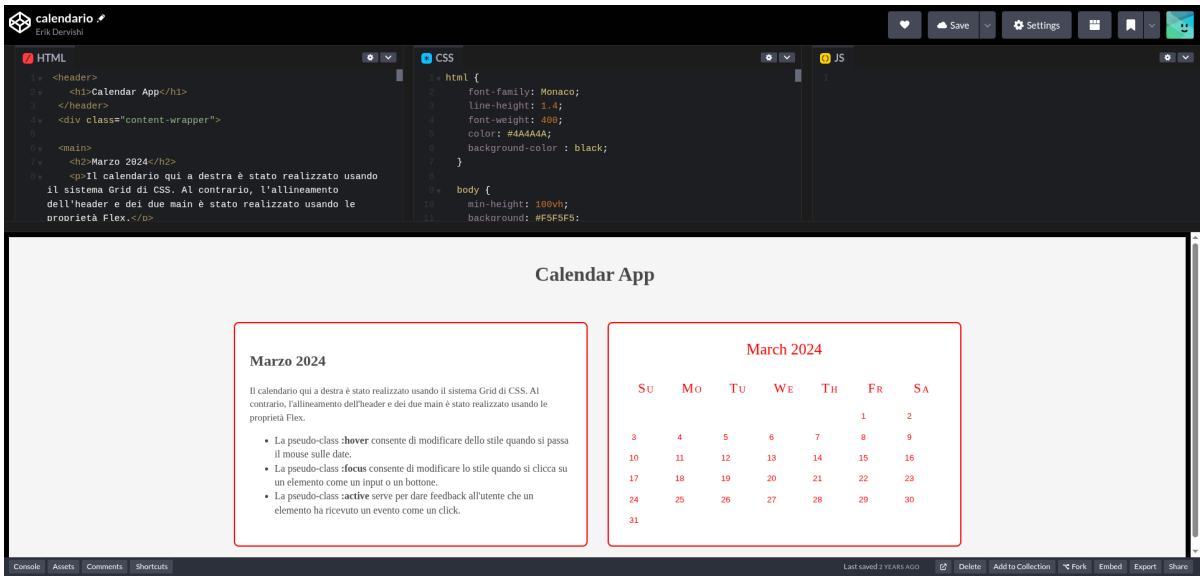

Figure 4.2: Codepen UI

4.2.1 Conceptual Similarities

The influence of CodePen is primarily seen in the user experience (UX) and the immediate feedback loop provided by the editor.

- **Live Preview Model:** The core paradigm of "input on the left, output on the right" is a hallmark of CodePen. The Stipula Workbench directly implements this model: as the user defines contract elements in the form-based editor, the corresponding Stipula source code is instantly generated and displayed in the 'Contract

Code‘ panel. This provides immediate validation that the user's actions are correctly translating into the formal language.

- **Encouragement of Experimentation:** The frictionless, real-time feedback loop pioneered by tools like CodePen encourages learning and experimentation. By seeing the code change instantly, users can quickly understand the relationship between the high-level concepts in the editor and the resulting Stipula syntax, flattening the learning curve.

4.2.2 Logical Differences

The primary difference stems from the nature of the "output" being generated.

- **Generated vs. Executed Output:** In CodePen, the output panel renders the direct, executable result of the user's code (a webpage). In the Stipula Workbench, the output is a static representation of the generated code, not a live execution of the contract. The actual execution is simulated separately in the ‘Stipula Interpreter‘. This separation is necessary because a Stipula contract is not a simple, runnable script but a stateful entity that requires sequential interaction.

- **Input Method:** CodePen's input is a free-form text editor. The Workbench, as mentioned, uses a structured, form-based input method. While both benefit from a live preview, the nature of the input is fundamentally different, reflecting the distinct goals of each application—creative freedom versus structured, error-resistant construction.

