Politechnika Śląska

Wydział Automatyki, Elektroniki i Informatyki

Kierunek Informatyka

Rozprawa doktorska

Jakość i bezpieczeństwo oprogramowania w przemyśle motoryzacyjnym – analiza standardów oraz opracowanie metody wspomagającej proces tworzenia oprogramowania.

Autor: mgr inż. Patryk Pankiewicz

Kierujący pracą: dr hab.inż. Andrzej Białas, prof. Łukasiewicz-EMAG

Gliwice, 2022

Spis treści

Spis treści **3**

1 Wstęp **7**
1.1 Motywacja i cele pracy 8
1.1.1 Teza 9
1.2 Doktorat wdrożeniowy 10
1.3 Opis pracy 11

2 Proces **13**
2.1 Procesy 15
2.1.1 ASPICE 15
2.1.2 V-Model 18
2.2 Standardy technologiczne 19
2.2.1 Standard AUTOSAR Classic 19
2.2.2 MISRA C 24
2.2.3 Metryki HIS 24
2.3 Normy o zasięgu międzynarodowym 25
2.3.1 ISO26262 25
2.3.2 ISO 25010 26
2.3.3 ISO 21434 28

3 Identyfikacja problemów **31**
3.1 Problemy skalowalności 31
3.1.1 Problem P1 – skalowalność ekstraktów diagnostycznych 31
3.2 Problemy sprzętowe 35
3.2.1 Problem P2 – ograniczone zasoby sprzętowe 35
3.3 Problemy złożoności 38
3.3.1 Problem P3 – wysoka złożoność oprogramowania . . . 38

3.4 Problemy czasu wykonania 44
3.4.1 Problem P4 – skomplikowane ograniczenia i zależności czasowe 44

4 Metodyka badań empirycznych 51
4.1 Opis stanowiska badawczego 51
4.1.1 Opis projektów komercyjnych 52
4.2 Opis metod badawczych 52
4.2.1 Metoda badawcza MB1 53
4.2.2 Metoda badawcza MB2 53
4.2.3 Metoda badawcza MB3 55
4.2.4 Metoda badawcza MB4 55
4.2.5 Metoda badawcza MB5 56
4.2.6 Metoda badawcza MB6 57
4.3 Metody badawcze a cele badawcze 57

5 Pakiet rozwiązań 59
5.1 Automatyzacja stosu diagnostycznego 59
5.1.1 Nawiązanie do zagadnienia badawczego 59
5.1.2 Sposób rozwiązania problemu badawczego 64
5.1.3 Walidacja i ocena rozwiązania 73
5.1.4 Odniesienie do norm jakości, bezpieczeństwa i zabezpieczeń 76
5.1.5 Dostarczenie pakietu wiedzy i wzorców projektowych dla inżynierów oprogramowania 80
5.2 Automatyzacja i architektura DLT 86
5.2.1 Nawiązanie do zagadnienia badawczego 86
5.2.2 Sposób rozwiązania problemu badawczego 89
5.2.3 Walidacja i ocena rozwiązania 93
5.2.4 Odniesienie do norm jakości, bezpieczeństwa i zabezpieczeń 95
5.2.5 Dostarczenie pakietu i wzorców projektowych dla inżynierów oprogramowania 98
5.3 Automatyzacja i architektura stosu NvM 101
5.3.1 Nawiązanie do zagadnienia badawczego 101
5.3.2 Sposób rozwiązania problemu badawczego 106
5.3.3 Walidacja i ocena rozwiązania 110

5.3.4 Odniesienie do norm jakości, bezpieczeństwa i zabezpieczeń . . . 112
5.3.5 Dostarczenie pakietu i wzorców projektowych dla inżynierów oprogramowania . . . 114
5.4 Obserwator systemu . . . 117
5.4.1 Nawiązanie do zagadnienia badawczego . . . 117
5.4.2 Sposób rozwiązania problemu badawczego . . . 119
5.4.3 Walidacja i ocena rozwiązania . . . 122
5.4.4 Odniesienie do norm jakości, bezpieczeństwa i zabezpieczeń . . . 125
5.4.5 Dostarczenie pakietu i wzorców projektowych dla inżynierów oprogramowania . . . 127

6 Wnioski i uwagi końcowe 133
6.1 Podsumowanie prac badawczych . . . 133
6.2 Podsumowanie pakietu rozwiązań . . . 135
6.3 Konkluzje i dyskusja . . . 139

Spis rysunków 143

Lista kodów źródłowych 145

Słownik skrótów ze skorowidzem 147

Słownik terminów ze skorowidzem 153

Bibliografia 155

Rozdział 1

Wstęp

Poziom skomplikowania oprogramowania w pojazdach samochodowych drastycznie wzrasta z roku na rok. Systemy wbudowane stosowane w samochodach odpowiadają bezpośrednio za bezpieczeństwo kierowcy oraz pasażerów. W obecnie produkowanych pojazdach, znajduje się ponad 100 jednostek obliczeniowych Electronic Control Unit – (ECU) – każda z nich jest odpowiedzialna za realizowanie części funkcjonalności powiązanych z jazdą, ale nie tylko. Producenci oferują coraz to większy zakres funkcji niezwiązanych z przemieszczaniem się – skomplikowane systemy multimedialne swoimi możliwościami zbliżają się do telefonów komórkowych. Po jednej z aktualizacji w samochodach marki Tesla, użytkownicy mogli nawet grać w gry komputerowe używając kierownicy [1]. Oczywiście samochód musiał być w trybie parkowania.

Z jednej strony samochody oferują coraz lepsze systemy zwiększające bezpieczeństwo, m.in.:

- systemy monitorujące zmęczenie kierowcy,
- systemy wspomagające jazdę, łącznie z autonomicznym poruszaniem się,
- systemy przygotowujące samochód do wypadku (napinając pasy, podwyższając zawieszenie, aby ustawić samochód w bezpieczniejszej pozycji itp.),
- systemy „eCall" (Emergency Call), dzwoniące na pogotowie po wypadku, aby przekazać informacje na temat pozycji samochodu, liczby pasażerów oraz stanu pojazdu.

