УДК 004.415:004.424
DOI: 10.18523/2617-3808.2022.5.31-40

Гулаєва Н. М., Кобєлєв М. Д.

РЕАЛІЗАЦІЯ ЧАТ-БОТІВ ІЗ ПОВЕДІНКОЮ, ЩО КЕРУЄТЬСЯ СКІНЧЕННИМ АВТОМАТОМ, У ГРАФІЧНОМУ КОНСТРУКТОРІ

Окреслено основні методи використання скінченних автоматів (СА) для моделювання й програмної реалізації поведінки об'єктів різної природи. Як приклад інтерактивної програми розглянуто чат-боти: основні поняття, методи класифікації, способи реалізації. Запропоновано розширення СА для моделювання поведінки текстового чат-бота, побудованого на правилах. Дано короткий огляд методів перетворення СА на програмний код у різних парадигмах програмування. Запропоновано динамічний підхід для виконання такого перетворення, зі збереженням специфікації СА в базі даних. Розроблено графічний конструктор, який дає змогу створювати модель чат-бота, базованого на правилах, у вигляді скінченного автомата та виконує трансформацію побудованої моделі для безпосереднього використання в месенджер-платформі Telegram.

Ключові слова: скінченний автомат, СА, model driven development, генерація кода, чат-бот, Telegram месенджер, мікросервісна архітектура, документна база даних, резидентна база даних.

Вступ

Природний, вичерпний і зрозумілий опис поведінкових аспектів програмної системи є обов'язковим на всіх етапах циклу її розроблення. Чітке розмежування потоку даних і потоку управління даними дає змогу підвищити надійність розроблюваної системи. Провести жорстку структуризацію опису логіки поведінки системи, чітко розмежувати опис логіки та опис семантики можна шляхом використання моделі скінченного автомата (СА) [6] під час проєктування та розробки програмного застосунку. В результаті також підвищується надійність розроблюваного програмного забезпечення та покращується його документування [24; 41; 50]. Зазначимо, що використання СА для проєктування та розроблення програмного забезпечення є реалізацією популярного підходу MDD (Model Driven Development) – розробки, керованої моделлю [39].

Скінченні автомати як метод проєктування та розроблення програмних застосунків

Розробленню загальних методів створення програм, поведінка яких описується СА, присвячено роботи різних авторів. Зокрема, Девід Гарел у [25] запропонував метод використання СА як візуального формалізму для специфікації поведінкових аспектів складних реактивних систем. При цьому модель СА розширено (виключні та ортогональні стани, вкладені стани, суперстани, дії при вході в стан та при виході з нього тощо), а метод названо методом *діаграм станів* (statecharts method). Метод отримав подальший розвиток: об'єктно-орієнтованим варіантом діаграм станів Девіда Гарела є *машини станів UML* (UML State Machines) [32]. У [41] ідеться про метод розроблення будь-яких програм на основі розширеної моделі СА (вкладені та викличні автомати, групи станів, переходи з пріоритетами); цей метод названо *автоматним програмуванням*, або *програмуванням з явним виділенням станів*. У [50] показано, як за допомогою СА або системи СА будувати опис надійного програмного застосунку; при цьому для СА введено кілька типів виходів (дій): на вході в стан та на виході з нього, при введенні даних та на переходах. Запропонований у [50] метод названо *віртуальним скінченним автоматом* (Virtual Finite State Machine); існує реалізація моделі виконання СА StateWORKS [44]. У [37] обговорюється планове, централізоване управління станів у контексті мобільних застосунків.

Згідно з класифікацією Девіда Гарела [24], будь-яку програмну систему можна віднести до одного з таких класів:

– *трансформаційні* системи, які виконують перетворення вхідних даних і повертають результат цього перетворення (наприклад, компілятори, архіватори);

– *інтерактивні* системи, які виконують операції введення/виведення в режимі діалогу з користувачем (наприклад, текстові редактори, чат-боти);
– *реактивні* системи, які підтримують безперервний взаємозв'язок із середовищем, реагуючи певним чином на зовнішні подразники (наприклад, системи контролю та управління фізичними пристроями).

Традиційною сферою використання СА є компілятори [6], хоча існує багато прикладів використання СА в задачах всіх трьох типів [1–3; 16; 19; 24; 25; 31; 33; 37; 41; 50; 53]. Разом з тим, на думку авторів, використанню СА в інтерактивних системах все ще не приділяють достатньо уваги, хоча використання СА для реалізації таких систем є природним.

Поняття та класифікація чат-ботів

Прикладом інтерактивної системи є чат-бот – комп'ютерна програма, з якою користувачі можуть вести діалог природною мовою в голосовому або текстовому форматі. Першим чат-ботом в історії вважають ELIZA – програму, розроблену 1966 року, завданням якої було імітувати діалог із психотерапевтом [51]. Сьогодні чат-боти широко використовують як віртуальні помічники, що вирішують типові завдання: відповідають на прості запитання користувачів, поширюють інформацію (рекламу, повідомлення служб), шукають інформацію за запитом, дають персональні рекомендації й консультують щодо вибору товару, збирають відгуки відвідувачів, виконують прості доручення, такі як бронювання, інвентаризація, відслідковування замовлень тощо. Чат-боти є популярними помічниками бізнесу, оскільки допомагають знизити витрати на обслуговування клієнтів та здатні швидко обслуговувати кілька користувачів одночасно.

