Kakuro Game Documentation

David Sequera, Daniel Gutierrez

November 27, 2023

Contents

1 Introduction **1**
1.1 Purpose . . . 2

2 Requirements **2**
2.1 Rules of the Game . . . 2
2.2 Roles and Actions . . . 2

3 System Requirements **2**
3.1 Interface Requirements . . . 2
3.2 Quality Atributes . . . 3
3.2.1 Performance . . . 3
3.2.2 Usability . . . 3
3.2.3 Maintainability . . . 3

4 Design **3**
4.1 Architecture . . . 3

5 Data Structures **5**
5.1 Logic Structures . . . 5
5.2 Main Structures . . . 6

6 Implementation **8**
6.1 Technologies . . . 8

7 Algorithms **8**
7.1 Board Initialization Algorithms . . . 8
7.2 Verification Algorithms . . . 11
7.3 Machine Player Algorithms . . . 12
7.3.1 Brute Force . . . 12
7.3.2 Search . . . 13
7.4 Game Board Algorithms . . . 13

8 Game Modes **16**

9 Conclusion **16**

1 Introduction

This document provides a description of the Kakuro game, including its rules, the requirements, data structures, design, implementation, algorithms and complexity technologies used for implementation, a detailed interface description, and how the game board interacts.

1.1 Purpose

The purpose of this project is to implement a Kakuro game using the algorithmic techniques learned in the course. The project is divided in the software design and implementation of the game, and the algorithmic techniques used to solve the different challenges game.

2 Requirements

In order to define the requirements of a game it is useful to take into account the rules of the game, the roles, and the actions that every roll could have.

2.1 Rules of the Game

Kakuro is a number puzzle game played on a grid. The objective is to fill in the grid with numbers from 1 to 9 such that:

- Each row and column must contain each number from 1 to 9 exactly once.
- The numbers in each **shaded** (stack) cell must sum up to the value given in that cell.
- No two cells in the same row or column can contain the same number.

2.2 Roles and Actions

There are 2 main roles that are defined in classic Kakuro game, user and board. The user tries to complete correctly the board and the board verifies if the solution given by the user is correct. Notice that this part section only emphasise the roles and actions that are importan for the game, but there are other derivate roles and actions that are relevant for the design of the system, this roles and actions that can be specified or inherit .

- **Board:** Must create the board distribution with a unique solution and verify that each row with a value and each column with a value sum
  - **Create Board:** Must create a board with the following rules:
    - The board must have a unique solution.
    - rows and columns don't have repeated numbers.
  - **Row:** The sum of the values in the row must be equal to the value of the row.
  - **Column:** The sum of the values in the column must be equal to the value of the column.
- **Player or user:** Must fill the board the cells with numbers from 1 to 9 such that the sum of the values in the row or column is equal to the value of the row or column.

3 System Requirements

This section talks about the specific system design requirements that are needed to implement the game. This part is centered in the specific requirements of the system with a more technical approach.

3.1 Interface Requirements

- The system must create a board and specify the size of the board.
- The system must give the user a guide to play the game.
- The system must give the user the option to play in different levels (the size of the board).
- The system should provide the user a response to the actions that the user makes.

3.2 Quality Atributes

This section talks about the quality attributes that are important for the system. The attributes are centered in making the system more usable and efficient, there are no metrics to measure the quality attributes, because that is not the purpose of this project. In order to measure usability the project only takes into account the use of good practices, and in terms of performance it wouln't be measured traditionally, instead the main algorithms are going to be analyzed in terms of complexity.

3.2.1 Performance

The system must be able to create a board with a unique solution in a reasonable time. The system must be able to verify if the solution given by the user is correct in a reasonable time.

3.2.2 Usability

The system must be easy to use and understand. The system must provide a guide to the user to play the game.

Practices:

- Use a good color palette contrast.
- Provide a guide to the user with the rules of the game and the action that can make.
- Give response to the user actions.

3.2.3 Maintainability

The system must be easy to maintain and extend. The system must be easy to understand and modify.

4 Design

In order to create a good system it is necessary to have a design that takes into a account the needs of the parts involved in the system. in this case the user expirience is very important part of the system, but also the game designer expirience. From the beginning of the project the goal of the system desing was to provide a useful and good user expirience, and the advantage for the game designer of not being tied to the previous implementations and being able to change segments of the game without affecting the other parts of the system.