Niewątpliwie, im więcej skomplikowanego oprogramowania znajduje się w samochodzie, tym więcej możliwych błędów, co może powodować nieprzewidywalne zachowanie samochodu lub systemów wspomagających. Firma Volvo wykonała w 2020 roku analizę dotyczącą statystyk oprogramowania [2], która wykazała, że każdy ich samochód posiada: ponad 100 milionów linii kodu, ponad 10 milionów instrukcji warunkowych oraz ponad 3 miliony funkcji wywoływanych w 30 milionach miejsc w kodzie. Nawet z zapewnieniem najlepszych standardów i norm, zebraniem grupy wybitnych programistów oraz wielopoziomowym procesem testowania, w tak ogromnej bazie oprogramowania pojawi się wiele błędów. Co więcej, wielkość i złożoność oprogramowania będą rosły. Można zatem przyjąć, że nie jesteśmy w stanie tworzyć w pełni deterministycznych rozwiązań. Musimy więc posiadać jak najwięcej narzędzi oraz metod, które wspomagają proces tworzenia oprogramowania, zapewniają jak najwięcej informacji na temat działania systemu, a także umożliwiają analizy skomplikowanych zależności.

Już w 2007 roku badania wykazały, że 35% problemów w samochodach pochodziło z elektronicznych systemów wbudowanych [3], dlatego producenci samochodów zjednoczyli się i wspólnie opracowali standard AUTomotive Open System ARchitecture (AUTOSAR), którego celem jest zwiększenie jakości i unifikacja oprogramowania systemów wbudowanych stosowanych w samochodach.

Przemysł samochodowy od lat stara się optymalizować oraz ciągle rozwijać wszystkie procesy wpływające na jakość, co finalnie ma gwarantować bezpieczeństwo i niezawodność. Pytanie, czy to wystarczy? Czy jesteśmy w stanie nadążyć za galopującym wzrostem technologicznym, coraz to nowymi językami oprogramowania, sposobami tworzenia kodu, nowymi architekturami mikrokontrolerów i stopniem skomplikowania systemów wbudowanych, który jest rezultatem tych wszystkich czynników?

1.1 Motywacja i cele pracy

Głównym celem pracy było stworzenie pakietu rozwiązań oraz narzędzi, stanowiących metodę wspomagającą proces tworzenia oprogramowania dla systemów wbudowanych, stosowanych w samochodach osobowych wykorzystujących standard AUTOSAR Classic.

Cele szczegółowe to:

1. Analiza elementów wpływających na jakość oprogramowania stosowanego w przemyśle samochodowym, w szczególności skupiając się na systemach powiązanych z jednostkami czasu rzeczywistego, najbardziej wpływającymi na bezpieczeństwo kierowcy.
2. Zidentyfikowanie obszarów, które posiadają jakiegokolwiek rodzaju wady lub luki.
3. Przeprowadzenie analiz oraz badań prowadzących do zminimalizowania możliwości wystąpienia błędów, zwiększenia jakości i bezpieczeństwa całego systemu.
4. Zaproponowanie rozwiązań dotyczących procesu tworzenia oprogramowania.
5. Zaproponowanie odpowiednich architektur, które zwiększą jakość produktu.
6. Zaproponowanie sposobu zmniejszenia ryzyka wynikającego z ogromnego stopnia skomplikowania oprogramowania.
7. Wskazanie obszarów, na które należy zwracać największą uwagę podczas realizowania projektów dotyczących systemów wbudowanych.
8. Ulokowanie wyników analiz oraz zaproponowanych rozwiązań na modelu jakości oprogramowania, opisanego w normie ISO 25010 [4].
9. Ulokowanie wyników analiz w kontekście norm bezpieczeństwa funkcjonalnego ISO 26262 [5] i cyberbezpieczeństwa ISO 21434 [6].

1.1.1 Teza

Teza postawiona przez autora pracy brzmi:

Poprzez wykorzystanie zaproponowanego w pracy pakietu rozwiązań stanowiącego metodę wspomagającą proces tworzenia oprogramowania, istnieje możliwość poprawy jakości, bezpieczeństwa funkcyjnego i cyberbezpieczeństwa oprogramowania w rozumieniu norm o zasięgu międzynarodowym – ISO 25010, ISO 26262 i ISO 21434.

Teza dotyczy oprogramowania w układach wbudowanych stosowanych w przemyśle motoryzacyjnym oraz wykorzystujących standard AUTOSAR Classic, stosowany w większości produkowanych seryjnie samochodów osobowych. Przykładami takich produktów są:

1. sterownik zarządzania baterią samochodu – Battery Management System (BMS),
2. główny sterownik pokładowy – Body Control Module (BCM), odpowiedzialny za pozyskiwanie i przetwarzanie sygnałów z innych sterowników,
3. sterownik komunikacyjny – Telematic Control Unit (TCU), odpowiedzialny za komunikację samochodu z chmurą i zdalne aktualizacje oprogramowania.

1.2 Doktorat wdrożeniowy

Rozprawa doktorska została zrealizowana w ramach programu doktoratów wdrożeniowych, mającego na celu zwiększenie współpracy przemysłu z jednostkami naukowymi. Autor rozprawy jest na stałe zatrudniony w przemyśle, co umożliwiło wdrożenie opracowanych naukowych konceptów. Rezultaty pracy zostały wdrożone w projektach komercyjnych. Wdrożone kody źródłowe są własnością klientów, ale wszystkie elementy koncepcyjne i architektoniczne proponowanej metody wspomagającej proces tworzenia oprogramowania, zostały zaprezentowane w taki sposób, aby odbiorcy mogli je zaimplementować we własnych projektach lub badaniach. Rozwiązania problemów badawczych zaprezentowano w postaci wzorców projektowych do wykorzystania przez twórców tego typu oprogramowania.

1.3 Opis pracy

Problem, który autor starał się rozwiązać, wynika z rosnącej liczby błędów w oprogramowaniu w przemyśle motoryzacyjnym, które wpływają bezpośrednio na niezawodność pojazdów, bezpieczeństwo i komfort kierowców. Praca skupiona została na jednostkach wbudowanych, używających standardu AUTOSAR Classic, czyli systemach powiązanych z krytycznymi układami dotyczącymi bezpieczeństwa użytkowników samochodów. Stanowi to istotę problemu, ponieważ komplikowanie oprogramowania nie osiągnie limitu i będzie stale rosnąć. Według autora potrzebne są zatem nowe metody, by utrzymać jakość i bezpieczeństwo na odpowiednim poziomie. Jednym z głównych celów rozprawy było ukazanie złożoności procesu tworzenia oprogramowania oraz wynikających z tego ryzyka błędów.

