УДК 004.41:004.05 https://doi.org/10.15407/pp2026.01.051

Ю.В. Єрофеєв, І.П. Сініцин

ПРОБЛЕМИ ЕФЕКТИВНОГО ТЕСТУВАННЯ СПЕЦІАЛЬНИХ ПРОГРАМНИХ ЗАСОБІВ ВБУДОВАНИХ СИСТЕМ ТА ЇХ ОПРАЦЮВАННЯ

Розглянуто проблему результативного й ресурсно ефективного тестування спеціальних програмних засобів (СПЗ) вбудованих систем (ВС) в умовах істотної різнорідності й швидкої еволюції апаратних платформ, мінливості конфігурацій і скорочення циклів еволюції СПЗ. Показано, що для сучасних вбудованих виробів характерні часткова незалежність життєвих циклів апаратної та програмної складових і зростання ролі керованої модернізованості СПЗ ВС. Обґрунтовано потребу в таких умовах не лише виявлення дефектів, а й додаткових функцій тестування: прогнозування (не)задовільної якості СПЗ і ВС загалом, підтримку рішень щодо інтеграції СПЗ і накопичення повторно використовуваних тестових активів. Узагальнено особливості типових архітектур ВС і підходів до конструювання СПЗ ВС, значущі для забезпечення вищезазначених додаткових функцій тестування, зокрема, рамкову та мікросервісну архітектури, розроблення, кероване тестами, гнучкі методології, розроблення на основі моделей, а також підходи безперервної інтеграції й безперервного тестування. Показано роль XiL-підходів, керування конфігураціями та контролювання зв'язків між артефактами для забезпечення відтворюваності середовища тестування. Запропоновано розглядати стале повторне використання робочих продуктів тестування в руслі парадигми лінійок програмних продуктів як основу для сталого підвищення зрілості процесів тестування СПЗ ВС та їх конструювання загалом.

Ключові слова: вбудована система, спеціальні програмні засоби, тестування, повторне використання, лінійка програмних продуктів, безперервна інтеграція, безперервне тестування, життєвий цикл, програмна архітектура, керування конфігурацією.

Yu.V. Yerofieiev, I.P. Sinitsyn

THE PROBLEMS OF EMBEDDED SYSTEMS SPECIAL SOFTWARE EFFECTIVE TESTING AND THEIR ELABORATION

The paper examines the problem of sustainable testing of special software for embedded systems under conditions of rapid hardware evolution, configuration variability, and shortened update cycles. It shows that modern embedded products are characterized by a partial separation of hardware and software life cycles, a growing role of controlled firmware updates, component reuse, and the integration of computer vision and artificial intelligence algorithms. It is substantiated that, in such conditions, testing should not be limited to defect detection, but should also support quality forecasting, integration decisions, and the accumulation of reusable testing assets. The study summarizes typical embedded-system architectures and approaches to constructing special software, including layered and microservice-oriented architectures, test-driven development, agile methods, model-based development, and continuous integration, continuous testing, quality assurance, and security practices. The role of XiL approaches, configuration management, and artifact traceability in ensuring environment reproducibility is highlighted. Sustainable reuse of testing work products within the software product line paradigm is proposed as a basis for increasing the maturity of development and testing processes for special embedded software.

Keywords: embedded system, special software, testing, reuse, software product line, continuous integration, continuous testing, life cycle, software architecture, configuration management.

©Ю.В. Єрофеєв, І.П. Сініцин, 2026

CC BY 4.0

Вступ

У сучасних вбудованих системах (ВС) спеціальні програмні засоби (СПЗ) є одним із найскладніших, найуразливіших і водночас найбільш критичних складників [1, 2]. На відміну від програмних засобів загального призначення, СПЗ ВС функціонують в умовах жорстких ресурсних обмежень, вимог реального часу або близьких до нього, обмежень енергоспоживання, а також тісної залежності від апаратної платформи й фізичного оточення через взаємодію з датчиками та виконавчими механізмами. У зв'язку з цим дефекти СПЗ можуть спричиняти не лише функціональні збої, а й відмови ВС загалом, порушення вимог безпеки, загрози й втрати для довкілля та життя й здоров'я людини.

За цих умов тестування СПЗ ВС має, крім традиційного виявлення дефектів СПЗ, виконувати ще й додаткові функції:

а) надавати адекватні дані для (не)кількісного прогнозування якості СПЗ і ухвалення рішень щодо їх інтеграції з операційною системою, процесором, апаратними засобами;

б) підтримувати прогнозування вірогідних технічних та організаційних проблем у життєвому циклі (ЖЦ) ВС;

в) забезпечувати накопичення тестових артефактів і допоміжних даних для постійного вдосконалення процесів тестування й конструювання СПЗ в умовах обмежених ресурсів.

Разом функції а)-в) зумовлюють зростання актуальності проблеми сталого повторного використання робочих продуктів тестування в ЖЦ СПЗ і ВС. загалом. Особливо важливою вона є для організацій-розробників, що створюють ВС різних типів у різних предметних областях і не обмежуються однією апаратною та програмною платформою. Саме розв'язання цієї проблеми створює підґрунтя для узгодженого досягнення вищезазначених цілей тестування та зрештою підвищення зрілості процесу конструювання ВС.

Метою статті є окреслення підходу до сталого повторного використання робочих продуктів тестування СПЗ ВС у рамках парадигми лінійок програмних продуктів (Software Product Lines), зумовленого особливостями ЖЦ сучасних ВС: різнорідністю апаратно-програмних платформ, стислими термінами, обмеженими обчислювальними ресурсами.

1. Особливості СПЗ ВС, значущі для їх тестування

Типові архітектури ВС і СПЗ ВС. Детальний аналіз сучасних ВС у різних предметних областях [2] висвітлює три узагальнені типи їхньої архітектури: рамкову (reference), із застосуванням шару абстракції обладнання та мікросервісну.

В історично першій *рамковій* архітектурі ВС (рис. 1) СПЗ безпосередньо взаємодіє з драйверами пристроїв та операційною системою, які в свою чергу взаємодіють з апаратними засобами [2, 3].

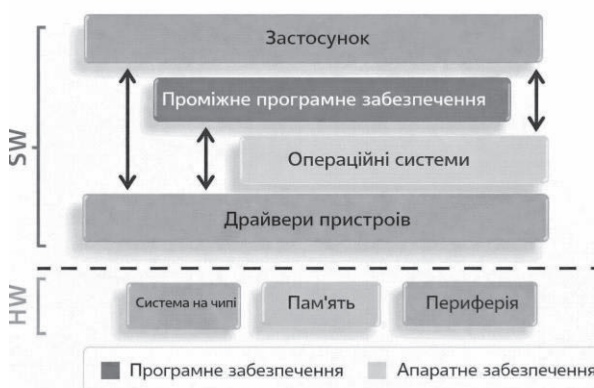

Рис. 1. Рамкова архітектура ВС

Подальший розвиток апаратних засобів зумовив поширення прошарованої архітектури [2] для СПЗ ВС, що являють собою монолітну збірку (так звану прошивку (firmware)). Ці СПЗ подано переліком програмних шарів. Кожний шар має принаймні один інтерфейс щодо суміжного шару, визначений його типом. Інтерфейси надають узгоджений доступ до функцій СПЗ, приховують внутрішні деталі їх реалізації та можуть виступати як обгортки для інтеграції несумісного коду між шарами [1].

