Systems Dynamics Interface Technical Report

Interdisciplinary Centre for Circular Chemical Economy

UK Research and Innovation

NICER PROGRAMME

HERIOT WATT UNIVERSITY

Systems Dynamics Interface Technical Report

Bing Xu[1*] , *Umit Bititci*[1] , *Rossen Kazakov*[1] , *Yonghan Zhang*[2], *Junhao Song*[2*]

[1]Edinburgh Business School, Heriot-Watt University, UK

[2]School of Mathematical and Computer Sciences, Heriot-Watt University, UK

This technical documentation outlines a System Dynamics (SD) model designed to support policy implementation by exploring strategies for transitioning to a circular economy in the UK chemical manufacturing sector. The model analyses dynamic interdependencies among key variables, providing a comprehensive framework for decision-making. Integrated into a web-based interactive platform via SDEverywhere, the SD model allows stakeholders to visualise scenario outcomes and adjust parameters in real-time. By combining rigorous system modelling with accessible web technologies, our research bridges academic research and practical decision-making through dynamic, data-driven simulations. ***Project Webpage*** *and* ***System Dynamics Interface****.*

---

* Corresponding Authors. Edinburgh Business School, Heriot-Watt University, Edinburgh, UK, EH14 4AS. Emails: b.xu@hw.ac.uk and junhao.song@hw.ac.uk. We would like to thank all project academics and industrial partners for the valuable comments they provided. This work was supported by the UKRI Interdisciplinary Centre for Circular Chemical Economy [Grant number EP/V011863/1, EP/V011863/2].

1. Introduction

The UK chemical manufacturing industry has traditionally followed a linear economic model, often described as the "take-make-dispose" paradigm [1]. This model has led to unsustainable resource consumption, environmental degradation, and substantial economic inefficiencies [2]. Consequently, there is an urgent need to transition towards a circular economy (CE) that prioritizes resource efficiency, waste reduction, and the optimisation of resource usage within the industry [3].

However, transitioning to a CE is inherently complex due to the interconnected dynamics of resource flows, regulatory frameworks, technological advancements, and market dynamics [4] [5]. Effectively capturing these interactions requires an advanced analytical approach capable of modelling feedback loops, accumulations, and delays–key characteristics of complex systems.

To address this challenge, we have developed a System Dynamics (SD) model [6] using Stock-Flow Diagrams (SFDs) [7] to represent and explore these nonlinear feedback mechanisms and dynamic interactions. This model serves as a powerful analytical tool to quantitatively assess policy scenarios and their impact on achieving a CE [8] [9]. The SD model is provided in a standardised model file format (*.mdl*), enabling further analysis and adaptation.

Recognising the importance of stakeholder engagement and practical applicability, the developed SD model has been integrated into a web-based interactive platform leveraging the capabilities of the SDEverywhere [10]. This innovative integration enables real-time exploration of dynamic scenarios through interactive simulation features:

- **Real-Time Parameter Adjustment**: Users can intuitively modify key model parameters using interactive sliders.
- **Multi-Scenario Comparison:** Stakeholders can directly compare graphical outputs representing different policy or technology scenarios [10].

The platform's interactive visualisation capabilities significantly improve the user experience, allowing stakeholders (e.g., policymakers, industry representatives, and researchers) to dynamically visualise and interpret trends through user-friendly graphical outputs. Key features include:

- **Dynamic Charts:** Interactive graphical plots displaying trends based on adjustable parameters, with functionality supporting zooming and detailed data point inspection.

- **High Speed Computing:** The SD model is compiled directly into JavaScript, ensuring exceptional computational performance with millisecond-level response times (typically less than 300 ms for a 10-year simulation).
- **Cross Platforms Compatibility:** The responsive design ensures seamless operation across various desktop and mobile devices, maximising accessibility and usability.

This technical report provides a detailed overview of the conceptual foundations and implementation processes underpinning our system dynamics simulation platform. Subsequent sections will document the technical approach, outline key model variables and structures, describe interface functionalities, evaluate performance benchmarks, and discuss future enhancements and wider application opportunities.