Struktura pracy jest następująca: w celu przedstawienia czytelnikowi kontekstu procesu tworzenia oprogramowania, w rozdziale 2 opisano procesy, standardy technologiczne oraz normy o zasięgu międzynarodowym, które wpływają na jakość i bezpieczeństwo oprogramowania. W rozdziale 3 przedstawiono zidentyfikowane przez autora problemy badawcze wraz z powiązanymi zagadnieniami badawczymi oraz omówiono je w kontekście istniejących badań naukowych. W rozdziale 4 opisano stanowiska badawcze używane podczas badań i analiz oraz wszystkie wykorzystane metody badawcze. W rozdziale 5 zaprezentowano proponowane elementy składające się na pakiet rozwiązań oraz finalną metodę wspomagającą proces tworzenia oprogramowania. Ponadto wskazano relację każdego zaproponowanego rozwiązania do zidentyfikowanych wcześniej problemów oraz zagadnień badawczych. Pracę podsumowano w rozdziale 6.

Wszystkie użyte w tekście skróty oraz terminy zostały opisane w słownikach na końcu pracy. Skrótów powiązanych ze standardem AUTOSAR użyto zgodnie ze słownikiem pojęć [7]. Rysunki przedstawiono w języku angielskim, ponieważ wdrożenia realizowane były w projektach międzynarodowych, a w opisach wyjaśniono ich znaczenie. W momencie druku rozprawy, artykuł badawczy autora na temat pamięci nieulotnej [8] jest w fazie redakcyjnej procesu publikacji. Specyfikacje standardu AUTOSAR są dostępne online, elementy bibliografii powiązane ze standardem mogą być wyszukane przez odniesienie się do ich nazwy. Cała dokumentacja standardu jest dostępna na stronie www.autosar.org.

Autorskie kody źródłowe oraz wyniki badań dostępne są na platformie GitHub pod adresem:

https://github.com/PatrykPankiewicz/Materialy

W pracy odnoszono się często do dwóch terminów:

Architektura – w kontekście standardu AUTOSAR; ogólna architektura oprogramowania została przedstawiona w rozdziale 2.2.1. Szczegóły są dostępne w dokumentacji stosu [9]. Architektura w kontekście rozprawy oznacza natomiast rozszerzenie istniejących rozwiązań o autorskie moduły oraz odpowiednie skonfigurowanie połączeń między modułami.

Automatyzacja – oznacza zaprojektowanie narzędzi przez autora, najczęściej w postaci skryptów, które ograniczają lub zastępują pracę inżynierów w trakcie procesu tworzenia oprogramowania.

Rozdział 2

Proces tworzenia oprogramowania motoryzacyjnych systemów wbudowanych

Jakość oraz bezpieczeństwo oprogramowania są rezultatami pracy wielu osób przyczyniających się do rozwoju produktu. W niektórych przypadkach, przy jednym układzie wbudowanym pracują setki osób. W celu utrzymania jak najwyższej jakości oprogramowania, przemysł motoryzacyjny używa wielu norm, standardów oraz procesów. Wszystkie te elementy nazywane są „state of the art", czyli zgodne z obecnymi zasadami tworzenia oprogramowania.

W przypadku jakichkolwiek problemów występujących w samochodach na drogach, ewentualna odpowiedzialność prawna jest określana na podstawie dowiedzenia odpowiedniej staranności tworzenia oprogramowania i wykorzystywania elementów znanych jako „state of the art".

W 2010 roku firma Toyota została obciążona karami zasądzonymi na kwotę ponad 1,6 miliarda dolarów przez sąd federalny w USA, jako rezultat problemów z systemami zarządzającymi pedałem gazu, które powodowały niekontrolowane przyspieszenie niektórych samochodów [10]. W ramach procesu sądowego zespół ekspertów zidentyfikował liczne naruszenia procesu tworzenia oprogramowania, jak na przykład niestosowanie ważnych elementów standardu budowania oprogramowania Motor Industry Software Reliability Association (MISRA) C. Z tego powodu, producenci samochodów wymagają od wszystkich swoich poddostawców stosowania określonych elementów procesu tworzenia oprogramowania, aby upewnić

się, że oprogramowanie systemów wbudowanych jest „zgodne ze sztuką" („state of the art"). Wymaganie dobrej znajomości procesu tworzenia oprogramowania systemów wbudowanych do samochodów, ogranicza dostępność specjalistów na rynku pracy. Często zespoły projektowe rozproszone są po całym świecie, przebywając w różnych krajach oraz często w różnych strefach czasowych.

W niniejszym rozdziale wyróżniono główne elementy wpływające na jakość i bezpieczeństwo oprogramowania. Dodatkowo uwzględniona została norma ISO 21434, opisująca aspekty cyberbezpieczeństwa. W ostatnich latach samochody wymieniają coraz to więcej danych z siecią Internet, co znacząco zwiększa zagrożenia wynikające z potencjalnych ataków. Dawniej, póki atakujący nie miał fizycznego dostępu do samochodu, praktycznie nie istniała szansa na wpłynięcie na sposób działania systemów sterujących. W przypadku ciągłego połączenia pojazdów z internetem, możliwość zdalnego włamania się do sieci samochodu jest znacząco większa. Z tego względu w 2021 roku została opublikowana norma o zasięgu międzynarodowym ISO 21434, opisująca aspekty cyberbezpieczeństwa w samochodach osobowych.