Найпоширенішим і водночас найпрактичнішим прошарком прошивки є шар *абстракції обладнання* (hardware abstraction level) [1]. По суті це прикладний програмний інтерфейс для взаємодії з облад-

нанням, який замінює доступи на рівні апаратних засобів викликами функцій вищого рівня (рис. 2).

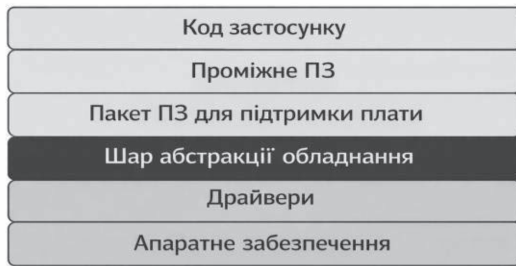

Рис. 2. Архітектура ВС з використанням шару абстракції обладнання

Саме він має забезпечити переносність і повторне використання СПЗ ВС та його позитивні ефекти: зниження вартості конструювання СПЗ і ВС загалом; підвищення рівня абстракції; зменшення кількості помилок завдяки багаторазовій експлуатації; спрощення масштабування, зокрема, перенесення на нові апаратні засоби й ВС у межах одного сімейства.

Нарешті для опрацювання обмежень розглянутих архітектур ВС протягом останньої декади набула активного розвитку мікросервісна архітектура (рис. 3).

Рис. 3. Мікросервісна архітектура ВС на базі вбудованої операційної системи

Мікросервісна архітектура подає користувацький застосунок множиною взаємодіючих автономних мікросервісів, зорієнтованих на його прикладні функції. Типова реалізація мікросервісу у ВС відповідає окремому процесу або задачі/набору задач ОС із власними ресурсами та механізмами ізоляції, якщо вони доступні. Концептуально мікросервіс містить код сервісу, вхідну чергу повідомлень, механізми надсилання вихідних повідомлень, логування та індикатори стану для моніторингу працездатності.

Мікросервісна архітектура є дефакто стандартною для гнучких методологій розроблення СПЗ, практик DevSecOps і конвеєра безперервної інтеграції й доставки (CI/CD). Низький рівень зв'язності сервісів полегшує й пришвидшує процес тестування, спрощує масштабування та заміну сервісів без істотного порушення роботи всього СПЗ.

Однак, водночас із окресленими перевагами, ця архітектура підвищує складність проєктування та створює накладні витрати на комунікації та пам'ять; децентралізований обмін повідомленнями може ускладнювати досягнення детермінованої поведінки в реальному часі та збільшувати час відгуку. Повна незалежність розгортання сервісів не завжди має місце в ресурсно-обмежених ВС, особливо за відсутності розвиненого механізму керування або в спрощених операційних - середовищах.

Тому застосування мікросервісів у ВС потребує попереднього аналізу вимог, обмежень платформи та декомпозиції функцій СПЗ ВС.

Зовнішнє ділове середовище конструювання СПЗ ВС. Однією з визначальних особливостей зовнішнього ділового середовища конструювання ВС є зростання впродовж останнього десятиліття ролі Інтернету речей – класу розподілених ВС, здатних збирати й передавати дані мережею без безпосередньої участі - людини.

Ця тенденція зумовлює істотну різнорідність методів і технологій конструювання СПЗ ВС, які зручно розподілити за двома форматами:

а) «традиційним», згідно з яким СПЗ ВС розробляють із вузькоспеціалізованою функціональною метою для певної апаратної платформи й визначеного сценарію застосування;

б) «диверсифікованим», який передбачає конструювання низки мінливих СПЗ ВС (і, можливо, самих ВС як послугу повного циклу) для різних споживачів та/або предметних областей із використанням подібних між собою, але мінливих, апаратних платформ (рис. 4).

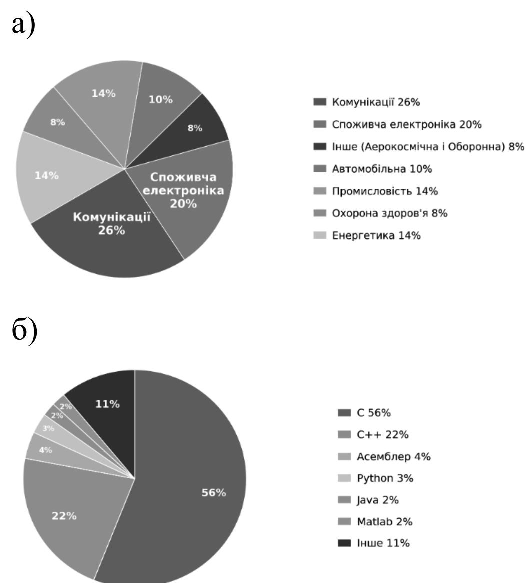

Рис. 4. Розподіл ВС: а) за предметними областями; б) за мовами програмування

Такі СПЗ не лише допускають певну повторюваність операцій конструювання як на низькому (драйвери, первинні завантажувачі, bootloaders) так і на вищих рівнях (модулі й бібліотеки), а й потребують її для підвищення якості та зрілості процесів конструювання. Основними мовами програмування для конструювання СПЗ ВС в обох форматах є C та C++. Однак розвиток сучасних архітектур СПЗ ВС, насамперед мікросервісної, зумовлює зростання частки інших МП, зокрема, Python і Java (рис. 4).

Внутрішнє ділове середовище конструювання СПЗ ВС. Для врахування описаних вище особливостей зовнішнього ділового середовища компанії-розробники активно вдосконалюють власні процеси та інструменти для забезпечення конкурентоспроможної якості СПЗ ВС.

«Традиційний» формат а), за якого СПЗ конструюють у де-факто спільному життєвому циклі (ЖЦ) з пристроєм, поступово вичерпує потенціал [1, 3]: він ускладнює швидке нарощування функцій, адаптацію до нових умов застосування та підтримку ВС упродовж тривалого строку експлуатації.

Ключовою ідеєю новітніх підходів, властивих «диверсифікованому» формату б), є організація відносно незалежних ЖЦ для СПЗ та апаратних засобів. Апаратна платформа еволюціонує за власним циклом (ревізії, компонентна база). Натомість СПЗ конструюють як постійно оновлюваний продукт: із можливістю регулярних випусків, підтримкою сумісності й контрольованими змінами без повного реінжинірингу СПЗ і тим паче ВС під кожне вдосконалення [1, 2].