Існує багато способів класифікації чат-ботів, виділимо такі [5; 27]:
– за способом комунікації: *текстові* (спілкуються зі співрозмовником через текстові повідомлення) та *голосові* (здатні розпізнавати та відтворювати мовлення й генерувати голосові відповіді);
– за методами оброблення вхідної інформації та породження відповіді: чат-боти, що побудовані *на основі правил* (rule-based), вибирають відповідь, застосовуючи фіксовану наперед визначену множину правил; чат-боти, що побудовані *на основі пошуку* (retrieval-based), є гнучкішими, обирають відповідь із множини можливих варіантів; чат-боти, що побудовані *на основі породження* (generative-based), генерують відповідь з урахуванням історії спілкування, для розробки таких чат-ботів застосовують методи машинного навчання;
– за обсягом знань (knowledge domain) або за метою (goal) виділяють два типи чат-ботів. До першого типу належать чат-боти, які розробляють для використання за конкретними сценаріями (бронювання готелю/квитка/житла, замовлення продукту, планування події, довідник тощо). Оскільки ці чат-боти фокусуються на певній галузі знань, їх називають *орієнтованими на завдання* (task-oriented), *закритого домену* (closed domain) та *вертикальними* (vertical) чат-ботами. Іноді проводять додаткову класифікацію цих чат-ботів за функціональністю (боти для продажів, лід-боти, транзакційні чат-боти, інформатори, асистенти тощо). До другого типу відносять чат-боти, розроблені для симуляції розмови, часто з розважальною метою. Оскільки такі чат-боти розробляють із можливістю відповідати на будь-які запитання, їх називають *не орієнтованими на завдання* (non-task oriented), *відкритого домену* (open domain), *горизонтальними* (horizontal), а також *розмовними* (conversational) чат-ботами;
– за платформою впровадження: чат-боти можуть функціонувати в месенджерах (Telegram, Slack, Viber, WhatsApp), соціальних мережах (Facebook), на сайтах, у мобільних застосунках, в системах управління тощо.

Надалі розглядатимемо тільки текстові, орієнтовані на завдання чат-боти, які побудовані на основі правил і призначені для вирішення конкретного завдання довідкового характеру в месенджер-платформі Telegram. Зазначимо, що поведінка чат-ботів на основі правил природно моделюється СА. Особливістю чат-ботів, призначених для використання в месенджер-платформах, є те, що месенджер-платформа повністю відповідає за зовнішній вигляд чату, а чат-боти виглядають як звичайні користувачі цієї платформи.

Існує два способи розроблення чат-ботів: у пропонованих сучасними платформами редакторах або шляхом написання власного програмного коду з використанням API, за допомогою якого можна управляти чат-ботом та отримувати повідомлення від користувачів [5; 21]. Перевагами написання власного програмного коду є гнучкість (немає обмежень щодо технологій розроблення або типу чат-бота), актуальність (можна використовувати найновіші можливості

наданого для відповідної платформи впровадження API), безпека (немає потреби передавати токени доступу до API третім особам). Водночас, для написання власного коду потрібні спеціальні знання та навички (щодо розроблення програмного коду, розгортання застосунку, налаштування інфраструктури тощо).

На жаль, не всі платформи, що допомагають легко створювати чат-бот користувачеві-непрограмісту, підтримують упровадження розробленого чат-бота в Telegram-месенджер. Розглянемо такі можливості створення чат-ботів Telegram без програмування. Manybot [30] – безкоштовний чат-бот Telegram: простий в опануванні, втім, має обмежену кількість сценаріїв використання (розсилка повідомлень підписникам). Потужнішими інструментами створення чат-ботів є BotSailor [10] та Sendpulse [40]. Ці інструменти дають можливість розробляти широкий спектр сценаріїв використання чат-бота (відстеження користувачів, статистика, власні функції Telegram тощо). Однак буває не завжди зручно дотримуватися запропонованих цими засобами підходів до моделювання поведінки користувачів. До того ж, зазначені інструменти є платними, а безкоштовні версії мають обмеження щодо кількості чат-ботів, підписників, повідомлень на місяць тощо. Ще однією альтернативою написанню програмного коду є використання NoCodeAPI, який дає змогу автоматизувати відповіді чат-бота на повідомлення користувачів. Утім, за такого підходу бракує стану користувача, що обмежує кількість сценаріїв використання чат-бота.

Враховуючи сказане, ми розробили візуальний конструктор текстового орієнтованого на завдання чат-бота, поведінка якого описується СА. Цей конструктор надає зручний інтерфейс для побудови СА – моделі поведінки чат-бота – та генерує код для безпосереднього використання побудованого чат-бота в месенджер-платформі Telegram.

Моделювання поведінки чат-бота за допомогою детермінованого СА

СА є зручним інструментом моделювання поведінки програми (зокрема й чат-бота) через його здатність по-різному реагувати на події: реакція включно з іншими факторами залежить від ланцюжка попередніх подій, який фактично кодується в поточному стані автомата.

Нагадаємо [6], що детермінований СА можна подати як список компонентів (S, s_{start}, F, I, O, δ, λ), де S – скінченна множина станів, $s_{start} \in S$ – початковий стан, $F \subset S$ – множина фінальних станів, I – скінченний вхідний алфавіт, O – скінченний вихідний алфавіт, $\delta : S \times I \to S$ – функція переходів, $\lambda : S \times I \to O$ – функція виходів. Для моделювання поведінки чат-бота, створюваного користувачем конструктора чат-ботів, визначимо ці компоненти так.

Початковий стан s_{start} – режим чекання, в якому перебувають усі чат-боти (екземпляри) до контакту з користувачем: антиспам-політика Telegram забороняє чат-ботам першими писати користувачам. Множина фінальних станів $F = \varnothing$: активований користувачем чат-бот залишається активним; у разі блокування чат-бота користувачем чат-бот перебуває в режимі чекання, доки користувач не продовжить листування. Множина станів $S' = S \backslash s_{start}$ визначається розробником чат-бота.

Визначення вхідного та вихідного алфавітів параметризуємо для надання максимальної гнучкості при створенні чат-ботів. Вхідний алфавіт I містить множину кодів, які може розпізнавати чат-бот. Ці коди описують ланцюжки символів, що надсилає користувач чат-бота, їх називають тригерами [10; 40]. Множина кодів містить множину команд (позначатимемо символом c) і множину текстових повідомлень (позначатимемо символом w). Наразі всі команди мають починатись символом «/», а текстові повідомлення описуються регулярними виразами. Наприклад, $I = \{c_{start}, w_{world\$}, w_{any}\}$, де c_{start} – команда «/start», $w_{world\$}$ – всі текстові повідомлення, що закінчуються «world», w_{any} – будь-які текстові повідомлення.