Podczas analizy wykonanej przez autora zidentyfikowano następujące elementy istotne dla rozwoju oprogramowania:

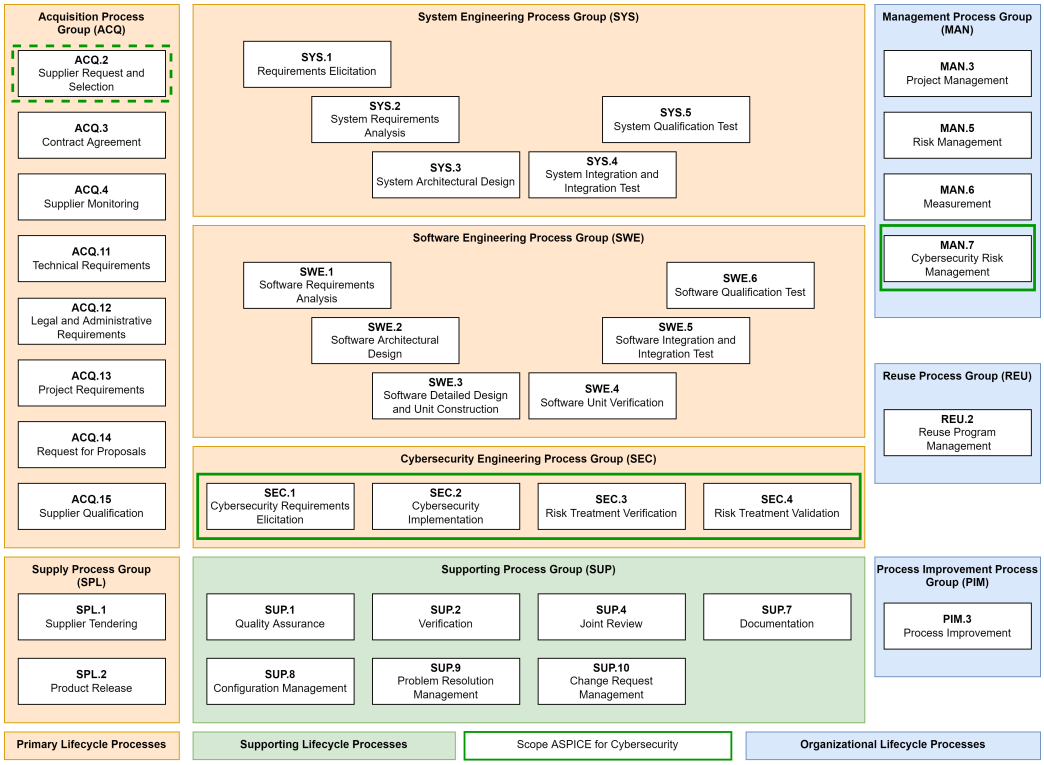
- standard AUTOSAR Classic,
- ewaluacja procesu tworzenia oprogramowania Automotive Software Process Improvement and Capability Determination (ASPICE), oparta na standardzie ISO 33061 [11],
- model tworzenia oprogramowania V-Model,
- metryki dotyczące oprogramowania Hersteller Initiative Software (HIS) [12],
- standard budowania oprogramowania MISRA C [13],
- norma bezpieczeństwa funkcyjnego ISO 26262 [5],
- norma jakości oprogramowania ISO 25010 [4],
- norma cyberbezpieczeństwa oprogramowania ISO 21434 [6].

Przedstawione w nich aspekty wyznaczają granicę badań i analiz przeprowadzonych przez autora. Zostały one przedstawione w kolejnych sekcjach, z podziałem na: procesy, normy i standardy.

2.1 Procesy

2.1.1 ASPICE

Standard ASPICE to referencyjny model procesu tworzenia oprogramowania w przemyśle motoryzacyjnym. W 2021 roku poza głównym modelem [14], zostały także wprowadzone dodatkowe elementy modelu dotyczące cyberbezpieczeństwa [15]. Szczegóły dotyczące modelu referencyjnego są dostępne w dokumentacji [14]. Rysunek 2.1 prezentuje model ASPICE wraz z wyróżnionymi kolorem zielonym elementami dotyczącymi cyberbezpieczeństwa. W kontekście rozważań niniejszej pracy



Rysunek 2.1. Model referencyjny procesu ASPICE

przeanalizowano głównie elementy grupy procesów Software Engineering (SWE), jako że są bezpośrednio powiązane z tworzeniem oprogramowania. ASPICE sprowadza się do oceny elementów procesu tworzenia oprogramowania pod względem jakości, powtarzalności oraz zgodności z

referencyjnym modelem. Każdy element procesu jest oceniany zgodnie z poziomami dojrzałości przedstawionymi w tabeli 2.1. Poziom 0 oznacza brak jakiejkolwiek definicji procesu, można go porównać do sytuacji, gdzie kilku inżynierów zdecydowałoby się na realizację danego przedsięwzięcia i bez większego zastanowienia rozpoczęłoby pracę. Efekty takiego projektu są nieprzewidywalne, niepowtarzalne oraz nacechowane wysokim ryzykiem. Poziom 5 oznacza proces ciągle optymalizowany, w pełni wdrożony i udokumentowany. Dobrym porównaniem jest proces wykorzystywany w sieciach gastronomicznych firmy McDonald's – gdziekolwiek na świecie nie skorzysta się z efektów pracy – jedzenie smakuje przewidywalnie, jest wykonane w określonych ramach czasowych, a jego jakość jest utrzymywana na odpowiednim poziomie. W przypadku oprogramowania motoryzacyjnego, wysoki poziom ASPICE określa podobne priorytety: powtarzalność jakości produktu, ukończenie projektu na czas i odporność procesu tworzenia oprogramowania na czynniki zewnętrzne (takie jak np. rotacja członków projektu). Standardowym poziomem ASPICE, oczekiwanym przez większość firm w przemyśle, jest poziom 3.

Wpływ ASPICE na jakość i bezpieczeństwo oprogramowania

Analizy ASPICE dokonano przy wykorzystaniu doświadczenia autora w przemyśle ([14], [15]). Wnioski są następujące:

1. Wymagania procesowe ASPICE są dobrze określone, nie zidentyfikowano możliwości mierzalnej poprawy.

2. Wdrożenie procesu jest bardzo ważne dla jakości oprogramowania.

3. Często występującym problemem jest integracja oprogramowania, zespołów i procesów dostawców.

4. Procesy często wymagają poprawy, ale przeważnie istnieją duże ograniczenia czasowe oraz finansowe.

5. Wymagania procesowe są czasami ignorowane przez pracowników, ponieważ brakuje świadomości na temat potencjalnych problemów wynikających z braku odpowiedniego procesu.

