

## ЗМІСТ

<b>АНОТАЦІЯ .....</b>	<b>3</b>
<b>ВСТУП .....</b>	<b>4</b>
<b>ПОСТАНОВКА ЗАДАЧІ .....</b>	<b>5</b>
<b>1 МОДИФІКАЦІЯ КЛІТИННИХ АВТОМАТІВ .....</b>	<b>6</b>
1.1 Огляд клітинних автоматів .....	6
1.2 Розширення вихідним інформаційним каналом .....	8
1.3 Розширення керуючим інформаційним каналом .....	9
1.4 Використання модифікованого автомату для розпізнавання необмеженої граматики.....	10
<b>2 МОДЕЛЬ ЕКОНОМІЧНИХ ВІДНОСИН .....</b>	<b>15</b>
2.1 Опис моделі.....	15
2.2 Використання керуючого та вихідного каналів.....	18
2.3 Експериментальне визначення оптимальної суми фіксованого податку .....	20
<b>3 ВІЗУАЛІЗАЦІЯ КЛІТИННИХ АВТОМАТІВ .....</b>	<b>33</b>
3.1 REACT ТА ІДЕЯ VIRTUAL DOM.....	33
3.2 CLOSURESCRIPT ТА OM .....	37
3.3 COMMUNICATING SEQUENTIAL PROCESSES ТА CORE.ASYNC .....	41
3.4 Огляд виконаної реалізації.....	45
<b>ВИСНОВКИ .....</b>	<b>58</b>
<b>СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ .....</b>	<b>59</b>
<b>ДОДАТОК А. ВИХІДНИЙ КОД ЗАСТОСУВАННЯ .....</b>	<b>60</b>

АНОТАЦІЯ  
кваліфікаційної роботи  
на здобуття академічного ступеня магістра

Тема: Моделювання процесів за допомогою модифікованих клітинних автоматів

Автор: Криворучко Іван Вікторович

Науковий керівник: Жежерун Олександр Петрович

Захищена «12» червня 2014 р.

Короткий зміст роботи:

В даній роботі було досліджено розширення клітинного автомату двома інформаційними каналами, що дозволяють надсилати спеціально сформовані повідомлення на всі клітини автомату, а також отримувати загальну інформацію про поточний стан автомату. Розширений таким чином автомат був використаний для створення розпізнавача необмеженої мови і для моделювання економічної взаємодії на ринку високотехнологічних товарів (розглянуто проблему вибору товару покупцем та формування ціни товару продавцем в умовах не лише необхідності конкурувати, а й сплачувати податки державі). Для відображення створених автоматів було реалізовано веб-застосування на основі бібліотек React, Om і підходу асинхронної взаємодії Communicating Sequential Processes (бібліотека `core.async`).

---

(підпис автора)

## ВСТУП

Від часів опублікування першого опису Гри Життя клітинні автомати набули широкого поширення серед різних поколінь математиків та програмістів. Простий за своєю суттю апарат дозволяє робити навіть малокваліфікованому спеціалісту цікаві експерименти, котрі переважно заключаються у генерації різноманітних візерунків та автономних віртуальних світів. Водночас, більш серйозні застосування клітинних автоматів хоч й існують, проте маловідомі широкому загалу.

Дещо зупинились і більш теоретичні дослідження клітинних автоматів: багато часу для їх аналізу витрачали науковці 1970-80х років, а зараз серйозні роботи присвячені даній моделі зустрічаються доволі рідко. Разом з тим, не зважаючи на всю зовнішню простоту моделі досі залишаються можливості її подальшого удосконалення, котрі можуть зробити клітинні автомати ще більш "виразною" мовою опису динамічних систем та процесів. Зокрема, цікавою виглядає можливість поєднання клітинних каналів з каналами даних, що дозволяють певним чином збирати та розсилати інформацію з усіх клітин автомату, таким чином доповнюючи локальну взаємодію клітин автомату можливістю глобальної взаємодії.

Можна припустити, що таке розширення моделі дозволить ще більш точно моделювати різноманітні процеси реального життя, особливо ті, які подібним чином поєднують локальні та глобальні зміни у своєму стані. Зокрема, такі клітинні автомати можуть виявитись гарною апроксимацією ринку, моделюючи як товарний обмін між різними його учасниками так і участь держави у формі збору податків та загального контролю ринку (якщо він присутній). Остання задача стає напрочуд актуальною у світлі останніх процесів, що відбуваються у нашому суспільстві - здатність відносно простим чином моделювати різні варіанти податкового законодавства може стати у нагоді під час можливих майбутніх реформ даної сфери.

## ПОСТАНОВКА ЗАДАЧІ

В рамках даної роботи буде досліджено можливість розширення стандартних клітинних автоматів двома додатковими каналами даних. Один з них на кожному кроці певним чином агрегує інформацію про поточний стан автомату та надає її користувачу, а другий - розповсюджує певну команду на всі клітини, кожна така команда змінює функцію переходу клітини в обхід звичних правил клітинного автомату. Модифікований таким чином автомат буде використаний для моделювання економічної взаємодії покупців, підприємств та держави. Також, буде створена система візуалізації клітинних автоматів з підтримкою описаних вище модифікацій.

# 1 МОДИФІКАЦІЯ КЛІТИННИХ АВТОМАТІВ

## 1.1 Огляд клітинних автоматів

Вперше клітинні автомати були запропоновані у 1940-50-х роках в роботах фон Неймана та Улама[1]. Проте наступні 30 років такий тип автоматів не дуже цікавив дослідників. У 1970-х Конвей запропонував так звану Гру Життя (Game of Life) - двовимірний клітинний автомат, що описувався простим набором правил, проте генерував напрочуд цікаво поведінку. В 1980-х глибоким дослідженням одновимірних клітинних автоматів займався Стівен Вольфрам, зокрема він запропонував їх класифікацію.

Клітинні автомати - особливий клас дискретних динамічних систем, дискретними в яких є час, простір та множина станів. Зазвичай клітинний автомат представляється як об'єднання однакових комірок (клітин), з'єднаних між собою. Разом клітини утворюють так звану клітинну решітку (форма якої може бути різноманітною). Клітина виступає скінченним автоматом, стан якого на кроці  $t+1$  визначається станом системи (а саме сусідів та самої клітини) на попередньому кроці  $t$  та набором локальних правил взаємодії сусідніх клітин. Рідше під час визначення стану використовуються глобальні правила - такі, що визначають вплив певної загальної змінної на всі клітини автомату.[2] Саме розширення такими глобальними правилами буде досліджено у даній роботі.

На кожному кроці кожна клітина приймає один із станів з наявної скінченної множини станів  $S$ , котре може розглядатись як  $k$ -вимірний алфавіт  $S = \{0, 1, \dots, k-1\}$ .

Якщо кожна клітина має  $2r$  сусідів, локальне правило переходу виглядає наступним чином  $\phi : S^{2r+1} \rightarrow S$ . Правило може бути детермінованим або стохастичним.

Для визначення сусідів конкретної клітини у двовимірному автоматі існує декілька підходів, найпопулярніші з яких "сусідство" фон Неймана та

"сусідство" Мура[3]. Перший розглядає лише чотири клітини, котрі безпосередньо торкаються поточної клітини (згори, знизу, зліва та справа). А другий підхід додає до цих чотирьох ще чотири клітини, що знаходяться на діагоналях відносно поточної клітини. Надалі буде розглядатись підхід Мура, тому правило переходу можна записати наступним чином  $a_{ij}^{(t+1)} = \phi(a_{ij}^{(t)}, a_{i,j+1}^{(t)}, a_{i+1,j}^{(t)}, a_{i+1,j+1}^{(t)}, a_{i,j-1}^{(t)}, a_{i-1,j}^{(t)}, a_{i-1,j-1}^{(t)}, a_{i-1,j+1}^{(t)}, a_{i+1,j-1}^{(t)})$ .

Так як автомат представляє собою обмежену сітку, постає проблема визначення сусідів клітин, котрі знаходяться на межах цієї сітки. Одним з найпопулярніших шляхів розв'язання проблеми є використання тороїдальної структури сітки - якщо клітина немає сусіда згори, цим сусідом виступає клітина на відповідній позиції знизу сітки, аналогічно відбувається "перехід" для інших клітин на межах.

Ітеративний процес роботи автомату полягає у тому, що на кожному кроці наведена вище функція застосовується до кожної клітини водночас. Тобто, кожна клітина оброблюється паралельно та незалежно, що робить клітинні автомати гарним інструментом моделювання процесів зі схожою паралельною структурою, наприклад, економічної взаємодії.

Зазвичай клітинні автомати у пам'яті комп'ютера представляються як масиви комірок, де кожна комірка відповідає певній клітині автомату. Водночас більшість автоматів описується в термінах певної підмножини "живих" станів та визначення сусідів клітин ("мертві" клітини можуть бути вираховані користуючись цією інформацією). Тому такий автомат можливо представити просто як множину "живих" клітин.[4] Саме такий варіант представлення (дещо модифікований для конкретних задач) використовувався у роботі.

## 1.2 Розширення вихідним інформаційним каналом

Першою розглянутою модифікацією є додавання до звичайного клітинного автомату так званого вихідного інформаційного каналу. Такий канал на кожній ітерації роботи автомату збирає інформацію про поточний його стан, певним чином агрегує її і відсилає створене повідомлення "у зовнішній світ".

Агрегуюча функція обирається відповідно до задачі, що моделюється автоматом. Прикладом може бути кількість клітин у певному стані, стан, котрий має найбільшу кількість представників, різноманітні предикати (зокрема, перевірка чи кількість клітин певного стану перевищує задану), тощо.

Надсилення повідомлення може здійснюватись багатьма різними способами, в даній роботі був обраний підхід Communicating Sequential Processes, так як запропоновані в ньому абстракції доволі вдало вписуються в описаний механізм. Більш детальний огляд CSP наведений у відповідному розділі, на даному етапі достатньо зазначити, що CSP передбачає створення так званих каналів (channels), у які незалежні процеси здатні направляти повідомлення для спілкування один з одним. По суті, канали у CSP виступають більш формалізованими чергами.

Сама по собі модифікація не додає нічого нового безпосередньо до виразної потужності самого автомату, проте його наявність, по-перше, спрощує деякі моменти роботи з клітинним автоматом, насамперед з визначенням необхідності зупинки автомату, по-друге, може виступати своєрідним індикатором, котрий визначає бажану поведінку наступної запропонованої модифікації - керуючого інформаційного каналу.

### 1.3 Розширення керуючим інформаційним каналом

Друга модифікація - додавання керуючого інформаційного каналу - змінює модель клітинних автоматів більш суттєво. Доданий канал на кожному кроці може прийняти повідомлення із зовнішнього світу (для передачі цих повідомлень у створеній реалізації також використовується CSP), котре містить команду - формальним чином сформована сутність, котра змінює функцію переходу автомату для всіх подальших ітерацій.

Фактично, отримуємо наступну функцію переходу (використовується "сусідство" Мура):  $a_{ij}^{(t+1)} = \phi(a_{ij}^{(t)}, a_{i,j+1}^{(t)}, a_{i+1,j}^{(t)}, a_{i+1,j+1}^{(t)}, a_{i,j-1}^{(t)}, a_{i-1,j}^{(t)}, a_{i-1,j-1}^{(t)}, a_{i-1,j+1}^{(t)}, a_{i+1,j-1}^{(t)})$  if  $c?() == false$  OR  $c(a_{ij}^{(t)}, a_{i,j+1}^{(t)}, a_{i+1,j}^{(t)}, a_{i+1,j+1}^{(t)}, a_{i,j-1}^{(t)}, a_{i-1,j}^{(t)}, a_{i-1,j-1}^{(t)}, a_{i-1,j+1}^{(t)}, a_{i+1,j-1}^{(t)})$  if  $c?() == true$ , де  $c(...)$  - отримана команда, а  $c?()$  - предикат, що визначає чи команда наразі активована. Як і агрегуюча функція вихідного каналу, команда, що буде надсилатись, буде відрізнятись для конкретних автоматів, більше того, для одного автомату може використовуватись декілька різних типів команд, вибір між застосуванням яких робиться зовнішнім світом (користувачем) і може засновуватись на показниках, котрі повертає вихідний інформаційний канал.

Потенційно, дана модифікація дозволяє деяким чином усунути детермінованість автомату (котра частково зберігається навіть при використанні стохастичних правил переходу) та надає користувачу змогу впливати на працюючий автомат. Для задач моделювання ця модифікація цікава насамперед тим, що дає змогу відносно простим шляхом симулювати зовнішній вплив на процес або систему: наприклад, державне регулювання економічної взаємодії корпорацій та робітників.

Розглянемо застосування даних модифікацій на прикладі конкретної задачі.



#### 1.4 Використання модифікованого автомату для розпізнавання необмеженої граматики

Розглянемо задачу створення розпізнавача мови, що описується необмеженою граматиною.

Нагадаємо, що необмежена граматика (тип-0) - найбільш загальний клас граматик у ієрархії Чомського, у якій на ліву та праву частини правил не накладаються ніякі обмеження. Мови, описані такими граматиками, є рекурсивно-зліченими та розпізнаються машиною Тюринга.[5] Представником даної граматики є наступна мова  $L = \{ ww : w \in A^* \}$ ,  $A = \{a, b\}$ , котра, по суті, описує мову усіх слів з парною кількістю літер, в яких перша половина слова ідентична другій половині, при чому літери належать алфавіту з двох символів. Для саме цієї мови і було створено клітинний автомат-розпізнавач.

В ньому використовується три основних стани - *dead*, *a* та *b*, перший з яких означає порожню клітину, а наступні два відповідають літерам алфавіту. Таким чином слово представляється послідовним горизонтальним ланцюжком клітин стану *a* та *b*:

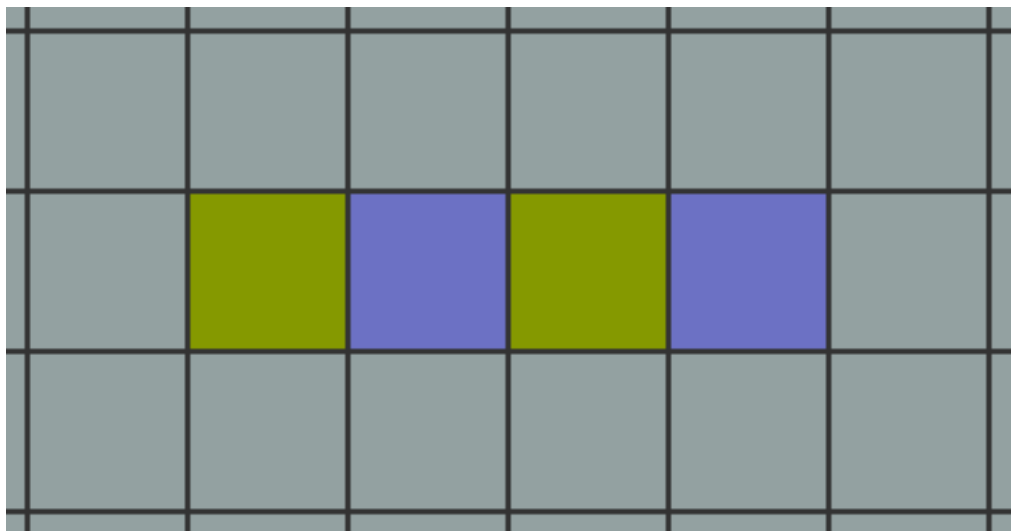


Рисунок 1

Стан/літера *a* показана зеленим кольором, а *b* - бузковим, сірий колір - "мертві" клітини. Для розпізнавання слова вводиться набір проміжних станів,

та відбувається певні перетворення в результаті яких на сітці автомату з'являється клітина у стані, що кодує успішне (*s*) або провальне (*f*) розпізнання слова. Кількість клітин у цих станах на кожному кроці передається у вихідний інформаційний канал, і коли кількість одного з цих станів перевищує нуль робота автомату зупиняється з успіхом або провалом (відповідне повідомлення показується у графічному інтерфейсі).

Правила переходу автомату виглядають наступним чином:

```
(cond
  (and (dead? s) (alive-only :right letter?)) :lc
  (and (dead? s) (alive-only :left letter?)) :rc
  (and (dead? s)
    (or (alive-only :right #{:lc})
        (alive-only :left #{:rc}))) :x
  (and (= :lc s)
    (between-l-r #{:x :a :b :dead} letter?)) right
  (and (= :rc s)
    (between-l-r letter? #{:x :a :b :dead})) left
  (and (letter? s) (between-l-r #{:lc} letter?)) :lc
  (and (letter? s) (between-l-r letter? #{:rc})) :rc
  (between-l-r #{:lc} #{:rc}) :f
  (or (and (= :lc s) (= :rc right))
      (and (= :lc left) (= :rc s))) :m
  (and (dead? s) (= :m top)) :n
  (and (dead? s) (letter? top) (= :n right)) :n
  (and (dead? s) (letter? top)
    (#{:n :a :b} left)) top
  (and (= :n s) (letter? right)) right
  (and (letter? s) (= :n left)) :n
  (and (letter? s) (= :x left) (= s bottom)) :x
  (and (letter? s) (= :x left)
    (letter? bottom) (not (= s bottom))) :f
  (and (= :m s) (= :x left)) :s
  :else s)
```

де *s* - поточний стан клітини, *dead?* - предикат, що визначає чи є клітина "мертвою", *alive-only* визначає, що живим є лише один сусід на певній позиції і він має певний стан, *between-l-r* перевіряє, що клітина знаходиться між двома клітинами у певних заданих станах, *left*, *right*, *top*, *bottom* - стани сусідів зліва, справа, згори, знизу, а всі символи, що починаються з : - позначки відповідних станів клітин.

Розглянемо основні моменти роботи автомату більш детально. Перш за все автоматом слово розбивається на дві рівні частини, якщо ж це виконати неможливо (кількість літер у слові непарна) з'являється клітина у стані *f*. Для

цього по одній мертвій клітині з боків слова утворюється стани  $lc$  та  $rc$  котрі з кожним кроком міняються місцями з клітиною справа та зліва відповідно. Коли вони стають сусідами одна одної, вони перетворюються у середину слова (стан  $m$ ), а межі виділяються станом  $x$ :

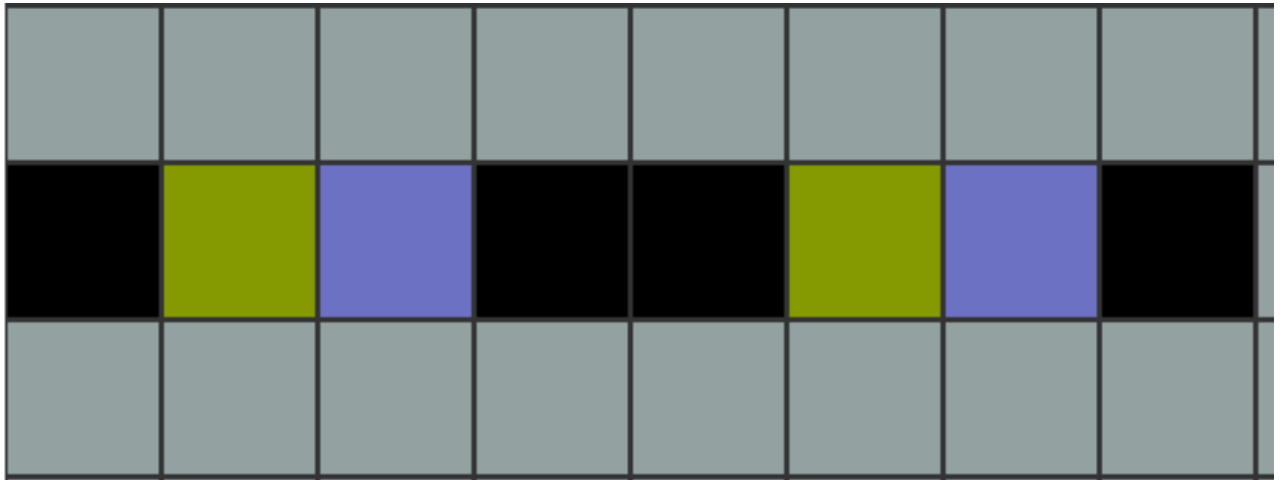


Рисунок 2

межі та середини слова виділені чорним кольором. Якщо автомат приходить до ситуації, коли у певної клітини лівий та правий сусіди у станах  $lc$  та  $rc$  відповідно, це значить, що кількість літер у слові непарна і дана клітина переходить у стан  $f$ .