Зокрема, у перспективній сфері автономних наземних комплексів вдосконалення процесів конструювання відбувається одночасно двома «траєкторіями» – апаратній і програмній, із різними темпами та обмеженнями.

Вирішальним чинником реалізації такого підходу стає гнучкість архітектури СПЗ і здатність до керованого оновлення як СПЗ загалом («прошивки»), так і окремих компонентів СПЗ – бібліотек, сервісів, модулів оброблення даних. Це дозволяє прийнятно швидко покращувати функції та якісні характеристики СПЗ, зберігаючи контроль над версіями, залежностями та конфігураціями в умовах мінливості платформ і сценаріїв використання.

Вдосконалювальні зміни стосуються двох взаємопов'язаних аспектів: організаційного (процесного) та інфраструктурного (технологічного).

До інфраструктурних тенденцій належать: удосконалення тестових стендів, зокрема, із підтримкою віддаленого доступу; розроблення та впровадження засобів автоматизації тестування СПЗ ВС у конфігураціях XiL (X-in-the-Loop), включно з MiL (Model-in-the-Loop), SiL (Software-in-the-Loop), PiL (Processor-in-the-Loop) та HiL (Hardware-in-the-Loop) [4], а також відповідних моделей середовища; наскрізна автоматизація тестів на різних рівнях; керування конфігураціями й артефактами повторного використання. Організаційно ключовими є перехід до безперервної інтеграції та регресійного тестування

(Continuous Integration (CI)), а також адаптація гнучких методів розроблення (Agile) до специфіки СПЗ ВС та обмежень апаратної платформи.

Доцільно розрізняти такі XiL-конфігурації:

– MiL (Model-in-the-Loop) – верифікація алгоритмів на рівні моделей;

– SiL (Software-in-the-Loop) – тестування виконуваного коду в симульованому середовищі;

– PiL (Processor-in-the-Loop) – виконання коду на цільовому процесорі з модельованими периферійними умовами;

– HiL (Hardware-in-the-Loop) – інтеграційне тестування із реальним апаратним компонентом у контурі.

Фактично конструювання СПЗ ВС перебуває в стані технологічної конкуренції, у межах якої постійно з'являються нові пристрої та конфігурації, а самі СПЗ безперервно оновлюються й ускладнюються (зокрема, бібліотеками й компонентами для реалізації алгоритмів комп'ютерного зору й штучного інтелекту). У цьому контексті дедалі більш запитаним стає підхід QAOps [5, 6] – сукупність організаційно-інфраструктурних практик, зорієнтована на спільну відповідальність за якість (Dev/QA/Ops), запровадження керованих критеріїв допуску змін (quality gates), забезпечення відтворюваності середовищ і накопичення повторно використовуваних тестових активів (зокрема, в межах лінійок програмних продуктів).

Актуальні особливості процесів ЖЦ СПЗ ВС і ВС. У таблиці 1 співставлено особливості процесів технічного керування та технічних процесів, що безпосередньо впливають на вимоги, архітектуру, інтеграцію та верифікацію/валідацію СПЗ у ЖЦ ВС, згідно з ДСТУ ISO/IEC/IEEE 12207, 1528.

Таблиця 1.

Особливості процесів ЖЦ СПЗ ВС

Процес ЖЦ	Об'єкт процесу		Особливість СПЗ ВС
	СПЗ	ВС	
Процеси технічного керування			
Керування ризиками (Risk Management)	Ризики системи: технічні, виробничі, експлуатаційні	Ризики ПЗ: дефекти, вразливості, технічний борг	Додаються ризики: обмежені ресурси, реальний час, залежність від платформи, кібербезпека, обмежений час доступності компонентів
Керування конфігурацією (Configuration Management)	Конфігурація виробу та варіантів: апаратна ревізія, BOM, інтерфейси	Конфігурація коду, збірок і релізів: версії, гілки, артефакти	Єдина ідентифікація конфігурації. Перелік (відомість) матеріалів програмного забезпечення. Специфікація складу апаратних компонентів
Управління інформацією (Information Management)	Артефакти системи: вимоги, архітектура, інтерфейси, протоколи випробувань	Артефакти ПЗ: код, збірки, тести, звіти	Простежуваність (traceability) артефактів від вимог до тестів і випусків/релізів
Забезпечення якості (Quality Assurance)	Забезпечення якості системи: відповідність вимогам та критеріям приймання	Забезпечення якості ПЗ: стандарти кодування, метрики, контроль дефектів	Атрибути якості: затримка, енергоспоживання, температурні режими, відмовостійкість, функціональна безпека та кібербезпека
Технічні процеси			
Визначення потреб і вимог заінтересованих сторін (Stakeholder Needs & Requirements)	Потреби користувачів/місії на рівні виробу	Потреби до ПЗ як складника системи	Уточнення потреб до автономності, сенсорів, зв'язку, експлуатаційних умов і обмежень

Визначення системних і програмних вимог (System/Software Requirements Definition)	Системні вимоги: функціональні та нефункціональні	Вимоги до ПЗ: функції, інтерфейси, обмеження	Інтерфейси з периферією, протоколи, обмеження пам'яті/CPU, вимоги до оновлення/відновлення, захищений завантажувач (secure boot)
Визначення архітектури (Architecture Definition)	Архітектура системи та розподіл функцій між підсистемами	Архітектура ПЗ: компоненти, інтерфейси, взаємодія	Код-дизайн апаратної і програмної частини , ізоляція безпечні межі взаємодії
Визначення проєктного рішення (Design Definition)	Проектування компонентів системи	Проектування модулів ПЗ	Проектування драйверів, станів і режимів
Системний аналіз (System Analysis)	Аналіз альтернатив і компромісів системного рівня	Аналіз рішень на рівні ПЗ	Компроміси: точність vs швидкодія, енергоспоживання vs продуктивність, пам'ять vs функціональність
Реалізація (Implementation)	Реалізація компонентів системи	Реалізація ПЗ, збірка та інтеграція залежностей	Крос-компіляція, оптимізації під платформу, захист ключів і секретів
Інтеграція (Integration)	Інтеграція на рівні системи	Інтеграція компонентів ПЗ	Робочий макет апартаного забезпечення, драйвери, інтеграція модулів комп'ютерного зору та AI-моделей; сумісність із ревізіями HW
Верифікація (Verification)	Перевірка відповідності вимогам	Верифікація ПЗ: огляди, аналіз, тестування	SiL/HiL, ін'єкція відмов (fault injection), fuzzing , аналіз покриття та часових характеристик
Перехід (Transition)	Передача в експлуатацію та/або виробництво	Реліз ПЗ та розгортання	Підписані образи, provisioning, OTA; перевірка у впливових умовах (температура/вібрації/зв'язок) і сценаріях відмов
Валідація (Validation)	Підтвердження придатності системи для потреб у цільовому середовищі	Валідація ПЗ в контексті системи	Польові випробування, acceptance-сценарії, оцінювання автономних функцій у репрезентативних умовах
Експлуатація (Operation)	Експлуатація системи	Експлуатація ПЗ	Телеметрія, логування, віддалена діагностика, контроль ресурсів і конфігурації
Супровід (Maintenance)	Супровід системи.	Супровід ПЗ.	Регулярні оновлення, підтримка та керування сумісністю варіантів.
Виведення з експлуатації (Disposal)	Виведення системи з експлуатації	Виведення ПЗ/даних з експлуатації	Безпечне стирання даних і ключів, deprovisioning; вимоги компласнсу (compliance)