Według autora dalsze badania dotyczące standardu ASPICE są bardziej powiązane z pracą kierownika projektu lub inżyniera jakości. Dalsze analizy

Poziom	Definicja	Opis
0	Incomplete Process	Brak procesu, brak formalnej definicji ról, osób odpowiedzialnych, dokumentacji, śledzenia zmian oraz narzędzi. Brak definicji jakościowych.
1	Performed Process	Proces jest częściowo wykonywany, ale nie jest powtarzalny, zależy od zespołu wykonującego konkretny projekt. Jakość i sposób wykonania prac projektowych nie jest łatwa do zapewnienia.
2	Managed Process	Proces jest zdefiniowany i udokumentowany. Definicje ról, narzędzi oraz przebiegu projektów są dostępne, a pracownicy są tego świadomi.
3	Established Process	Proces jest zdefiniowany i zaimplementowany, projekty są realizowane zgodnie z procesem, który jest w stanie zapewnić zaplanowany wynik prac.
4	Predictable Process	Proces jest zdefiniowany, mierzalny oraz monitorowany. Parametry prac projektowych są zbierane w celu predykcji wyników oraz zapewnienia zaplanowanej jakości.
5	Innovating Process	Proces jest zdefiniowany i ciągle udoskonalany, bazując na istniejących statystykach oraz monitorowaniu przebiegu projektów. Każdy projekt przyczynia się do udoskonalania procesu, identyfikowania oraz usuwania problemów.

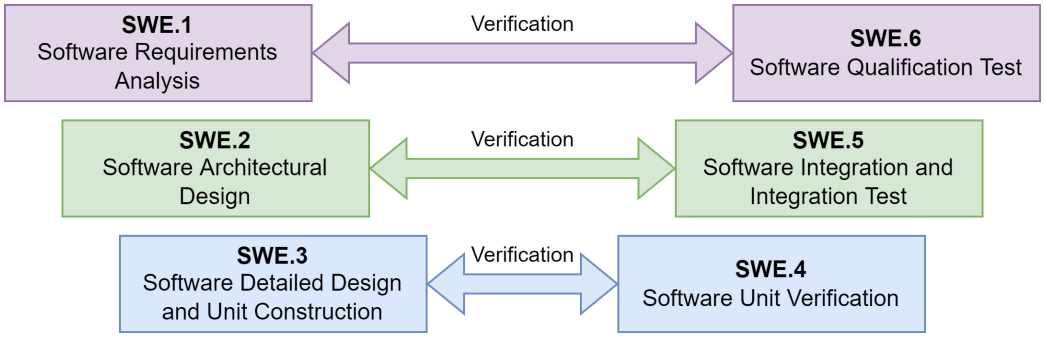
Tablica 2.1. Poziomy dojrzałości procesu tworzenia oprogramowania ASPICE

ASPICE w kontekście ram pracy skupiających się na standardzie AUTOSAR zostały ocenione jako niezasadne.

2.1.2 V-Model

Definicja modelu

V-Model jest to proces wytwarzania systemów, uważany w przemyśle motoryzacyjnym jako „zgodny ze sztuką" (ang. „state of the art"). V-Model przedstawiony na rysunku 2.2 wyznacza sposób realizowania projektów zgodny z jego kształtem – zaczynając od lewej strony litery V, wyznacza kolejne etapy tworzenia produktu, aż do prawej górnej części „V", czyli końcowej walidacji. Model można stosować zarówno do wytwarzania



Rysunek 2.2. Warstwowa walidacja dla modelu V

oprogramowania oraz innych systemów (na przykład elektroniki, lub elementów mechanicznych). Główną ideą modelu jest warstwowa pozioma weryfikacja każdej fazy koncepcyjnej. Oznacza to, że: wysokopoziomowe wymagania klienckie są weryfikowane przez końcowe testy kwalifikacyjne, architektura oprogramowania jest weryfikowana przez testy integracyjne i szczegółowa specyfikacja modułowa jest weryfikowana przez testy jednostkowe. Pozioma weryfikacja została przedstawiona graficznie na rysunku 2.2. W ten sposób każda warstwa prac inżynieryjnych ma odpowiedni sposób sprawdzenia jej poprawności.

Historycznie model V został stworzony w nawiązaniu do kaskadowego modelu tworzenia oprogramowania [16], obecnie uważanego za przestarzały. Istnieje jednak wiele opracowań i implementacji modelu w projektach używających zwinnych technik prowadzenia projektu, np. [17]. Metodyki

zwinne są obecnie stosowane w większości projektów dotyczących oprogramowania w przemyśle motoryzacyjnym, a wybrane sposoby ich wdrożenia zostały opisane przez autora [18].

Wpływ modelu V na jakość i bezpieczeństwo oprogramowania

Podobnie jak w przypadku ASPICE, wpływ modelu V na jakość oprogramowania jest znaczący, ale powiązany z metodami prowadzenia projektów skupiającymi się na zarządzaniu projektami, a nie pracach czysto technicznych. W artykule badawczym [19] widoczny jest również związek ze standardem AUTOSAR, który zawiera dużo bardziej techniczne spojrzenie na proces tworzenia oprogramowania. Kolejna publikacja [20] prezentuje sposoby optymalizacji czasu trwania prac projektowych oraz próby zwiększenia jakości produktu, które opierają się na liczbie defektów obecnych przed i po wprowadzeniu proponowanej metody. Implementacje modelu V są dobrze zdefiniowane i ciągle ulepszane [21]. Bazując na wykonanym przez autora przeglądzie literatury oceniono, że jego zaplanowane badania i analizy są bardziej skupione na technicznym aspekcie tworzenia oprogramowania.

2.2 Standardy technologiczne

2.2.1 Standard AUTOSAR Classic

Pojęcie AUTOSAR ma wiele znaczeń. Przede wszystkim jest to organizacja skupiająca wiele firm z rynku motoryzacyjnego: producentów samochodów, firmy pracujące przy oprogramowaniu, producentów mikrokontrolerów oraz elektroniki i wielu innych firm. Na rysunku 2.3 przedstawiono listę partnerów na rok 2020. W niniejszej pracy skoncentrowano się na „Classic Platform", gdyż obecnie jest to platforma przeznaczona dla sterowników z największymi wymaganiami dotyczącymi jakości, bezpieczeństwa funkcjonalnego oraz cyberbezpieczeństwa. Platforma ta jest używana w systemach czasu rzeczywistego, które posiadają największe wymagania czasowe (często rzędu mili- czy mikrosekund)i ograniczone zasoby sprzętowe, a także są zaklasyfikowane jako najbardziej wpływające na bezpieczeństwo kierowcy – nawet Automotive Safety Integrity Level (ASIL) D. Znaczenie poziomów ASIL zostało opisane w rozdziale 2.3.1. W tabeli 2.2 wyróżniono natomiast