Далі, для порівняння частин слова на рівність використовується наступна процедура: клітини автомату, що знаходяться безпосередньо під "літерами" слова та його серединою переходить у стан  $n$ , а надалі літери правої частини слова переміщаються на відповідні позиції новоствореного нижнього ряду та починає рухатись вліво шляхом обміну станами з клітинами зліва, що перебувають у стані  $n$ . Рух зупиняється якщо зліва "мертва" клітина, або клітина-літера:

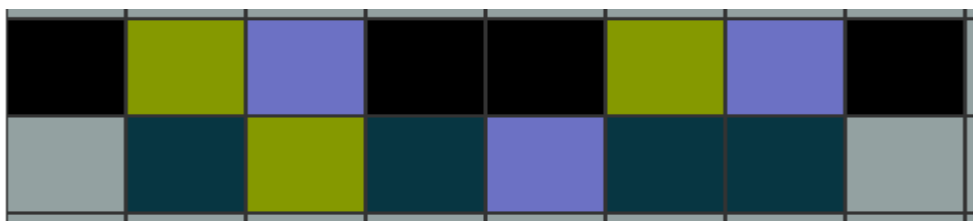


Рисунок 3

клітини у стані  $n$  зображені темно-зеленим кольором. Якщо рух клітини у нижньому ряді зупинився і при цьому вона перебуває у такому ж стані, як клітина над нею - верхня клітина переходить у стан  $x$ , якщо ж стани не співпадають - верхня клітина переходить у стан  $f$  (розпізнання слова провалилось). Коли всі клітини лівої частини слова переходять у стан  $x$  (тобто всі клітини нижнього ряду перебували у такому ж стані, як відповідні ним клітини лівої половини слова), це значить, що слово розпізнано успішно, і одна з клітин середини слова переходить у стан успіху  $s$ :

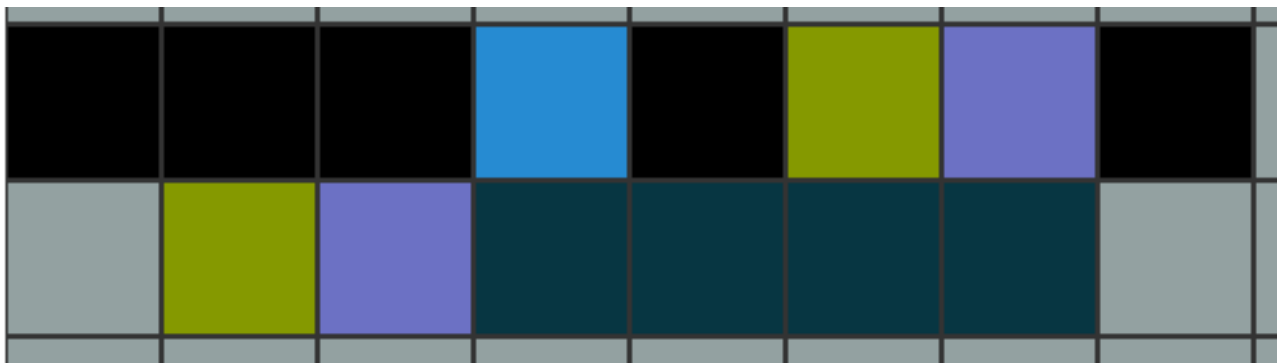


Рисунок 4

"успішна" клітина виділена блакитним кольором.

За допомогою керуючого каналу автомату можливо вказати, що певна літера має поводитись як "підстановка" (wildcard) - тобто, при перевірці на рівність дана літера буде рівна не лише собі, а й іншій літері. Наприклад, початкове слово, котре не мало б розпізнатись:

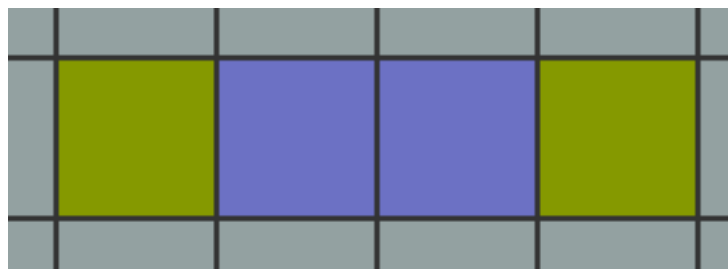


Рисунок 5

Але якщо надіслати команду, що  $a$  і  $b$  розпізнаються як "підстановки", слово буде розпізнано автоматом успішно:

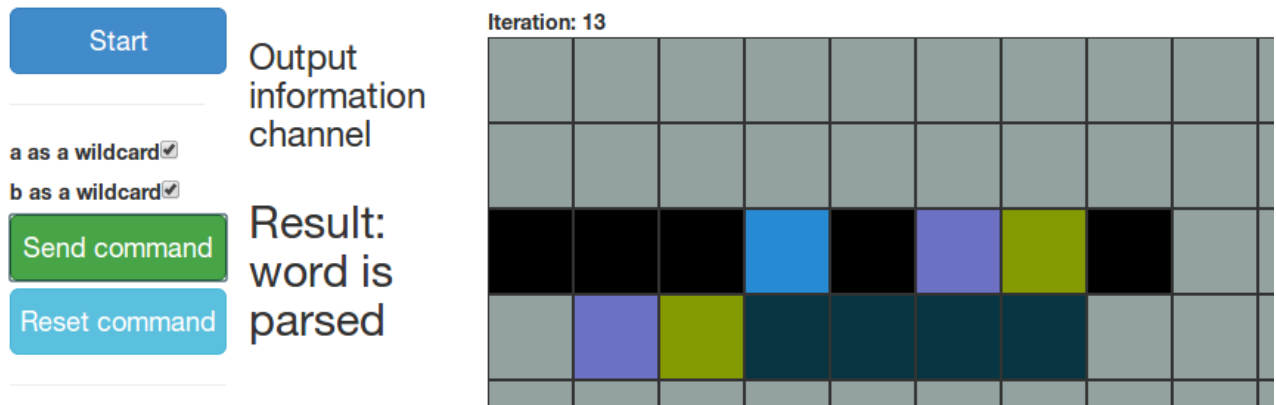


Рисунок 6

Таким чином, ми впевнелись, що клітинний автомат здатний розпізнавати такий клас мов, котрий відповідає машинам Тюринга, водночас, за допомогою запропонованих каналів даних вдалось дещо спростити використання автомату (визначення моменту зупинки), а також розширити його можливості додатковими опціональними правилами поведінки за допомогою керуючого каналу. Розглянемо використання каналів у більш складній моделі.

## 2 МОДЕЛЬ ЕКОНОМІЧНИХ ВІДНОСИН

### 2.1 Опис моделі

Для перевірки дієздатності запропонованої модифікації клітинного автомату було вирішено промодельовати економічну взаємодію покупців та виробників на ринку інформаційних послуг/сервісів: він характеризується схильністю покупців до вибору товару, котрий вже є у його оточення, що ідеально вписується у загальну схему роботи клітинного автомату, а також характеризується більш-менш стійкою прив'язкою клієнта до обраного товару.

За основу була взята модель описана у статті "Modelling the Spatial Dimension of Economic Systems with Cellular Automata"[2]. Сітка автомату обмежена - містить  $N$  клітин, кожна з яких представляє одного покупця, а сусіди (використовується "сусідство" Мура, тобто кожна клітина має вісім сусідів), відповідно, є оточенням конкретного покупця. На ринку змагаються  $M$  компаній, котрі не представлені безпосередньо на сітці автомату, натомість кожна клітина-покупець на певній ітерації володіє товаром одної з цих компаній - відповідно стан клітини показує, який товар вона обрала. Тобто, алфавіт станів  $S$  можна описати наступним чином  $S = \{0, 1, \dots, M\}$ , де 0 відповідає відсутності товару, а товару 1, ...,  $M$  компаніям з відповідним порядковим номером.

Всі покупці є однаковими у тому розумінні, що мають однакову функцію корисності. Вона включає в себе три основні компоненти:

- кількість сусідів з певним товаром
- загальний відсоток ринку, "зайнятий" цим товаром
- вартість товару

При цьому куплений товар має так званий строк придатності - на кожному кроці роботи автомату куплений товар може "зламатись" з базовою ймовірністю  $d$ , тим самим перевівши автомат у стан "без товару". Для конкретної компанії базова ймовірність зламатись може як збільшитись, так і зменшитись в залежності від поточної середньої ціни. Реальна ймовірність "зламатись"  $rd$  обчислюється наступним чином:

$$rd(k) = d * p(k) / (E_{l=1}^M p(l) / M)$$

де  $k$  - номер компанії,  $p(i)$  - ціна товару  $i$ -тої компанії.

Обирають та купують товари лише ті клітини, котрі під час даної ітерації не мають ніякого товару, тобто покупці, що вже мають певний товар, не змінюють його, допоки він не "зламається".

Формально функцію корисності покупця з координатами  $(i,j)$  для товару від компанії  $k$  можна описати наступним чином:

$$U_{i,j}(k) = (b_{i,j}(k)/nn)^a * s(k) * pr(k)^p$$

де  $b_{i,j}(k)$  - кількість сусідів поточної клітини, що володіють товаром від компанії  $k$ ,  $nn$  - загальна кількість сусідів (фактично, є константою, що дорівнює восьми, бо використовується тороїдальна сітка),  $s_k$  - частина ринку, що належить компанії  $k$ ,  $pr(k)$  - ціна товару компанії  $k$ ,  $a$  та  $p$  - параметри алгоритму, що відповідають за локальне "поширення" товару та його вартість відповідно.

З даної функції корисності можна отримати і відношення переваг між різними виробниками для конкретного покупця, яке водночас визначає з якою ймовірністю покупець обере товар певної компанії:

$$RP_{i,j}^k = U_{i,j}(k) / E_{l=1}^M U_{i,j}(l)$$

Компанії отримують гроші не лише за продаж товару, але й за його використання: тобто, на кожній ітерації компанія отримує  $pr(k) * n(k)$  одиниць прибутку, де  $n(k)$  - поточна кількість клітин у стані  $k$  (покупців, що

користуються товаром даної компанії). При цьому на кожного покупця компанії витрачає фіксовану суму  $es$  (двадцять одиниць) для підтримання надання товару.

Присутня у моделі і держава. На кожній ітерації компанії додаткову сплачують певний податок державі, а сама держава витрачає фіксовану суму  $eg$  (одна одиниця) на підтримку існування кожної клітини автомату. Період роботи автомату розбивається на роки: параметр  $T$  вказує тривалість року у ітераціях (за замовченням один рік триває дванадцять ітерацій).

Якщо на певній ітерації капітал компанії стає менше нуля, вона оголошується банкрутом, втрачає всіх своїх покупців (вони переходять у стан "без товару") та надалі більше не продає товар.

Функція переходу станів клітини є доволі простою:

```
{:state
  (cond
    (or (< (rand) (* (:depreciation env)
                    (/ current-price avg-price)))
      (bankrupt? s))
    :without-good

    (without-good? s)
    (if-let [c (weighted
                (user-preferences env global-share n-states))]
      c
      :without-good)

    :else s)}
```

де  $(:depreciation\ env)$  - значення параметра  $d$ ,  $(bankrupt?\ s)$  визначає чи є банкрутом компанія-постачальник поточного купленого товару,  $(user-preferences\ env\ global-share\ n-states)$  знаходить ймовірності обрати товар певної компанії даною клітиною, а  $weighted$  робить зважений вибір відповідно до знайдених ймовірностей,  $:without-good$  позначає стан "без товару".

Модель підтримує три основні схеми оподаткування: відсоток з чистого доходу, відсоток з обороту та фіксована сума. Кожний із схем відповідає свій параметер, котрий можна налаштовувати за допомогою керуючого каналу. Під оборотом розуміється сума надходжень від покупців без віднімання витрат на



їх підтримання, а під чистим доходом - з відніманням цих витрат.

Перейдемо до більш докладного огляду використання інформаційних каналів.

## 2.2 Використання керуючого та вихідного каналів

Як згадувалось раніше керуючий канал можна використовувати для зміни параметрів алгоритму, зокрема  $a$ ,  $p$ ,  $d$ , розміри відсотку/суми податків. А у вихідний канал на кожній ітерації автомат направляє інформацію про поточний стан моделі, а саме поточні значення всіх параметрів, що налаштовуються керуючим каналом, розмір витрат на утримання компанії та державою, загальна кількість компаній, вартості товарів, капітал держави та компанії, сумарний прибуток та чистий прибуток, отримані компаніями за поточний рік, сумарні витрати на підтримання існування користувачів за поточний рік, розподіл ринку між компаніями та поточна схема оподаткування, обрана кожною з компаній. Найцікавішим є останній показник.

На початку роботи моделі кожна компанія платить як податок відсоток з обороту, проте кожного "року" (за замовченням дванадцять ітерацій) обробник інформації з вихідного каналу надсилає у керуючий канал команду, яка для кожної компанії, можливо, змінює її обрану схему оподаткування. Для визначення яку саме схему обрати на наступний рік використовується доволі наївна функція:

```
(defn conservative-corp-tax
  [tax-rate income-tax-rate fixed-tax
   capital-incomings capital-expenditures]
  (let [tax (* tax-rate capital-incomings)
        income-tax (* income-tax-rate
                       (- capital-incomings capital-expenditures))]
    {:type :change-taxation-type
     :cmd
     (->> [[[:rate tax] [:income-rate income-tax] [:fixed fixed-tax]]
            (sort-by second)
            first
            first)}))
```

На вхід вона приймає поточні розміри відсотку з прибутку, обороту та суми фіксованого податку, а також сумарний прибуток та витрати за минулий рік. Для кожного типу оподаткування підраховується сума податку, яку б треба було заплатити за минулий рік, і з поміж отриманих значень обирається найменше, яку і стає обраною схемою на наступний рік. У даній функції можна відмітити і загальну структуру команди даного автомату: вона є асоціативним масивом з двома ключами - *type* (дія, яку треба виконати) та *cmd* (інформаційне поле специфічне для кожного типу команд). Для команди зміни схеми оподаткування значенням інформаційного поля виступає позначка обраного типу оподаткування.

Окрім зміни схеми оподаткування кожні півроку обробник вихідного каналу для кожної компанії надсилає команду зміни вартості товару. Функція, що визначає ціну на наступне півріччя також є доволі простою:

```
(defn conservative-corp-price
  [[competitor-count share capital-diff price]]
  (cond
    (< capital-diff 0)           {:type :change-price
                                :cmd (* price 1.5)}
    (and (< share (/ 1.0 competitor-count))
         (> price 100))         {:type :change-price
                                :cmd (/ price 3)}
    (and (< share (/ 1.0 competitor-count))
         (> price 10))          {:type :change-price
                                :cmd (/ price 1.5)}
    (and (< share (/ 1.0 competitor-count 2))
         (> price 100))         {:type :change-price
                                :cmd (/ price 4)}
    (and (< share (/ 1.0 competitor-count 2))
         (> price 10))          {:type :change-price
                                :cmd (/ price 2)}
    (>= share 0.33)             {:type :change-price
                                :cmd (* price 1.2)}
    (>= share 0.5)              {:type :change-price
                                :cmd (* price 1.5)}
    :else                        {:type :change-price
                                :cmd price}))
```

Функція приймає на вхід кількість конкурентів, відсоток ринку, що "належить" компанії, розмір капіталу компанії, чистий прибуток отриманий нею за попередні півроку та поточну ціну. Якщо за попередній період компанія зазнала збитків або вона завоювала більше половини всього ринку, то ціна збільшується у півтора рази, якщо завойовано третину ринку - ціна

збільшується у 1.2 рази. Зменшується ціна у декількох різних ситуаціях, коли частка ринку поточної компанії стає меншою, ніж її розмір за рівномірного розподілу між усіма конкурентами.

Дослідимо результати моделювання за умови використання різних розмірів податкових ставок та суми фіксованого податку.

### 2.3 Експериментальне визначення оптимальної суми фіксованого податку

Зі створеною моделлю було виконано декілька експериментів для виявлення найсприятливіших умов використання фіксованого податку, як такого, що вимагає найменше бюрократичних операцій, а отже потенційно найменш придатний до різних корупційних схем і, водночас, простіший у "використанні" як для платника, так і для держави.