2. Interface Implementation: SDEverywhere Integration

2.1 Toolchain and Technologies

Our System Dynamics Interface employs a robust and flexible technological toolchain, primarily built around the **SDEverywhere** framework [10]. SDEverywhere converts Vensim [12] system dynamics models (in *.mdl* format) into JavaScript code, enabling efficient web deployment and interactive simulation.

Tech Stack:

- **Frontend:** HTML, ECharts.js, and chartjs-plugin-zoom.js (interactive drag-and-zoom features).
- **Backend:** Node.js environment (version 18+, LTS version 20 recommended).

SDEverywhere's primary role is transforming the Vensim *.mdl* model into an optimised JavaScript module, supporting rapid simulation responses.

2.2 Implementation Steps

The following steps outline the implementation procedures for creating or updating the System Dynamics Interface:

A) Building a New .mdl Model:

- **Step A1:** Prepare the project directory containing only the .mdl file:

```
cd my-project-folder
```

- **Step A2:** Run the SDEverywhere setup script to initialise project files:

```
npm create @sdeverywhere@latest
```

The setup script provides default configurations suitable for most initial implementations, which can later be customised through sde.config.js and related configuration files.

- **Step A3:** Install required visualisation plugins:

```
npm install chartjs-plugin-zoom@0.7.7
```

- **Step A4:** Replace the default generated files in packages/app/ with custom-developed files:

  packages/app/index.html
  packages/app/package.json
  packages/app/src/graph-view.ts
  packages/app/src/index.css
  packages/app/src/index.js

- **Step A5:** Configure model parameters by modifying CSV files in the ./config folder:
  colors.csv: Define chart colour schemes.
  graphs.csv: Set chart types, titles, and axes configurations.
  inputs.csv: Specify adjustable input parameters and their ranges.
  model.csv: Describe variables, definitions, and equations.
  outputs.csv: List simulation outputs and their descriptions.
  strings.csv: Manage interface text content.

- **Step A6:** Launch the development server for testing:

```
npm run dev
```

B) Updating an Existing .mdl Model:

- **Step B1:** Replace the existing .mdl file with the new model file, and remove outdated build folders:

```
rm -rf baselines sde-prep
npm install -g @sdeverywhere/cli
npm install @sdeverywhere/plugin-check
```

- **Step B2:** Adjust the configuration in sde.config.js to align with new model specifics, ensuring proper parameter mappings. Example configuration snippet (sde.config.js):

```
export async function config() {
  return {
    genFormat: 'js',
    modelFiles: ['sdms.mdl'],
    watchPaths: ['config/**', 'sdms.mdl'],
    plugins: [
      workerPlugin({ outputPaths: ['path/to/worker.js'] }),
      checkPlugin(),
      vitePlugin({ name: 'app', apply: { development: 'serve' }, config: {
configFile: 'path/to/vite.config.js' } }),
    ]
  }
}
```

- **Step B3:** Restart the development server to verify updates:

```
npm run dev
```

2.3 Configuration Explanation:

The simulation interface is governed by several CSV files within the ./config directory:

colors.csv: Configures chart curve colours.

- id: Colour index.
- hex code: Hexadecimal colour code (beginning with #).

graphs.csv: Defines the layout and type of visual charts.

- id: Graph identifier.
- type: Chart type (line).
- title: Graph title.
- axis: Axes labels and scaling details.

inputs.csv: Specifies user-adjustable simulation parameters.

- id: Parameter identifier.
- name: Parameter name.
- default: Default value.
- range: Acceptable parameter range.

model.csv: Describes the structure of the SD model.

- id: Variable identifier.
- name: Variable name.
- definition: Variable description.
- equation: Mathematical equation or logic.

outputs.csv: Lists output indicators generated by the model.

- id: Output identifier.
- name: Indicator name.
- unit: Measurement unit.
- description: Indicator explanation.

strings.csv: Controls textual interface content.