Rysunek 2.3. Partnerzy konsorcjum AUTOSAR (Źródło [22])

przeznaczenie trzech platform: AUTOSAR Classic, AUTOSAR Adaptive oraz niestandaryzowanych elementów. Jak można zauważyć największe

	AUTOSAR Classic	**AUTOSAR Adaptive**	**System bez AUTOSAR**
Wymagania czasowe	Wysokie, rzędu mikrosekund	Średnie, rzędu milisekund	Niskie, rzędu sekund
Kryteria bezpieczeństwa	Wysokie, do ASIL D	Średnie, do ASIL B	Niskie, QM
Wymagana moc obliczeniowa	Niska, 1000 DMIPs	Wysoka, >20000 DMIPs	Wysoka, 10000 DMIPs

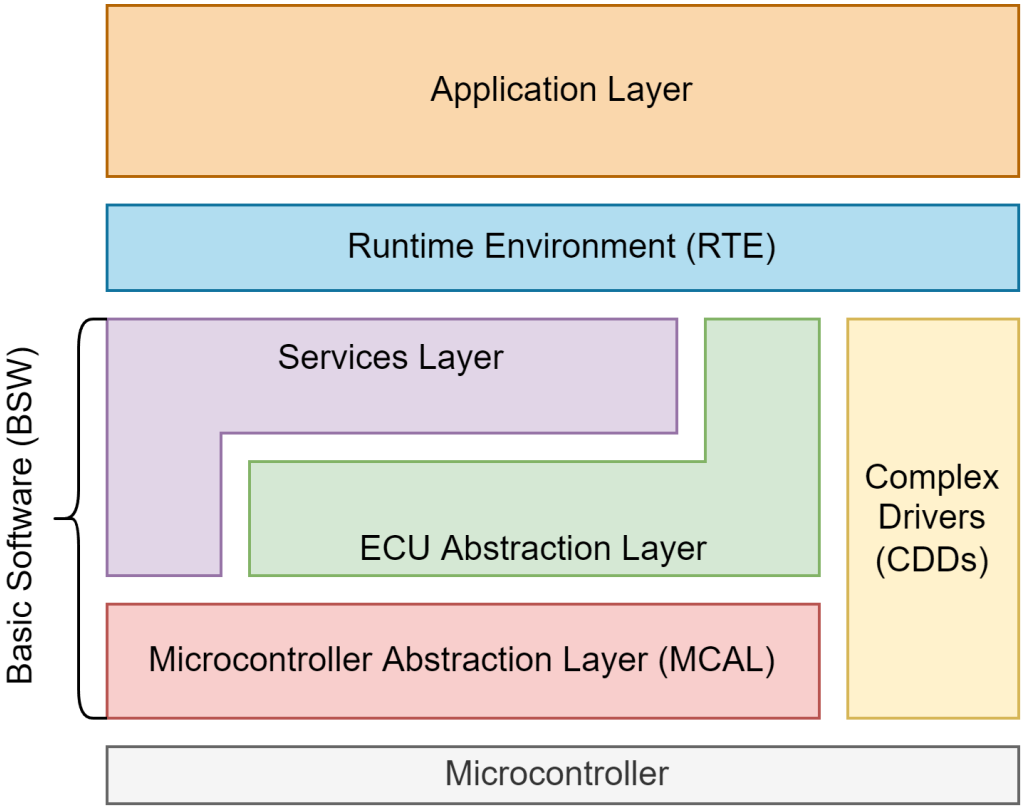
Tablica 2.2. Porównanie platform AUTOSAR

wymagania czasowe, największy wpływ na bezpieczeństwo i największe ograniczenia sprzętowe (najmniejsza dostępna moc obliczeniowa) posiada platforma AUTOSAR Classic. Więcej informacji na temat AUTOSAR można znaleźć w prezentacji [22].

AUTOSAR – Architektura oprogramowania

Pojęcie AUTOSAR oznacza również architekturę oprogramowania określonego przez standard, co przedstawiono na diagramie 2.4.

Architektura ta definiuje trzy główne warstwy oprogramowania:



Rysunek 2.4. Stos AUTOSAR Classic

1. Application Layer – warstwa logiki biznesowej. Implementowane są w niej algorytmy specyficzne dla danego systemu AUTOSAR. Warstwa ta zapewnia tylko częściowe wsparcie dla rozwoju oprogramowania w tym obszarze, nie reguluje dokładnego działania systemu. Głównymi jej elementami są komponenty nazywane Software Component – (SWC), odpowiedzialne na przykład za:

   (a) sterowanie szybami w samochodzie,
   (b) sterowanie ogrzewaniem,
   (c) sterowanie ładowaniem baterii,

   Stanowią one generalnie wszystkie komponenty realizujące specyficzne wysokopoziomowe funkcje danego systemu.

2. Basic Software – (BSW) – warstwa oprogramowania bazowego określonego przez AUTOSAR. Część ta jest bardzo dobrze zdefiniowana przez standard poprzez: określenie wymagań co do każdego modułu, określenie interfejsów, architektury oraz zależności danego modułu. BSW jest dodatkowo podzielona na:

   (a) Services Layer – warstwa standardowych serwisów, takich jak system operacyjny: zarządzanie pamięcią nieulotną, diagnostyką, zarządzanie stanem systemu, monitorowanie oprogramowania itp.,

   (b) ECU Abstraction Layer – warstwa zapewniająca abstrakcję dla sprzętu i peryferiów, niezależnie od ich rodzaju i lokalizacji (celem jest możliwość tworzenia oprogramowania w sposób niezależny od elektroniki wykonawczej),

   (c) Microcontroller Abstraction Layer – warstwa zawierająca sterowniki z bezpośrednim dostępem do sprzętu – rejestrów mikrokontrolera oraz peryferiów,

   (d) Complex Drivers – warstwa realizująca wszystkie niestandardowe elementy systemu, które nie są zdefiniowane przez BSW oraz elementy, które mają bardzo zaawansowane zależności i wysokie wymagania czasowe.