У всіх проведених дослідях більшість параметрів моделі було встановлено у однакові значення, а варіювались лише податкові ставки. Фіксовані параметри мали наступні значення:

- сітка розміром 33x33 (кількість клітин  $N = 1156$ )
- вісім компаній ( $M = 8$ )
- базова ймовірність "поломки" товару 3 відсотки ( $d = 0.03$ )
- параметри функції корисності покупця:  $a = 1$ ,  $p = -1$  (тобто, враховується і локальна поширеність товару, і його ціна)
- один рік триває дванадцять ітерацій роботи автомату ( $T = 12$ )
- початковий капітал держави тисяча одиниць капіталу, компанії починають з нульовим капіталом
- на підтримання існування клітини держава витрачає одну одиницю

капіталу ( $eg = 1$ ), а компанія - двадцять ( $ec = 20$ )

- початкова ціна товару у всіх компаній однакова і становить двадцять одиниць капіталу
- початкова схема оподаткування - податок на оборот для всіх компаній
- кожний експеримент тривав двісті п'ятдесят ітерацій

Також, на початку роботи моделі кожна компанія мала однакову кількість покупців рівномірно розподілених по сітці автомату наступним чином (світло-тілесним кольором виділені покупці без товару, інші вісім кольорів відповідають восьми наявним компаніям):

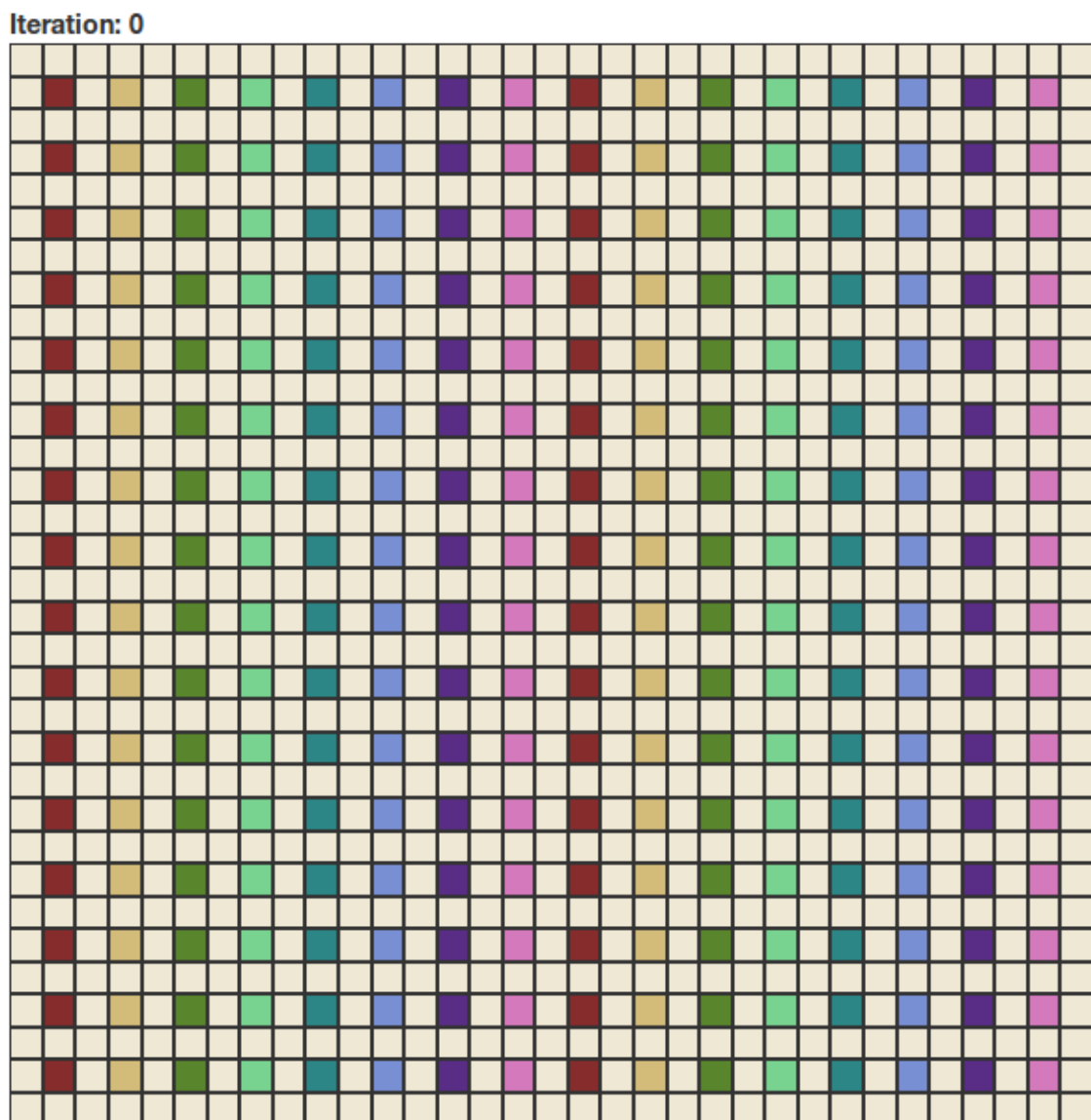


Рисунок 7

Під час першого експерименту рівень податку на оборот був встановлений у п'ять відсотків, податку на прибуток - десять відсотків, а фіксований податок становив дві тисячі одиниць капіталу. Фінальний стан автомату виглядав наступним чином:

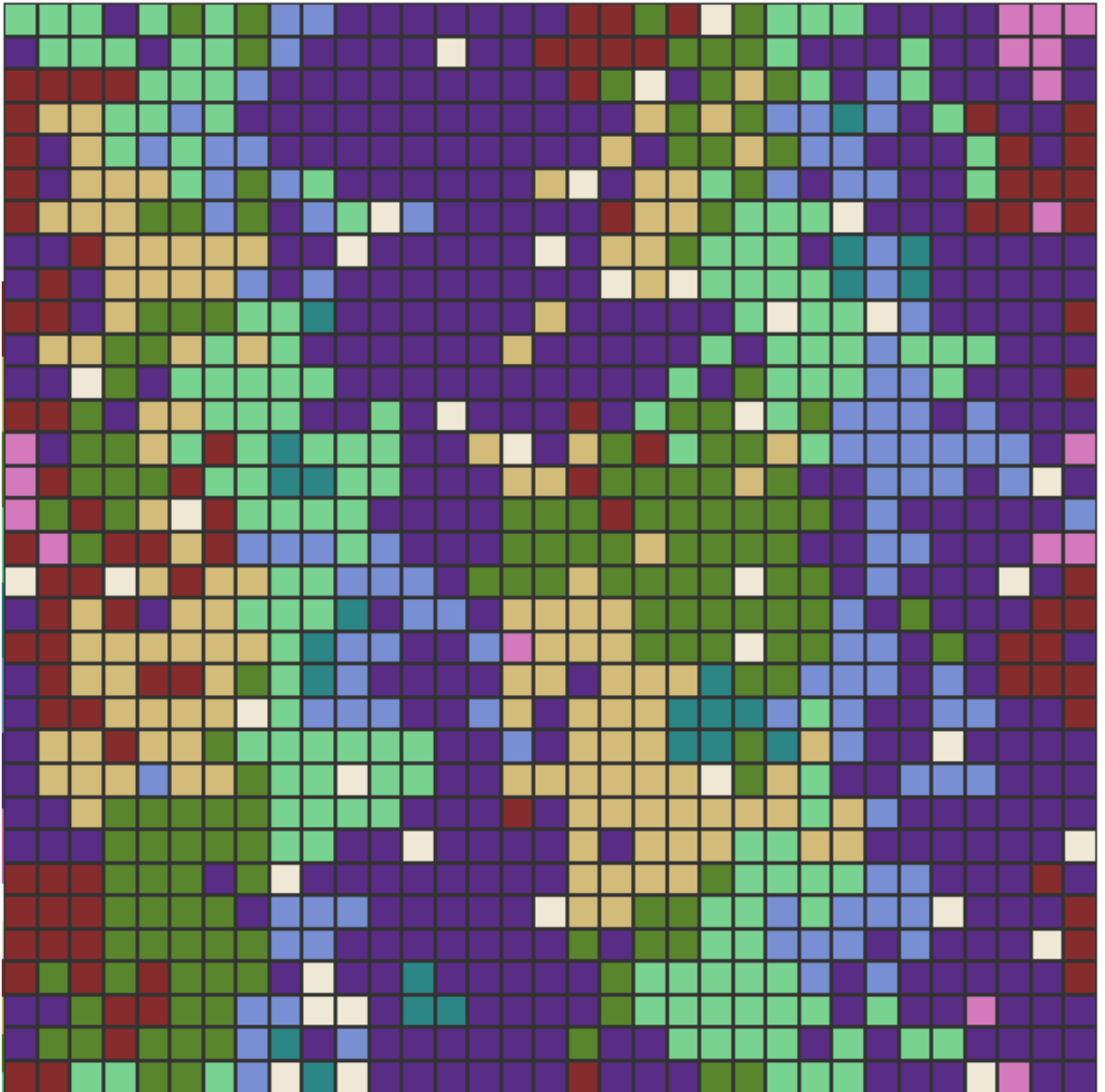


Рисунок 8

Результуючий капітал держави становив 2611491 одиниць капіталу, три відсотки покупців були без товару, а показники компаній були наступні:

<b>номер компанії</b>	<b>капітал</b>	<b>частина ринку, %</b>	<b>ціна</b>	<b>схема оподаткування</b>
1	412949	8	29	на прибуток
2	489895	11	29	на прибуток
3	470440	13	44	фіксований
4	476682	13	44	на прибуток
5	301667	2	29	на прибуток
6	339532	9	44	фіксований
7	1409245	35	92	фіксований
8	429412	1	29	на прибуток

Таблиця 1

Як бачимо, сьома компанія завоювала більше третини ринку, а п'ята та восьма майже зникли з нього. При цьому більшість компаній обрали податок на прибуток, хоча й фіксований також мав деяке поширення, зокрема був обраний найбагатшою сьомою компанією. Слід зазначити, що протягом роботи автомату в моменти рівномірного розподілу ринку між компаніями домінував фіксований податок. Середній капітал компанії становив 541227 одиниць капіталу, а середня ціна товару - 42.

Для наступного експерименту рівень податку на оборот становив вісім відсотків, податок на прибуток - двадцять відсотків, а фіксований податок -

тисячу п'ятсот одиниць капіталу. Експеримент завершився наступним станом автомату:

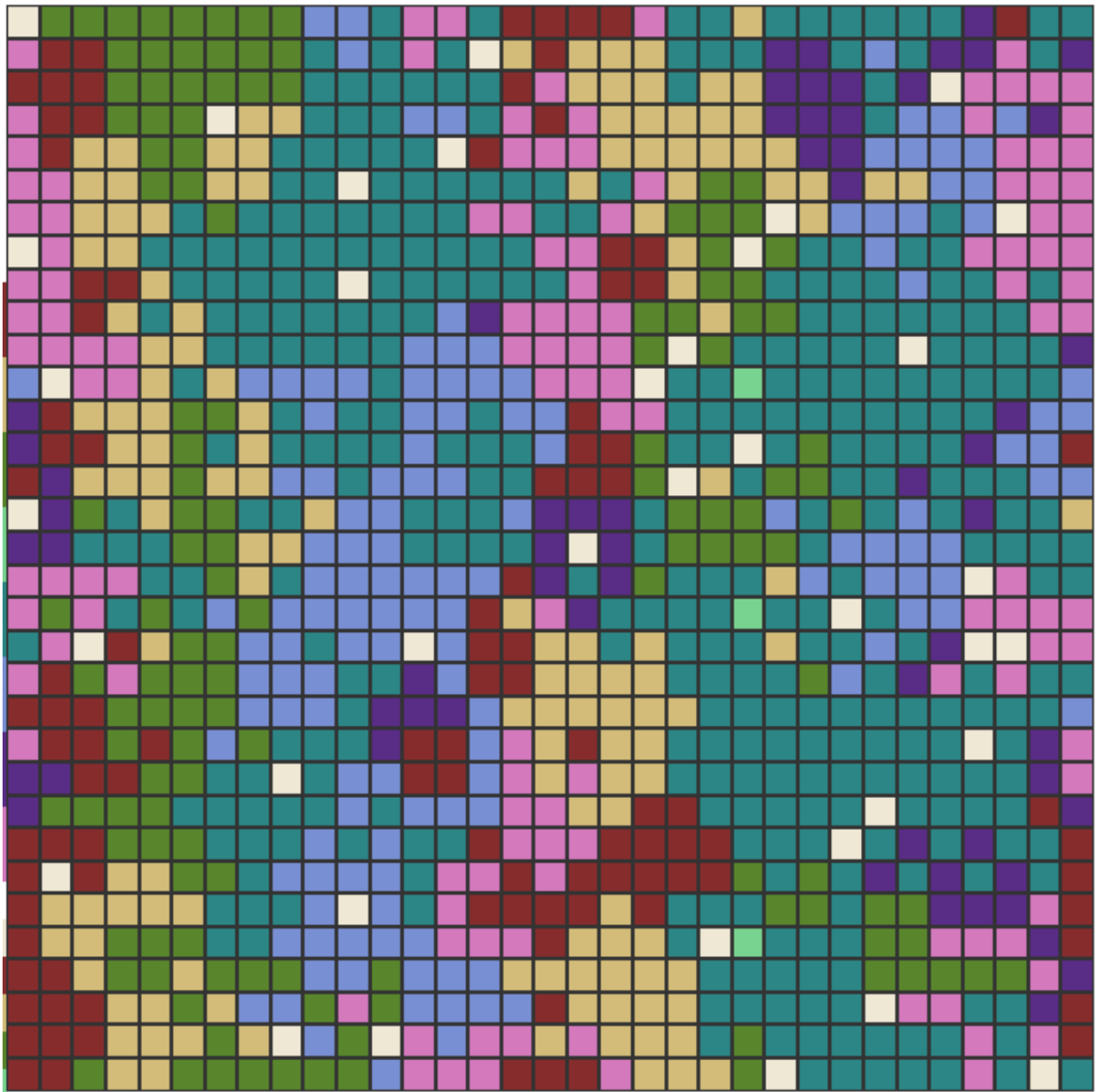


Рисунок 9

Капітал держави склав 2607058 одиниць, а частка покупців без товару лишилась, як і в попередньому експерименті (три відсотки).

Показники	компаній		були	такі:
номер компанії	капітал	частина ринку, %	ціна	схема оподаткування
1	272234	9	29	фіксований
2	514368	12	29	фіксований
3	551082	12	44	фіксований
4	154002	0	29	на прибуток
5	808375	32	44	фіксований
6	387382	12	44	фіксований
7	361451	5	44	фіксований
8	584177	11	44	фіксований

Таблиця 2

Цього разу знову одна з компаній зайняла близько третини ринку (п'ята), водночас, одна компанія взагалі була витіснена з ринку (четверта). Серед схем оподаткування абсолютно домінує фіксований податок, при цьому середній капітал компанії склав 454133 одиниць, а середня ціна товару - 39.

У порівнянні з попереднім експериментом фіксований податок став найпоширенішим, зібрані державою кошти істотно не зменшились, а середня ціна товару, навпаки, стала дещо меншою, водночас середні статки компаній стали не такими великими (хоча й зменшився розрив між найбагатшою та найбіднішою компаніями).



Зменшимо фіксований податок до п'ятисот одиниць не змінюючи при цьому параметри інших податків. Фінальний стан автомату:

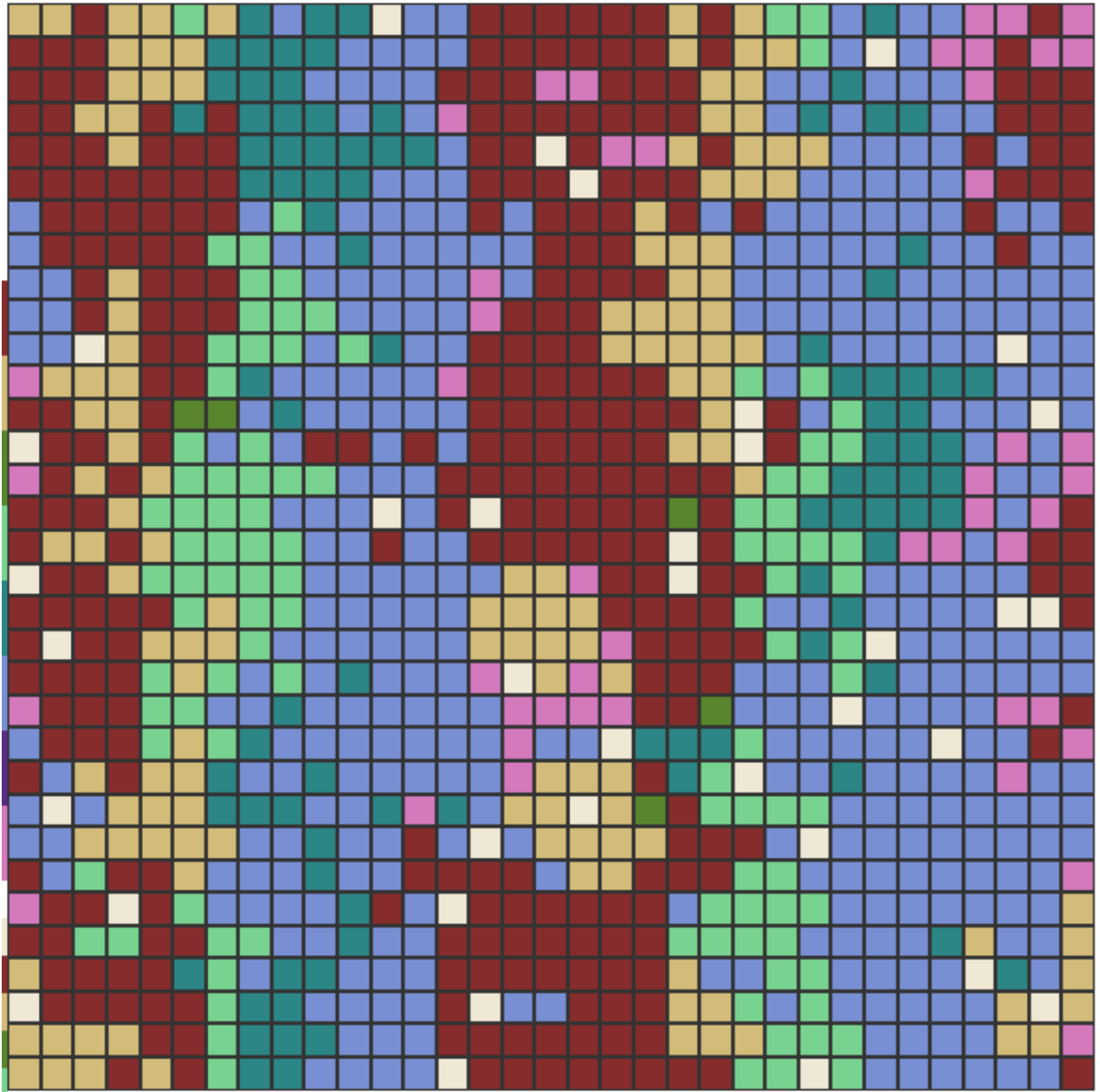


Рисунок 10

Капітал держави склав 772422, а відсоток покупців без товару знову не змінився - три відсотки.

Показники ж компаній стали такі:

<b>номер компанії</b>	<b>капітал</b>	<b>частина ринку, %</b>	<b>ціна</b>	<b>схема оподаткування</b>
1	759421	28	29	фіксований
2	480175	12	29	фіксований
3	403934	0	66	фіксований
4	327852	9	29	фіксований
5	345665	8	19	фіксований
6	605431	33	29	фіксований
7	514635	0	9	фіксований
8	489739	4	29	фіксований

Таблиця 3

Цього разу дві компанії отримали приблизно по третині ринку (перша та шоста), а дві зникли (третя та сьома), фіксований податок став єдиним, що використовується, середній капітал досягнув 490856 одиниць, середня ціна товару - 27.

Таким чином, було отримано найменшу ціну товару (що добре для покупців) та найбільший середній капітал компаній, водночас отримані державою гроші зменшились більше, ніж в тричі. Спробуємо знайти компромісний варіант, збільшивши фіксований податок до тисячі двохсот п'ятдесяти одиниць.