- id: Text element identifier.
- key: Textual string key.
- value: Displayed text content.

These steps ensure the successful deployment and reliable operation of our interactive, web-based System Dynamics Interface.

3. Interactive Visualisation Features

This section describes the visual design and core interactive functionalities of the frontend interface, ensuring intuitive user interaction and immediate observation of the impacts resulting from real-time parameter adjustments. The overview of the interface is illustrated in Figure 3.1.

Figure 3.1 The overview of SD interface.

3.1 Real-Time Parameters

The interactive platform incorporates an intuitive parameter control panel allowing users to dynamically adjust input parameters via interactive sliders. These sliders are defined and managed in the frontend JavaScript (index.js) and styled through customised CSS (index.css).

Interactive Sliders: They are initialised within the JavaScript file (index.js), specifically through the addSliderItem function. Sliders correspond directly to parameters defined in inputs.csv, and dynamically update model parameters through event listeners:

```
slider.on('change', change => {
```

```
    const start = spec.defaultValue;
    const end = change.newValue;
    slider.setAttribute('rangeHighlights', [{ start, end }]);
    updateValueElement(change.newValue);
    sliderInput.set(change.newValue);
  });
```

Real-Time Feedback: Numerical values displayed alongside each slider provide real-time parameter feedback, formatted consistently according to specification (format() function in the index.js):

```
function format(num, formatString) {
  switch (formatString) {
    case '.1f':
      return num.toFixed(1);
    case '.2f':
      return num.toFixed(2);
    case '.0M':
      return (num/1000000).toFixed(0) + 'M';
    default:
      return num.toString();
  }
}
```

Cross Platforms Compatibility: Interaction mechanisms accommodate both desktop and mobile usage. Desktop browsers support direct slider dragging, whereas mobile browsers primarily support click-to-jump due to platform-specific limitations.

Flexible Configuration: Parameters, ranges, defaults, and descriptions are centralised in the .csv based configuration (inputs.csv), allowing straightforward updates without codebase modifications.

3.2 Dynamic Charts and Scenarios

The system visualisation leverages dynamic charts that update instantaneously as simulation parameters change. Visual representations are facilitated by Chart.js library [11], enabling robust interactive capabilities.

Dynamic Diagrams: These charts visualise relationships among critical system variables (e.g., Pollution vs GDP), automatically updating as inputs are adjusted, ensuring real-time responsiveness.

Time-Series Analysis: Charts can represent multiple variables simultaneously (up to 5), each with distinct colour coding, clearly set in colors.csv. Data sets for the graphs are fetched and managed via the JavaScript logic in graph-view.ts and rendered in index.js:

```
function createGraphView(graphSpec) {
  return {
```

```
    spec: graphSpec,
    getSeriesData: (varId, sourceName) => {
      const series = model.getSeriesForVar(varId, sourceName);
      return series.map(point => ({
        x: point.x,
        y: point.y,
        label: format(point.y, graphSpec.yFormat)
      }));
    }
  };
}
```

Scenario Comparison (Dual-Chart Display): Two charts can be displayed concurrently for comparative scenario analysis. Initialisation occurs explicitly in the frontend JavaScript (index.js):

```
function initGraphsUI() {
  const firstGraphSpec = [...coreConfig.graphs.values()][0];
  const secondGraphSpec = [...coreConfig.graphs.values()][1];
  showGraph(firstGraphSpec);
  showGraph1(secondGraphSpec);
}
```

Interactive Chart Controls:

Zoom and Pan (Desktop & Mobile): Enhanced interactivity through chartjs-plugin-zoom enables detailed exploration of plotted data:

Desktop:

- Wheel Zoom: Scroll wheel zoom capability.
- Mouse Drag: Left-click and drag functionality for chart navigation.