Chapter 5

Project Documentation: The Wiki

A fundamental aspect of ensuring the adoption and usability of any software tool, especially one aimed at a diverse audience, is the quality of its documentation. For this reason, an integral part of the Stipula Workbench's development was the creation of a comprehensive and modern project wiki. This wiki is not merely a manual, but a strategic tool designed to guide new users from foundational concepts to the creation of complex contracts.

5.1 Objectives of the Wiki

The design of the wiki was guided by three main objectives:

- **Accessibility for Non-Programmers:** The primary goal was to create a welcoming entry point for users without a programming background, particularly legal professionals and students. The terminology, examples, and structure were chosen to connect familiar legal concepts (such as "agreement," "obligation," "penalty") to their formal implementation in Stipula.
- **Guided and Practical Learning:** Instead of presenting purely theoretical documentation, the wiki adopts a hands-on approach. It guides the user through the actual use of the Workbench, encouraging learning-by-doing and providing immediate visual feedback through clear, commented code examples.
- **Comprehensive Reference:** Beyond being an onboarding tool, the wiki serves as a complete reference manual for more experienced users, documenting the language's features, common design patterns, and the capabilities of the integrated analysis tools.

5.2 Structure and Content

To achieve these objectives, the wiki was organized into logical sections, easily navigable via a table of contents.

- **Core Concepts:** This introductory section explains the four pillars of the Stipula language (state-aware programming, linear asset management, events for obligations, and the agreement primitive). Each concept is presented first in legal terms and then in technical terms, accompanied by a short, simple code example illustrating its practical application.
- **Practical Guide to the Workbench:** This chapter is a step-by-step tutorial that teaches users how to build their first contract using the editor. It follows a realistic use case (a rental contract), showing how to use each card of the interface, from defining the parties to writing the functional clauses and analyzing the result.
- **Common Legal Patterns:** To demonstrate Stipula's expressive power, this section illustrates how to model common yet complex legal scenarios. It presents patterns for managing escrowed funds, resolving disputes via a trusted authority, and integrating real-world data (oracles) through a 'DataProvider'.
- **Analysis Tools:** Here, the integrated analysis tools are explained in simple terms. It clarifies what concepts like "liquidity" and "reachability" mean in the context of a contract and why their automatic verification is crucial for preventing errors and vulnerabilities.
- **Frequently Asked Questions (FAQ):** A final section that concisely answers the most common questions, resolving practical doubts about the use of the language and the Workbench.

In conclusion, the wiki is an essential component of the user experience. It transforms the Stipula Workbench from a simple development tool into a complete learning platform, democratizing access to the creation of formal and secure legal contracts.

Chapter 6

Conclusion

6.1 Summary of Work

The Stipula language [4, 6] provides a formally robust framework for defining digital legal contracts, but its adoption has been hindered by the complexity of its syntax and a fragmented, command-line-based toolchain. The core motivation for this thesis was to address this usability challenge directly. The primary objective was to design and implement the Stipula Workbench, a unified, web-based integrated development environment (IDE) that abstracts away this complexity and makes the power of formal methods accessible to non-specialized users, particularly those in the legal domain. The implemented solution is a modern, service-oriented web application. It features a concept-first, graphical interface built with React that allows users to construct contracts visually. This frontend is supported by a Node.js backend that orchestrates communication with external, pre-existing tools for static analysis and interactive interpretation.

6.2 Synthesis of Results

The project successfully met the key objectives outlined in Chapter 1. The Stipula Workbench effectively lowers the barrier to entry by providing an intuitive, form-based editor. It delivers a seamless workflow by integrating the editor, analyzer, and interpreter into a single interface. The live-preview model offers immediate feedback crucial for learning, and the entire system promotes correctness by design. The final product is not merely a tool for writing code, but an educational platform that empowers users to design, analyze, and test formally correct digital contracts with confidence.

6.3 Future Work

While the Stipula Workbench provides a solid foundation, there are several avenues for future development that could further enhance its capabilities and user experience:

- **Support for other decidable fragments of Stipula:** The language has other subsets with different formal properties [7, 13]; the Workbench could be extended to support their unique features.
- **Enhanced Code Editor:** Adding syntax highlighting, auto-completion, and inline error checking to the manual code editor would improve the experience for expert users.
- **Graphical State Machine Visualization:** A feature to graphically visualize the contract's states and the transitions between them could provide powerful insights into the contract's logic and flow.
- **User Account System:** Implementing a user authentication and database system would allow users to save, manage, and share their contract projects online.