Вихідний алфавіт O описує множину дій чат-бота; вони задаються кодами дій та супровідною інформацією. Наразі кодами дій є: «do nothing» – не виконувати жодної дії; «send message» – надіслати текстове повідомлення користувачеві; «make API request» – зробити запит на сторонній сервіс; «save user data» – зберегти інформацію про користувача у базі даних (БД). Текстові повідомлення, які чат-бот надсилає користувачеві, можуть бути визначені заздалегідь (сталі текстові повідомлення) або формуватись динамічно з використанням шаблонних змінних – послідовностей символів, які будуть замінені певним значенням перед відправленням повідомлення користувачеві. В розробленому нами конструкторі шаблонні змінні задаються парою {*DOMAIN.variable*}. DOMAIN вказує на джерело значень змінної: «UPD» (значення слід взяти з повідомлення, що надійшло через Telegram) або «DB» (значення слід взяти з БД, що заповнюється виконанням дій із кодом «save

user data»). Variable задає назву шаблонної змінної (наразі передбачені шаблонні змінні «message_date» – дата та час відправлення повідомлення, «message_text» – текст повідомлення, «message_from_first_name» – ім'я користувача чат-бота, «message_from_language_code» – мова інтерфейсу користувача чат-бота та інші). Наприклад, у разі визначення відповіді чат-бота «You wrote: {*DOMAIN.variable*}», користувач, що надіслав повідомлення «Hello», отримає відповідь чат-бота «You wrote: Hello».

Для забезпечення можливості виконання кількох дій у відповідь на дію одного тригера (в термінах класичної теорії СА – читання одного символу з вхідного ланцюжка), функцію виходів визначимо так: $\lambda : S \times I \rightarrow O^n$. Реалізацію функції переходів і функції виходів визначає розробник чат-бота.

За наведеного визначення компонент СА ехо-бот, задачею якого є дублювання повідомлень користувача, може бути заданий так: $S = \{s_{start}, s_{echo}\}$, $I = \{c_{start}, w_{any}\}$, $O \supseteq$ {«do nothing», «send message»}, а функції переходів та виходів задамо за допомогою діаграми станів СА, де вхідні повідомлення (тригери) зображено на дузі, а вихідні – над/під дугою (рис. 1). Чат-бот активується командою користувача «/start», після чого дублює всі повідомлення користувача, які не є командами.

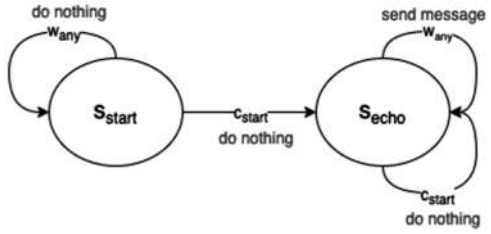

Рис. 1. Діаграма станів ехо-бота

Користувачеві конструктора чат-ботів – розробникові свого чат-бота – може знадобитись можливість змінювати функції переходів і виходів, не видаляючи попередні версії цих функцій. Для забезпечення можливості збереження історії змін функцій переходів і виходів розширимо СА неробочими (disabled) переходами; такі переходи задаватимемо відношеннями (а не функціями) переходів і виходів та порядковим номером їх створення. Зазначимо, що для конкретного стану, вхідного сигналу та коду дії тільки один перехід має бути робочим (в іншому разі отримаємо суперечливу множину умов переходів). У разі видалення робочого переходу поточним робочим переходом автоматично призначається неробочий перехід із найбільшим порядковим номером. Розширення моделі СА неробочими переходами є близьким до використання переходів із пріоритетами з [41].

Методи програмної реалізації СА

Для тестування та експлуатації змодельованого за допомогою СА чат-бота необхідно виконати перетворення СА на програмний код. Іншими словами, необхідно розробити транслятор СА.

Існує багато робіт, присвячених опису методів реалізації СА в процедурній (структурній) [38; 41; 48], функціональній [45] та особливо в об'єктно-орієнтованій [4; 8; 17; 26; 28; 34; 38; 42; 48] парадигмі. Значну увагу до методів генерації коду в об'єктно-орієнтованій парадигмі пояснюють популярністю об'єктно-орієнтованого підходу й тим, що багато понять із різних розширень моделі СА, які активно використовують, не підтримуються безпосередньо популярними об'єктно-орієнтованими мовами програмування [4; 17; 28; 42]. Запропоновані методи різняться ступенем підтримки елементів специфікації СА (елементів із розширень СА), підходами до проєктування класів, а також іншими важливими характеристиками розробки програмного забезпечення, такими як модульність, розширюваність, витрати пам'яті (використання дискового простору та оперативної пам'яті в процесі виконання), легкість розуміння та підтримки, ефективність, гнучкість тощо. В систематичному огляді [17] на основі аналізу 53 джерел зроблено висновок, що значна частина наявних методів реалізації СА в об'єктно-орієнтованій парадигмі ані реалізує повною мірою деякі поняття з моделі СА, ані надає стратегії реалізації, яка б враховувала релевантні якісні аспекти розроблення програмного забезпечення. В [42] також наголошується на тому, що більшість відомих методів мають серйозні недоліки: їх важко зрозуміти та підтримувати, вони мають невисоку продуктивність, залежать від властивостей певної мови програмування й не реалізують деяких понять із популярних розширень СА.

Звернемо увагу на існування низки бібліотек, утиліт, фреймворків, які виконують автоматичну генерацію коду за побудованою моделлю (специфікацією СА) [7; 9; 11–13; 15; 18; 22; 26; 28; 29; 34–36; 43; 44; 46; 47; 49; 52]. Хоча використання таких засобів може знизити часові витрати на розроблення, є зауваження стосовно повноти, семантики та ефективності згенерованого ними коду [34]. На жаль, немає оглядових робіт, де б проводилось ґрунтовне порівняння належної кількості різних пропонованих сьо-

годні автоматизованих засобів генерування коду за моделлю СА. Аналіз наявних праць дає змогу зробити висновок, що більшість таких засобів не підтримують усі необхідні поняття з популярних розширень СА [8; 34], генерують складний громіздкий код (що ускладнює його читання та аналіз, отже, ставить під сумнів можливість зробити адекватні висновки щодо структури коду, згенерованого машиною) [8; 17], не мають належної гнучкості або не дотримуються певних правил проєктування (наприклад, важливо, щоб код був подібним для подібних СА) [8; 28], згенерований код не оптимізований у частині швидкості оброблення подій [34]. Іноді згенерований код залежить від бібліотек, що надаються постачальником відповідної утиліти, і цей факт впливає на переносимість коду [34]. До того ж, часто цільовою мовою програмування генерується лише частина коду (код структури автомата), а поведінку об'єкта (оброблення сигналів) програміст має прописувати окремо [17]. Також не завжди виконується перевірка коректності побудованого СА.