Mobile:

- Pinch-to-Zoom: Two-finger gestures allow scaling.
- Single-Finger Pan: Long-press and drag functionality.

```
const chartConfig: ChartConfiguration = {
    type: 'line',
    data,
    options: {
        plugins: {
        zoom: {
          zoom: {
            enabled: true,
            mode: 'xy',
            speed: 0.1,
            rangeMin: {
```

```
              // Format of min zoom range depends on scale type
              x: -1,
            },
            rangeMax: {
                // Format of max zoom range depends on scale type
              x: 11,
            },
          },
          pan: {
            enabled: true,    // Enable panning
            mode: 'xy',         // Allow panning in the x and y directions
            rangeMin: {
              // Format of min pan range depends on scale type
              x: -1,
            },
            rangeMax: {
                // Format of max pan range depends on scale type
                x: 11,
            },
          }
        }
      }
    }
}
```

Reset Zoom Feature: Reset buttons are clearly implemented in JavaScript (index.js), enabling easy resetting of zoom levels to default views:

```
const ResetZoomButton = document.getElementById(\"resetZoom\");
ResetZoomButton.onclick = function() {
  graphView.chart.resetZoom();
};
```

Interactive Tooltip: Data point tooltips provide instant feedback about precise data values and timestamps upon mouse hover as shown in Figure 3.2, enhancing accuracy and interpretability. This functionality is directly handled by Chart.js internal settings defined in the GraphView initialisation:

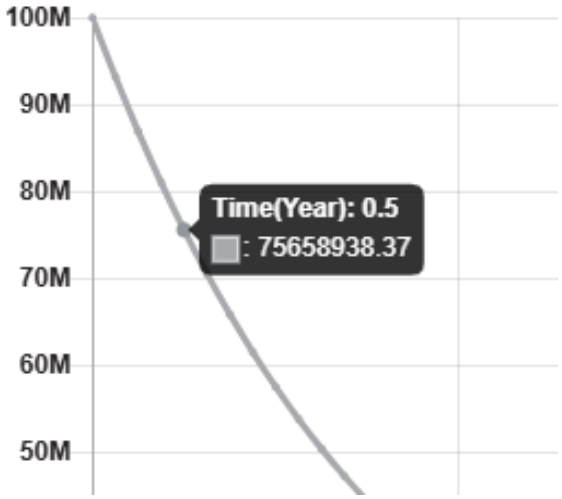

Figure 3.2 Tooltip of the plot.

```
const graphView = new GraphView(canvas, viewModel, options, tooltipsEnabled,
xAxisLabel, yAxisLabel);
```

The tooltip feature is implemented within the chart configuration using the Chart.js library and provides real-time display of precise data values and corresponding time points when users hover over specific data points. This functionality is defined in the chart configuration object as follows:

```
const chartConfig: ChartConfiguration = {
    type: 'line',
    data,
    options: {
        tooltips: {
        enabled: true, // TODO: Make configurable
        callbacks: {
          label: function(tooltipItem, data) {
            const dataset = data.datasets[tooltipItem.datasetIndex];
            const value = parseFloat(String(tooltipItem.yLabel)).toFixed(2);
            const label = dataset.label || '';
            return `${label}: ${value}`;
          },
          title: function(tooltipItems, data) {
            const title = tooltipItems[0].xLabel;
            return `Time(Year): ${title}`;
          }
        }
      }
    }
}
```

Technical Implementations:

JavaScript Integration: Comprehensive frontend logic is encapsulated in `index.js` and `graph-view.ts`, managing event listeners, parameter updates, graph rendering, and data handling.

CSS Responsive Layout (`index.css`): Ensures consistency across devices through flexible layout management:

```
.graph-inner-container {
  width: 100%;
}
#top-graph-canvas, #top-graph-canvas1 {
  width: 100%;
  min-height: 250px;
}
```

Development Overlay: The overlay component (`dev-overlay.js`) assists during development, providing quick debugging messages:

```
export function initOverlay() {
  const overlayElem =
document.getElementsByClassName('overlay-container')[0];
  updateOverlay(overlayElem, messagesHtml);
}
```

4. Deployment and Maintenance

This section outlines the deployment requirements, procedures, and recommended maintenance strategies for efficiently managing and updating the System Dynamics visualisation platform.