These potential improvements highlight the extensibility of the Workbench's architecture and its potential to evolve into an even more comprehensive tool for formal contract engineering.

References

[1] Dan Abramov and Tom Lerner. *React Docs*. Accessed: October 10, 2025. 2023.

[2] Mike Cantelon et al. *Node.js in Action*. Manning Publications, 2014. ISBN: 978-1617290572.

[3] CodePen. *CodePen: Online Code Editor and Front End Web Development*. `https://codepen.io/`. Accessed: October 10, 2025. 2024.

[4] Silvia Crafa and Cosimo Laneve. "Programming Legal Contracts: A Beginners Guide to *Stipula*". In: *The Logic of Software. A Tasting Menu of Formal Methods - Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*. Ed. by Wolfgang Ahrendt et al. Vol. 13360. LNCS. Cham: Springer, 2022, pp. 129–146.

[5] Silvia Crafa, Cosimo Laneve, and Adele Veschetti. *Stipula Prototype*. Version 1. Available on github: `https://github.com/stipula-language`. July 2022.

[6] Silvia Crafa et al. "Pacta sunt servanda: Legal contracts in *Stipula*". In: *Science of Computer Programming* 225 (2023), p. 102911.

[7] Giorgio Delzanno et al. "Decidability Problems for Micro-Stipula". In: *Proc. 27th Int. Conference COORDINATION 2025*. Vol. 15731. Lecture Notes in Computer Science. Springer, 2025, pp. 133–152.

[8] Ethereum Foundation. *Remix IDE*. `https://remix-project.org/`. Accessed: October 10, 2025. 2024.

[9] Samuele Evangelisti. "Analisi di Contratti Legali in ". Bachelor's Thesis. University of Bologna, 2024.

[10] Ian Fette and Alexey Melnikov. *The WebSocket Protocol*. RFC 6455, IETF. `https://tools.ietf.org/html/rfc6455`. 2011.

[11] Reiner Hähnle, Cosimo Laneve, and Adele Veschetti. "Formal Verification of Legal Contracts: A Translation-based Approach". In: *to appear in Proc. 20th Int. Conference on Integrated Formal Methods (iFM 2025)*. Lecture Notes in Computer Science. Springer, 2025.

[12] Cosimo Laneve. "Liquidity analysis in resource-aware programming". In: *J. Log. Algebraic Methods Program.* 135 (2023), 100889:1–18.

[13] Cosimo Laneve. “Reachability Analysis in Micro-Stipula”. In: *Proc. 26th International Symposium on Principles and Practice of Declarative Programming, PPDP 2024*. ACM, 2024, 17:1–17:12.

[14] Cosimo Laneve, Alessandro Parenti, and Giovanni Sartor. “Integration of Statutory Norms in Computable Contracts”. submitted to a journal. September, 2025.

[15] Cosimo Laneve, Alessandro Parenti, and Giovanni Sartor. “Legal Contracts Amending with Stipula”. In: *Proc. 25th Int. Conf.COORDINATION*. Ed. by Sung-Shik Jongmans and Antónia Lopes. Vol. 13908. LNCS. Cham: Springer, 2023, pp. 253–270. DOI: 10.1007/978-3-031-35361-1_14.

[16] Don Norman. *The Design of Everyday Things: Revised and Expanded Edition*. Basic Books, 2013. ISBN: 978-0465050659.

[17] OpenJS Foundation. *Express - Node.js web application framework*. https://expressjs.com/. Accessed: October 10, 2025. 2024.

[18] Elia Venturi. “Implementazione di un’interfaccia grafica per la scrittura di contratti legali nel linguaggio ”. Bachelor’s Thesis. University of Bologna, 2023.