3. Runtime Environment – (RTE) – warstwa zapewniająca odizolowanie BSW od warstwy aplikacji. RTE definiuje serwisy komunikacyjne służące do wymiany danych pomiędzy komponentami logiki biznesowej (pomiędzy SWC) i pomiędzy komponentami logiki biznesowej a stosem BSW. Innymi słowy, dzięki RTE dany komponent może wysłać zapytanie o obecne napięcie na akumulatorze samochodu bez wiedzy, który sterownik posiada ten sygnał. Zadaniem RTE jest pozyskanie tego sygnału i zaraportowanie obecnego napięcia.

AUTOSAR – Tworzenie oprogramowania

AUTOSAR określa również standardy, które są dostępne na stronie internetowej www.autosar.org, takie jak:

1. Software Requirements Specifications (SRS) – dokumenty zawierające zestaw wymagań, które muszą zostać spełnione przez dany komponent,

aby zapewnić zgodność ze standardem AUTOSAR, np.: dokumentacja sterowników analogowo-cyfrowych, diagnostyki, pamięci nieulotnej itp.

2. Software Specification (SWS) – dokumenty zawierające opis sposobu działania danego komponentu oraz jego relacje z innymi warstwami, np. dokumentacja wymagań dla sterowników analogowo-cyfrowych, diagnostyki, pamięci nieulotnej.

3. Technical Reports (TR) – dokumenty zawierające informacje pomocnicze, pomagające poruszać się po elementach standardu, np. lista modułów BSW, lista skrótów, katalog wzorców projektowych.

4. Explanatory Documents (EXP) – dokumenty wyjaśniające dane zagadnienie, np. prezentacja architektury AUTOSAR, prezentacja Virtual Function Bus (VFB).

Dostęp do całej dokumentacji AUTOSAR jest darmowy. Implementacja stosu AUTOSAR Classic wymaga opłacenia i stanowi bardzo obszerne oprogramowanie produkowane tylko przez kilka firm na całym świecie. Proces rozwoju oprogramowania dla systemu wbudowanego składa się z następujących elementów:

- konfiguracja stosu AUTOSAR,
- generowany kod stosu AUTOSAR Classic,
- dostarczony kod stosu AUTOSAR Classic,
- oprogramowanie pisane ręcznie – warstwa aplikacji, Complex Device Driver – (CDD).

Wpływ standardu AUTOSAR Classic na jakość, bezpieczeństwo oraz cyberbezpieczeństwo został określony na podstawie analiz jego poszczególnych warstw. Bazując na 10-letnim doświadczeniu autora w tworzeniu oprogramowania dla systemów wbudowanych, określono moduły, które wykazywały zależności z kryteriami norm ISO 25010, ISO 26262 oraz ISO 21434. Kolejnym krokiem było zdefiniowanie problemów badawczych powiązanych z wybranymi modułami i zagadnień badawczych, które mogły pomóc w ich rozwiązaniu. Zidentyfikowane problemy i zagadnienia badawcze zostały przedstawione w rozdziale 3.

2.2.2 MISRA C

MISRA C to zestaw zasad dotyczących tworzenia oprogramowania, opracowany przez organizację Motor Industry Software Reliability Association (MISRA), używany w przemyśle motoryzacyjnym oraz uważany za zgodny ze sztuką. Stosowanie zasad MISRA C jest wymagane we wszystkich projektach, gdzie stosowany jest standard AUTOSAR, zgodnie z [23]. MISRA C jest bardzo dobrze zdefiniowana, każda zasada jest zakwalifikowana jako jedna z trzech kategorii: „mandatory", „required" lub „advisory", co określa ich priorytet. Dodatkowo dla każdej zasady przedstawione są zalecenia, przykłady oraz uzasadnienie.

MISRA C została poddana analizie przez autora i zaklasyfikowana jako kompletny i bardzo ważny zestaw zasad, w którym nie zauważa się możliwości optymalizacji lub dalszych opracowań.

2.2.3 Metryki HIS

Metryki HIS to metryki dotyczące oprogramowania, które początkowo były ustanowione przez zespół producentów samochodów. Ich głównym celem było zwiększenie utrzymywalności oprogramowania, obniżenie poziomu skomplikowania i zwiększenie jakości. Dokonano autorskiego przeglądu samych metryk i istniejącej literatury, aby ocenić, czy istnieje obszar do realizowania prac badawczych w kontekście jakości i bezpieczeństwa oprogramowania.

Na szczególne zainteresowanie zasługuje pracą [24], gdzie przedstawiono wnikliwe spojrzenie zespołu badaczy na różne metryki oprogramowania, zamieszczając przegląd 38 artykułów i 112 metryk,oceniając je w odniesieniu do norm ISO 25010 oraz ISO 26262. Celem pracy było zapewnienie analizy, która wspomoże wybór właściwej metryki do odpowiedniego zastosowania komercyjnego. W artykule zaprezentowano katalog metryk oraz wykazano ich nawiązanie do atrybutów jakości oprogramowania z normy ISO 25010 – utrzymywalności (ang. maintainability). W pracy [25] podkreślono także wagę metryk i przedstawiono raport liczby naruszeń w badanym projekcie komercyjnym. Obecnie konkretne wartości metryk są wymagane przez producentów samochodów, a sama dokumentacja została zintegrowana ze standardem AUTOSAR.

Po przeprowadzej przez autora analizie literatury zaklasyfikowano

metryki jako kompletne rozwiązanie, podobnie jak MISRA.

2.3 Normy o zasięgu międzynarodowym

2.3.1 ISO26262

Definicja normy i jej składowe

Norma ISO 26262 [5] jest adaptacją normy IEC 61508 [26] dla przemysłu motoryzacyjnego, która adresuje zagrożenia wynikające z błędnego działania systemów Electrical/Electronic (E/E), wykluczając nominalne działanie systemów bezpieczeństwa. Innymi słowy, norma definiuje sposób zachowania w przypadku, gdy którykolwiek z systemów zadziała niezgodnie z jego przeznaczeniem i w ten sposób spowoduje zagrożenie dla użytkownika samochodu.

Poziomy ASIL i ich znaczenie

Opisane wcześniej ryzyko jest określane przy użyciu tak zwanych poziomów ASIL (od ASIL A do ASIL D). System E/E może być określony jako Quality Management (QM), co oznacza, że nie ma wpływu na bezpieczeństwo funkcyjne w postrzeganiu normy ISO 26262. Jeżeli wpływ na bezpieczeństwo istnieje, to najniższą klasyfikacją jest poziom ASIL A, a najwyższą ASIL D. Przykładami systemów ASIL A są systemy oświetlenia wnętrza, których awaria mogłaby oślepić kierowcę. Przykładami systemów ASIL D, są systemy powiązane z układem jezdnym, hamulcami, poduszkami powietrznymi itp., których awaria mogłaby doprowadzić do wypadku i zagrożenia życia użytkownika samochodu.