Враховуючи сказане, а також наявність запропонованих нами розширень СА, було прийнято рішення писати власну реалізацію генератора коду за побудованою користувачем моделлю СА.

Програмна реалізація СА – моделі чат-бота

Найпоширенішим способом реалізації СА в процедурній парадигмі (а часто і в програмах, написаних в об'єктно-орієнтованому стилі) є інструкція вибору [50]. Цей спосіб полягає у використанні двох вкладених операторів switch – один для визначення стану автомата, інший для визначення входу. Такий підхід характеризується високою швидкодією. Недоліками підходу є дублювання коду, розмір якого може бути досить великим (що є критичним, наприклад, для вбудованих машин, в яких обсяг пам'яті та процесора обмежений), а також складність читання та супроводу [17; 26; 48].

Для реалізації СА в об'єктно-орієнтованій парадигмі найчастіше використовують шаблон State [23] або його модифікації. Поведінка СА варіюється залежно від його стану; шаблон State реалізує таку поведінку за допомогою поліморфізму. Ідея полягає у введенні абстрактного класу для представлення різних станів автомата та в реалізації специфічної для кожного стану поведінки в підкласах. Слабка підтримка реалізації певних аспектів (розширення класичної моделі СА, необхідність створення додаткового екземпляра класу СА для кожного клієнта, робота з пам'яттю) спонукала розробників до побудови різноманітних модифікацій зазначеного шаблону [17; 48]. Зазначимо, що один чат-бот може обслуговувати багатьох клієнтів одночасно, отже, вимога до ефективного використання пам'яті є критичною.

Обидва описані підходи є статичними: за їх використання складно проводити динамічну модифікацію СА, а легкість модифікації СА є однією з основних вимог до конструктора чат-ботів. Динамічний підхід до реалізації СА передбачає використання у пам'яті програми таблиць (масивів) для збереження функцій переходів і виходів [14]. Цей підхід є гнучким (таблиці можна змінювати під час виконання програми, отже, можливою є динамічна модифікація структури автомата), втім, гнучкість досягається за рахунок погіршення продуктивності. В літературі обговорюють й інші недоліки зазначеного підходу (експоненційне зростання розміру таблиці зі збільшенням кількості вхідних змінних; можливість використовувати тільки числові значення як входи автомата; складність додавання дій при переході між станами; подання логіки переходів у однорідному табличному форматі, що робить умови складнішими для розуміння) та пропонують модифікації [17; 23; 50].

Для забезпечення належної гнучкості (СА часто модифікується в конструкторі) та швидкості трансляції побудованої користувачем діаграми станів СА у виконуваний код (користувач чат-бота має отримувати відповіді без затримок) було прийнято таке архітектурне рішення.

Структура СА, створюваного розробником чат-бота, зберігається в документі формату JSON у документній БД MongoDB. Документні БД, на відміну від реляційних, є не повністю структурованими БД (semi-structured), що дає змогу легко модифікувати структуру документа. В задачі реалізації чат-бота такий підхід є доречним через можливу наявність різної кількості дій на різних переходах, а також необхідність зберігати історію модифікацій функцій переходів і виходів. Ключами в JSON документах, що зберігають структуру СА, є ідентифікатори станів СА, а значеннями – піддокументи, що описують поведінку СА в цьому стані залежно від отриманого входу. Піддокументи з ключем «cmd_triggers» містять перелік тригерів-команд, а піддокументи з ключем «msg_triggers» – перелік тригерів-повідомлень. Для кожного тригера (команди або повідомлення) зберігається масив піддокументів з описом дій чат-бота та позначкою «disabled» за необхідності. Функція перехо-

дів СА задається кодом дії «change_state», функція виходів – кодами «do nothing», «send message», «make API request», «save user data». Наприклад, подану на рис. 1 діаграму станів ехо-бота буде збережено в документі JSON так:

```
{
 «init»: {
  «cmd_triggers»: {
   «/start»: [
    { «type»: «change_state», «options»: {«state»:
«echo-state» },
    { «type»: «send_message», «options»: { «text»:
«Hello, I am echo-bot» }}
   ]
  },
  «msg_triggers»: {},
 },
 «echo-state»: {
  «cmd_triggers»: {},
  «msg_triggers»: {
   «.*»: [
    { «type»: «send_message», «options»: { «text»:
«{{ MESSAGE_TEXT }}» }}
   ]
  }
 }
}
```

Збереження структури СА в БД MongoDB є зручним для реалізації процесу проєктування чат-бота в конструкторі чат-ботів. Водночас, отримання з MongoDB інформації про структуру чат-бота під час спілкування з ним у месенджері Telegram не забезпечуватиме належної швидкості експлуатації чат-бота. Для мінімізації часу відповіді чат-бота на повідомлення користувача інформація про СА дублюється в резидентній (in-memory) БД Redis, що є базою «ключ-значення»; в збережених документах ключем є ідентифікатор чат-бота, а значенням – серіалізований JSON документ зі структурою СА. Також у БД Redis у структурі даних HASH зберігаються стани чат-бота для всіх його користувачів: ключ – ідентифікатор чат-бота, поле – ідентифікатор користувача, значення поля – стан чат-бота для цього користувача. Зауважимо, що для запобігання втраті інформації у разі перебоїв у роботі резидентної БД слід увімкнути відповідні налаштування (persistence) БД Redis.