Отримали наступний фінальний вигляд автомату:

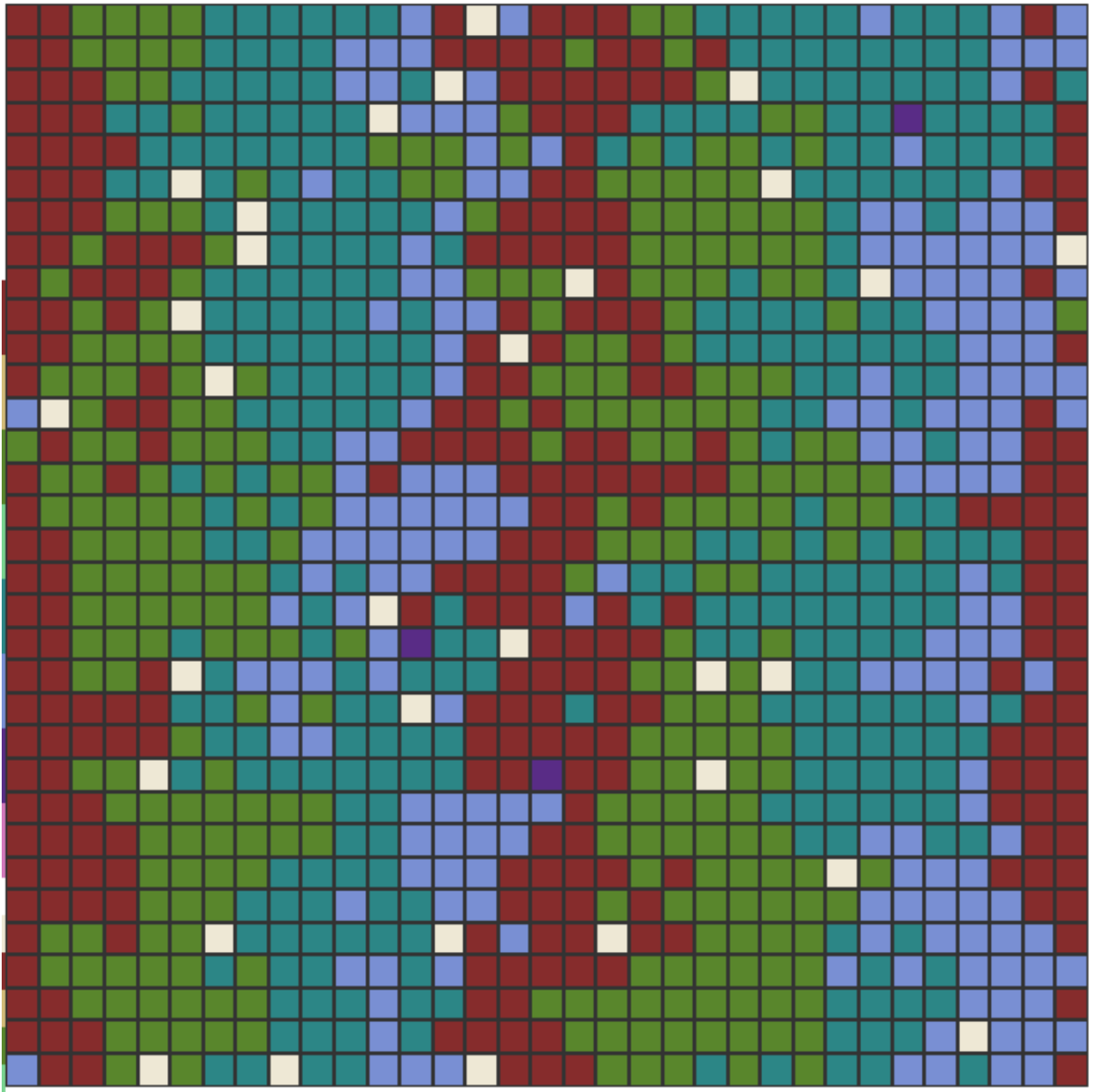


Рисунок 11

Капітал держави склав 2023379, без товару залишилось два відсотки покупців.

А от фінальний стан компаній значно змінився:

<b>номер компанії</b>	<b>капітал</b>	<b>частина ринку, %</b>	<b>ціна</b>	<b>схема оподаткування</b>
1	430663	24	29	фіксований
2	279634	0	56	на оборот
3	879705	26	51	фіксований
4	243732	0	9	на оборот
5	482569	28	29	фіксований
6	303563	16	20	фіксований
7	394928	0	50	фіксований
8	377015	0	168	на оборот

Таблиця 4

Чотири компанії зникли з ринку, а інші чотири розділили ринок приблизно порівну (одна з них мала трохи меншу частку покупців, а інші - майже однакові). При цьому всі "живі" компанії використовували фіксований податок, а середня ціна їх товарів сягнула 32. Середній капітал склав 423976 одиниці.

Бачимо, що результати другого експерименту є найзбалансованішими серед тих, де фіксований податок домінує, тобто можна припустити, що розмір фіксованого податку у тисячу п'ятсот одиниць є наближеним до оптимального для даної моделі.

Провівши нескладні підрахунки, на основі цих результатів можна отримати одну з можливих формул для визначення фіксованого податку  $ft$  для нашої моделі:

$$ft = 0.03 * N^2 * avgPrice$$

де  $avgPrice$  - поточна середня вартість товару,  $N$  – розмір сторони сітки.

Проведемо ще один експеримент з динамічною зміною фіксованого податку: за основу використаємо другий експеримент, і додамо туди ще одне правило обробки інформації з вихідного каналу: за одну ітерацію до кінця кожного року за наведеною формулою буде обраховуватись новий розмір фіксованого податку та відправлятись команда у керуючий інформаційний канал про зміни цього податку на нове значення. Отримані результати є доволі цікавими:

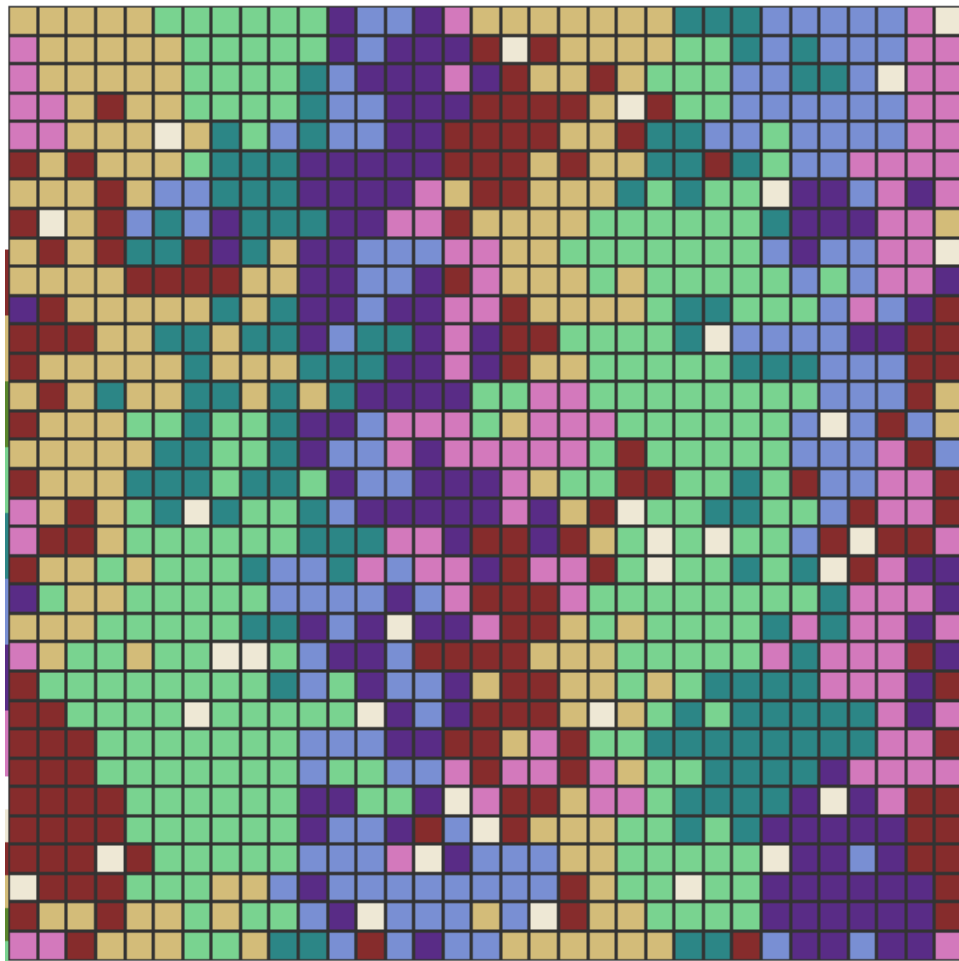


Рисунок 12

Фінальні статки держави досягнули 1911748, а відсоток покупців без товару повернувся до вже добре знайомого значення у три відсотки. Розмір фіксованого податку на останній ітерації склав 968 одиниць, при цьому впродовж роботи автомату він коливався в межах від 783 до 2450 одиниць.

<b>номер компанії</b>	<b>капітал</b>	<b>частина ринку, %</b>	<b>ціна</b>	<b>схема оподаткування</b>
1	144043	13	29	фіксований
2	213365	15	29	фіксований
3	114202	0	7	на оборот
4	251794	22	29	фіксований
5	122877	10	19	на дохід
6	139140	12	19	фіксований
7	166972	12	19	на дохід
8	150882	10	19	на дохід

Таблиця 5

Бачимо, що середній капітал компаній зменшився до 162909, а середня вартість товару до 23, фіксований податок обрала половина компаній (слід зазначити, що протягом роботи автомату всі компанії обирали лише фіксований податок, а інші типи були обрані лише наприкінці заданого періоду симуляції). Але при цьому, не враховуючи одної компанії, котра зникла з ринку, інші розділили покупців приблизно порівну (друга та четверта "захопили" дещо більше за рахунок "знищеної" третьої), тобто можна сказати, що залучення динамічного фіксованого податку дозволили протримати модель у стані балансу довше, ніж протягом попередніх експериментів.

Зрозуміло, що отримані результати є доволі синтетичним і не відображають усієї складності реального життя, проте навіть за ними можна відстежувати певні загальні тенденції у поведінці ринку та його реакцію на зовнішні зміни (зміну податків). Потенційно ж, отриманий інструмент можна доволі легко зробити більш потужним шляхом додавання додаткових зовнішніх команд (котрі здатні гарно просимулювати втручання держави, наприклад, можливість встановлення максимально можливої ціни товару), а також використанням більш "розумних" клітин, котрі будуть виступати не лише покупцями, а й найманими робітниками, що працюють на компанії. Також цікавим розширенням може бути додавання динамічної зміни ставок й інших податків, а також опціональної здатності уникнення компанією оплати податків з можливим отриманням штрафу на наступній ітерації.

### 3 ВІЗУАЛІЗАЦІЯ КЛІТИННИХ АВТОМАТІВ

#### 3.1 React та ідея virtual DOM

React[6] - бібліотека від Facebook, створена для побудови компонованих графічних HTML-інтерфейсів. На відміну від багатьох інших подібних рішень, дана бібліотека цілковито відмовилась від звичних шаблонів (HTML документів "покращених" додатковими директивами, котрі динамічно замінюються на необхідний вміст) і натомість пропонує використовувати компоненти - фактично, JavaScript функції, що генерують HTML-розмітку, та дотримуються певних вимог. Кожний компонент приймає як параметри два елементи - props (незмінні дані) та state (змінні дані).

Ключовою особливістю React є те, що бібліотека автоматично підтримує відображення у актуальному стані - компонент перемальовується у випадку зміни даних, що використовуються ним. На перший погляд це здається надто повільним, проте розробники використали цікавий підхід, котрий можна назвати virtual DOM. Під час першої ініціалізації компоненту викликається метод render, котрий генерує легковісне представлення відображення, з якого створюється рядок з HTML-розміткою та додається у тіло документу. Надалі виклики методу render призводять до повторної генерації проміжного представлення, котре порівнюється з попереднім варіантом, з порівняння знаходиться мінімальний набір змін, котрі необхідну виконати для перетворення попереднього відображення у поточний варіант. Перетворюються у розмітку та застосовуються вже ці, переважно невеликі, зміни. Можна сказати, що компонент у React - це простий скінченний автомат, котрий генерує різне відображення в залежності від поточного стану (значення даних, що використовуються компонентом).

Обробники подій додаються до компонентів просто як HTML-властивість певного DOM-елементу, ключем якої є ім'я події у camelCase, а значенням - функція, що оброблюватиме подію. У React імплементована своя система обробки подій, котра намагається забезпечувати сумісність зі



специфікацією W3C в незалежності від броузеру, що використовується. Додатково для всіх обробників значення JS-контексту автоматично встановлюється у поточний компонент. Також, використовується делегація подій - обробники приєднуються не безпосередньо до відповідних DOM-елементів, а створюється один загальний обробник, котрий приєднується до кореневого елемента і викликає необхідний користувацький обробник в залежності від події, що відбулась.

Компоненти бібліотеки є модульними - кожен з них може бути частиною іншого компоненту та містити компоненти у собі. Якщо компонент містить інші компоненти, він є їх власником та відповідальний за заповнення props всіх компонентів, що "належать" йому. Більш формально, компонент є власником всіх компонентів, що створюються у його render-методі.

Кожен компонент обов'язково має визначити метод render - генерує опис одного DOM-елементу, котрий мусить бути відображеним на сторінці. Даний метод має бути чистим (як у понятті чиста функція) - якщо на вхід подаються однакові параметри, результат має також бути однаковим. Водночас, React визначає доволі детальний життєвий цикл компонентів, для модифікації обробки кожного з його етапів достатньо додатково визначити відповідний метод під час опису компоненту. Передбачені наступні фази:

### **getInitialState**

викликається один раз перед додаванням компоненту у DOM; значення, що повертається, використовується як початкове для state

### **getDefaultProps**

викликається один раз перед додаванням компоненту у DOM; значення, що повертається, використовується як початкове для props

### **componentWillMount**

викликається один раз перед додаванням компоненту у DOM

**componentDidMount**

викликається одразу після додавання компоненту у DOM

**componentWillReceiveProps**

викликається під час отримання компонентом нових props, але перед генерацією нового відображення

**shouldComponentUpdate**

викликається перед генерацією відображення після отримання нових props або state; якщо метод повертає false, генерація нового відображення не буде проводитись

**componentWillUpdate**

викликається перед генерацією нового відображення після отримання нових props або state

**componentDidUpdate**

викликається одразу після генерації нового відображення після отримання нових props або state

**componentWillUnmount**

викликається перед тим, як компонент буде видалений з DOM

Для опису HTML-елементів всередині компонентів пропонується набір функцій з простору імен React.DOM, кожна з яких відповідає певному HTML-тегу. Наприклад, посилання створюється наступним чином:

```
var link = React.DOM.a({href: 'http://google.com'}, 'Google');
```

Також розробники створили простий трансформатор синтаксису JavaScript-файлів JSX, котрий дозволяє створювати HTML-елементи у JavaScript-кодi майже так само, як у звичайних HTML-файлах. Наприклад, попередній приклад буде виглядати так:

```
/** @jsx React.DOM */  
var link = <a href="http://google.com">Google</a>
```

Коментар на початку обов'язковий, він виконує функцію аналогічну прагма-конструкціям у C++, вказуючи інтерпретатору виконати трансформацію синтаксису перед інтерпретацією.

Для прив'язки створеного компонента до елемента на сторінці використовується функція `React.renderComponent`, першим параметром якої є компонент, а другим - DOM-елемент, до якого компонент має бути прив'язаним. Наведемо приклад створення та прив'язки композитного компоненту - списку коментарів від різних авторів[7]:

```
/** @jsx React.DOM */
var data = [
  {author: "Pete Hunt", text: "This is one comment"},
  {author: "Jordan Walke", text: "This is *another* comment"}
];

var Comment = React.createClass({
  render: function() {
    return (
      <div className="comment">
        <h2 className="commentAuthor">
          {this.props.author}
        </h2>
        {this.props.children}
      </div>
    );
  }
});

var CommentList = React.createClass({
  render: function() {
    var commentNodes = this.props.data.map(function (comment) {
      return <Comment author={comment.author}>{comment.text}</Comment>;
    });
    return (
      <div className="commentList">
        {commentNodes}
      </div>
    );
  }
});

var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.props.data} />
      </div>
    );
  }
});

React.renderComponent(
  <CommentBox data={data} />,
  document.getElementById('content')
);
```

Компонент `CommentBox` є кореневим компонентом, він приймає на вхід список коментарів та передає їх компоненту `CommentList`, котрий перетворює кожний елемент списку у компонент `Comment` та відображає їх одним списком.

### 3.2 ClojureScript та Om

Для реалізації застосування використовувалась мова програмування ClojureScript. Вона є варіантом мови Clojure, котрий запускається не на JVM, а компілюється у JavaScript. Хоча й ClojureScript не має всіх можливостей мови Clojure (зокрема, багатопоточності), але підтримує базову семантику даної мови, зокрема незмінні структури даних, "лінійні" послідовності, функції вищого порядку, простори імен та навіть макроси. Основна відмінність між Clojure та ClojureScript - якщо перша мова працює спільно з Java та має можливості інтероперабельності з саме цією мовою, то друге вже використовує JavaScript у ролі "батьківського" середовища та інтероперує з ним.

ClojureScript генерує немініфікований та необфускований JavaScript код, котрий надалі передається на вхід Google Closure Compiler[8], котрий і оптимізує отриманий код. Разом з компілятором до складу Google Closure входить набір низькорівневих бібліотек для більшості буденних задач, що виникають під час розробки JavaScript застосувань - наприклад, надсилання запитів до сервера. ClojureScript надає до них повний доступ без необхідності їх додаткового завантаження та додавання на сторінку.

Цікавою особливістю є те, що і компілятор ClojureScript, і компілятор Google Closure запускається на JVM - перший компілятор написаний на Clojure, а другий на Java.

Загалом процес перетворення ClojureScript у готовий до використання JavaScript код відбувається наступним чином[9]:

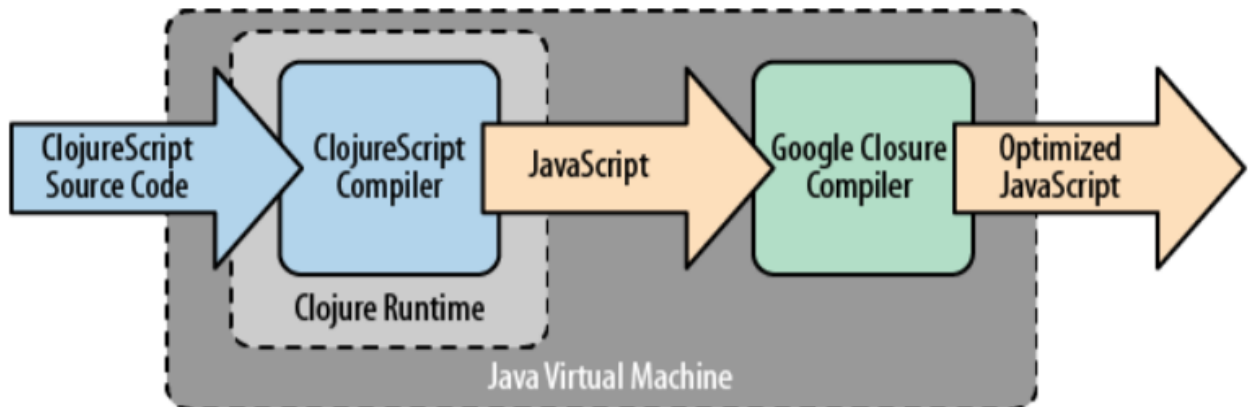


Рисунок 13

Генерація HTML-відображення у застосуванні здійснюється за допомогою бібліотеки Om[10]. Вона є обгорткою над розглянутою раніше JS-бібліотекою React, а також додає власну модель збереження стану застосування.

Om, як і React, пропонує використовувати незалежні, але зв'язані один з одним, компоненти, проте на відміну від React, всі дані, що передаються між компонентами є незмінними. Завдяки цьому та особливостям реалізації незмінності у Clojure/ClojureScript вдається досягти ще більшої швидкості роботи, адже для визначення того, чи потрібно перемальовувати компонент достатньо порівняти указники параметрів, а не їх вміст.