4.1 Architecture

The project is a monolit architecture, this means that the project is on a single code base, altough is divided in different components based on the logic divitions of the game. this components communicate through interfaces, and hides the implementation of the other components. The main components are the following:

- **User Iterface:** This component is in charge of the user interaction with the game, it paints the board from the game interface and communicates the actions of the user to the game interface .
- **Game Iterface:** is in charge of the game logic, it creates the board and verifies if the solution given by the user is correct.
- **Machine Player:** are the algorithms that are used to solve the game. It sends the solution to the game interface.

The following diagrams show the architecture of the system. The most important diagram is the component diagram, because it shows the components and the interfaces that are used to communicate the components.

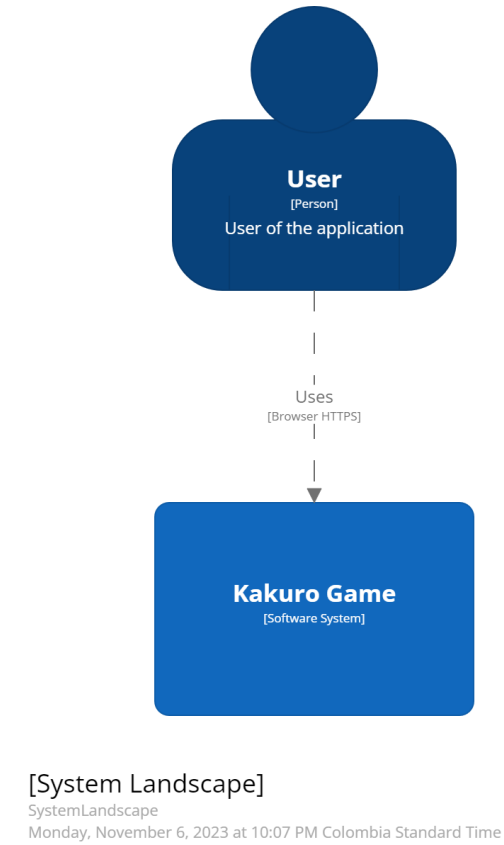

Figure 1: Context Diagram

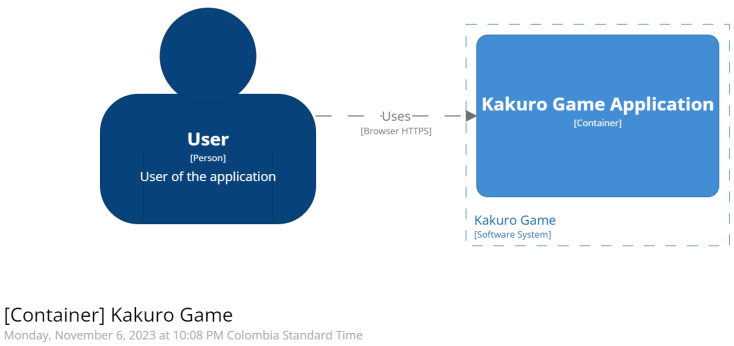

Figure 2: Containers Diagram

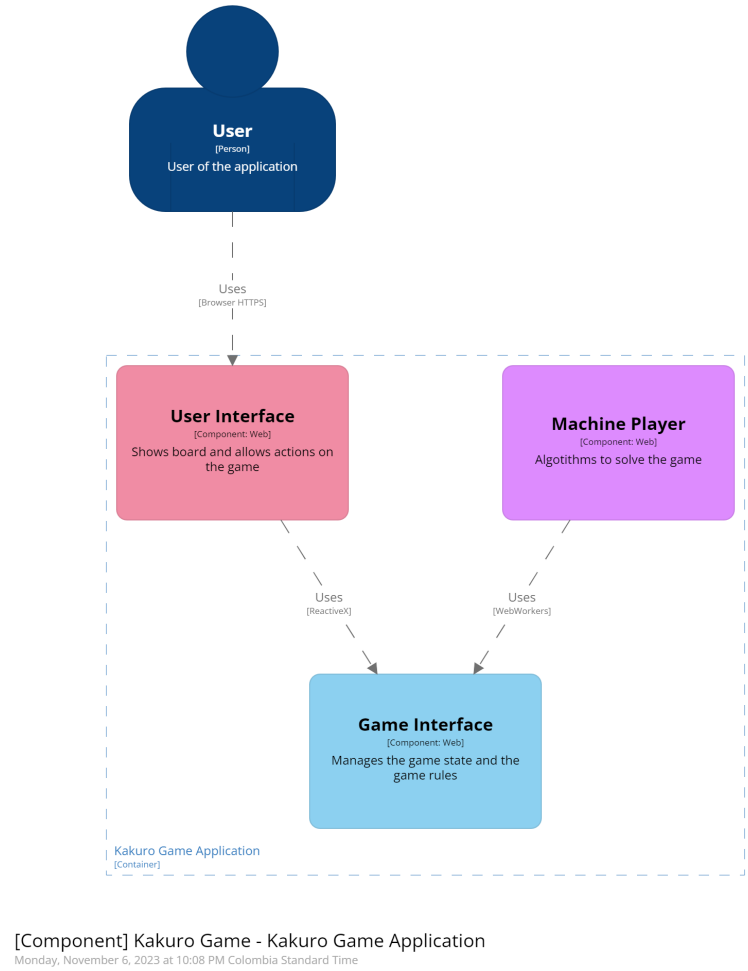

Figure 3: Components Diagram

This architecture is meant to be as separated as possible, this means that the components doesnt need to know the implementation of the other components, instead they only need to know the interface of the other components. This architecture allows to change the implementation of the components without affecting the other components. So the system is not tied to a specific implementation of the other components, it can implemented a serius of different data structures in order to allow the designer of the game to try as much ideas as wanted.

5 Data Structures

As it was mentioned in the previous section the system is not tied to a specific implementation of the other components, in this case the system uses a series of data structures that could be changed if any other solution is proposed. This structures are part of the game interface.

5.1 Logic Structures

Part of the design of the system inclueds a previous analysis of the data structures that are going to be used in the system:

- **Board:** Saves all the cells and the state of the game.
- **Cell:** A cell can be of three types:

1. **Blocked Cell:** These cells are not part of the game and are usually indicated with a different color.