Транслятор (інтерпретатор) СА реалізований як мікросервіс мовою Go. Мікросервіс реагує на подію «update» від Telegram: вибирає із БД Redis структуру СА відповідного чат-бота та стан поточного користувача, класифікує отримане від користувача повідомлення (команда або текстове повідомлення), шукає необхідний тригер і послідовно виконує всі передбачені для нього дії (без позначки «disabled»). Описаний підхід забезпечує високу швидкість відповіді чат-бота, необхідну гнучкість (оновлена структура автомата вибирається з БД), легкість читання (серіалізований формат збереження даних) та є ефективним щодо пам'яті, причому завдання вчасного оновлення та перевикористання даних (із кешу) вирішується засобами системи керування БД. Також збереження специфікації СА у форматі JSON практично унеможливлює ін'єкції коду, отже, підвищує безпеку застосунку. Фрагмент коду оброблення події подано нижче.

```
func runMessage(ctx *ExecutionContext) error {
  msg := ctx.Upd.Message.Text
  state := ctx.User.State
  keys := ctx.Bot.States[state].MsgTriggers
  for pattern, triggers := range keys {
    match, _ := regexp.MatchString(pattern, msg)
    if match {
      for _, trigger := range triggers {
        err := runAction(&trigger, ctx)
        if err != nil {
          return err
        }
      }
      break
    }
  }
  return nil
}
```

Загальна архітектура конструктора чат-ботів

Конструктор чат-ботів має мікросервісну архітектуру; основні компоненти та їх взаємодію подано на рис. 2. Сервіс front-end відповідає за інтерфейс (UI) користувача – розробника чат-бота, відображення та збереження модельованого чат-бота в документній БД. Сервіс bot-execution відповідає за оброблення подій від Telegram і за виконання відповідних дій СА; структура СА зчитується з резидентної «ключ-значення» БД. Сервіс bot-management виконує конвертацію моделі чат-бота з документної БД у формат БД «ключ-значення», реагуючи на події оновлення інформації у документній БД.

MDD є методологією розроблення програмного забезпечення, що ґрунтується на понятті моделі та її перетворення. СА та його розширення є широко використовуваним засобом моделювання поведінки об'єктів. Поведінка простого чат-бота природно моделюється СА. Розроблений візуальний конструктор дає змогу будувати модель чат-бота й одразу використовувати цей чат-бот у месенджер-платформі Telegram. Саме такий підхід до розроблення програмного забез-

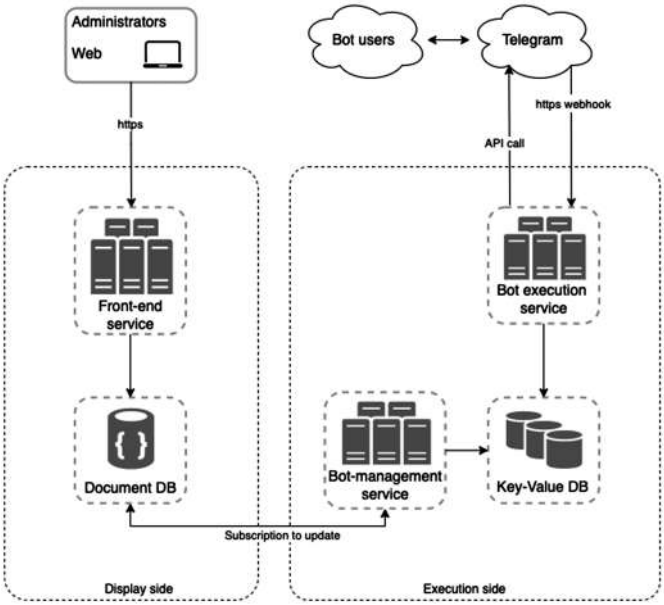

Рис. 2. Діаграма архітектури конструктора чат-ботів

печення – побудова моделі та її трансформація у виконуваний код – пропагується методологією MDD [39].

Висновки

Для генерації коду за моделлю СА вибрано динамічний підхід, причому структура чат-бота зчитується з резидентної БД. Такий підхід є гнучким, ефективним щодо часу й пам'яті, а структура автомата, що зчитується з БД, є зрозумілою для аналізу людиною.

Для збереження історії змін чат-бота СА було розширено неробочими переходами, також дозволяється кілька дій на одному переході. Коректність розроблюваного користувачем СА частково перевіряється на етапі моделювання, а саме: будь-який перехід має вести з одного стану в інший; для кожної вхідної події та коду дії СА умова тільки одного робочого переходу є істиною (несуперечливість); у будь-який стан автомата має вести шлях з початкового стану (досяжність).

Подальшу роботу заплановано вести за такими напрямами:

– удосконалення етапу генерації виконуваного коду (зокрема, використання спеціальних утиліт для роботи з регулярними виразами);
– ускладнення моделі СА шляхом розширення вхідного алфавіту – зокрема, додавання можливості відправляти повідомлення за розкладом (окремий сервіс-планувальник генеруватиме події, на які реагуватиме автомат);
– моделювання еволюції СА для вибору найефективнішого способу подання інформації (зауважимо, що еволюцію СА вперше розглянуто в [20]; підхід отримав назву «еволюційне програмування»).

Список літератури

1. Гулаєва Н. М. Автоматний підхід до організації підтримки мережевих камер у багатоклієнтських системах / Н. М. Гулаєва // Тез. доп. II Міжнар. конф. «TAAPSD'2005». – 2005. – С. 82–86.
2. Круковский М. Ю. Автоматно-графовая формальная модель композитного документооборота / М. Ю. Круковский // Математические машины и системы. – 2006. – № 2. – С. 87–95.
3. Коротунов С. Ю. Аналіз підходів до моделювання та верифікації кіберфізичних систем / С. Ю. Коротунов, Г. В. Табунщик // Радіоелектроніка, інформатика, управління. – 2020. – № 3. – С. 57–68. https://doi.org/10.15588/1607-3274-2020-3-5
4. Aabidi M. H. A New Approach for Code Generation from UML State Machine / M. H. Aabidi, A. Jakimi, E. H. Kinani, M. Elkoutbi // International Review on Computers and Software. – 2013. – No. 8. – Pp. 500–506.
5. Adamopoulou E. An Overview of Chatbot Technology / E. Adamopoulou, L. Moussiades // IFIP International Conference on Artificial Intelligence Applications and Innovations. – 2020. – Pp. 373–383. https://doi.org/10.1007/978-3-030-49186-4_31
6. Aho A. V. Compilers: Principles, Techniques, and Tools / A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. 2nd Edition. – Boston : Pearson/Addison Wesley, 2007. – 952 p.
7. ArgoUML [Electronic resource]. – Mode of access: https://argouml-tigris-org.github.io/.