Таблиця 1 надає процесне підґрунтя для проєктування стратегії тестування саме СПЗ ВС, визначення артефактів і меж відповідальності між рівнями СПЗ і ВС та ідентифікації специфічних для СПЗ ВС чинників – обмежених ресурсів, режиму реального часу, довготривалого супроводу, постійного оновлення та вимог - кібербезпеки.

Еволюція моделей ЖЦ СПЗ ВС, особливості якої підсумовано в табл. 2, зумовлена зростанням складності програмно-апаратної інтеграції, підвищенням вимог до швидкості внесення змін, а також необхідністю забезпечення простежуваності, надійності й безпеки на всіх рівнях розроблення. Якщо традиційнуV-модель зорієнтовано на послідовне проходження

етапів специфікації, проєктування, реалізації, верифікації та валідації, то подальший розвиток підходів пов'язаний із переходом до більш гнучких ітеративних схем, які скорочують цикл зворотного зв'язку між розробленням, інтеграцією та тестуванням.

Для сфери конструювання ВС така трансформація має особливе значення, оскільки програмні компоненти розробляють у тісному зв'язку з апаратним середовищем, обмеженнями цільової платформи та вимогами до функціональної безпеки й кібербезпеки.

Розвиток V-моделі до гнучких метолологій (Agile) і далі до Embedded DevSecOps відображає перехід від переважно послідовного та документно-орієнтованого підходу до безперервного, автоматизованого й ризик-орієнтованого керування ЖЦ ВС і СПЗ ВС. Якщо V-модель забезпечує високу формалізацію та простежуваність, то Agile підвищує адаптивність до змін, а Embedded DevSecOps поєднує гнучкість із безперервною інтеграцією, тестуванням, контролем безпеки та відповідності вимогам.

Таблиця 2.

Еволюція моделей ЖЦ СПЗ ВС

Модель	Ключова особливість	Вимоги	Інтеграція	Тестування СПЗ ВС	Безпека
V-модель	Послідовний, фазовий підхід	Фіксуються на початку	Переважно пізня	За рівнями верифікації та валідації	Окремий напрям робіт
Agile	Ітеративний та інкрементальний підхід	Уточнюються впродовж розроблення	Регулярна, поетапна	У межах ітерацій, з автоматизацією	Враховується в процесі
Embedded DevSecTestOps	Безперервний, автоматизований підхід	Динамічне керування і простежуваність	Безперервна (CI/CD/CT)	Безперервне, автоматизоване, багаторівневе	Інтегрована в повний цикл розробки

2. Актуальні методології конструювання СПЗ ВС

Одним із підходів до конструювання СПЗ ВС є адаптація принципів об'єктно-орієнтованого програмування до процедурної мови C. У цьому випадку інкапсуляція, композиція, обмежений поліморфізм та інші ООП-конструкції реалізують через структури даних, вказівники на функції, явні інтерфейси та усталені шаблони виклику. Такий підхід застосовують, коли необхідно забезпечити модульну організацію коду, повторне використання компонентів і формалізовану взаємодію між ними без переходу до повноцінного C++. У конструюванні СПЗ ВС це пов'язано з вимогами до їхнього функціонування й сумісності з ресурсними обмеженнями цільових платформ.

Подальший розвиток методологій конструювання СПЗ ВС пов'язаний із перенесенням акценту з побудови структури коду на керування процесом його поетапного формування. У [3] підхід Test-Driven Development (TDD) визначено як методологію, де реалізацію функціональності здійснюють через послідовність формування тестів, написання мінімально необхідного коду та подальший рефакторинг. За такого підходу тестування розглядають не як завершальний етап перевіряння, а як складову процесу конструювання програмної архітектури. Для ВС це означає можливість раннього виявлення дефектів, зниження складності налагодження та підвищення керованості змін у програмних компонентах.

У контексті СПЗ ВС TDD адаптують до апаратно-залежного середовища за

рахунок подвійно таргетованого тестування (dual-target testing), ізоляції залежностей від апаратної частини й операційної системи, а також використання тестових двійників (test doubles), зокрема, підробок (fakes), шпигунів (spies), імітацій (mocks). Це дає змогу перевіряти логіку модулів поза цільовою платформою, виконувати поетапне нарощування функціональності та підтримувати рефакторинг без безпосередньої залежності від наявності апаратного середовища. Таким чином, TDD у структурі сучасних методологій конструювання СПЗ ВС пов'язують з ітеративною розробкою, автоматизованим модульним тестуванням і поетапним формуванням архітектури СПЗ ВС.

Поряд із TDD у створенні ВС застосовують гнучкі (agile) методи розроблення, зорієнтовані на ітеративність, інкрементність і скорочення циклу зворотного зв'язку [7]. У межах такого підходу вимоги організують у журнал завдань (backlog), розробку виконують короткими ітераціями, а перевіряння ухвалених рішень інтегрують у поточний процес робіт. Для ВС це пов'язано з тим, що програмні складники зазвичай уточнюють одночасно з апаратною платформою, тому жорстко послідовні моделі не завжди забезпечують належну узгодженість між етапами розроблення.

Разом із тим у сфері СПЗ ВС agile-підходи зазвичай налаштовують з урахуванням вимог реального часу, апаратно-програмної розробки, обмеженості обчислювальних ресурсів і необхідності випробувань на обладнанні. На практиці це реалізують за рахунок окремих елементів, зокрема, коротких циклів планування, частішої інтеграції, раннього тестування та інтенсивної командної взаємодії. У такому трактуванні гнучкі методи розглядають як процесне підґрунтя для підвищення узгодженості між розробленням, інтеграцією та перевіркою програмних компонентів.

Іншим напрямом розвитку методологій конструювання СПЗ ВС є перенесення основного акценту з вихідного коду на модель системи. Model-based Development (MBD) [8, 9] визначають як підхід, де основним артефактом є модель, що описує структуру, поведінку, часові характеристики та взаємодію системи із зовнішнім середовищем. У межах цього підходу виконують моделювання, верифікаці. та раннє тестування до завершення реалізації на цільовій платформі. Надалі модель уточнюють до рівня, на якому можливе автоматизоване генерування програмного коду або інших реалізаційних артефактів.