2. **Input Cell:** These cells are user-editable and can contain values from 1 to 9.
3. **Stack Cell:** These cells represent the sums of rows and columns. They hold a tuple of two elements, indicating the sum for the row and column.

5.2 Main Structures

For this implementation of the game it uses mainly two data structures are used in the game:

Simple Structures

- **Board:** A 2-dimensional array of cells.
- **Type Board**: A 2-dimensional array of types of cells.
- **Cell:**

  1. **Blocked Cell:** These cells are represent by an integer assinged with 0.
  2. **Input Cell:** These cells are represented by an integer between 1 to 9, or 0 if it has not being fill with a number.
  3. **Stack Cell:** These cells are represented by a tuple of two elements, indicating the sum for the row and column.

  an alternative could be to use all tuples and differentiate them with "[#, #]" for blocked cells, "[N, N]" for stack cells, and "[#, N]" for input cells .

Complex Structures

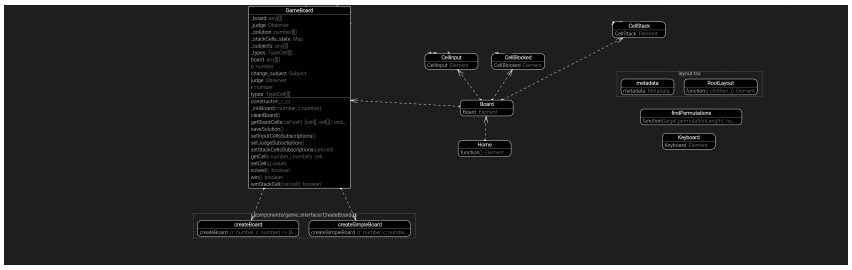

Figure 4: Class Diagram

- **Game:**
  Principals Procedures

  First

  Analisis
  Procedure Description: createBoard

  The **createBoard** procedure plays a central role in generating Kakuro game boards of varying complexity in LaTeX. This procedure initializes two matrices: the **board** matrix, responsible for storing cell values, and the **types** matrix, which designates the type of each cell (input, blocked, or part of a vertical or horizontal sum). Utilizing auxiliary functions, this comprehensive procedure sets the board's structure by placing blocked cells on the border, establishing a path for input cells, marking black cells, assigning values to input cells, and determining stack cells for both rows and columns.

Relation with Other Functions:

The **createBoard** procedure collaborates with several auxiliary functions to achieve a sophisticated Kakuro game board. Shared functions include setting blocked cells on the border (**setBlockedCellsBorder**), marking black cells (**setBlackCells**), assigning values to input cells (**setInputCellsValue**), and establishing stack cells for rows and columns (**setRowsStackCells** and **setColumnsStackCells**). This procedure, along with its auxiliary counterparts, provides a modular and scalable approach to crafting diverse Kakuro puzzles with varying levels of challenge.