8. Badreddin O. B. Enhanced code generation from UML composite state machines / O. B. Badreddin, T. C. Lethbridge, A. Forward, M. Elaasar, H. I. Aljamaan, M. Garzón // 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014). – 2014. – Pp. 235–245.
9. Boost Meta State Machine [Electronic resource]. – Mode of access: https://www.boost.org/doc/libs/1_79_0/libs/msm/doc/HTML/index.html.
10. BotSailor [Electronic resource]. – Mode of access: https://botsailor.com/.
11. BOUML [Electronic resource]. – Mode of access: https://www.bouml.fr.
12. BridgePoint/xtUML [Electronic resource]. – Mode of access: https://xtuml.org/.
13. C and C++ Finite State Machine Framework [Electronic resource]. – Mode of access: https://www.block-net.de/Programmierung/cpp/fsm/fsm.html.
14. Cargill T. C++ Programming Style / T. Cargill. – Reading, Mass. : Addison-Wesley, 1992. – 248 p.
15. Concurrent Hierarchical State Machine (CHSM) [Electronic resource]. – Mode of access: http://chsm.sourceforge.net/.
16. David R. Lane changing behavior recognition based on Artificial Neural Network-based State Machine approach / R. David, S. Rothe, D. Söffker // 2021 IEEE International Intelligent Transportation Systems Conference (ITSC). – 2021. – Pp. 3444–3449. https://doi.org/10.1109/ITSC48978.2021.9564919
17. Domínguez E. A systematic review of code generation proposals from state machine specifications / E. Domínguez, B. Pérez, A. Rubio, M. Zapata // Information and Software Technology. – 2012. – Vol. 54, No. 10. – Pp. 1045–1066. https://doi.org/10.1016/j.infsof.2012.04.008
18. Enterprise Architect [Electronic resource]. – Mode of access: https://www.sparxsystems.com.au/.
19. Fauzi R. Defense behavior of real time strategy games: Comparison between HFSM and FSM / R. Fauzi, M. Hariadi, S. Nugroho, M. Lubis // Indonesian Journal of Electrical Engineering and Computer Science. – 2019. – No. 13. – Pp. 634–642. https://doi.org/10.11591/ijeecs.v13.i2.pp634-642
20. Fogel L. J. Artificial Intelligence through Simulated Evolution / L. J. Fogel, A. J. Owens, M. J. Walsh. – New York : J. Wiley&Sons, 1966. – 184 p.
21. Følstad A. Future directions for chatbot research: an interdisciplinary research agenda / A. Følstad, T. Araujo, E. Law et al. // Computing. – 2021. – No. 103. – Pp. 2915–2942. https://doi.org/10.1007/s00607-021-01016-7
22. FSMGenerator [Electronic resource]. – Mode of access: http://fsmgenerator.sourceforge.net/.
23. Gamma E. Design Patterns: elements of reusable object-oriented software / E. Gamma, R. Helm, R. Johnson, J. Vlissides. – Westford : Addison-Wesley, 1994. – 417 p.
24. Harel D. On the development of reactive systems / D. Harel, A. Pnueli // Logic and Models of Concurrent Systems. – 1985. – No. 13. – Pp. 477–498. https://doi.org/10.1007/978-3-642-82453-1_17
25. Harel D. Statecharts: A visual formalism for complex systems / D. Harel // Scientific Computer Programming. – 1987. – Vol. 8, No. 3. – Pp. 231–274. https://doi.org/10.1016/0167-6423(87)90035-9
26. Hung M. A Generalized FSM Implementation Framework / M. Hung // Advances in Smart Vehicular Technology, Transportation, Communication and Applications. – 2019. – Pp. 465–470. https://doi.org/10.1007/978-3-030-04582-1_54
27. Hussain S. A Survey on Conversational Agents/Chatbots Classification and Design Techniques / S. Hussain, O. Ameri Sianaki, N. Ababneh // Workshop on Web, Artificial Intelligence and Network Applications (WAINA-2019). – 2019. – Pp. 946–956. https://doi.org/10.1007/978-3-030-15035-8_93
28. Lopes S. F. An easy-to-use and flexible object-oriented framework for extended finite state machines / S. F. Lopes, S. Silva, J. L. Monteiro // IEEE 10th International Conference on Industrial Informatics. – 2012. – Pp. 35–40. https://doi.org/10.1109/INDIN.2012.6301360
29. Machina.js [Electronic resource]. – Mode of access: http://machina-js.org/.
30. Manybot [Electronic resource]. – Mode of access: https://manybot.io/.
31. Miao H. Modeling Web Browser Interactions Using FSM / H. Miao, Z. Qian, T. He // 2nd IEEE Asia-Pacific Service Computing Conference (APSCC 2007). – 2007. – Pp. 211–217. https://doi.org/10.1109/APSCC.2007.13
32. OMG, Unified Modeling Language (OMG UML) Ver. 2.5.1 Specification [Electronic resource]. – Mode of access: https://www.omg.org/spec/UML/2.5.1/PDF.
33. Park H. Item Measurement for Logistics-Oriented Belt Conveyor Systems Using a Scenario-Driven Approach and Automata-Based Control Design / H. Park, A. Van Messem, W. De Neve // 2020 IEEE 7th International Conference on Industrial Engineering and Applications (ICIEA). – 2020. – Pp. 271–280. https://doi.org/10.1109/ICIEA49774.2020.9102044
34. Pham V. C. Complete Code Generation from UML State Machine / V. C. Pham, A. Radermacher, S. Gérard, S. Li // 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017). – 2017. – Pp. 208–219. https://doi.org/10.5220/0006274502080219
35. Ricca F. Using UniMod for Maintenance Tasks: An Experimental Assessment in the Context of Model Driven Development / F. Ricca, M. Leotta, G. Reggio et al. // 2012 4th International Workshop on Modeling in Software Engineering, MiSE. – 2012. – Pp. 77–83. https://doi.org/10.1109/MISE.2012.6226018
36. Rosmaro [Electronic resource]. – Mode of access: https://rosmaro.js.org/.
37. Salmre I. Chapter 5. Our Friend, the State Machine / I. Salmre // Writing Mobile Code: Essential Software Engineering for Building Mobile Applications. – Addison-Wesley, 2005. – 771 p.
38. Samek M. Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems, 2nd Edition / M. Samek. – CRC Press, 2008. – 728 p. https://doi.org/10.1201/b16463
39. Selic B. The pragmatics of model-driven development / B. Selic // IEEE Software. – 2003. – Vol. 20, No. 5. – Pp. 19–25.
40. Sendpulse [Electronic resource]. – Mode of access: https://sendpulse.com/ru/features/chatbot.
41. Shalyto A. A. SWITCH Technology: An Automated Approach to Developing Software for Reactive Systems / A. A. Shalyto, N. I. Tukkel' // Programming and Computer Software. – 2001. – No. 27. – Pp. 260–276. https://doi.org/10.1023/A:1012392927006
42. Spinke V. An object-oriented implementation of concurrent and hierarchical state machines / V. Spinke // Information and Software Technology. – 2013. – Vol. 55, No. 10. – Pp. 1726–1740. https://doi.org/10.1016/j.infsof.2013.03.005
43. Stateflow [Electronic resource]. – Mode of access: https://www.mathworks.com/products/stateflow.html.
44. StateWORKS [Electronic resource]. – Mode of access: http://www.stateworks.com/.
45. Thompson S. Regular Expressions and Automata using Haskell: technical report [Electronic resource] / S. Thompson. – University of Kent, 2000. – Mode of access: https://kar.kent.ac.uk/22057/.
46. Umodel [Electronic resource]. – Mode of access: https://www.altova.com/umodel.
47. UMPLE [Electronic resource]. – Mode of access: https://cruise.umple.org/umpleonline/.
48. van Gurp J. On the implementation of finite state machines / J. van Gurp, J. Bosch // Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications. – 1999. – Pp. 172–178.