Для ВС MBD пов'язано з інтеграцією етапів формування вимог, проєктування, тестування та реалізації в межах єдиного модельного подання. Застосування моделі як основного джерела опису системи забезпечує трасування вимог, повторне використання компонентів, аналіз поведінки та узгодження програмної й апаратної частин. Такий підхід застосовується під час розроблення складних кіберфізичних і багатодоменних систем, у яких поєднуються алгоритмічний опис, часовий аналіз, симуляція та автоматизоване генерування коду.

Наступний етап еволюції методологій конструювання СПЗ ВС пов'язаний з інтеграцією процесів розроблення, тестування, постачання та гарантування безпеки в межах безперервного циклу Embedded DevSecOps [3, 6], насамперед у конструюванні СПЗ ВС мовою C. Його поширення зумовлене апаратними обмеженнями цільових платформ, тривалим ЖЦ ВС, вимогами до надійності функціонування та необхідністю контролювання ризиків на пізніх етапах впровадження ВС.

В [1] сучасне проєктування ВПЗ розглянуто через архітектуру програмного забезпечення, процеси Agile і DevOps, а також практики розроблення. До складу відповідних інженерних практик віднесено проєктування безпечних застосунків, статичний аналіз, перевірка коду (code review), оцінювання покриття коду та використання CI/CD. У цьому контексті Embedded DevSecOps розглядають як процесну основу для інтеграції вимог безпеки та якості до повного циклу розроблення СПЗ ВС мовою C.

Окреме місце серед актуальних підходів займає фреймворк-орієнтоване розроблення СПЗ ВС на основі C++ і Qt. У цьому випадку C++ застосовують для реалізації прикладної логіки, системних серві-

сів і обчислювальних компонентів, тоді як комерційний фреймворк Qt забезпечує засоби конструювання інтерфейсу (насамперед Qt Quick і QML) організації взаємодії між програмними модулями (механізм signals and slots), оброблення подій і досягнення переносності між апаратно-програмними платформами. Такий підхід застосовують, зокрема, для ВС на базі вбудованої ОС Linux, де поєднують вимоги до продуктивності, модульної організації та супроводження СПЗ ВС. Тоді C++ і Qt становлять інструментальну основу для створення модульного та переносного СПЗ ВС, зокрема панелей керування, людино-машинних інтерфейсів та інших пристроїв із розвиненою візуальною складовою.

Актуальні методології конструювання СПЗ ВС охоплюють як підходи до структурної організації програмного коду, так і процесні, модельні та фреймворк-орієнтовані засоби розроблення. Їх застосуванню притаманне поєднання вимог до надійності, передбачуваності виконання, інтеграції з апаратною платформою, тестовності, безпеки та супроводженості СПЗ ВС.

У практиці Embedded DevSecOps безперервне тестування доцільно запроваджувати як багаторівневу систему перевірок, де окремі завдання можуть бути виконані послідовно або паралельно залежно від їхнього призначення та вартості виконання. До такої системи зазвичай належать автоматизоване збирання, версії СПЗ, модульне тестування, аналіз якості коду, перевірка залежностей, тестування інтеграції компонентів, а також спеціалізовані безпекові процедури, що доповнюють класичний CI/CD-конвеєр. Важливо, що результати кожного запуску накопичують фактичні дані про тривалість, успішність і чутливість тестових процедур, які стають підґрунтям для подальшої оптимізації CI/CD-конвеєра, раціонального розподілу тестових завдань і підвищення відтворюваності контролю якості. Таким чином, безперервне тестування в Embedded DevSecOps (рис. 5) слугує не лише засобом оперативного виявлення помилок, а й механізмом системного керування якістю, зокрема, та безпекою СПЗ ВС протягом усього ЖЦ ВС [6].

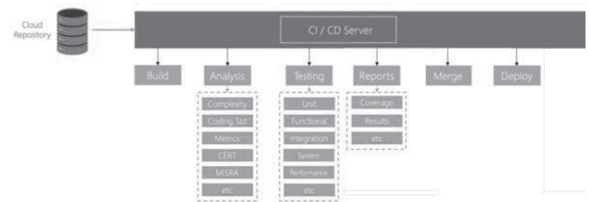

Рис. 5. Позиція безперервного тестування СПЗ ВС в Embedded DevSecOps

3. Особливості рівнів і видів тестування СПЗ ВС

Позиція тестування СПЗ ВС у життєвому циклі ВС. Рівні та види тестування спеціалізованого програмного забезпечення вбудованих систем доцільно розглядати як загальну методичну основу, успадковану від усталених практик тестування програмного забезпечення. У цьому підході рівні тестування охоплюють компонентний, інтеграційний, системний і приймальний рівні, тоді як види тестування описують аспект перевірки, зокрема, функційну та нефункційну відповідність. Однак для СПЗ ВС ця класифікація набуває прикладної специфіки: об’єкт тестування має бути оцінено не лише як програмний код, а й як складник програмно-апаратної системи, який функціонує у взаємодії з процесором, шинами обміну, сенсорами, виконавчими механізмами та часовими обмеженнями. Тому для СПЗ ВС тестові рівні фактично відображають не лише їхню логічну декомпозицію, а й ступінь наближення тестового оточення до реального середовища функціонування СПЗ.

Саме з цієї причини в ЖЦ ВС широко застосовують сімейство конфігурацій X-in-the-Loop (XiL) [4] як предметно-орієнтоване уточнення класичних рівнів тестування [5]. На ранніх етапах застосовують Model-in-the-Loop (MiL), де верифікують алгоритми та керувальну логіку на рівні моделей без реального апаратного виконання. Далі Software-in-the-Loop (SiL) дає змогу перевіряти згенерований або реалізований програмний код у середовищі розроблення, зіставляючи його поведінку з моделлю. Processor-in-the-Loop (PiL) переносить перевірку на цільовий процесор або його еквівалентний симулятор, що важливо для виявлення ефектів, пов’язаних із компі-

лятором, архітектурою процесора та часовими характеристиками виконання. Наступний етап Hardware-in-the-Loop (HiL), передбачає підключення реального контролера до моделі фізичного об’єкта, що виконується в реальному часі, і тому є ключовим засобом перевірки інтеграції програмної та апаратної частин. Таким чином, XiL-послідовність конкретизує, як саме класичні рівні тестування реалізуються в контексті СПЗ вбудованих систем.