4.1 Deployment Requirements

The visualisation interface is designed as a **static webpage**. As such, it can be deployed rapidly on standard web servers or hosted effortlessly via GitHub Pages for maximum accessibility and ease of maintenance.

Static Web Hosting: No database or dynamic backend required.

Node.js Environment: `Node.js` LTS version 20 or later (recommended version 22) is necessary to build the project locally or within CI pipelines.

Git and GitHub Account: Essential for source control and quick deployment using GitHub Actions.

4.2. Deployment Procedures

The recommended deployment strategy is through **GitHub Pages**, leveraging automated deployment scripts configured via GitHub Actions.

4.2.1 GitHub Pages Configuration

Ensure the GitHub repository is configured to use GitHub Pages by selecting the `gh-pages` branch as the source for the site.

4.2.2 Deployment Commands (Local Deployment)

To manually build and preview locally, the following commands are utilised:

```
npm install
```

```
npm run build
```

The static site output will be generated within the directory:

```
./packages/app/public
```

Deploy this directory content directly onto any static web server (e.g., Nginx, Apache).

An example GitHub Actions workflow (`.github/workflows/deploy.yml`) automates the deployment process upon trigger:

```yaml
name: Deploy to GitHub Pages
on:
  workflow_dispatch:
    branches:
      - master  # Branch to deploy from

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Setup Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '22'  # Recommended Node.js version

      - name: Install dependencies
        run: npm install

      - name: Build project
        run: npm run build

      - name: Deploy to GitHub Pages
        uses: peaceiris/actions-gh-pages@v3
        with:
          github_token: ${{ secrets.WORKFLOWS }}
          publish_dir: ./packages/app/public  # Static output directory
```

Workflow Explanation:

- **Checkout code**: Pulls the latest version of the master branch.
- **Setup Node.js**: Set up Node.js environment for building the project.
- **Install dependencies**: Installs project dependencies defined in package.json.
- **Build project**: Executes build command generating static files.
- **Deploy to GitHub Pages**: Pushes the built site to the gh-pages branch for hosting.

This automation significantly streamlines the deployment process, reducing manual intervention and potential errors.

4.3 Maintenance strategy

A clear maintenance strategy is essential for the long-term usability and accuracy of the visualisation tool. It involves regularly updating the system dynamics model and frontend interface components through a structured pipeline.

Step-by-step updating procedure:

1. **Replace the .mdl file**:
    - Substitute the current model (sdms.mdl) in the root directory with an updated .mdl file.
2. **Update configuration** (sde.config.js):
    - Adjust configurations or parameters in the config directory (inputs.csv, outputs.csv, etc.) to match changes in the new model.
3. **Rebuild the project**:

```
npm run build
```

4. **Deploy updated model**:
    - Push changes to GitHub to trigger the automated deployment pipeline via GitHub Actions.

Updating the Frontend Interface:

1. **Update source files**: Modify frontend code (index.js, graph-view.ts, index.css, and other frontend assets).
2. **Local testing**: `npm run dev`
3. **Build static assets**: `npm run build`
4. **Automated deployment**:
   Commit and push changes to the master branch. GitHub Actions pipeline automatically deploys updated assets to GitHub Pages.

Recommended Practices for SD Interface Project Effective Maintenance:

- **Branch management**: Use branches (`feature/`, `fix/`) for specific enhancements or fixes before merging into the `main` or `master` branch, maintaining clarity in change history.
- **Dependency management**: Regularly update dependencies, specifically security patches and critical updates, via `npm`: `npm run update`

5. Appendices

5.1 Dependencies

- ❖ `Node.js`: Version 20 or later (recommended 22 for latest stability and performance).
- ❖ `npm`: Package manager to handle the installation and management of all dependencies.