49. Visual Paradigm [Electronic resource]. – Mode of access: https://www.visual-paradigm.com/.
50. Wagner F. Modeling Software with Finite State Machines: A Practical Approach / F. Wagner, R. Schmuki, T. Wagner, P. Wolstenholme. – New York : Auerbach Publications CRC Press, 2006. – 392 p. https://doi.org/10.1201/9781420013641
51. Weizenbaum J. ELIZA – a computer program for the study of natural language communication between man and machine / J. Weizenbaum // Communications of the ACM. – 1966. – Vol. 9, No. 1. – Pp. 36–45. https://doi.org/10.1145/365153.365168
52. YAKINDU Statechart Tools [Electronic resource]. – Mode of access: https://www.itemis.com/en/yakindu/state-machine/.
53. Zuzak I. A Finite-State Machine Approach for Modeling and Analyzing RESTful Systems / I. Zuzak, I. Budiselic, G. Delac // Journal of Web Engineering (JWE). – 2011. – Vol. 10, No. 4. – Pp. 353–390.

References

Aabidi, M. H., Jakimi, A., Kinani, E. H., & Elkoutbi, M. (2013). A New Approach for Code Generation from UML State Machine. International Review on Computers and Software, 8, 500–506.

Adamopoulou, E., & Moussiades, L. (2020). An Overview of Chatbot Technology. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*, 373–383. https://doi.org/10.1007/978-3-030-49186-4_31

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, Techniques, and Tools* (2nd Edition). Pearson/ Addison Wesley.

ArgoUML. https://argouml-tigris-org.github.io/.

Badreddin, O. B., Lethbridge, T. C., Forward, A., Elaasar, M., Aljamaan, H. I., & Garzón, M. (2014). Enhanced code generation from UML composite state machines. In 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014), 235–245.

Boost Meta State Machine. https://www.boost.org/doc/libs/1_79_0/libs/msm/doc/HTML/index.html.

BotSailor. https://botsailor.com/.

BOUML. https://www.bouml.fr.

BridgePoint/xtUML. https://xtuml.org/.

C and C++ Finite State Machine Framework. https://www.blocknet.de/Programmierung/cpp/fsm/fsm.html.

Cargill, T. (1992). *C++ Programming Style*. Addison-Wesley.

Concurrent Hierarchical State Machine (CHSM). http://chsm.sourceforge.net/.

David, R., Rothe, S., & Söffker, D. (2021). Lane changing behavior recognition based on Artificial Neural Network-based State Machine approach. In *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, 3444–3449. https://doi.org/10.1109/ITSC48978.2021.9564919

Domínguez, E., Pérez, B., Rubio, A., & Zapata, M. (2012). A systematic review of code generation proposals from state machine specifications. *Information and Software Technology*, *54* (10), 1045–1066. https://doi.org/10.1016/j.infsof.2012.04.008

Enterprise Architect. https://www.sparxsystems.com.au/.

Fauzi, R., Hariadi, M., Nugroho, S., & Lubis, M. (2019). Defense behavior of real time strategy games: Comparison between HFSM and FSM. *Indonesian Journal of Electrical Engineering and Computer Science*, *13*, 634–642. https://doi.org/10.11591/ijeecs.v13.i2.pp634-642

Fogel, L. J., Owens, A. J., & Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution*. J. Wiley&Sons.

Følstad, A., Araujo, T., Law, E., Brandtzaeg, P., Papadopoulos, S., Reis, L., ..., Luger, E. (2021). Future directions for chatbot research: an interdisciplinary research agenda. *Computing*, *103*, 2915–2942. https://doi.org/10.1007/s00607-021-01016-7

FSMGenerator. http://fsmgenerator.sourceforge.net/.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley.

Gulayeva, N. (2005). Avtomatnyi pidkhid do orhanizatsii pidtrymky merezhevykh kamer u bahatokliientskykh systemakh. In *Tez. dop. II Mizhnar. konf. TAAPSD2005*, 82–86 [in Ukrainian].

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Scientific Computer Programming*, *8* (3), 231–274. https://doi.org/10.1016/0167-6423(87)90035-9

Harel, D., & Pnueli, A. (1985). On the development of reactive systems. *Logics and Models of Concurrent Systems*, *13*, 477–498. https://doi.org/10.1007/978-3-642-82453-1_17

Hung, M. (2019). A Generalized FSM Implementation Framework. Advances in Smart Vehicular Technology, Transportation, Communication and Applications, 465–470. https://doi.org/10.1007/978-3-030-04582-1_54

Hussain, S., Ameri Sianaki, O., & Ababneh, N. (2019). A Survey on Conversational Agents/Chatbots Classification and Design Techniques. *Workshop on Web, Artificial Intelligence and Network Applications (WAINA-2019)*, 946–956. https://doi.org/10.1007/978-3-030-15035-8_93

Korotunov, S., &. Tabunshchyk, H. (2020). Analiz pidkhodiv do modeliuvannia ta veryfikatsii kiberfizychnykh system. *Radioelektronika, informatyka, upravlinnia*, *3*, 57–68 [in Ukrainian]. https://doi.org/10.15588/1607-3274-2020-3-5

Krukovskij, M. (2006). Avtomatno-grafovaja formal'naja model' kompozitnogo dokumentooborota. *Matematicheskie mashiny i sistemy, 2,* 87–95 [in Russian].