Всі дані, необхідні для генерації відображення, зберігаються у одному кореневому контейнері, котрий змінюється за допомогою механізму транзакцій. Зазвичай контейнером виступає звичайний асоціативний масив, що має деревоподібну структуру (тобто, має багато вкладених асоціативних масивів), кожна "гілка" цього масиву використовується одним з (або декількома) компонентами, що входять до складу застосування.

Всередину компоненту передається не сам контейнер з даними, а так званий курсор, що описує певний шлях всередину контейнеру (на

концептуальному рівні курсор схожий на таке поняття зі сфери структур даних як zipper). Фактично, курсор обмежує область видимості даних для компонента лише необхідними йому даними, водночас зберігає можливість оновлювати ці дані з компоненту, не знаючи про загальну структуру контейнера з даними.

Для кожного з етапів життєвого циклу компоненту React передбачений бібліотекою передбачений свій протокол - для того, щоб визначити особливу поведінку під час конкретного етапу, достатньо реалізувати відповідний протокол. Кожний Om-компонент має визначити щонайменше протокол, котрий описує render-фазу (безпосередню генерацію HTML, протокол IRender або IRenderState). Під час створення екземпляру компонента йому обов'язково передається курсор на певну частину загального контейнера з даними. Дані з курсору доступні напряму під час фази render, проте під час інших фаз до них можна звертатись лише як до атому (один з базових елементів Clojure/ClojureScript) - попередньо потрібно "розіменувати" (dereference) посилання, котрим, по суті, є курсор.

Для генерації HTML можна використовувати стандартне API React'у React.DOM (що і було обрано для даної роботи), також існує декілька Om-специфічних додаткових бібліотек.

Прив'язка компоненту до DOM-елементу здійснюється за допомогою функції om/root, котра дещо відрізняється від варіанту React. Першим аргументом так само приймається компонент, а от другим вже виступає контейнер з даними. Також приймається третій параметр - асоціативний масив з додатковою конфігурацією, зокрема DOM-елементом, до якого слід прив'язати компонент.

Наведемо Om-компоненти, що відповідають вищеописаному прикладу для React:

```
(ns example
  (:require [om.core :as om :include-macros true]
            [om.dom :as dom :include-macros true]))

(def data (atom [{:author "Pete Hunt" :text "This is one comment"}
                 {:author "Jordan Walke" :text "This is *another* comment"}]))

(defn comment [comment owner]
  (reify
    om/IRender
    (render [this]
      (dom/div #js {:className "comment"}
        (dom/h2 #js {:className "commentAuthor"} (:author comment))
        (:text comment)))))

(defn comment-list [comments owner]
  (reify
    om/IRender
    (render [this]
      (dom/div #js {:className "commentList"}
        (om/build-all comment comments)))))

(defn comment-box [data owner]
  (reify
    om/IRender
    (dom/div #js {:className "commentBox"}
      (dom/h1 nil "Comments")
      (om/build comment-list data))))

(om/root comment-box data
  {:target (. js/document (getElementById "content"))})
```

Цікавою відмінністю від Om від React'у є обробники подій. Якщо в останньому зазвичай обробник подій несе у собі безпосередню логіку, котра має виконатись у відповідь на події, в Om часто використовують інший підхід - обробник лише переправляє інформацію про подію в канал даних, створений за допомогою реалізації CSP для Clojure/ClojureScript. Обробка нових повідомлень у каналі зазвичай відбувається у "батьківському" компоненті, в якому і створюються всі необхідні канали: їх створення відбувається під час фази `getInitialState`, а додавання обробників повідомлень з каналів - під час `componentWillMount`. Таким чином можливе значно спростити код компонентів-дітей (бо вся їх логіка це просто пересилання повідомлень), і зосередити обробку всіх подій в одному місці. Загалом, даний підхід досить схожий на PubSub, проте й має свої особливості. Розглянемо його детальніше в наступному розділі.

### 3.3 Communicating Sequential Processes та core.async

Communicating Sequential Processes - формальна мова опису схем взаємодії у паралельних системах, котра є представником так званих алгебр процесів (process calculi) і базується на передачі повідомлень каналами (channel). Вперше CSP було описано у 1978 році в роботі С. А. Р. Хоара.[11]

Найяскравішими представниками підходу є мови програмування оссам, Limbo, Go та бібліотека core.async. Донедавна підхід рідко використовувався у практичних задачах, але відносно часто залучався для специфікації та верифікації паралельних аспектів складних систем, наприклад мікропроцесора INMOS T9000 Transputer. Цікавим використанням було моделювання системи управління відмовами створеної для використання на Міжнародній Космічній станції, котре проводилось Бременським інститутом безпечних систем та Daimler-Benz Aerospace - за допомогою CSP дослідники довели, що їх система не містить deadlock'ів та livelock'ів. При цьому під час процесу моделювання та аналізу було виявлено певну кількість помилок, котрі не могли бути знайдені звичайними тестами. Схожим чином Praxis High Integrity Systems залучило CSP для верифікації своєї системи Certification Authority для смарт-карт - компанія стверджує, що завдяки цьому їх рішення має набагато менший рівень відмов, ніж пропозиції конкурентів.

В останні роки CSP починають все частіше використовувати і у повсякденних прикладних задачах (як от, побудова асинхронних веб-застосунків, графічних інтерфейсів), насамперед через поширення мови програмування Go, модель паралельності якої повністю будується на даному підході.

CSP дозволяє описувати системи як набір незалежних компонент-процесів, котрі взаємодіють один з одним лише за допомогою передачі повідомлень. Взаємодія між процесами один з одним та з середовищем



описується за допомогою спеціальних операторів алгебри процесів.

Поєднуючи декілька основних примітивів різним чином, можливо легко описувати навіть доволі складні системи.

Існує два основних класи примітивів:

### **події**

акт спілкування або взаємодії. Вважається, що вони є неподільними та миттєвими.

### **примітивні процеси**

процеси, що описують базову поведінку, наприклад, STOP (процес, що нічого не передає - deadlock) та SKIP (описує успішне завершення роботи)

Найважливішими операторами є:

### **префікс**

поєднує подію і процес, породжуючи новий процес. Наприклад,  $a \rightarrow P$  описує процес, що чекає на подію  $a$ , а після цього поводить себе як процес  $P$

### **детермінований вибір**

описує процес, робота якого розвивається за двома різними шляхами в залежності від того, яка подія надійде з оточення. Наприклад, якщо відбувається подія  $a$  буде виконуватись процес  $P$ , а якщо подія  $b$  - то процес  $Q$

### **недетермінований вибір**

схожий на попередній, але розвиток процесу не залежить від оточення, а чекає на те, поки відбудуться обидві події, а потім "самостійно" (в залежності від внутрішнього стану системи) визначає, яку гілку виконання обрати

### **перетинання (interleave)**

описує процес, що поєднує в собі незалежну одночасну роботу двох

інших процесів

### **паралельний інтерфейс (interface parallel)**

процес, що поєднує в собі роботу одночасну роботу двох інших процесів, при цьому вони мусять синхронізуватись один з одним - вказані в операторі події можуть бути оброблені, лише коли обидва процеси здатні їх обробити

### **сховування (hiding)**

описує процес, котрий не оброблює вказані події, які оброблюються базовим процесом

Хоча дана теорія доволі схожа з моделлю акторів, існує декілька фундаментальних відмінностей у примітивах, що використовуються у даних підходах:

- процеси в CSP анонімні, а актори мають "ідентичність";
- в CSP передача повідомлень вимагає явної "зустрічі" між відсилачем та отримувачем повідомлення - повідомлення не може бути надіслане допоки нема отримувача, здатного його прийняти. Натомість, в моделі акторів надсилання повідомлень є повністю асинхронним;
- CSP використовує явні канали для передачі повідомлень, а модель акторів передає їх іменованим отримувачам (акторам).

Як вказувалось раніше, в моєму застосуванні використовується реалізація CSP для мови програмування Clojure/ClojureScript, а саме бібліотека `core.async`. Вона надає можливість асинхронного програмування за допомогою каналів. Основними задачами бібліотеки є:

- надати інструменти для незалежних потоків дій, що взаємодіють за допомогою подібних до черг каналів;
- підтримка "справжніх" потоків та сумісного використання потоків з пулу;
- базуватись на CSP, водночас розвиваючи її.[12]

Як було сказано раніше про CSP, головною ознакою каналів є те, що вони блокуючі. Водночас, бібліотека надає можливість створювати буферизовані канали, здатні приймати задану кількість повідомлень без необхідності їх моментального отримання.

Використання бібліотеки у Clojure та ClojureScript дещо відрізняється, так як моє застосування написане на ClojureScript, надалі зосередимось на розгляді API саме цієї версії.

Використання бібліотеки починається з імпортування визначених у ній функцій:

```
(require-macros '[cljs.core.async.macros :refer :all])
(require '[cljs.core.async :refer :all])
```

Для створення каналу використовується наступна функція:

```
(chan)
```

Створений канал можливо "закрити" - він перестав приймати в себе нові повідомлення, але ті, що вже в ньому присутні, все ще доступні для читання. Коли всі повідомлення з каналу вичерпані, він повертає nil.

```
(close! (chan))
```

Так як JavaScript однопоточний, бібліотека використовує аналог легковісних потоків для створення уявлення багатопоточної обробки повідомлень. Для цього операції треба "огортати" код у спеціальний макрос go, котрий передає дії на асинхронне виконання, повертає інший канал, в якому з'явиться результат операцій і дозволяє програмі продовжувати своє виконання. Базовими операціями над каналом є запис та читання повідомлень:

```
(let [c (chan 3)]
  (go (>! c "hello")
    (<! c)))
```

Якщо потрібно чекати на появу повідомлення в одному з декількох каналів, можна використовувати операцію alts!:

```
(let [c1 (chan) c2 (chan)]
  (go (while true
    (let [[v ch] (alts! [c1 c2])]
      (println "Read" v "from" ch))))
  (go (>! c1 "hi"))
  (go (>! c2 "there")))
```

### 3.4 Огляд виконаної реалізації

Застосування було виконано у формі так званого single-page application, при чому серверна частина у нього відсутня, всі обрахунки відбуваються на клієнтській стороні. Як зазначалось у попередніх розділах, застосування написане на мові ClojureScript з використанням бібліотеки Om для генерації HTML та його відображення, core.async для взаємодії компонентів застосування, також був використаний CSS-фреймворк Bootstrap для надання стандартним HTML-елементам більш гарного зовнішнього вигляду. Для базових функцій клітинного автомату були написані генераційні тести за допомогою бібліотек clojurecript.test та double-check, код візуалізації не тестувався.

Код застосування зберігається у хостингу репозиторіїв GitHub як проект з відкритим кодом <https://github.com/gsnemark/beatha>, а саме скомпільоване застосування розміщене у публічному доступі за допомогою безкоштовного сервісу хостингу статичних сторінок GitHub Pages і доступне за посиланням <http://gsnemark.github.io/beatha/>. При кожному оновленні коду на GitHub автоматично запускаються тести, а також компілюється весь проект та завантажується на GitHub Pages (якщо тести закінчились успішно) за допомогою сервісу Travis-CI. Таким чином у мережі завжди доступний не лише вихідний код, а й найсвіжіша працююча версія застосування.

Застосування складається з двох основних екранів: головне меню та відображення обраного клітинного автомату. Кожний з екранів є окремим Om-компонентом. При запуску застосування спершу показується головне меню.

Воно містить кнопки запуску одного з трьох можливих клітинних автоматів:

### Cellular automata experiments

Game of Life
Unrestricted language parser
Economic model

Рисунок 14

Перша кнопка запускає звичайний варіант Game of Life без будь-якого розширення інформаційними каналами. Друга - розпізнавач необмеженої граматики, а третя - модель економічних відносин. Через свою допоміжну роль, компонент виконаний не у звичному для Om стилі, а дещо спрощений: обробники натискань на кнопки викликають відображення наступного екрану напрямку, це відображення відбувається шляхом знищення поточного екрану з DOM-елементу та відмальовки наступного екрану у цьому ж DOM-елементі. Розглянемо детальніше кожен з наявних клітинних автоматів.

Екран з реалізацією Game of Life надає базові можливості взаємодії з клітинним автоматом:

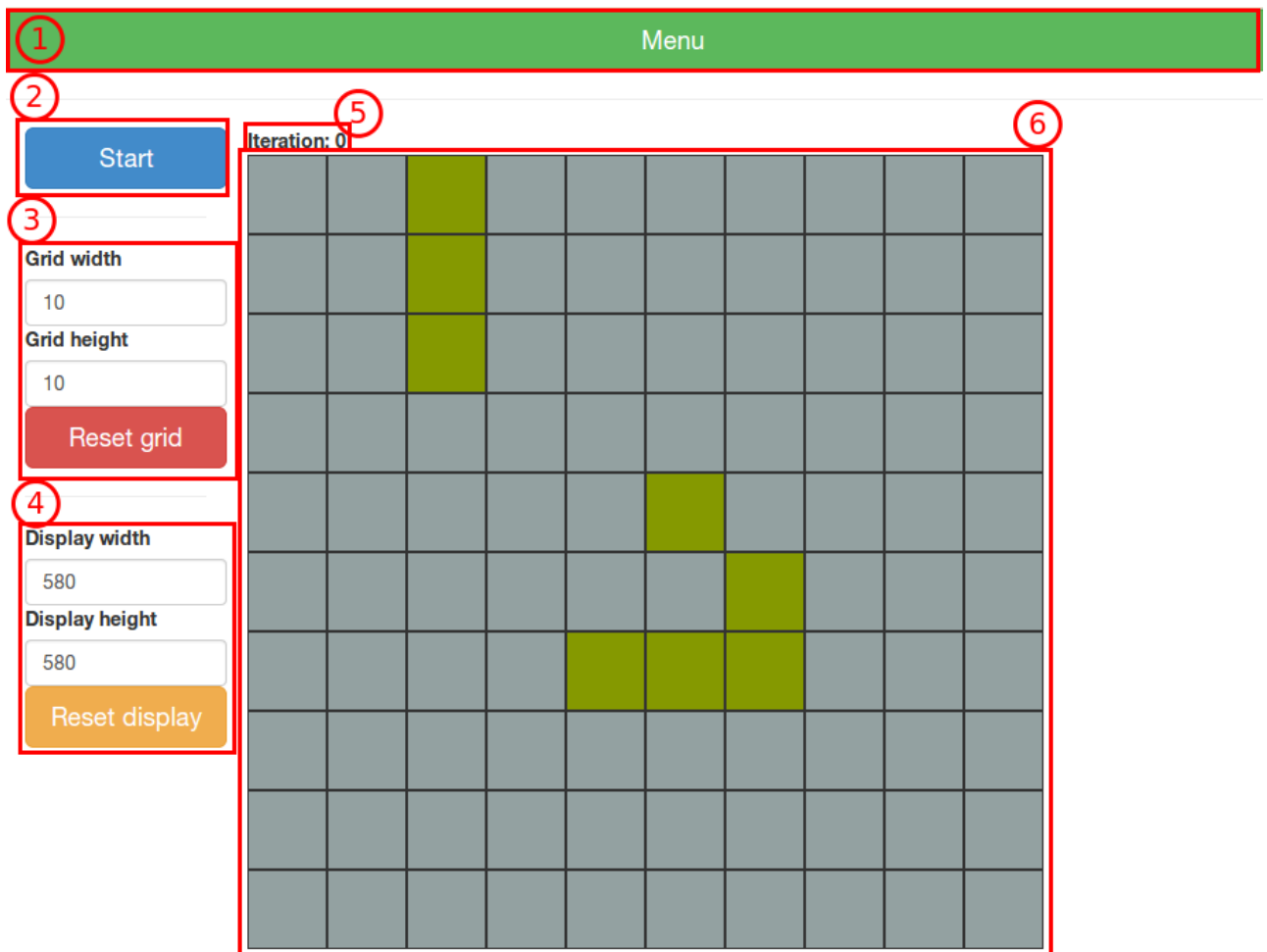


Рисунок 15

Згори знаходиться кнопка (1), котра дозволяє знищити поточний автомат та повернутись у головне меню.

У правій частині екрану знаходиться показник поточної ітерації (5) та сітка автомату (6). Вона може перебувати у двох станах - налаштування та симуляція. У стані налаштування можливо змінювати певні глобальні параметри сітки автомату (розглянуто далі), а натиснення на комірку призводить до зміни її поточного стану на наступний: для Game of Life таких стани два - мертвий (клітина сірого кольору) та живий (клітина зеленого кольору), за замовченням всі клітини мертві. У режимі симуляції сітка вже не сприймає натиснення на себе користувачем, а покроково змінює свій стан

відповідно до правил автомату кожні пів секунди. Для Game of Life правила наступні:

```
(cond
  (and alive? (or (< n 2) (> n 3))) {state :dead}
  (and alive? (or (= n 2) (= n 3))) {state :alive}
  (and (not alive?) (= n 3))      {state :alive}
  :else                          cell)
```

де  $n$  - кількість живих сусідів (використовується "сусідство" Мура), *cell* поточний стан клітини, *alive?* - булевий вираз, що описує поточний стан клітини (чи клітина жива).

У лівій частині екрану розміщується блок налаштування клітинного автомату, котрий складається з наступних частин:

2. Кнопка запуску/зупинки роботи автомату ("Start") - якщо автомат ще не у режимі симуляції переводить його у цей режим, інакше - зупиняє симуляцію та переводить автомат у режим налаштування.
3. Блок налаштування розмірності сітки - дозволяє змінити кількість стовпчиків та рядків комірок сітки. У поля заноситься бажана розмірність сітки, а кнопка оновлює сітку відповідно до введених значень, при чому стан сітки скидається у початковий (всі клітини "мертві"). Доступно лише у режимі налаштування.
4. Блок налаштування візуального розміру сітки - дозволяє вказати скільки пікселів по ширині та висоті займає сітка на екрані. У поля заноситься бажана ширина та висота, а натиснення кнопки призводить до їх реальної зміни. Доступно лише у режимі налаштування.

Кожен з вищеописаних елементів є самостійним Om-компонентом, котрі об'єднані в одному батьківському, в якому зосереджений код обробки подій та описане взаємне розміщення компонентів-дітей. Початковий загальний стан застосування описується наступним чином:

```
(def app-state {automaton {grid {width 10 :height 10 :cells {}}
  :display {width 580 :height 580}
  :util {started false :iteration 0}}
  :command {}})
```

Асоціативний масив складається з двох основних частин:

`command` - містить специфічну для задачі інформацію стосовно керуючого каналу та `automaton` - містить безпосередньо асоціативний масив з інформацією про сам клітинний автомат. У ньому під ключем `grid` зберігається ще один асоціативний масив, який містить поточну розмірність сітки, а також асоціативний масив з клітинами - ключем цього масиву є вектор з координатою клітини (наприклад, `[1 1]`), а значенням - специфічне для конкретного автомату представлення стану клітини (для Game of Life це `{:state :alive}`), як зазначалось раніше, клітини зі "стандартним" станом у даному масиві не зберігаються (для Game of Life це мертві клітини). Під ключем `display` зберігається ширина та висота сітки у пікселях, а під ключем `util` - допоміжна інформація: `started` містить булеву змінну, котра визначає чи знаходиться автомат у режимі симуляції, а `iteration` - номер поточної ітерації.