Library / Plugin	Version
@sdeverywhere/build	0.3.4
@sdeverywhere/check-core	0.1.2
@sdeverywhere/cli	0.7.23
@sdeverywhere/plugin-check	0.3.14
@sdeverywhere/plugin-config	0.2.4
@sdeverywhere/plugin-vite	0.1.8
@sdeverywhere/plugin-wasm	0.2.3
@sdeverywhere/plugin-worker	0.2.3
chartjs-plugin-zoom	0.7.7

5.2 Licensing

All incorporated libraries and frameworks are open-source and licensed under permissive terms, supporting flexible adaptation and further development. The System Dynamics Interface (SDsimulator) is all under the MIT License. Copyright (c) Heriot-Watt University.

Library	License	Summary
SDEverywhere	MIT License	Permissive license allowing reuse and modification.
chartjs-plugin-zoom	MIT License	Free for personal and commercial use with attribution.

The **MIT License** [13] ensures that both SDEverywhere and chartjs-plugin-zoom can be freely used, modified, and distributed with minimal restrictions, thereby enabling this platform to be adapted and extended in future research and policy applications.

References

[1] T. Franklin, P. Styring, and T. Baker, "Financing a circular chemical economy."

[2] N. M. Bocken and S. W. Short, "Unsustainable business models–Recognising and resolving institutionalised social and environmental harm," *J. Cleaner Prod.*, vol. 312, p. 127828, 2021.

[3] B. Xu, U. Bititci, M. Marques, M. Jiang, and J. Xuan, "Towards a circular chemical economy: Stakeholders' perspectives," 2021. https://www.thechemicalengineer.com/features/towards-a-circular-chemical-economy-stakeholders-perspectives/

[4] J. Kirchherr, N. H. N. Yang, F. Schulze-Spüntrup, M. J. Heerink, and K. Hartley, "Conceptualizing the circular economy (revisited): an analysis of 221 definitions," *Resour. Conserv. Recycl.*, vol. 194, p. 107001, 2023.

[5] M. Roci, N. Salehi, S. Amir, S. Shoaib-ul-Hasan, F. M. Asif, A. Mihelič, and A. Rashid, "Towards circular manufacturing systems implementation: A complex adaptive systems perspective using modelling and simulation as a quantitative analysis tool," *Sustain. Prod. Consum.*, vol. 31, pp. 97–112, 2022.

[6] L. Schoenenberger, A. Schmid, R. Tanase, M. Beck, and M. Schwaninger, "Structural analysis of system dynamics models," *Simul. Model. Pract. Theory*, vol. 110, p. 102333, 2021.

[7] G. P. Richardson, "Reflections on the foundations of system dynamics," *Syst. Dyn. Rev.*, vol. 27, no. 3, pp. 219–243, 2011.

[8] J. Song, Y. Yuan, K. Chang, B. Xu, J. Xuan, and W. Pang, "Exploring public attention in the circular economy through topic modelling with twin hyperparameter optimisation," *Energy AI*, vol. 18, p. 100433, 2024.

[9] D. C. Pigosso and T. C. McAloone, "Making the transition to a circular economy within manufacturing companies: The development and implementation of a self-assessment readiness tool," *Sustain. Prod. Consum.*, vol. 28, pp. 346–358, 2021.

[10] Climate Interactive, "SDEverywhere: Translate Vensim models to C and web apps," *GitHub*, 2018. [Online]. Available: https://github.com/climateinteractive/SDEverywhere. [Accessed: Mar. 12, 2025].

[11] Chart.js Contributors, "chartjs-plugin-zoom: Zoom and pan interactions for Chart.js," *GitHub*, 2017. [Online]. Available: https://github.com/chartjs/chartjs-plugin-zoom. [Accessed: Mar. 12, 2025].

[12] Ventana Systems, "*Vensim* (Version 10.2.1)" [Computer software]. 2024. [Online]. Available: https://vensim.com/. [Accessed: Mar. 12, 2025].

[13] Open Source Initiative, "The MIT License," [Online]. Available: https://opensource.org/licenses/MIT. [Accessed: Mar. 12, 2025].