Для складних кіберфізичних ВС (транспортних тощо) цю послідовність доповнюють конфігурації Driver-in-the-Loop (DiL) та Vehicle-in-the-Loop (ViL). У галузі автоматизованих транспортних засобів DiL-тестування зазвичай означає включення людини-оператора або водія до замкнутого контуру тестування, коли вже поведінку ВС загалом оцінюють не лише за формальними критеріями, а й з урахуванням реакції людини на сценарії керування, інтерфейси та динаміку об’єкта. VIL із свого боку поєднує реальний транспортний засіб із віртуальним середовищем. Його застосовують для зближення полігонних і віртуальних випробувань. У результаті конфігурації MiL, SiL, PiL, HiL, DiL і ViL слід трактувати не як альтернативу традиційним рівням тестування, а як ієрархію середовищ для перевірки, у межах якої послідовно зростає реалізм відтворення умов експлуатації і, відповідно, достовірність оцінювання властивостей СПЗ ВС. У цьому контексті концепція *зсуву ліворуч* (рис. 6) означає перенесення дій з тестування на максимально ранні стадії ЖЦ СПЗ, коли ще можна дешево усунути дефекти вимог, архітектури, моделей і програмної логіки.

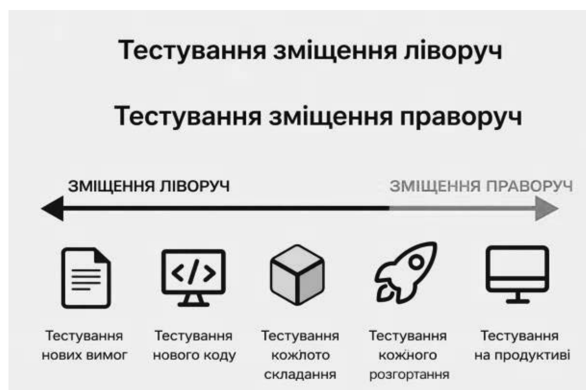

Рис. 6. Концепція зсуву ліворуч і зсуву праворуч у процесі тестування СПЗ ВС

Для СПЗ ВС це охоплює ранню верифікацію вимог, аналіз архітектурних рішень, MiL-перевірки, статичний аналіз, модульне тестування, а також SiL і PiL тестування до повної інтеграції з апаратними засобами. Натомість *зсув праворуч* акцентує потребу у продовженні перевірянь після інтеграції та розгортання, коли СПЗ ВС і ВС загалом демонструють свої критичні властивості вже в реальних або наближених до реальних умовах експлуатації. Для ВС це означає HiL, DiL, ViL тестування, аналіз телеметрії, експлуатаційний моніторинг, перевірка оновлень і довготривалі випробування стабільності. Отже, для СПЗ ВС зсув ліворуч і зсув праворуч доцільно розглядати як взаємодоповнювальні кращі практики, а XiL-конфігурації – як практичний механізм їх реалізації в безперервному контурі гарантування якості.

У контексті тестування СПЗ ВС доцільно розмежовувати стандартизовані техніки проєктування тестів і підходи типу чорної, білої та сірої скриньки. Згідно з ДСТУ ISO/IEC/IEEE 29119-3, 29119-4, до стандартизованих належать насамперед специфікаційно-орієнтовані, структурно-орієнтовані та досвідно-орієнтовані техніки. Їх застосовують під час проєктування та реалізації тестів на різних рівнях, зокрема, також і для СПЗ ВС з урахуванням апаратних обмежень, часових вимог та залежності від периферії.

Підходи фазингу [10, 11] та чорної й білої скриньки описуть не стандартну класифікацію технік тестування, а ступінь доступу тестувальника до внутрішньої структури об’єкта тестування: від перевірки лише зовнішньої поведінки до аналізу внутрішніх структур, покриття коду та проміжного комбінованого варіанта.

Серед цих технік *фазинг* слід розглядати як окрему спеціалізовану техніку динамічного автоматизованого тестування, а не як самостійний «рівень» тестування. Його сутність полягає в масованому поданні на вхід програми випадкових, спеціальним чином модифікованих або цілеспрямовано згенерованих даних для виявлення аварійних завершень, порушень роботи з пам’яттю та інших дефектів. У най-

простішому вигляді фазинг тяжіє до моделі чорної скриньки, але сучасні варіанти, зокрема, керований покриттям (coverage-guided) фазинг [10], фактично наближають його до сірої скриньки, оскільки передбачають внутрішній зворотний зв'язок щодо виконання СПЗ для спрямування генерації нових тестів. Для СПЗ ВС така техніка є особливо цінною, але водночас складною через тісну прив'язку СПЗ до регістрів, переривань і периферійних інтерфейсів.

У сучасних практиках тестування СПЗ ВС їх фазинг поєднують з емуляцією та автоматичним моделюванням периферії. У [12] прямо зазначено, що динамічне тестування та фазинг СПЗ ВС істотно обмежені апаратними залежностями й низькою масштабованістю, хоча запропонований авторами підхід дає змогу тестувати firmware у більш апаратно-незалежний спосіб.

Перспективні підходи до організації тестування СПЗ ВС. У практиці розроблення й тестування СПЗ ВС сформувалося кілька стійких центрів компетенцій, кожен з яких охоплює окремий клас інструментів, платформ і послуг, пов'язаних із забезпеченням якості, верифікацією та випробуванням.

Статичний аналіз і якість коду (Parasoft, IAR, MathWorks, Synopsys / Black Duck (Coverity)). Засоби цього напряму застосовують для тестування коду СПЗ без їх виконання, виявлення дефектів, потенційно небезпечних конструкцій, порушень стандартів програмування та підтвердження відповідності вимогам до надійності й безпеки.

HIL і випробувальні стенди (dSPACE, National Instruments). Компанії цього сегменту спеціалізуються на побудові апаратно-програмних стендів для випробувань у реальному часі, моделюванні середовищ функціонування та інтеграції програмних компонентів із реальними контролерами, платами й периферією.

Фазинг (Code Intelligence, Defensics). Методи та засоби цього класу зорієнтовано на автоматизоване формування великої кількості некоректних, граничних або несподіваних вхідних даних з метою виявлення відмов і вразливостей СПЗ ВС.

Новітні підходи: цифрові двійники та агентний штучний інтелект

Перспективним напрямом розвитку тестування СПЗ ВС є використання цифрових двійників як основи безперервного тестування. Цифровий двійник доцільно розглядати як виконувану модель СПЗ ВС та їх середовища, узгоджену з реальною конфігурацією за параметрами, інтерфейсами й сценаріями функціонування. Такий підхід дає змогу перенести значну частину регресійних тестів до автоматизованих конфігурацій із залученням реальних пристроїв. Практична реалізація зазвичай охоплює моделі середовища, бібліотеки (не)штатних і граничних сценаріїв, автоматизоване виконання в CI/CD-конвеєрі, а також валідування самого двійника шляхом калібрування та підтвердження прийнятної точності моделі.

Іншим новітнім підходом є застосування агентного штучного інтелекту для підтримки процесів тестування. Тоді агентний штучний інтелект виконує допоміжні функції в задачах, які складно масштабувати вручну: генерування тестових заготовок на основі вимог та інтерфейсів, адаптацію тестів у разі зміни конфігурацій і варіантів СПЗ ВС, аналіз журналів і логів виконання, а також добір сценаріїв для розширення покриття. Доцільним є також його застосування для поглиблення тестування безпеки, зокрема, під час підготування фазингу. Водночас у критичних і регульованих предметних областях такий підхід має бути застосований лише за умови контролю результатів, коли артефакти проходять формалізовані процедури перевірки, аналізу та аудиту з боку експерта в предметній області.