Батьківський компонент визначає сім `core.async` каналів:

### **change-grid-dimensions**

оброблює зміну розмірності автомату, заповнюється повідомленнями під час натиснення на кнопку з блоку 3, повідомлення містить двохелементний вектор з новою логічною шириною та висотою сітки

### **change-display-dimensions**

оброблює зміну візуального розміру сітки, заповнюється повідомленнями під час натиснення на кнопку з блоку 4, повідомлення містить двохелементний вектор з новою шириною та висотою сітки

### **cell-state-changed**

змінює стан заданої клітини відповідно до правил автомату, повідомлення надсилаються з основного циклу застосування кожні пів секунди для кожної клітини автомату, повідомленням є двохелементний вектор з координатами клітини, стан якої потрібно змінити на наступний

### **started**

змінює режим роботи автомату на симуляцію або налаштування,



заповнюється повідомленнями під час натиснення на кнопку з блоку 2, повідомлення містить булеву змінну

### **reset**

скидає параметри автомату у значення за замовченням, заповнюється повідомленнями обробником повідомлень для каналу `change-grid-dimensions` (при зміні розмірності сітки), а також кодом, специфічним для певних автоматів

### **output-info-channel, command-info-channel**

оброблюють повідомлення з вихідного та керуючого інформаційних каналів відповідно, будуть розглянуті пізніше

Батьківський компонент розроблений таким чином, щоб візуалізувати роботу довільного клітинного автомату, а не якогось конкретного. Для цього були створені два протоколи, реалізувавши які опис клітинного автомату може бути візуалізований створеним компонентом.

Перший протокол описує загальну роботу клітинну автомату.

```
(defprotocol AutomatonSpecification
  "Describes basic interactions with particular set of cellular automata rules."
  (default-cell [this] "Default cell of the given automata.")
  (next-initial-state [this state]
    "Returns next possible initial state after the given one.")
  (next-grid [this grid]
    "Transforms given grid according to automata's rules."))
```

Функція *default-cell* повертає опис комірки за замовченням (котра, не має зберігатись у пам'яті). *next-initial-state* здійснює перебирання початкових станів, приймаючи на вхід певний стан, повертає наступний, що слідує за ним. Використовується дана функція у режимі налаштування для створення початкової конфігурації автомату. Остання функція *next-grid* є основною - отримавши на вхід опис поточної сітки автомату, генерує її наступну ітерацію (як згадувалось раніше, сітка описується асоціативним масивом з ключами *width*, *height*, *cells*).

Другий протокол описує розроблену модифікацію клітинного автомату, у Game of Life він, фактично, не використовується (містить реалізацію-заглушку).

```
(defprotocol InformationChannelsSpecification
  "Describes interactions with information channels which augment the regular
  cellular automata:

  - command channel is a source of external commands which augment rules of
    automata;
  - output channel is filled by the automata itself with aggregate
    information about current state."
  (process-command-channel [this ic]
    "Handles messages from the command information channel.")
  (fill-output-info-channel [this oc grid]
    "Sends a message about the current automata's state."))
```

Перша функція, *process-command-channel* приймає на вхід `core.async` канал, котрий моделює керуючий інформаційний канал та має специфічним для конкретного автомату чином оброблювати команду, що приходять у даний канал. А друга, *fill-output-info-channel*, приймає на вхід поточний стан сітки автомату, а також `core.async` канал, що описує вихідний інформаційний канал, і має надіслати певне повідомлення про агрегований стан сітки на цей канал.

Наступний автомат - парсер необмеженої граматики, використовує модифікацію автомату за допомогою цих інформаційних каналів, і тому зовнішній вигляд екрану з автоматом для нього дещо відрізняється:

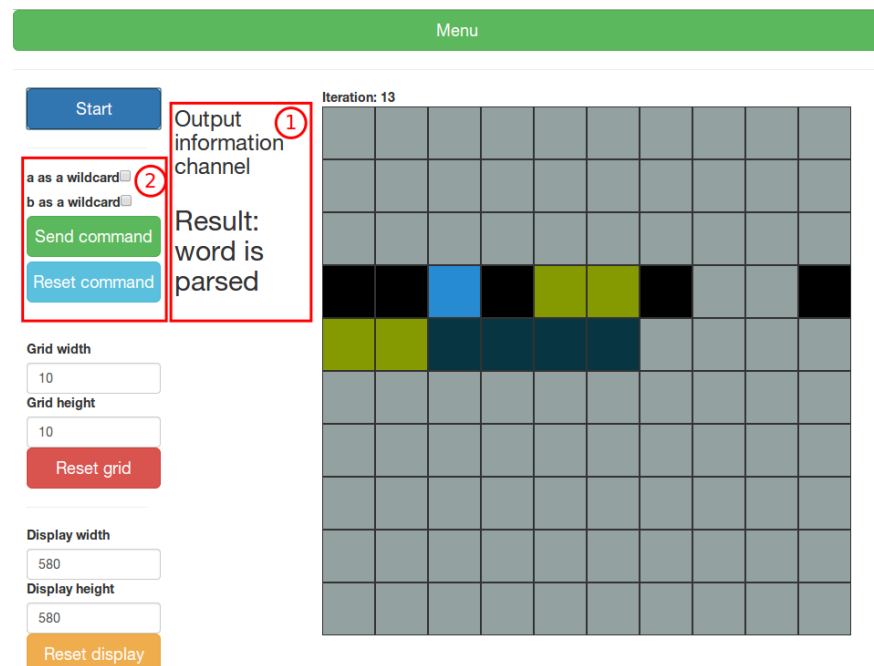


Рисунок 16

До попереднього варіанту відображення додаються два нові блоки:

1. Відображення повідомлень з вихідного інформаційного каналу - для парсера у цьому блоці з'являється повідомлення про те, чи належить задане слово граматиці (відповідно, буде виведено одне з повідомлень "Result: word is parsed" або "Result: word is not parsed"). Якщо на вихідний канал ще не приходило повідомлень, блок не показується. Для обробки цих повідомлень парсер додає у загальний стан застосування ще один ключ `result`, котрий містить булеву змінну, яка показує чи успішно закінчився процес розпізнавання.
2. Блок надсилання команд - містить засоби надсилання на керуючий інформаційний канал автомату команд. Для парсера у блоці розміщується чекбокси, що дозволяють сформувати команду (дві опції, які можна вибрати водночас - використовувати літеру "a" як wildcard, використовувати літеру "b" як wildcard), а також кнопка для надсилання сформованої команди та очищення автомату від попередньо надісланої команди (відміняє її дію). Даний блок доступний як в режимі налаштування, так і в режимі симуляції, завдяки чому команди можна надсилати вже під час роботи автомату.

Зрозуміло, що робота з інформаційними каналами для кожного автомату буде своя, тому батьківський `Om`-компонент за замовченням не містить компонентів-дітей для відображення даної взаємодії. Натомість, він приймає опціональний параметр - реалізацію ще одного протоколу, котра визначає складові елементи такого відображення, і інтегрує ці елементи у загальний інтерфейс. Зокрема, використовує функції з цього протоколу для обробки повідомлень з раніше згаданих каналів *output-info-channel* (заповнюється у головному циклі застосування на кожному кроці роботи автомату, повідомлення відповідають повідомленням, що надіслані у вихідний інформаційний канал), *command-info-channel* (заповнюється компонентом, що

відповідає за керуючий інформаційний канал, повідомлення містять команди, що надсилаються у автомат). Протокол виглядає наступним чином:

```
(defprotocol CellularAutomatonAppCustomization
  (automaton-specific-css [this data]
    "Return additional CSS rules for given automaton that should be added
    to the page.")
  (automaton-configuration-view [this]
    "Generates Om component that renders automaton-specific configuration
    block.")
  (automaton-command-view [this]
    "Generates Om component that renders block for sending commands to
    automaton. Please ensure the same component is returned on each call and
    the new one is generated (i. e., anonymous function).")
  (automaton-command-initial-state [this]
    "Returns the default stated for the command view.")
  (automaton-command-reset [this command-info-channel]
    "Resets any changes created by the command sent to the automata.")
  (automaton-output-handler [this data owner msg]
    "Handles messages posted by the cellular automaton.")
  (automaton-output-view [this]
    "Generates Om component that renders any changes produced by
    the handler. Please ensure the same component is returned on each call and
    the new one is generated (i. e., anonymous function).")
  (automaton-output-reset [this data owner]
    "Resets changes produced by the handler."))
```

Функція *automaton-specific-css* дозволяє додати на сторінку додаткові глобальні CSS-правила, специфічні для певного автомату. Насамперед використовується, якщо необхідно генерувати динамічні правила на основі певних параметрів автомату. Наступна функція, *automaton-configuration-view* визначає Om-компонент, котрий використовується для налаштування специфічних для автомату параметрів (приклад буде наведено далі). Функція *automaton-command-view* має повернути компонент, котрий відображає блок надсилання команд, *automaton-command-reset* відмінює дію команди на автомат (за замовченням викликається під час зміни розмірності автомату), *automaton-output-handler* оброблює повідомлення, що надходять у створений батьком канал *output-info-channel*, *automaton-output-view* повертає компонент, котрий відображає блок візуалізації повідомлень з вихідного інформаційного каналу, *automaton-output-reset* переводить створений минулою функцією компонент у початковий вигляд (за замовченням викликається під час зміни розмірності автомату).

Останній автомат - економічна модель, окрім модифікацій інтерфейсу присутніх у попередньому автоматі, дана модель використовує додатковий

блок конфігурації, специфічної для автомату (*automaton-configuration-view* з щойно розглянутого протоколу):

Рисунок 17

Надається можливість задати наступні додаткові параметри автомату:

### **кількість компаній**

скільки компаній приймають участь у ринковій боротьбі

### **тривалість "року"**

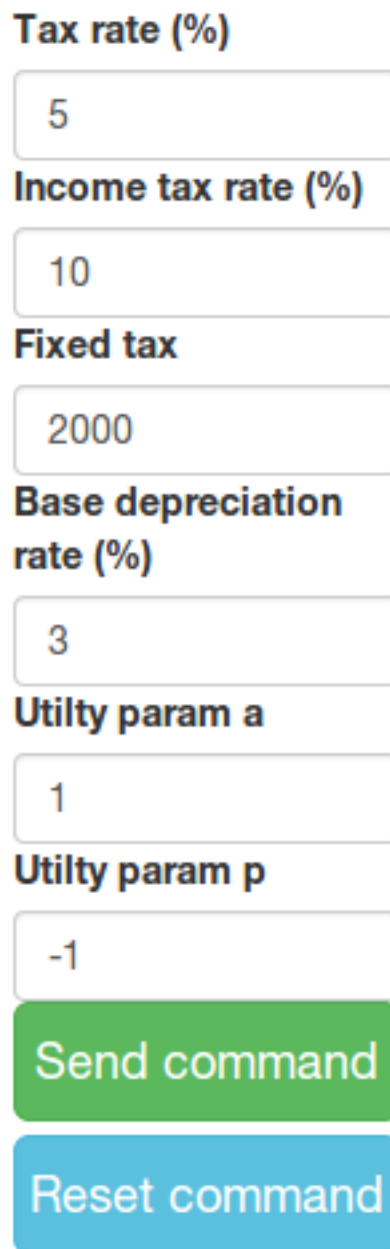
зі скількох ітерацій роботи автомату складається економічний "рік"

### **кількість виконаних ітерацій**

як довго модель має працювати до зупинки

Кожна клітина автомату відповідає покупцю, а її стан показує товар якої компанії придбаний даним користувачем. Так як кількість компаній є параметром, то й кількість станів клітини не є відомою наперед - через це даний автомат використовує генерацію додаткових CSS-стилів (*automaton-specific-css*): перед запуском моделі для кожного стану підбирається унікальний колір, котрий описується у окремому CSS-правилі.

Блок надсилання команд даного автомату є більш складним, ніж у розпізнавача мови:



**Tax rate (%)**

5

**Income tax rate (%)**

10

**Fixed tax**

2000

**Base depreciation rate (%)**

3

**Utlity param a**

1

**Utlity param p**

-1

**Send command**

**Reset command**

Рисунок 18

Він дозволяє редагувати широкий набір параметрів алгоритму, а саме розмір відсоткових ставок на оборот та прибуток, величину фіксованого податку, ймовірність відмови від поточного товару, а також параметри функції корисності покупця ( $a$  - внесок локальних вподобань та  $p$  - внесок ціни товару).

Більш цікавим є і блок відображення даних з вихідного інформаційного каналу:

Output information channel	
<b>Capital</b>	
Government:	37271.79753086419
Corporation 1:	55000.59259259259
Corporation 2:	45229
Corporation 3:	13516.444444444447
Corporation 4:	41042.066666666668
<b>Market share</b>	
Without good:	2%
Corporation 1:	24%
Corporation 2:	28%
Corporation 3:	25%
Corporation 4:	21%
<b>Good prices</b>	
Corporation 1:	19.75308641975309
Corporation 2:	40
Corporation 3:	29.629629629629633
Corporation 4:	64
<b>Taxation types</b>	
Corporation 1: rate	
Corporation 2: rate	
Corporation 3: income-rate	
Corporation 4: rate	
<b>Tax rate: 5%</b>	
<b>Income tax rate: 10%</b>	
<b>Fixed tax: 2000</b>	
<b>Base depreciation rate: 3%</b>	
<b>Utility function params:</b>	
a: 1 p: -1	

Рисунок 19

На ньому можна в реальному часі бачити детальну інформацію про стан автомату: рівень грошових запасів уряду та компаній, поточний розподіл ринку у відсотках, ціни на товар від різних компаній, обрану кожною компанією схему оподаткування, а також поточні усіх значення параметрів, що надсилаються командою. Цікавою особливістю є те, що для легшої візуальної ідентифікації всі показники певної компанії додатково виділяються таким самим кольором, що і відповідний стан клітини автомату.

Обраний для створення графічного інтерфейсу інструмент хоч і є доволі молодим та малопоширеним виявився надзвичайно зручним у використанні та підтримці. Завдяки обраній у Om моделі збереження та поширення даних особливих проблем не викликає створення універсальних компонентів, що з легкістю можуть бути використані для побудови подібних, проте все-таки різних інтерфейсів, а водночас і залишає можливість налаштування елементів під конкретний випадок (створення, умовно кажучи, компонентів вищого порядку). Зокрема, у створеній програмі інтерфейс усіх трьох автоматів базується на єдиному базовому компоненті котрий, за необхідності, розширюється додатковими підкомпонентами.

У процесі розробки деякі елементи інтерфейсу частково переписувались декілька разів: завдяки явній ставці на повну незалежність компонентів один від одного та їх спілкування за допомогою каналів `core.async` навіть відсутність статичної типізації та зміна вигляду вихідного коду, що безпосередньо потрапляє на сторінку, унаслідок компіляції Google Closure не склали значних перепон для безболісної зміни частин загальної системи. Загалом, використання Om та React, а також `core.async` виявилось незвичним, але продуктивним, підходом до створення веб-інтерфейсів.



## ВИСНОВКИ

В рамках даної магістерської роботи я дослідив розширення клітинних автоматів двома додатковими інформаційними каналами, котрі роблять автомат більш придатним для моделювання нетривіальних систем.

Як практичну частину було створено розпізнавач необмеженої формальної мови, а також модель економічної взаємодії компаній, держави та покупців на ринку високотехнологічних послуг.

Для візуалізації створених автоматів я ознайомився з новітньою бібліотекою створення веб-інтерфейсів користувача від Facebook React та її ClojureScript-обгорткою Om. Був розглянутий і засіб асинхронного програмування на основі CSP core.async. Використовуючи перелічені технології було створено single-page веб-застосування здатне відображати роботу довільного клітинного автомату прямокутної форми. Додатково застосування надає можливість налаштовувати відображення інформації з вихідного інформаційного каналу, зв'язаного з автоматом, та блоку надсилання команд на керуючий інформаційний канал даного автомату.

## СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Клеточные автоматы. Реализация и эксперименты [Электронный ресурс] / Лев Наумов, Анатолий Шалыто – <http://www.ict.edu.ru/ft/001854/klet.pdf>
2. Modelling the Spatial Dimension of Economic Systems with Cellular Automata [Электронный ресурс] / Max Keilbach – <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.3209>
3. Cellular automaton [Электронный ресурс] / Wikipedia – [https://en.wikipedia.org/wiki/Cellular\\_automaton](https://en.wikipedia.org/wiki/Cellular_automaton)
4. Chas Emerick. Clojure Programming / Chas Emerick, Brian Carper, Christophe Gran – O'Reilly, 2012 – 607 с.
5. Unrestricted grammar [Электронный ресурс] / Wikipedia – [http://en.wikipedia.org/wiki/Unrestricted\\_grammar](http://en.wikipedia.org/wiki/Unrestricted_grammar)
6. Why did we build React? [Электронный ресурс] / Pete Hunt – <http://facebook.github.io/react/blog/2013/06/05/why-react.html>
7. Interactivity and Dynamic UIs [Электронный ресурс] / React docs – <http://facebook.github.io/react/docs/interactivity-and-dynamic-uis.html>
8. Closure Tools [Электронный ресурс] / Google Closure docs – <https://developers.google.com/closure/>
9. Stuart Sierra. ClojureScript Up and Running / Stuart Sierra, Luke VanderHart — O'Reilly, 2012 — 100 с.
10. Om [Электронный ресурс] / David Nolen – <https://github.com/swannodette/om>
11. Communicating sequential processes [Электронный ресурс] / Wikipedia – [https://en.wikipedia.org/wiki/Communicating\\_sequential\\_processes](https://en.wikipedia.org/wiki/Communicating_sequential_processes)
12. Clojure core.async Channels [Электронный ресурс] / Rich Hickey – <http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html>

## ДОДАТОК А. ВИХІДНИЙ КОД ЗАСТОСУВАННЯ

**project.clj** – опис залежностей проекту та інших мета-даних

```
(defproject beatha "0.0.1-SNAPSHOT"
  :description "Cellular automata experiments."
  :url "https://gsnewmark.github.io/beatha/"
  :license {:name "Eclipse Public License – v 1.0"
            :url "http://www.eclipse.org/legal/epl-v10.html"
            :distribution :repo}

  :min-lein-version "2.3.4"

  :source-paths ["src/clj" "src/cljs"]

  :dependencies [[org.clojure/clojure "1.6.0"]
                 [org.clojure/clojurescript "0.0-2227"]
                 [org.clojure/core.async "0.1.303.0-886421-alpha"]
                 [om "0.6.2"]
                 [com.facebook/react "0.9.0.2"]
                 [org.webjars/bootstrap "3.1.1-1"]]

  :plugins [[lein-cljsbuild "1.0.4-SNAPSHOT"]]

  :hooks [leiningen.cljsbuild]

  :cljsbuild
  {:builds {:beatha
            {:source-paths ["src/cljs"]
             :compiler
             {:output-to "dev-resources/public/js/beatha.js"
              :optimizations :advanced
              :pretty-print false}}}}})
```