Design

Entrance:

$$r : \text{Number of rows}$$
$$c : \text{Number of columns}$$

Exits:

$$\texttt{createBoard}(r, c) : \begin{bmatrix} \text{Matriz } 1 \times 2 \\ \text{Dúo que contiene una matriz board y otra matriz types} \end{bmatrix} = \begin{bmatrix} \text{board} \\ \text{types} \end{bmatrix}$$

Procedure createBoard
Require: TypeCell.INPUT, type of board entry to place numbers.

Procedure `createBoard`(r, c)
1: `board[][]` ← createMatrix(r, c, 0)
2: `types[][]` ← createMatrix(r, c, TypeCell.INPUT)
3: `setBlockedCellsBorder(types)`
4: `setBlackCells(types, r, c);`
5: `setInputCellsValue(types,board, r, c)`
6: `setColumnsStackCells(types, board, r, c)`
7: `return board, types`

Complexity

For the previous procedure, code inspection was used to find the complexity, so:

- The time complexity is dominated by the function calls `createMatrix`, `setBlockedCellsBorder`, `setBlackCells`, `setInputCellsValue`, and `setColumnsStackCells`.
- The time complexity of `createMatrix` is $O(r \times c)$ since it creates two matrices of size $r \times c$.
- `setBlockedCellsBorder` and `setBlackCells` loop through the `types` arrays once, so they both have complexity $O(r \times c)$.
- `setInputCellsValue` loops through the `types` array and performs operations that depend on the size of the board, so its complexity is also $O(r \times c)$.
- `setColumnsStackCells` loops through the `types` and `board` arrays once, so its complexity is $O(r \times c)$.
- In summary, the total complexity of the procedure `createBoard` is $O(r \times c)$.

Invariant

`board` contains the game board information, and `types` contains information about the type of each cell on the board.

6 Implementation

6.1 Technologies

The Kakuro game is implemented using the following technologies:

- **Framework:** Next.js
- **Language:** TypeScript
- **Styles:** Tailwind CSS
- **Reactive Programming:** RxJS
- **Web**: Web Workers

7 Algorithms

7.1 Board Initialization Algorithms

Problem Analysis

The problem involves initializing a game board with specific rules, including setting input cells, stack cells, blocked cells, and calculating row and column sums.

Problem Design

In addressing the given challenge, we have devised two distinct board initialization algorithms: `createSimpleBoard` and `createBoard`.

- `createSimpleBoard`: This algorithm is designed to generate a game board with straightforward rules, prioritizing simplicity and ease of understanding.
- `createBoard`: In contrast, this algorithm is more intricate. It encompasses a comprehensive set of rules for initializing the board. Notably, it incorporates the placement of black cells, input cells, and stack cells, guided by specific criteria and conditions.

These algorithms offer flexibility in catering to different requirements — from a basic and uncomplicated board generation to a more sophisticated approach that considers various factors for a nuanced game board initialization.

Complexity

The complexity of the algorithms is influenced by the size of the board (r x c). The implementation takes advantage of TypeScript and uses the 'TypeCell' enum for cell types.

Procedure Description: createSimpleBoard

The **createSimpleBoard** procedure is designed to generate a foundational Kakuro game board in LaTeX. This function produces two crucial matrices: the **board** matrix for storing cell values and the **types** matrix, which indicates the type of each cell (input, blocked, or part of a vertical or horizontal sum). The procedure initializes the board with simple input cells using the **setSimpleInputCells** auxiliary procedure. It then calculates and assigns sum cells for both rows and columns while ensuring the proper marking of blocked cells. This procedure establishes the groundwork for a basic Kakuro puzzle, providing simplicity in the generated boards.

Relation with Other Procedures:

The **createSimpleBoard** procedure collaborates with other auxiliary functions shared with the more intricate **createBoard** function. Both functions utilize common auxiliary operations, such as setting blocked cells and determining sum cells for rows and columns. While **createSimpleBoard** specializes in crafting straightforward Kakuro boards, **createBoard** is designed for more complex and challenging game levels. The shared auxiliary functions create a cohesive relationship, allowing for a modular and scalable approach to Kakuro board generation in different game scenarios.

Procedure createMatrix

The `createMatrix` procedure is designed to generate a two-dimensional array, representing a matrix, based on the given inputs. It requires three parameters: $rows$, indicating the number of rows in the matrix; $cols$, specifying the number of columns; and def, which sets the default value for each element in the matrix. The resulting matrix is initialized with the provided default value.

Relation with Other Functions

The `createMatrix` procedure serves as a fundamental utility in various matrix-related operations within our program. It is frequently employed by other procedures, such as `createBoard`, to create the initial state of game boards. Additionally, it is a building block for procedures like `setBlockedCellsBorder` and `setInputCellsValue`, which require the generation of matrices with specific dimensions and default values. Therefore, `createMatrix` plays a crucial role in facilitating the initialization and configuration of matrices used throughout the application.