Перспективним є поєднання цифрового двійника з агентним штучним інтелектом в об'єднаному xiL-контурі безперервного тестування СПЗ. Тоді агент із функціями штучного інтелекту відбирає або, за потреби, формує релевантні сценарії, ініціює їх виконання на цифровому двійнику, узагальнює результати прогонів, готує

описи дефектів і підтримує актуальність тестового набору під час еволюції СПЗ ВС.

4. Авторський підхід до опрацювання проблем тестування СПЗ ВС

Зіставлення проаналізованих вище особливостей СПЗ ВС і процесу безперервного тестування СПЗ у SiL-конфігурації Embedded DevSecOps дозволяє уточнити проблеми ефективної організації цього процесу [13] на підтримку його додаткових функцій а)-в), зафіксованих у Вступі статті:

– ефективне керування артефактами тестування СПЗ, зокрема їх адміністрування й контрольоване повторне використання як у межах окремих циклів DevSecOps, так і між ними (P_1);

– забезпечення повноаспектного наскрізного простежування між самими артефактами тестування та між ними й іншими артефактами DevSecOps (P_2);

– адаптування перспективних процесів і технік динамічного тестування, регламентованих ДСТУ ISO/IEC/IEEE 29119-3, 29119-4, з урахуванням ускладнюючих особливостей СПЗ ВС (P_3);

– своєчасне ґрунтовне відстеження якості СПЗ ВС, насамперед кібер- і функціональної безпеки, прийнятної для подальшого тестування СПЗ на цільових платформах у конфігураціях PiL, HiL, ViL, DiL, її гарантування й засвідчення (P_4);

– ефективне розподілення й контролювання відповідальності й повноважень між різними командами в Embedded DevSecOps: конструювання, тестування, інформаційної безпеки, операційного персоналу (P_5);

– постійне (не)кількісне оцінювання рівня зрілості процесу безперервного тестування СПЗ і здобуття інформації щодо його вдосконалення (P_6).

Авторський підхід до опрацювання проблем P_1-P_6 поєднує:

а) конструктивну постановку спільної для P_1-P_6 *технічної задачі*, а саме: розроблення моделей і методів ефективного розгортання й використання безпечного конвеєра безперервного автоматизованого тестування СПЗ ВС у siL-конфігурації в Embedded DevSecOps для диверсифікованого конструювання мінливих СПЗ ВС;

б) засади розв’язання цієї задачі, підсумовані положеннями B_1–B_7;

в) першочергові кроки з реалізації підходу S_1.–S_7.

Запропоновані засади та кроки підсумовано в таблиці 3.

Таблиця 3.

Сутність авторського підходу до опрацювання проблем тестування СПЗ мінливих ВС

Засади підходу	Кроки реалізації засад
Організація конвеєра (pipeline) безперервного автоматизованого тестування СПЗ ВС у TDD-стилі [3] як підґрунтя для ефективного CI/CD конвеєра розгортання СПЗ в DevSecOps (B_1)	Побудова моделі властивостей (віртуальної) лінійки СПЗ мінливих ВС шляхом виокремлення в CI/CD конвеєрі конструювання СПЗ обов’язкових фрагментів, спільних для всіх СПЗ, і, відповідно, опціональних, властивих лише окремим СПЗ (S_1)
Формалізація обох конвеєрів у парадигмі лінійки програмних продуктів (Software Product line) згідно з ISO/IEC 26554 (для цільової Лінійки тестів мінливих СПЗ) і ДСТУ ISO/IEC 26550 (як (віртуальної) лінійки самих СПЗ) (B_2)	Побудова на підставі отриманої моделі властивостей технологічної моделі процесу автоматизованого тестування СПЗ як процесу розгортання й використання Лінійки їх тестів , а саме вкладених моделей конвеєрів тестування домену й застосунків та їх інфраструктури, згідно із засадами [13] і B_1, B_2 (S_2)
Декларативне моделювання процесів тестування домену та застосунків у парадигмі Pipeline as a Code [14] як часткових конвеєрів, вкладених до формованої Лінійки тестів, а інфраструктури тестування СПЗ – у	Налаштування традиційних технік тестування коду на рівнях від модульного до системного для СПЗ ВС та їх інтеграція з техніками поглибленого статичного аналізу [6] і фазингу [10,11] у модельованих вкладених кон-

Засади підходу	Кроки реалізації засад
парадигмі Infrastructure as a Code [15] (B_3)	веєрах тестування домену й застосунків (S_3)
Декларативне моделювання процесів розгортання та адміністрування інфраструктури Лінійки тестів у руслі підходів GitOps [16] і QAOps [5, 6] для адекватного розподілу відповідальностей і взаємної довіри різних команд в Embedded DevSecOps (B_4)	Розроблення методів ситуативного вибору оптимального інструмента, переважно з відкритим кодом, для основних видів статичного й динамічного тестування в модельованих конвеєрах рівня домену й застосунків (S_4) згідно з [17]
Застосування інструментальних засобів з відкритим кодом для декларативного конфігураційного керування автоматизованими конвеєрами Лінійки тестів та їх інфраструктурою (B_5)	Формування повної й ненадлишкової системи метрик якості запропонованої Лінійки тестів та розроблення процедур виявлення її неприйнятного зниження на їх підставі (S_5)
Забезпечення автоматизованого виявлення, у формованих конвеєрах Лінійки тестів, порушень відповідності тестованого СПЗ релевантним міжнародним і галузевим стандартам, насамперед щодо кібер- та інформаційної безпеки (B_6)	Аналіз ефективності інструментів з відкритим кодом для декларативного керування конвеєрами та інфраструктурою як кодом, де-факто стандартизованих підходами GitOps і QAOps, і визначення їх рамкового переліку для розгортання й використання запропонованої Лінійки тестів (S_6)
Безперервне відстеження якості запропонованої Лінійки тестів із вчасним виявленням її неприпустимого зниження та можливостей і способів удосконалення Лінійки (B_7)	Розроблення макетного зразка інструментального засобу вибору оптимального інструмента для певного виду тестування за допомогою методів кроку S_4, його інтеграція до процесів запропонованої Лінійки тестів разом із вибраними засобами з кроку S_6 та апробація отриманого рішення у вітчизняній організації-розробнику мінливих СПЗ ВС (S_7)

Реалізація й ситуативне уточнення кроків S_1.–S_7 є предметом подальших досліджень авторів.