**profiles.clj** – опис допоміжних профілів побудови проекту для тестування, розробки та збирання production-версії

```
{:shared {:clean-targets ["out" :target-path]

          :test-paths ["test/cljs"]
          :resources-paths ["dev-resources"]

          :dependencies [[com.cemerick/double-check "0.5.7-SNAPSHOT"]]
          :plugins [[com.cemerick/clojurescript.test "0.3.1-SNAPSHOT"]]

          :cljsbuild
          {:builds
           {:beatha
            {:source-paths ["test/cljs"]
             :compiler {:libs [""]}
                       :optimizations :whitespace
                       :pretty-print true}}}}

          :test-commands
          {"phantomjs"
           ["phantomjs" :runner
            "dev-resources/public/js/beatha.js"]}}}

:prod [:shared
       {:cljsbuild
        {:builds
         {:beatha
```

```

{:source-paths []
 :compiler {:libs []
            :optimizations :advanced
            :pretty-print false
            :preamble ["react/react.min.js"]
            :externs ["react/externs/react.js"]}}}}}]

:tdd [:shared]

:dev [:shared
      {:source-paths ["dev-resources/tools/http" "dev-resources/tools/repl"]

       :dependencies [[ring "1.2.1"]
                      [compojure "1.1.6"]
                      [enlive "1.1.5"]]

       :plugins [[com.cemerick/austin "0.1.5-SNAPSHOT"]]

       :cljsbuild
       {:builds {:beatha
                  {:source-paths ["dev-resources/tools/repl"]}}}

       :injections
       [(require '[ring.server :as http :refer [run]]
                  'cemerick.austin.repls)
        (defn browser-repl-env []
          (reset! cemerick.austin.repls/browser-repl-env
                  (cemerick.austin/repl-env)))
        (defn browser-repl []
          (cemerick.austin.repls/cljs-repl
            (browser-repl-env)))}}}]

```

**script/publish\_app.sh** – скрипт для автоматичного розгортання застосування за допомогою Travis CI

```

#!/bin/bash

if [ "$TRAVIS_BRANCH" == "master" ]; then
  cd $HOME
  git clone --quiet https://:${GH_TOKEN}@github.com:gsnewmark/beatha > /dev/null

  cd beatha
  lein2 with-profile prod do clean, compile

  git checkout gh-pages
  cp dev-resources/public/js/beatha.js js/beatha.js
  git add js/beatha.js
  git commit -m 'Deploy to Github Pages'

  git remote set-url origin https://:${GH_TOKEN}@github.com:gsnewmark/beatha.git
  git push -fq origin gh-pages > /dev/null 2>&1
fi

```

**src/cljs/beatha/automaton.cljs** – реалізація логіки створених клітинних автоматів

```

(ns beatha.automaton
  (:require-macros [cljs.core.async.macros :refer [go]])
  (:require [cljs.core.async :refer [put! <!]]
            [clojure.set :as set]))

```

```

(defprotocol AutomatonSpecification
  "Describes basic interactions with particular set of cellular automata
  rules."
  (default-cell [this] "Default cell of the given automata.")
  (next-initial-state [this state]
    "Returns next possible initial state after the given one.")
  (next-grid [this grid]
    "Transforms given grid according to automata's rules.))

(defprotocol InformationChannelsSpecification
  "Describes interactions with information channels which augment the regular
  cellular automata:

  - command channel is a source of external commands which augment rules of
    automata;
  - output channel is filled by the automata itself with aggregate
    information about current state."
  (process-command-channel [this ic]
    "Handles messages from the command information channel.")
  (fill-output-info-channel [this oc grid]
    "Sends a message about the current automata's state.))

;;; State of world is described as a set of living cells, where each cell is
;;; vector of coordinates [x y].

(defn neighbours
  "Generates seq with neighbour cells of the given cell.

  Intended to work with a two-dimensional grid representation where each cell
  is encoded as pair of it's coordinates in a world."
  ([[x y]]
    {:pre [(not (neg? x)) (not (neg? y))]}
    (for [dx [-1 0 1] dy [-1 0 1] :when (not= 0 dx dy)]
      [(+ x dx) (+ y dy)]))
  ([width height coords]
    {:pre [(pos? width) (pos? height)]}
    (map (fn [[x y]]
      (let [x (if (< x 0) (+ width x) x)
            y (if (< y 0) (+ height y) y)]
        [(rem x width) (rem y height)]))
      (neighbours coords))))

(defn- transition
  [this grid dead-state tfn]
  (let [{:keys [width height cells]} grid]
    (->>
      (range height)
      (mapcat
        (fn [y] (map (fn [x] (tfn this width height cells x y)) (range width))))
        (remove #(let [[cell] (vals %)] (nil? cell)))
        (remove #(let [[cell] (vals %)] (= (:state cell) dead-state)))
        (reduce merge {})
        (assoc grid :cells))))

(def default-automata
  (reify
    AutomatonSpecification
    (default-cell [_] {:state :dead})
    (next-initial-state
      [_ state]
      ({:dead :alive :alive :speaking :speaking :dead} state))
    (next-grid [_ grid] grid))

```



```

letter? #{:a :b}
dead? #(= :dead %)
alive-only
(fn [dir pred]
  (and (pred (dir n-states))
        (every? dead? (vals (dissoc n-states dir))))))

between-l-r
(fn [left-states right-states]
  (and (left-states (:left n-states))
        (right-states (:right n-states)))))

command (get @ext-command :command (fn [_ _] false)))

{[x y]
 (or (command n-states cell)
     {:state
      (cond
        (and (dead? s) (alive-only :right letter?)) :lc
        (and (dead? s) (alive-only :left letter?)) :rc
        (and (dead? s)
              (or (alive-only :right #{:lc})
                  (alive-only :left #{:rc}))) :x
        (and (= :lc s)
              (between-l-r #{:x :a :b :dead} letter?)) right
        (and (= :rc s)
              (between-l-r letter? #{:x :a :b :dead})) left
        (and (letter? s) (between-l-r #{:lc} letter?)) :lc
        (and (letter? s) (between-l-r letter? #{:rc})) :rc
        (between-l-r #{:lc} #{:rc}) :f
        (or (and (= :lc s) (= :rc right))
            (and (= :lc left) (= :rc s))) :m
        (and (dead? s) (= :m top)) :n
        (and (dead? s) (letter? top) (= :n right)) :n
        (and (dead? s) (letter? top)
              (#{:n :a :b} left)) top
        (and (= :n s) (letter? right)) right
        (and (letter? s) (= :n left)) :n
        (and (letter? s) (= :x left) (= s bottom)) :x
        (and (letter? s) (= :x left)
              (letter? bottom) (not (= s bottom))) :f
        (and (= :m s) (= :x left)) :s
      :else
      s)}})))))

```

#### InformationChannelsSpecification

```

(process-command-channel [this ic]
  (go (while true (let [cmd (<! ic)] (reset! ext-command cmd)))))
(fill-output-info-channel [this oc grid]
  (put! oc (->> grid
                 :cells
                 vals
                 (group-by :state)
                 (#(select-keys % [:s :f]))
                 (map (fn [[k v]] [k (count v)]))
                 (into {}))
                 (merge {:s 0 :f 0}))))))

```

```

(defn- weighted
  [m]
  (when-not (empty? m)
    (let [w (reductions #(+ % %2) (vals m))
          r (rand (last w))]
      (nth (keys m) (count (take-while #(<= % r) w))))))

```

```

;;; from https://groups.google.com/d/msg/clojure/UdFLYjLvNRs/NqlA7wnLCE0J
(defn- deep-merge
  "Recursively merges maps. If vals are not maps, the last value wins."
  [& vals]
  (if (every? map? vals)
      (apply merge-with deep-merge vals)
      (last vals)))

(defn corp-keyword [i] (keyword (str "corp-" i)))

(defn- corp-params
  [n pfn]
  (into {}
        (map (fn [i]
                 (let [corp (corp-keyword i)]
                   [corp (pfn corp i)]))
              (range 1 (inc n)))))

(def market-model-default-params
  {:tax-rate 0.05
   :income-tax-rate 0.1
   :fixed-tax 2000
   :expenditures-per-cell
   {:government 1 :corp 20}
   :depreciation 0.03
   :utility-params
   {:a 1 :p -1 :q 0}
   :utility
   (fn [a p price global-share local-share]
     (* global-share
        (Math/pow local-share a)
        (Math/pow price p)))))

(defn market-model-default-state
  [n]
  (merge market-model-default-params
        {:corp-quantity n
         :taxation-type
         (corp-params n (constantly :rate))
         :prices
         (corp-params n (constantly 100))
         :capital
         (merge {:government 1000} (corp-params n (constantly 0)))
         :capital-diff
         (corp-params n (constantly 0))
         :capital-incomings
         (corp-params n (constantly 0))
         :capital-expenditures
         (corp-params n (constantly 0))}))

(defn conservative-corp-price
  [[competitor-count share capital capital-diff price]]
  (cond
    (< capital-diff 0)
    {:type :change-price
     :cmd (* price 1.5)}
    (and (< share (/ 1.0 competitor-count))
         (> price 100))
    {:type :change-price
     :cmd (/ price 3)}
    (and (< share (/ 1.0 competitor-count))
         (> price 10))
    {:type :change-price
     :cmd (/ price 1.5)}
    (and (< share (/ 1.0 competitor-count 2))
         (> price 100))
    {:type :change-price

```





```

                                (/ v total-pref)
                                (/ 1 (count available-goods))))))
                                p))))
    (into {}))))))
(reify
 AutomatonSpecification
 (default-cell [_] {:state :without-good})
 (next-initial-state [_ state]
  (let [corp-quantity (:corp-quantity @env)
        corp-states-progression
        (merge {:without-good
                  (if (> corp-quantity 0) :corp-1 :without-good)}
                (corp-params corp-quantity
                              (fn [corp i]
                                (if (< i corp-quantity)
                                    (corp-keyword (inc i))
                                    :without-good))))))
        (get corp-states-progression state :without-good))]
    (next-grid [this grid]
      (let [env-atom env
            env @env
            corp-quantity (:corp-quantity env)
            cell-quantity (* (:width grid) (:height grid))
            global-share (compute-global-share corp-quantity grid)

            update-capitals
            (fn [env key]
              (let [cell-quantity (* cell-quantity (get global-share key 0))
                    income (* cell-quantity (get-in env [:prices key] 0))
                    exp (* cell-quantity
                           (get-in env [:expenditures-per-cell :corp] 0))
                    tax (if (> income 0)
                          (condp = (get-in env [:taxation-type key] :rate)
                            :rate (* (+ income exp) (get env :tax-rate 0))
                            :income-rate
                              (* income (get env :income-tax-rate 0))
                            :fixed (get env :fixed-tax 0))
                          0)
                    diff (- income tax exp)]
                (-> env
                  (update-in [:capital key] + diff)
                  (update-in [:capital :government] + tax)
                  (update-in [:capital-diff key] + diff)
                  (update-in [:capital-incomings key] + income)
                  (update-in [:capital-expenditures key] + exp))))))

            (swap! env-atom
              (fn [e]
                (-> (reduce update-capitals e (keys (:prices e)))
                  (update-in
                     [:capital :government]
                     (fn [capital]
                       (- capital
                          (* cell-quantity
                             (get-in e [:expenditures-per-cell :government]
                                       0))))))))))

            (transition
             this grid :without-good
             (fn [this width height cells x y]
               (let [get-cell (partial get-cell this cells)
                     n-states
                     (ordered-neighbour-states get-cell width height x y)

                     {:keys [top-left top top-right left right
                             bottom-left bottom bottom-right]
                     }

```

```

n-states

cell (get-cell [x y])
s (:state cell)
without-good? (fn [s] (= :without-good s))
avg-price (/ (apply + (vals (:prices env)))
              (:corp-quantity env))
current-price (get-in env [:prices s])
bankrupt? (fn [c] (and
                  (not (without-good? s))
                  (< (get-in env [:capital c] -1) 0))))

{[x y]
 {:state
  (cond
   (or (< (rand) (* (:depreciation env)
                    (/ current-price avg-price)))
    (bankrupt? s))
   :without-good

  (without-good? s)
  (if-let [c (weighted
               (user-preferences env global-share n-states))]
   c
   :without-good))

 :else s)}})))))

InformationChannelsSpecification
(process-command-channel [this ic]
  (go
    (while true
      (let [msg (<! ic)
            {:keys [type cmd]} msg]
        (condp = type
          :reset
            (reset! env
                     (market-model-default-state (:corp-quantity @env)))

          :change-corp-quantity
            (reset! env (market-model-default-state cmd))

          :change-price
            (let [[corp price] cmd]
              (swap! env #(-> %
                               (assoc-in [:prices corp] price)
                               (assoc-in [:capital-diff corp] 0))))

          :change-taxation-type
            (let [[corp taxation-type] cmd]
              (swap! env #(-> %
                               (assoc-in [:taxation-type corp]
                                         taxation-type)
                               (assoc-in [:capital-incomings corp] 0)
                               (assoc-in [:capital-expenditures corp] 0))))

          :do-nothing
            nil

          :config
            (swap! env deep-merge cmd))))))
  (fill-output-info-channel [this oc grid]
    (let [env @env]
      (put! oc (-> env
                    (dissoc :utility))

```

```
(assoc :global-user-share
      (compute-global-share
        (:corp-quantity env) grid)))))))))
```

**src/cljs/beatha/core.cljs** – реалізація графічного інтерфейсу та логіки надсилання команд на керуючий інформаційний канал, обробки даних з вихідного інформаційного каналу

```
(ns beatha.core
  (:require-macros [cljs.core.async.macros :refer [go alt!]])
  (:require [om.core :as om :include-macros true]
            [om.dom :as dom :include-macros true]
            [cljs.core.async :refer [put! chan <!]]
            [beatha.automaton :as a]))

(enable-console-print!)

(def app-state {:automaton {:grid {:width 10 :height 10 :cells {}}
                             :display {:width 580 :height 580}
                             :util {:started false :iteration 0}}
               :command {}})

(defn handle-int-config-change
  [e owner state key]
  (let [value (.. e -target -value)]
    (if (re-matches #"^-?[0-9]+" value)
      (om/set-state! owner key (js/parseInt value))
      (om/set-state! owner key (get state key)))))

(defn handle-float-config-change
  [e owner state key]
  (let [value (.. e -target -value)]
    (if (re-matches #"^-?[0-9]+(\\.[0-9]+)?" value)
      (om/set-state! owner key (js/parseFloat value))
      (om/set-state! owner key (get state key)))))

(defn navigation-button
  [text f]
  (dom/button
    #js {:type "button"
         :className "btn btn-success btn-lg btn-block"
         :onClick (fn [] (f))}
    text))

(def ^:private empty-view
  (fn [data owner] (reify om/IRender (render [_] (dom/span nil "")))))

(defn num->percent [n] (str (Math/floor (* n 100)) "%"))

(defn normalize
  [n min max]
  (cond
    (< n min) min
    (> n max) max
    :else n))

(defn is-number? [v] (not (js/isNaN v)))

(defn keyword->str
  [k]
```

```

    (when (keyword? k)
      (.substr (str k) 1)))

(defn random-hsl-colors
  [total]
  (->> (range total)
    (map (partial * (/ 360 total)))
    (interleave (cycle ["50%,35%" "50%,65%"]))
    (partition 2)
    (map #(apply str (reverse %)))))

(defn grid-config-view
  [data owner]
  (reify
    om/IRenderState
    (render-state [_ {:keys [width height change-grid-dimensions] :as state}]
      (let [started (:started data)]
        (dom/div
          #js {:role "form" :className "automaton-grid-control"}
          (dom/label nil "Grid width")
          (dom/input
            #js {:type "text" :className "form-control" :value width
                  :disabled started
                  :onChange
                    #(handle-int-config-change % owner state :width)}})
          (dom/label nil "Grid height")
          (dom/input
            #js {:type "text" :className "form-control" :value height
                  :disabled started
                  :onChange
                    #(handle-int-config-change % owner state :height)}})
          (dom/button
            #js {:type "button" :className "btn btn-danger btn-lg btn-block"
                  :disabled started
                  :onClick #(put! change-grid-dimensions [width height])}
            "Reset grid")))))

(defn display-config-view
  [data owner]
  (reify
    om/IRenderState
    (render-state [_ {:keys [width height change-display-dimensions]
                       :as state}]
      (let [started (:started data)]
        (dom/div
          #js {:role "form" :className "automaton-display-control"}
          (dom/label nil "Display width")
          (dom/input
            #js {:type "text" :className "form-control" :value width
                  :disabled started
                  :onChange
                    #(handle-int-config-change % owner state :width)}})
          (dom/label nil "Display height")
          (dom/input
            #js {:type "text" :className "form-control" :value height
                  :disabled started
                  :onChange
                    #(handle-int-config-change % owner state :height)}})
          (dom/button
            #js {:type "button" :className "btn btn-warning btn-lg btn-block"
                  :disabled started
                  :onClick #(put! change-display-dimensions [width height])}
            "Reset display")))))

```

```

"Reset display"))))))))

(defn cell-view
  [data owner]
  (reify
    om/IRenderState
    (render-state [_ {:keys [cell-state-changed x y]}]
      (let [{:keys [width height state started]} data
            st #js {:width width :height height}]
        (dom/div #js {:style st
                      :className
                      (apply str
                        (interpose " " ["automaton-cell" (name state)]))
                      :onClick #(when-not started
                                  (put! cell-state-changed [x y]))}))))))

(defn grid-view
  [data owner]
  (reify
    om/IRenderState
    (render-state [_ state]
      (let [grid-width (get-in data [:grid :width])
            grid-height (get-in data [:grid :height])
            cell-width (/ (get-in data [:display :width]) grid-width)
            cell-height (/ (get-in data [:display :height]) grid-height)
            default-cell (:default-cell state)]
        (apply dom/div #js {:className "automaton-grid"}
          (dom/b nil "Iteration: " (get-in data [:util :iteration] 0))
          (mapv (fn [y]
                  (apply dom/div #js {:className "automaton-row row"}
                    (mapv
                      (fn [x]
                        (om/build
                          cell-view
                          (merge (get-in data [:grid :cells [x y]]
                                          default-cell)
                                {:started (:started data)
                                 :width cell-width
                                 :height cell-height})
                        {:init-state
                         {:cell-state-changed
                          (:cell-state-changed state)

                          :x x :y y}}))
                      (range grid-width))))
                  (range grid-height))))))

(defprotocol CellularAutomatonAppCustomization
  (automaton-specific-css [this data]
    "Return additional CSS rules for given automaton that should be added
    to the page.")
  (automaton-configuration-view [this]
    "Generates Om component that renders automaton-specific configuration
    block.")
  (automaton-command-view [this]
    "Generates Om component that renders block for sending commands to
    automaton. Please ensure the same component is returned on each call and
    the new one is generated (i. e., anonymous function).")
  (automaton-command-initial-state [this]
    "Returns the default stated for the command view.")
  (automaton-command-reset [this command-info-channel]
    "Resets any changes created by the command sent to the automata."))