Procedure setBlockedCellsBorder

The `setBlockedCellsBorder` procedure takes a matrix of cell types (`types`) as input and modifies it to set the border cells as blocked. The function begins by extracting the dimensions of the input matrix (`r` and `c`). Subsequently, it iterates over the rows and columns, assigning the `TypeCell.BLOCKED` value to the cells located at the beginning of each row and column. This process effectively creates a border of blocked cells around the matrix.

Relation with Other procedures

The `setBlockedCellsBorder` procedure plays a crucial role in the initialization of game boards, particularly in scenarios where certain cells need to be marked as blocked to establish game constraints. This procedure is often called within broader functions like `createBoard` to ensure that the outer edges of the board are appropriately designated as blocked cells. Additionally, it collaborates with other functions like `setRowsStackCells` and `setColumnsStackCells`, contributing to the overall configuration of the board's structure.

Procedure setBlackCells

The `setBlackCells` procedure takes a matrix of cell types (`types`) along with the number of rows (`r`) and columns (`c`) as input. It aims to randomly set a quarter of the cells in the matrix as black (`TypeCell.BLOCKED`). The function begins by calculating the target count of black cells based on the total number of cells (`r * c`). It then iterates until the current count of black cells (`currentBlackCount`) reaches the target count. In each iteration, the function generates random indices (`i` and `j`) and checks if the corresponding cell is not already blocked. If not, it marks the cell as blocked and increments the black cell count.

Relation with Other Procedures

The `setBlackCells` Procedure is an integral part of the board creation process, often called within functions like `createBoard`. Its role is to introduce a controlled randomness to the placement of black cells, contributing to the diversity and challenge of the generated game boards.

This Procedur collaborates with other initialization procedure like `setBlockedCellsBorder` and `setSimpleInputCells` to collectively configure the board's structure and properties.

Procedure setSimpleInputCells

Esta función se encarga de establecer las celdas de entrada de manera simple en el tablero. Utiliza una secuencia predefinida de números del 1 al 9, y asigna valores a las celdas de entrada de manera aleatoria, evitando conflictos con las celdas vecinas en la misma fila y columna. Si todas las celdas están ocupadas con números excluidos, la celda actual se bloquea. Este procedimiento asegura un inicio del juego con un conjunto equilibrado de números de entrada en el tablero.

Relation with Other Procedures

Esta función está estrechamente relacionada con la inicialización del tablero de juego. Se encarga de establecer de manera aleatoria y equilibrada las celdas de entrada en el tablero, siguiendo las restricciones del juego del Sudoku. Además, interactúa directamente con las funciones de creación y manipulación de matrices, así como con la función que bloquea celdas en el borde del tablero. La ejecución exitosa de 'setSimpleInputCells' contribuye significativamente a la creación de un tablero de Sudoku válido y desafiante para el jugador.

Procedure setInputCellsValue

The procedure setInputCellsValue initializes the input cells on the board. It iterates through each cell, identified as an input cell in the types matrix, and populates it with a random number from the sequence [1, 2, 3, 4, 5, 6, 7, 8, 9]. Before assigning a number to the input cell, it excludes numbers already present in the same row and column. If all numbers are excluded, the cell type is set to TypeCell.BLOCKED. The process continues until all input cells are filled.

Relation with Other Procedures

This procedure plays a crucial role in the initialization of the board, providing initial values to the input cells. The assigned values are essential for the subsequent execution of functions that involve the board's constraints and operations, ensuring the input cells meet the required conditions. The randomized assignment process prevents pre-determined patterns and enhances the diversity of generated boards.

Procedure setInputCellsValue

The procedure setRowsStackCells iterates through each row, from the last to the first, and sets stack cells based on the sum of numbers in each row. It calculates the sum by iterating through each column in the current row, excluding blocked cells. If the sum is greater than zero, it designates the current cell as a stack cell (TypeCell.STACK) and assigns a tuple [sum, 0] to represent the row's cumulative sum. This process ensures that stack cells capture the cumulative sum of numbers in each row.

Relation with Other Procedures

This procedure contributes to the initialization of stack cells in the board. The stack cells, identified by TypeCell.STACK, store cumulative row sums, which are essential for meeting the constraints of the puzzle. The values stored in these stack cells serve as a foundation for subsequent operations and functions that involve row-wise calculations and constraints.