Lopes, S. F., Silva, S., & Monteiro, J. L. (2012). An easy-to-use and flexible object-oriented framework for extended finite state machines. In *IEEE 10th International Conference on Industrial Informatics, 35–40*. https://doi.org/10.1109/INDIN.2012.6301360

Machina.js. http://machina-js.org/.

Manybot. https://manybot.io/.

Miao, H., Qian, Z., & He, T. (2007). Modeling Web Browser Interactions Using FSM. In *2nd IEEE Asia-Pacific Service Computing Conference (APSCC 2007)*, 211–217. https://doi.org/10.1109/APSCC.2007.13

OMG, Unified Modeling Language (OMG UML) Ver. 2.5.1 Specification. https://www.omg.org/spec/UML/2.5.1/PDF.

Park, H., Van Messem, A., & De Neve, W. (2020). Item Measurement for Logistics-Oriented Belt Conveyor Systems Using a Scenario-Driven Approach and Automata-Based Control Design. In 2020 IEEE 7th International Conference on Industrial Engineering and Applications (ICIEA), 271–280. https://doi.org/10.1109/ICIEA49774.2020.9102044

Pham, V. C., Radermacher, A., Gérard, S., & Li, S. (2017). Complete Code Generation from UML State Machine. In *5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, 208–219. https://doi.org/10.5220/0006274502080219

Ricca, F., Leotta, M., Reggio, G., Tiso, A., Guerrini, G., & Torchiano, M. (2012). Using UniMod for Maintenance Tasks: An Experimental Assessment in the Context of Model Driven Development. In *2012 4th International Workshop on Modeling in Software Engineering, MiSE*, 77–83. https://doi.org/10.1109/MISE.2012.6226018

Rosmaro. https://rosmaro.js.org/.

Salmre, I. (2005). Chapter 5. Our Friend, the State Machine. In *Writing Mobile Code: Essential Software Engineering for Building Mobile Applications*. Addison-Wesley.

Samek, M. (2008). *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems, 2nd Edition*. CRC Press. https://doi.org/10.1201/b16463

Selic, B. (2003). The pragmatics of model-driven development. *IEEE Software, 20* (5), 19–25.

Sendpulse. https://sendpulse.com/ru/features/chatbot.

Shalyto, A. A., & Tukkel', N. I. (2001). SWITCH Technology: An Automated Approach to Developing Software for Reactive Systems. *Programming and Computer Software*, *27*, 260–276. https://doi.org/10.1023/A:1012392927006

Spinke, V. (2013). An object-oriented implementation of concurrent and hierarchical state machines. *Information and Software Technology*, *55*. https://doi.org/10.1016/j.infsof.2013.03.005

Stateflow. https://www.mathworks.com/products/stateflow.html.

StateWORKS. http://www.stateworks.com/.

Thompson, S. (2000). Regular Expressions and Automata using Haskell (Technical report). University of Kent. https://kar.kent.ac.uk/22057/.

UModel. https://www.altova.com/umodel.

UMPLE. https://cruise.umple.org/umpleonline/.

van Gurp, J., & Bosch, J. (1999). On the implementation of finite state machines. In *Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications*, 172–178.

Visual Paradigm. https://www.visual-paradigm.com/.

Wagner, F., Schmuki, R., Wagner, T., & Wolstenholme, P. (2006). *Modeling Software with Finite State Machines: A Practical Approach*. New York: Auerbach Publications CRC Press. https://doi.org/10.1201/9781420013641

Weizenbaum, J. (1966). ELIZA – a computer program for the study of natural language communication between man and machine. *Communications of the ACM, 9* (1), 36–45. https://doi.org/10.1145/365153.365168

YAKINDU Statechart Tools. https://www.itemis.com/en/yakindu/state-machine/.

Zuzak, I., Budiselic, I., & Delac, G. (2011). A Finite-State Machine Approach for Modeling and Analyzing RESTful Systems. *Journal of Web Engineering (JWE)*, *10* (4), 353–390.

N. Gulayeva, M. Kobieliev

IMPLEMENTATION OF FSM BASED CHAT-BOTS IN A GRAPHICAL DESIGNER

Finite state machine (FSM) is a powerful tool to model object behavior. Using FSM and its extensions to model program behavior followed by the automatic generation of executable code is the approach encouraged by the model-driven development (MDD) – a software development methodology based on the concepts of model and model transformation.

In this paper, a brief overview of FSM-based common methods to model and develop software programs of any nature is given. These methods include David Harel's statecharts, UML State Machines, Virtual Finite State Machine, etc. Examples of all types of software systems (transformational, interactive, reactive) implemented using FSM are cited.

Chat-bots as an example of an interactive software system are considered: concept, classification methods, implementation techniques. A graphical designer of rule-based chat-bots to be integrated in the messenger Telegram is developed and implemented. In this designer, chat-bot behavior is modeled using FSM.

Formal method to model a rule-based chat-bot using FSM is provided. The FSM concept is extended by disabled transitions to save history of transition changes made during the FSM design process. A brief overview of code generation methods from FSM specification is done; advantages and disadvantages of the most popular approaches are considered. Dynamic approach to generate code by FSM specification saved in DB is proposed. To implement this approach, document MongoDB and in-memory key-value Redis DB are used; FSM is kept as a JSON-document. This approach is efficient in flexibility, speed and memory needs.

Architecture diagram of developed chat-bot graphical designer is given. It has the microservice architecture. The FSM model-to-code transformation is carried out by the bot-execution service written using compiled language Go. Other services include the front-end (UI for end-user, CRUD API for chat-bot) and the bot-management (synchronization of document and key-value databases) services.

Keywords: finite state machine, FSM, model driven development, code generation, chat-bot, Telegram messenger, microservice architecture, document database, in-memory database.

Матеріал надійшов 14.08.2022