Висновки

Запропоновано линійку тестів згідно зі стандартом ISO/IEC 26554 практиками QAOps для відокремленого тестування СПЗ ВС на рівнях від модульного до системного, де емуляцію замінено на використання тестових двійників (mocks, SiL) для модулів СПЗ і положеннями специфікації вимог до СПЗ.

Література

1. Beningo J. Embedded software design: A practical approach to architecture, processes, and coding techniques. Apress, 2022.
2. Lacamera D. Embedded systems architecture: Design and write software for embedded devices to build safe and connected systems. 2nd ed. Packt Publishing, 2023.
3. Grenning J.W. Test-Driven Development for Embedded C. Pragmatic Bookshelf, 2011.
4. Association for Standartization of Automation and Measuring Systems (ASAM). Generic Simulator Interface. Specification. Part 1, 2024. – 396 p.
5. Clokie K. A practical guide to Testing in DevOps – Leanpub, 2017. – 128 p.
6. Hornbeek M., Wakeman D. Continuous Testing, Quality, Security, and Feedback: Essential strategies and secure practices for DevOps, DevSecOps, and SRE transformations – Packt Publishing, Limited, 2024. – 420 p/
7. Salo O., Abrahamsson P. Agile methods in European embedded software development organisations: A survey on the actual use and usefulness of Extreme Programming and Scrum. IET Software. 2008. Vol. 2. No 1. P. 58–64.
8. Böhm W. et al. (Eds.): Model-Based Engineering of Collaborative Embedded Systems. ISBN 978-3-030-62135-3. Springer, Jan. 2021 – 358 p.
9. Nicolescu G., Mosterman P.J. (eds.). Model-Based Design for Embedded Systems. CRC Press, 2009.
10. Yun J., Lee I., Xu M., Kim T. Fuzzing of embedded systems: A survey. ACM

Computing Surveys. 2022. Vol. 55. No 7. Article 132. P. 1–33.
11. Eisele M., Ebert D., Huth C., Zeller A. Fuzzing embedded systems using debug interfaces. Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA ’23). 2023. P. 1031–1042.
12. Feng B., Mera A., Lu L. P²IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. 29th USENIX Security Symposium (USENIX Security 20). 2020. P. 1237–1254.
13. Ierofeiev I., Sinitsyn I., Slabospitska O. Embedded Software Testing Issues and Addressing Them with Software Product Lines Paradigm. CEUR Workshop Proceedings. 2024. Vol. 4053. P. 13–23.
14. Soni M. Hands-on Pipeline as YAML with Jenkins: A Beginner's Guide to Implement CI/CD Pipelines for Mobile, Hybrid, and Web Applications Using Jenkins BPB Publications, 2021 – 320 p.
15. Wang R.Infrastructure as Code, Patterns and Practices. With examples in Python and Terraform – Manning, 2022. – 400 p.
16. Salecha R. Practical GitOps : Infrastructure Management Using Terraform, AWS, and GitHub Actions – Apress L. P., 1st ed., 2022 –270 p.
17. Єрофеєв Ю.В., Слабоспицька О.О. Embedded DevSecOps у хмарі: засоби та перспективи оптимізації. Proc. of the XVIII Int. scientific and practical conf. «Information technologies and automation–2025». 2025 – P. 837.

References

1. Beningo J. Embedded software design: A practical approach to architecture, processes, and coding techniques. Apress, 2022.
2. Lacamera D. Embedded systems architecture: Design and write software for embedded devices to build safe and connected systems. 2nd ed. Packt Publishing, 2023.
3. Grenning J.W. Test-Driven Development for Embedded C. Pragmatic Bookshelf, 2011.
4. Association for Standartization of Automation and Measuring Systems (ASAM). Generic Simulator Interface. Specification. Part 1, 2024. – 396 p.
5. Clokie K. A practical guide to Testing in DevOps – Leanpub, 2017. – 128 p.
6. Hornbeek M., Wakeman D. Continuous Testing, Quality, Security, and Feedback: Essential strategies and secure practices for DevOps, DevSecOps, and SRE transformations – Packt Publishing, Limited, 2024. – 420 p/
7. Salo O., Abrahamsson P. Agile methods in European embedded software development organisations: A survey on the actual use and usefulness of Extreme Programming and Scrum. IET Software. 2008. Vol. 2. No 1. P. 58–64.
8. Böhm W. et al. (Eds.): Model-Based Engineering of Collaborative Embedded Systems. ISBN 978-3-030-62135-3. Springer, Jan. 2021 – 358 p.
9. Nicolescu G., Mosterman P.J. (eds.). Model-Based Design for Embedded Systems. CRC Press, 2009.
10. Yun J., Lee I., Xu M., Kim T. Fuzzing of embedded systems: A survey. ACM Computing Surveys. 2022. Vol. 55. No 7. Article 132. P. 1–33.
11. Eisele M., Ebert D., Huth C., Zeller A. Fuzzing embedded systems using debug interfaces. Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA ’23). 2023. P. 1031–1042.
12. Feng B., Mera A., Lu L. P²IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. 29th USENIX Security Symposium (USENIX Security 20). 2020. P. 1237–1254.
13. Ierofeiev I., Sinitsyn I., Slabospitska O. Embedded Software Testing Issues and Addressing Them with Software Product Lines Paradigm. CEUR Workshop Proceedings. 2024. Vol. 4053. P. 13–23.
14. Soni M. Hands-on Pipeline as YAML with Jenkins: A Beginner's Guide to Implement CI/CD Pipelines for Mobile, Hybrid, and Web Applications Using Jenkins BPB Publications, 2021 – 320 p.
15. Wang R.Infrastructure as Code, Patterns and Practices. With examples in Python and Terraform – Manning, 2022. – 400 p.
16. Salecha R. Practical GitOps : Infrastructure Management Using Terraform, AWS, and GitHub Actions – Apress L. P., 1st ed., 2022 –270 p.
17. Ierofeiev I., Slabospitska O. Cloude Embedded DevSecOps: Optimization Tools and Perspectives. Proc. of the XVIII Int. scientific and practical conf. «Information technologies and automation–2025». 2025. – P. 837.

Дата першого надходження до видання: 10.03.2026
Внутрішня рецензія отримана: 14.03.2026
Зовнішня рецензія отримана: 15.04.2026
Дата прийняття статті до друку: 19.03.2026
Дата публікації: 16.04.2026

Про авторів:

[1]*Єрофеєв Юрій Володимирович,*
аспірант ІПС НАНУ
Yerofieiev Yuriy,
Post-graduate student
http://orcid.org/0009-0006-8985-2729.

[2]*Сініцин Ігор Петрович,*
доктор технічних наук,
директор.
Sinitsyn Igor,
Ph.D (doctor, technical sciences),
director
http://orcid.org/0000-0002-4120-0784.

Місце робот авторів:

[1]Інститут програмних систем
НАН України
Institute of Software Systems
National Academy of Sciences of Ukraine
тел. +38-044-522-62-42
E-mail: www.iss.nas.gov.ua