Procedure setColumnsStackCells

The procedure setColumnsStackCells iterates through each column, from the last to the first, and sets stack cells based on the sum of numbers in each column. It calculates the sum by iterating through each row in the current column, excluding blocked and existing stack cells. If the sum is greater than zero, it designates the current cell as a stack cell (TypeCell.STACK) and assigns a tuple [0, sum] to represent the column's cumulative sum. If the current cell is already marked as a stack cell, it updates the existing stack cell by adding the new sum to the existing row sum.

Relation with Other Procedures

This procedure complements the initialization of stack cells in the board by handling the column-wise calculations. The stack cells, identified by TypeCell.STACK, store cumulative column sums, which are crucial for fulfilling puzzle constraints. The values stored in these stack cells contribute to the overall structure of the puzzle and serve as a basis for subsequent puzzle-solving algorithms.

7.2 Verification Algorithms

Analisis

The code provided implements the method `setJudgeSubsctiption`, which is responsible for establishing a subscription (`_judge`) to a set of cells of type `TypeCell.STACK`. Here is an analysis of the problem and the code:

- **Goal:** The goal of the function is to establish a subscription to a specific set of cells of type `TypeCell.STACK` and perform actions when these cells emit events through their `Subjects` .

Design

Entrances:

There are no direct input parameters to the function, but it is assumed that there are properties and methods related to the object containing the function.

Exits:

There are no explicit exits from the function, but actions such as updating the internal state and emitting events are performed via the `Subject change_subject`.

Procedure `setJudgeSubscription` Require: Subject, observable event observables, is an object or data structure that emits a sequence of items or events over time. push, is a method to add in an array an object TypeCell.STACK, board input type for tracks winStackCell, function to know if the board is fine.

Procedure `setJudgeSubscription`():

```
1: change_subject ← new Subject<void>()
2: stackCells ← []
3: For each cell in board:
4:      If cell.type is TypeCell.STACK:
5:          stackCells.push(getCell(cell.i, cell.j))
6: observables ← stackCells.flatMap(cell => cell.subject)
7: observable ← merge(...observables)
8: _judge ← {
9:      next<- cell ⇒
10:          _stackCells_state.set(`cell.i−cell.j`, winStackCell(cell))
```

```
11:         change_subject.next()
14: }
15: observable.subscribe(_judge)
```

Complexity:

La complejidad temporal depende principalmente del número de celdas del tipo `TypeCell.STACK` y de la complejidad de las acciones realizadas en los métodos de `_judge`. En general, la función puede tener una complejidad lineal.

Invariant:

La invariante asegura que, antes de suscribir el objeto `_judge` al observable, se han inicializado correctamente los `Subjects` y se han agregado las celdas relevantes al arreglo `stackCells`.

7.3 Machine Player Algorithms

7.3.1 Brute Force

Analisis

The problem addressed by the `play` function is related to performing actions in a puzzle game based on the current state of the board and cell types. Here is an analysis of the problem and function:

Design

Entrances:

$board$: matrix board

$types$: matrix data types

Exits:

- `response: ActionMessage` - An object that contains the actions and response of the game.

Procedure play
Require: TypeCell.INPUT, type of board entry to place numbers. TypeCell.STACK, board input type for tracks getNumberOfInpuCells, function to know the amount of input cells clean_permutation, function to know the permutations of the possible numbers of a track Math.random(), function to get a random number push, means adding something into the array

Procedure `play(board[][], types[][])`

```
1: r, c = board.length, board[0].length
2: i=0, j=0
3: if (stack_cells.size === 0) {
4:     for (i = 0; i < r; i++) {
5:         for (j = 0; j < c; j++) {
6:             if (types[i][j] !== TypeCell.STACK) continue
7:             [row_count, column_count] = getNumberOfInpuCells(board, types, i, j)
8:             stack_cells.set(`i,j`, {i, j, row_permutations <- get(board[i][j][0],
row_count).slice(0),
9:                     col_permutations: get(board[i][j][1], column_count).slice(0)})
10:         }
11:     }
13:     clean_permutation(board, types, stack_cells)
15: }
16: action[] = []
17: stack_cells( <= {
18:     [i, j] = [cell.i, cell.j]
```

```
19:     [row_permutations, col_permutations] = [cell.row_permutations,
cell.col_permutations]
20:     row_per = row_permutations[Math.floor(Math.random() *
row_permutations.length)]
21:     col_per = col_permutations[Math.floor(Math.random() *
col_permutations.length)]
22:     if(row_per)
23:         row_per.forEach( <= {
24:             actions.push({i, j:  j+index+1, value})}
25:         }
26:     if(col_per)
27:         col_per.forEach( <= {
28:             actions.push({i:  i+index+1, j, value})}
29:         }
30: }
31: response= {"ok"}
32: return response
```

Complexity:

Time:

- Creating the `board` and `types` arrays takes time $O(r \times c)$, since they are initialized with sizes proportional to r and c.
- `setBlockedCellsBorder` and `setBlackCells` iterate over all the cells of the arrays, so their time is $O(r \times c)$.
- `setInputCellsValue`, `setRowsStackCells`, and `setColumnsStackCells` also loop through all cells of the arrays, so each is $O(r \times c)$.
- The creation of `stack_cells` and the call to `clean_permutation` has total complexity $O(1)$.