```

```

(automaton-output-handler [this data owner msg]
  "Handles messages posted by the cellular automaton.")
(automaton-output-view [this]
  "Generates Om component that renders any changes produced by
  the handler. Please ensure the same component is returned on each call and
  the new one is generated (i. e., anonymous function).")
(automaton-output-reset [this data owner]
  "Resets changes produced by the handler.")

(declare render-menu-view)

(defn gen-app-view
  ([automaton-spec]
    (gen-app-view automaton-spec
      (reify
        CellularAutomatonAppCustomization
        (automaton-specific-css [_ _] "")
        (automaton-configuration-view [_] empty-view)
        (automaton-command-view [_] empty-view)
        (automaton-command-initial-state [_] {})
        (automaton-command-reset [_ _])
        (automaton-output-handler [_ _ _ _])
        (automaton-output-view [_] empty-view)
        (automaton-output-reset [_ _ _])))))
  ([automaton-spec customization]
    (fn [data owner]
      (reify
        om/IInitState
        (init-state [_]
          {:reset (chan)
           :change-grid-dimensions (chan)
           :change-display-dimensions (chan)
           :cell-state-changed (chan)
           :started (chan)
           :output-info-channel (chan)
           :command-info-channel (chan)}))
        om/IWillMount
        (will-mount [_]
          (let [reset-c (om/get-state owner :reset)
                grid-c (om/get-state owner :change-grid-dimensions)
                display-c (om/get-state owner :change-display-dimensions)
                cell-state-c (om/get-state owner :cell-state-changed)
                started-c (om/get-state owner :started)
                output-info-c (om/get-state owner :output-info-channel)
                command-info-c (om/get-state owner :command-info-channel)]
            (a/process-command-channel automaton-spec command-info-c)
            (go (while true
                  (alt!
                    reset-c
                    ([reset-grid?]
                     (automaton-command-reset customization command-info-c)
                     (automaton-output-reset customization data owner)
                     (when reset-grid?
                       (om/transact!
                        data [:automaton :grid]
                        (fn [grid] (assoc grid :cells {}))))
                     (om/update! data [:automaton :util :iteration] 0))))
                  grid-c
                  ([[width height]]
                   (put! reset-c false)
                   (om/transact!
                    data [:automaton :grid]

```

```

      (fn [grid]
        (assoc grid
          :width width :height height :cells {}))))

display-c
([[width height]]
 (om/transact!
  data [:automaton :display]
  (fn [display]
    (assoc display :width width :height height))))

cell-state-c
([[x y]]
 (om/transact!
  data [:automaton :grid :cells]
  (fn [grid]
    (let [cell (get grid [x y]
                     (a/default-cell automaton-spec))
          state (:state cell)
          cell (assoc cell
                      :state
                      (a/next-initial-state automaton-spec
                        state))]
      (assoc grid [x y] cell)))))

started-c
([started]
 (om/update! data [:automaton :util :started] started)
 (if started
  (om/set-state!
   owner :update-loop-id
   (js/setInterval
    (fn []
      (om/transact!
       data
        [:automaton :grid]
        (partial a/next-grid automaton-spec))
      (om/transact!
       data [:automaton :util :iteration] inc)
      (a/fill-output-info-channel
       automaton-spec
       output-info-c
       (get-in @data [:automaton :grid]))
      (or (om/get-state owner :animation-step) 1000)))
    (when-let [id (om/get-state owner :update-loop-id)]
      (js/clearTimeout id))))))

output-info-c
([o] (automaton-output-handler
      customization data owner o))))))

om/IRenderState
(render-state [_ state]
 (dom/div
  #js {:className "container-liquid"}
  (dom/style #js {:media "screen" :type "text/css"}
   (automaton-specific-css customization data))
  (dom/div
   #js {:className "row"}
   (navigation-button "Menu" render-menu-view)
   (dom/hr nil))
  (dom/div
   #js {:className "row"}
   (dom/div
    #js {:className "col-sm-2"}

```



```

(let [started (get-in data [:automaton :util :started])]
  (dom/div
    #js {:className "row"}
    (dom/button
      #js {:type "button"
            :className "btn btn-primary btn-lg btn-block"
            :onClick #(put! (:started state) (not started))}
      (if started "Stop" "Start"))))
  (dom/hr nil)
  (dom/div
    #js {:className "row"}
    (om/build
      (automaton-command-view customization)
      (:command data)
      {:init-state
       (merge
        (select-keys state [:command-info-channel])
        (automaton-command-initial-state customization))}))
    (dom/div
      #js {:className "row"}
      (om/build (automaton-configuration-view customization)
        (merge (get-in data [:automaton :specific])
          (get-in data [:automaton :util])
          (get-in data [:automaton :grid]))
        {:init-state state}))
    (dom/div
      #js {:className "row"}
      (om/build grid-config-view
        (get-in data [:automaton :util])
        {:init-state
         {:change-grid-dimensions
          (:change-grid-dimensions state)
          :width (get-in data [:automaton :grid :width])
          :height
          (get-in data [:automaton :grid :height])}}))
    (dom/hr nil)
    (dom/div
      #js {:className "row"}
      (om/build display-config-view
        (get-in data [:automaton :util])
        {:init-state
         {:change-display-dimensions
          (:change-display-dimensions state)
          :width (get-in data [:automaton :display :width])
          :height
          (get-in data [:automaton :display :height])}})))
    (let [output-view-present?
          (not= (automaton-output-view customization)
                empty-view)]
      (dom/div
        nil
        (dom/div
          #js {:className "col-sm-2"
                :hidden (not output-view-present?)}
          (dom/div
            #js {:className "row"}
            (dom/h3 nil "Output information channel")
            (om/build (automaton-output-view customization) data)))
          (dom/div
            #js {:className (if output-view-present?
                              "col-sm-8"
                              "col-sm-10")}
            (dom/div
              #js {:className "row"}

```

```

      (om/build grid-view
        (:automaton data)
        {:init-state
         {:cell-state-changed (:cell-state-changed state)
          :default-cell
          (a/default-cell automaton-spec)}})))))))))

(declare unrestricted-language-parser-customization)

(defn unrestricted-language-parser-command-view
  [data owner]
  (reify
    om/IRenderState
    (render-state [_ state]
      (let [c (:command-info-channel state)
            a-wildcard (:a-wildcard data)
            b-wildcard (:b-wildcard data)

            wildcard-fn
            (fn [wildcard neighbour-states cell]
              (let [{:keys [top-left top-top-right left right
                           bottom-left bottom-bottom-right]}
                    neighbour-states]
                (if (and (#{:a :b} (:state cell)) (= :x left)
                        (= wildcard bottom))
                  {:state :x}
                  false)))
            a-wildcard-fn (partial wildcard-fn :a)
            b-wildcard-fn (partial wildcard-fn :b)]
        (dom/div
          nil
          (dom/label nil "a as a wildcard")
          (dom/input
            #js {:type "checkbox"
                  :checked a-wildcard
                  :onClick
                  (fn [] (om/transact! data [:a-wildcard] not))}))
          (dom/label nil "b as a wildcard")
          (dom/input
            #js {:type "checkbox"
                  :checked b-wildcard
                  :onClick
                  (fn [] (om/transact! data [:b-wildcard] not))}))
          (dom/button
            #js {:type "button"
                  :className "btn btn-success btn-lg btn-block"
                  :onClick
                  #(cond
                     (and b-wildcard (not a-wildcard))
                     (put! c {:command b-wildcard-fn})

                     (and a-wildcard (not b-wildcard))
                     (put! c {:command a-wildcard-fn})

                     (and a-wildcard b-wildcard)
                     (put! c {:command
                              (fn [neighbours s]
                                (if-let [a-res (a-wildcard-fn neighbours s)]
                                  a-res
                                  (b-wildcard-fn neighbours s))))))
                  :else (fn [_ _] false))})
            "Send command"))

```

```

      (dom/button
        #js {:type "button"
              :className "btn btn-info btn-lg btn-block"
              :onClick
                #(automaton-command-reset
                  unrestricted-language-parser-customization c)}
        "Reset command")
      (dom/hr nil))))))

(defn unrestricted-language-parser-output-view
  [data _]
  (reify
    om/IRender
    (render [_]
      (dom/div
        #js {:className "row"
              :hidden (not (#{:success :failure} (:result data))))}
        (dom/h2
          nil
          "Result: " (if (= :success (:result data))
                        "word is parsed"
                        "word is not parsed"))))))))

(def unrestricted-language-parser-customization
  (reify
    CellularAutomatonAppCustomization
    (automaton-specific-css [_ _] "")
    (automaton-configuration-view [_] empty-view)
    (automaton-command-view [this] unrestricted-language-parser-command-view)
    (automaton-command-initial-state [_] {})
    (automaton-command-reset [_ input-channel] (put! input-channel {}))
    (automaton-output-handler [_ data owner {:keys [s f]}]
      (when (or (> s 0) (> f 0))
        (om/transact!
          data (fn [d] (assoc d :result (if (> s 0) :success :failure))))
        (put! (om/get-state owner :started) false)))
    (automaton-output-view [_] unrestricted-language-parser-output-view)
    (automaton-output-reset [_ data _]
      (om/transact! data (fn [d] (assoc d :result :none))))))

(defn corps [n] (map a/corp-keyword (range 1 (inc n))))

(defn generate-corp-css
  [corp color]
  (str "." (keyword->str corp) " { background-color: hsl(" color "); }"))

(defn market-command-view
  [data owner]
  (reify
    om/IRenderState
    (render-state [_ {:keys [tax-rate income-tax-rate fixed-tax depreciation
                             a p command-info-channel]
                      :as state}]
      (dom/div
        #js {:role "form" :className "economic-model-control"}

        (dom/span
          nil
          (dom/label nil "Tax rate (%)")
          (dom/input
            #js {:type "text" :className "form-control" :value tax-rate
                  :onChange
                    #(handle-int-config-change % owner state :tax-rate))}))

```

```

(dom/span
  nil
  (dom/label nil "Income tax rate (%)")
  (dom/input
    #js {:type "text" :className "form-control" :value income-tax-rate
         :onChange
         #(handle-int-config-change % owner state :income-tax-rate)}))

(dom/span
  nil
  (dom/label nil "Fixed tax")
  (dom/input
    #js {:type "text" :className "form-control" :value fixed-tax
         :onChange
         #(handle-int-config-change % owner state :fixed-tax)}))

(dom/label nil "Base depreciation rate (%)")
(dom/input
  #js {:type "text" :className "form-control" :value depreciation
       :onChange
       #(handle-int-config-change % owner state :depreciation)}))

(dom/label nil "Utilty param a")
(dom/input
  #js {:type "text" :className "form-control" :value a
       :onChange #(handle-float-config-change % owner state :a)}))

(dom/label nil "Utilty param p")
(dom/input
  #js {:type "text" :className "form-control" :value p
       :onChange #(handle-float-config-change % owner state :p)}))

(dom/button
  #js {:type "button"
       :className "btn btn-success btn-lg btn-block"
       :onClick
       #(->> {:tax-rate
              (normalize (/ (js/parseFloat tax-rate) 100) 0 1.0)
              :income-tax-rate
              (normalize (/ (js/parseFloat income-tax-rate) 100) 0 1.0)
              :depreciation
              (normalize (/ (js/parseFloat depreciation) 100) 0 1.0)
              :fixed-tax (js/parseInt fixed-tax)}
              (filter (comp is-number? second))
              (merge
                {:utility-params (->> {:a (js/parseFloat a)
                                       :p (js/parseFloat p)}
                                       (filter (comp is-number? second))
                                       (into {})))})
              (assoc {:type :config} :cmd)
              (put! command-info-channel)))
       "Send command")

(dom/button
  #js {:type "button"
       :className "btn btn-info btn-lg btn-block"
       :onClick
       #(put! command-info-channel
              {:type :config
               :cmd a/market-model-default-params})}
       "Reset command")

(dom/hr nil))))

```

```

(defn gen-market-info-box
  [f path]
  (fn [data owner]
    (reify
      om/IRender
      (render [_]
        (let [corp (:corp data)
              corp-str (keyword->str corp)]
          (dom/div
            #js {:className corp-str}
            (apply str "Corporation " (get (.split corp-str "-") 1) ": ")
            (f (conj path corp))))))))))

(defn market-output-view
  [data owner]
  (let [get-market-info (fn [path] (get-in data (cons :market-state path)))]
    (reify
      om/IRender
      (render [_]
        (let [corps
              (map (fn [c] {:corp c})
                   (corps
                     (get-in data [:automaton :specific :corp-quantity] 4)))]
          (dom/div
            #js {:className "row"
                  :hidden (not (contains? data :market-state))}
            (dom/b nil "Capital")
            (dom/div
              nil
              "Government: " (get-market-info [:capital :government]))
            (apply
              dom/span nil
              (om/build-all
                (gen-market-info-box get-market-info [:capital]) corps))

            (dom/b nil "Market share")
            (dom/div
              #js {:className "without-good"}
              "Without good: "
              (num->percent
                (get-market-info [:global-user-share :without-good]))
            (apply
              dom/span nil
              (om/build-all
                (gen-market-info-box
                  (comp num->percent get-market-info)
                  [:global-user-share])
                corps))

            (dom/b nil "Good prices")
            (apply
              dom/span nil
              (om/build-all
                (gen-market-info-box get-market-info [:prices]) corps))

            (dom/b nil "Taxation types")
            (apply
              dom/span nil
              (om/build-all
                (gen-market-info-box (comp keyword->str get-market-info)
                  [:taxation-type]) corps))

            (dom/span

```

```

nil
(dom/b nil "Tax rate: ")
(dom/span
  nil
  (num->percent (get-market-info [:tax-rate])))
(dom/div nil))

(dom/span
  nil
  (dom/b nil "Income tax rate: ")
  (dom/span
    nil
    (num->percent (get-market-info [:income-tax-rate])))
  (dom/div nil))

(dom/span
  nil
  (dom/b nil "Fixed tax: ")
  (dom/span nil (get-market-info [:fixed-tax]))
  (dom/div nil))

(dom/b nil "Base depreciation rate: ")
(dom/span
  nil
  (num->percent (get-market-info [:depreciation])))
(dom/div nil)

(dom/b nil "Utility function params: ")
(dom/div
  nil
  (dom/span nil "a: " (get-market-info [:utility-params :a]))
  (dom/span
    nil " p: " (get-market-info [:utility-params :p]))))

(defn market-model-configuration-view
  [data owner]
  (reify
    om/IInitState
    (init-state
      [_]
      {:config-changed (chan)
       :year-period (:year-period data)
       :stop-after (:stop-after data)
       :corp-quantity (:corp-quantity data)})
    om/IWillMount
    (will-mount [_]
      (let [config-c (om/get-state owner :config-changed)
            reset-c (om/get-state owner :reset)]
        (go
          (while true
            (let [[corp-quantity year-period stop-after] (<! config-c)]
              (om/transact! data #(assoc %
                                           :corp-quantity corp-quantity
                                           :year-period year-period
                                           :stop-after stop-after))
              (put! reset-c true))))))
    om/IRenderState
    (render-state [_ {:keys [corp-quantity year-period config-changed
                             stop-after command-info-channel]
                      :as state}]
      (let [started (:started data)]
        (dom/div
          #js {:role "form" :className "automaton-economic-model-control"}
          (dom/label nil "Quantity of corporations"))

```

```

(dom/input
  #js {:type "text" :className "form-control" :value corp-quantity
       :disabled started
       :onChange
       #(handle-int-config-change % owner state :corp-quantity)})
(dom/label nil "Year period (num. of iterations)")
(dom/input
  #js {:type "text" :className "form-control" :value year-period
       :disabled started
       :onChange
       #(handle-int-config-change % owner state :year-period)})
(dom/label nil "Stop after iteration (0 - never)")
(dom/input
  #js {:type "text" :className "form-control" :value stop-after
       :disabled started
       :onChange
       #(handle-int-config-change % owner state :stop-after)})
(dom/button
  #js {:type "button" :className "btn btn-danger btn-lg btn-block"
       :disabled started
       :onClick #(do (put! config-changed
                           [corp-quantity year-period stop-after])
                     (put! command-info-channel
                           {:type :change-corp-quantity
                            :cmd corp-quantity}))})

  "Update")
(dom/hr nil))))))

(def market-model-customization
  (reify
    CellularAutomatonAppCustomization
    (automaton-specific-css [_ data]
      (let [corp-quantity
            (get-in data [:automaton :specific :corp-quantity] 4)
            corps (corps corp-quantity)]
        (->> (map generate-corp-css corps (random-hsl-colors corp-quantity))
              (interpose "\n")
              (apply str))))
    (automaton-configuration-view [_] market-model-configuration-view)
    (automaton-command-view [_] market-command-view)
    (automaton-command-initial-state [_]
      (let [params a/market-model-default-params]
        (merge (:utility-params params)
              {:tax-rate
               (apply str (butlast (num->percent (:tax-rate params))))
               :income-tax-rate
               (apply str (butlast (num->percent (:income-tax-rate params))))
               :fixed-tax (:fixed-tax params)
               :depreciation
               (apply str
                 (butlast (num->percent (:depreciation params)))))))))
    (automaton-command-reset [_ command-channel]
      (put! command-channel {:type :reset}))
    (automaton-output-handler [_ data owner msg]
      (let [corp-quantity
            (get-in @data [:automaton :specific :corp-quantity] 4)
            year-period
            (get-in @data [:automaton :specific :year-period] 12)
            stop-after
            (get-in @data [:automaton :specific :stop-after] 0)
            iteration
            (get-in @data [:automaton :util :iteration] 0)
            corps (corps corp-quantity)
            competitor-count
```

```

(count (filter (fn [[k v]] (>= v 0))
              (dissoc (:capital msg) :government)))
command-info-c (om/get-state owner :command-info-channel)
started-c (om/get-state owner :started)]
(if (and (not (= stop-after 0))
        (>= iteration stop-after))
    (put! started-c false)
    (do
      (when (zero? (mod iteration (/ year-period 2)))
        (doseq [corp corps]
          (let [cmd
                (update-in (a/conservative-corp-price
                          (cons competitor-count
                                ((juxt (:global-user-share msg)
                                       (:capital msg)
                                       (:capital-diff msg)
                                       (:prices msg))
                                   corp)))
                  [:cmd]
                  (fn [p] [corp p])))]
            (put! command-info-c cmd))))
      (when (zero? (mod iteration year-period))
        (doseq [corp corps]
          (let [cmd
                (update-in (a/conservative-corp-tax
                          (:tax-rate msg)
                          (:income-tax-rate msg)
                          (:fixed-tax msg)
                          (get-in msg [:capital-incomings corp])
                          (get-in msg [:capital-expenditures corp]))
                  [:cmd]
                  (fn [p] [corp p])))]
            (put! command-info-c cmd))))))
      (om/transact! data (fn [d] (assoc d :market-state msg))))))
(automaton-output-view [_] market-output-view)
(automaton-output-reset [_ data _]
  (om/transact! data (fn [d] (dissoc d :market-state))))))

```

```

(defn render-cellular-automaton
  ([view] (render-cellular-automaton view {}))
  ([view additional-data]
   (om/root
    view
    (assoc-in app-state [:automaton :specific] additional-data)
    {:target (. js/document (getElementById "app"))
     :init-state {:animation-step 500}})))

(defn menu-view
  [data owner]
  (reify
   om/IRender
   (render [_]
    (dom/div #js {:className "app-menu"}
      (dom/div #js {:className "page-header"}
        (dom/h1 nil "Cellular automata experiments"))
      (navigation-button
       "Game of Life"
       (partial render-cellular-automaton (gen-app-view a/game-of-life)))
      (navigation-button
       "Unrestricted language parser"
       (partial render-cellular-automaton
        (gen-app-view

```



```

        a/unrestricted-language-parser
        unrestricted-language-parser-customization)))
(navigation-button
 "Economic model"
 (partial render-cellular-automaton
  (gen-app-view
   a/market-model market-model-customization)
  {:corp-quantity 4
   :year-period 12
   :stop-after 0}))))))

(defn render-menu-view
  []
  (om/root
   menu-view
   app-state
   {:target (. js/document (getElementById "app"))}))

(render-menu-view)

```