Therefore, the total complexity is $O(r \times c)$.

Invariant:
the invariant guarantees that the `board` and `types` arrays contain the desired configuration of the game board.

7.3.2 Search

7.4 Game Board Algorithms

Procedure initBoard

The procedure initBoard initializes the board (board) and its corresponding types based on the provided number of rows (r) and columns (c). It checks the dimensions of the board, and if both r and c are less than 5, it invokes the createSimpleBoard procedure to generate a simple board. Otherwise, it utilizes the createBoard procedure for a more complex board.

Relation with Other Procedures

This procedure is a crucial part of the puzzle initialization process. It determines whether to create a simple board or a more complex one based on the input dimensions. The decision to use a simple or complex board may depend on factors such as the size of the puzzle or the desired difficulty level. The initialization sets the stage for subsequent puzzle-solving algorithms and user interactions.

Procedure cleanBoard

The procedure cleanBoard aims to reset the board by setting the values of input cells to zero. Input cells, identified by TypeCell.INPUT, represent the initial values provided as part of the puzzle. By resetting these cells to zero, the board is effectively cleaned, allowing for a fresh start or the preparation for a new puzzle.

Relation with Other Procedures

This procedure is often employed in puzzle-solving algorithms to prepare the board for the application of constraint-solving techniques. By cleaning the board, the algorithm can start with a consistent and well-defined initial state, ensuring that subsequent operations build upon a blank canvas. This process is integral to the overall puzzle-solving workflow, enhancing the efficiency and effectiveness of the solving algorithms.

Procedure setInputCellsSubscriptions

The `setInputCellsSubscriptions` procedure initializes and subscribes subjects for black cells in the board. It creates a 2D array of subjects (_subjects) corresponding to input cells. Then, it iterates through the board and, for each black cell marked as TypeCell.INPUT, it creates a new subject and associates it with the corresponding cell. Afterward, it subscribes to stack cells using the `setStackCellsSubscriptions` procedure.

Relation with Other Procedures

This procedure is closely related to `setStackCellsSubscriptions`, which handles subscriptions for stack cells. The two procedures work together to set up the necessary subscriptions for various cell types in the board.

Procedure setStackCellsSubscriptions

The `setStackCellsSubscriptions` procedure takes a cell as input and sets up subscriptions for the corresponding stack cells in both rows and columns. It retrieves the stack cells in the row and column using the `getStackCells` function and creates observables for each. The procedure then merges these observables, mapping each emission to the input cell. Finally, it subscribes to the merged observables and saves the resulting subjects for rows and columns in the _subjects array.

Relation with Other Procedures

This procedure is an integral part of the broader system for managing cell subscriptions. It is often used in conjunction with procedures like `setInputCellsSubscriptions` to establish a comprehensive subscription system for different cell types on the board.

Procedure setJudgeSubscription

The `setJudgeSubscription` procedure plays a pivotal role in the overall functionality of the Sudoku-solving system. It begins by initializing a `change_subject`, which serves as a communication channel for signaling changes in the stack cells. Simultaneously, it populates an array `stackCells` by iterating over the entire board and collecting every cell marked as a stack cell.

Next, the procedure generates an observable (`observable`) by flat-mapping the subjects of each stack cell. This observable effectively amalgamates all individual cell subjects into one observable stream. The judge object (_judge) is then defined with the necessary handlers: `next`, `error`, and `complete`. The `next` handler updates the internal state (_stackCells_state) and triggers the `change_subject` whenever a stack cell is modified.

Moreover, the judge object is a key element in the coordination of the Sudoku-solving logic. It serves as a central hub for processing changes in stack cells and responding accordingly. By subscribing the observable to the judge object, the procedure establishes a dynamic link that allows continuous monitoring of the stack cells.

Relation with Other Procedures

The `setJudgeSubscription` procedure is intricately linked with other components of the Sudoku-solving system. It often works in tandem with procedures like `setStackCellsSubscriptions`, which sets up subscriptions for individual stack cells. Additionally, it complements the broader architecture by coordinating with procedures responsible for updating the state of the board and handling various game-related functionalities. This inter-procedural collaboration ensures a seamless flow of information and facilitates real-time responsiveness to changes in the Sudoku board.

Procedure solved

The `solved` function determines whether the current Sudoku board is solved or not. It achieves this by comparing each cell in the internal board (_board) with the corresponding cell in the solution (_solution). The function returns a boolean value, `true` if all cells match, indicating a solved board, and `false` otherwise.

Relation with Other Procedures

The `solved` function plays a crucial role in the Sudoku-solving process. It is often used as a condition to check whether the puzzle has been successfully solved or if additional steps need to be taken. Integration with procedures responsible for updating and manipulating the board ensures that the solving algorithm operates within the context of the puzzle's state.

Procedure getCell

The `getCell` function retrieves information about a specific cell on the Sudoku board based on the provided row index (`i`) and column index (`j`). It returns a cell object with attributes such as row index (`'i'`), column index (`'j'`), cell value (`value`), cell type (`type`), and the associated subject (`subject`).

Relation with Other Procedures

The `getCell` function serves as a utility for obtaining detailed information about a particular cell. It is often used in conjunction with procedures that require specific cell details, such as setting up subscriptions or validating moves. The seamless integration of `getCell` enhances the overall modularity and clarity of the Sudoku-solving system.

Procedure setCell

The `setCell` function updates the value of a specified cell on the Sudoku board. It takes as inputs the row index (`i`), column index (`j`), and the new cell value (`value`). The function first checks if the target cell is an input cell; otherwise, it logs an error and exits. If the cell is valid, it updates the internal board (_board) and notifies the associated subject with the new cell information.

Relation with Other Procedures

The `setCell` function is closely related to the `getCell` function, as it uses it to retrieve the subject (`Subject<cell>`) associated with the modified cell. Additionally, procedures that rely on real-time updates, such as those managing subscriptions or judging the solution, may be triggered by changes made through `setCell`. The function's integration into the Sudoku-solving system ensures consistent and controlled modifications to the puzzle.

Procedure winStackCell

The `winStackCell` function determines whether a given cell wins the stack it belongs to. It takes a cell as input and extracts its value, assuming it is in the format `[rowValue, colValue]`. The function then obtains the cells in the same row and column of the stack using `getStackCells`. Afterward, it calculates the sum of values in both the row and column cells. The cell wins the stack if and only if the

row and column sums match the respective values.

Relation with Other Procedures

The `winStackCell` function is closely related to the `getStackCells` function, which provides the necessary information about the cells in the same row and column of the stack. Additionally, this procedure is utilized within the judging mechanism, specifically in the `setJudgeSubscription` procedure, to determine the winning state of stack cells. It plays a crucial role in evaluating the correctness of the Sudoku solution.

Procedure win:

The `win` function determines whether the game is won or not. It checks if there are stack cells in the current state (indicated by `_stackCells_state.size`) and if all the stack cells are in a winning state (indicated by `Array.from(_stackCells_state.values()).every(v => v)`). The function returns a boolean value indicating the overall game status.

Relation with Other Procedures

The `win` procedure is closely related to procedures that involve stack cells, such as `setJudgeSubscription` and `winStackCell`. It relies on the state of individual stack cells to determine the overall game-winning condition. Additionally, the `win` function is likely used in the game logic or UI to determine when the game has been successfully completed.

Function getStackCells:

The `getStackCells` function takes a cell as input and returns an ordered pair of arrays of cells representing the stack cells in the same row and column as the input cell. If the input cell is not of type `TypeCell.STACK`, the function returns `undefined`. The function uses the `getCell` function to retrieve the cells in the specified rows and columns, considering the input cell's position.

Relation with Other Functions

The `getStackCells` function is often used in conjunction with procedures that involve stack cells, such as `setRowsStackCells` and `setColumnsStackCells`. It provides a convenient way to obtain the stack cells associated with a given cell, which can be crucial for various game logic and state management tasks.

8 Game Modes

The game only has one classic mode with the rules specified at the beginning of the document, and it is meant to be played with boards of 3x3, 5x5, and 10x10. Altough it supports different sizes and types of boards they are not the main goal of the project. On the other hand, the solving algorithms are a little more capable (around 2 times the length of the maximun size of any or both dimensions), just to ensure it is capable of solving the smaller ones.

9 Conclusion

This document provides an overview of the Kakuro game, including its rules, the technologies used for implementation, and an in-depth description of the game's interface and communication with React.