Opis głównych elementów normy

Norma ISO 26262 składa się z wielu części, które określają m.in. słownik pojęć, elementy powiązane z zarządzaniem, fazą koncepcyjną projektów, projektowaniem elementów elektronicznych. Proces analizy bezpieczeństwa funkcyjnego zaczyna się od określenia opisów przedmiotów analiz, określenia ich celów bezpieczeństwa (ang. safety goals), przygotowania wymagań powiązanych z bezpieczeństwem funkcjonalnym, realizowania zidentyfikowanych elementów zgodnie z modelem V, a kończy

na opisaniu i oddaniu całościowej dokumentacji dotyczącej bezpieczeństwa funkcjonalnego. W kontekście niniejszej rozprawy, główną częścią normy jest część 6, czyli „Product development at the software level".

Część ta została poddana analizie przez autora, a na jej podstawie odniesiono proponowane części metody wspomagającej proces tworzenia oprogramowania do wymagań stawianych przez ISO 26262-6. Każde z proponowanych rozwiązań zawiera odniesienie do wymagań normy, na które wpływa.

2.3.2 ISO 25010

Definicja normy i jej składowe

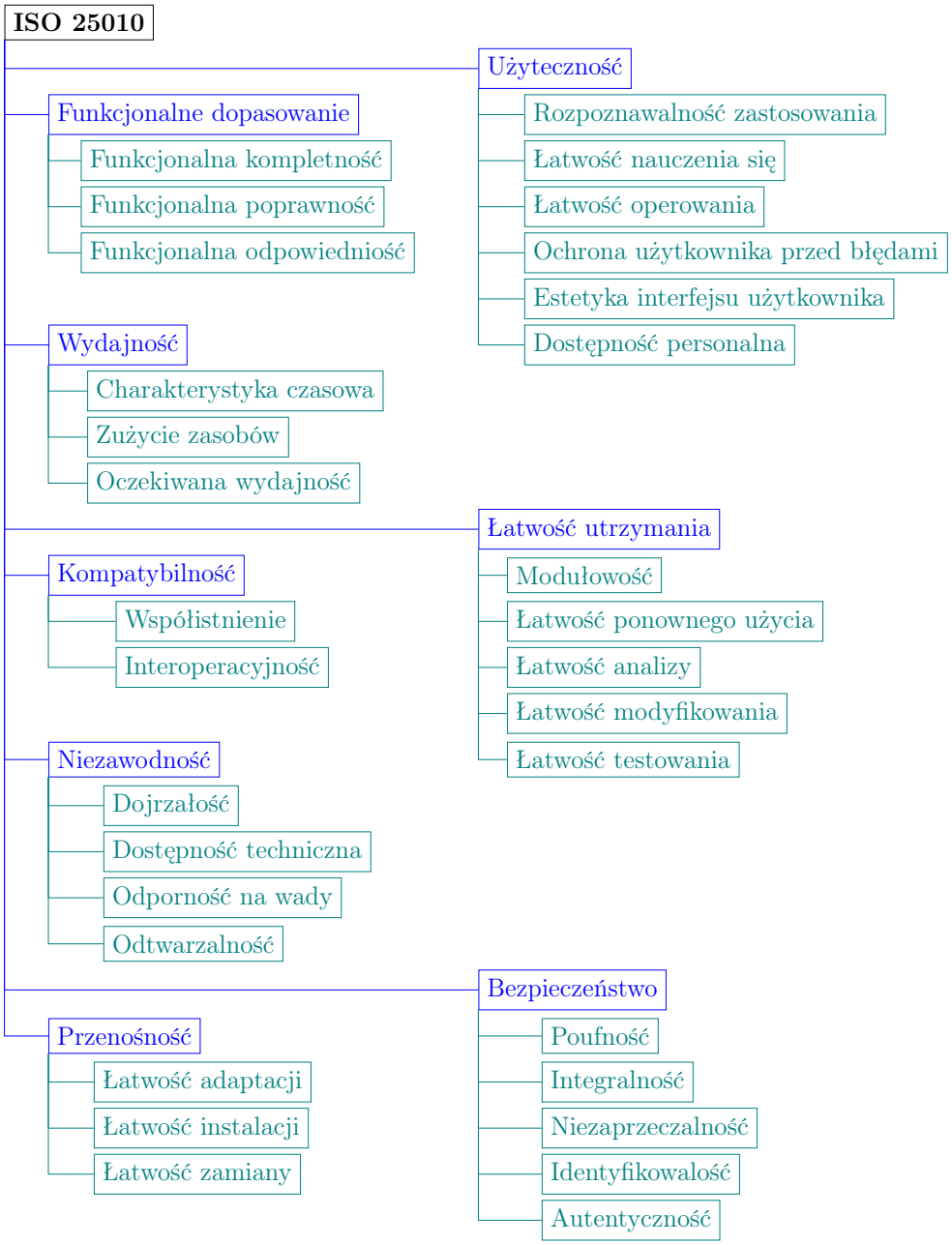
Norma ISO 25010 [4] to norma zawierająca model jakości oprogramowania, która określa jakie właściwości oprogramowania można brać pod uwagę w celu ewaluacji danego produktu. W kontekście normy ISO 25010, jakość oznacza stopień, w jakim system zaspokaja wymagania i potrzeby interesariuszy, przez co zapewnia wartość. Opisane wymagania i potrzeby są reprezentowane przez model jakości opisywany w normie. Składa się on z 8 kategorii. ISO 25010 jest normą dotyczącą oprogramowania, która nie dotyczy wyłącznie przemysłu motoryzacyjnego. W praktyce wybrane elementy normy stanowią podstawę do oceny jakości konkretnych warstw oprogramowania w samochodach osobowych.

Na bazie własnego doświadczenia komercyjnego oszacowano, że norma będzie odpowiednia w ocenie jakości oprogramowania. Podobne opinie zostały przedstawione przez autorów prac [27], czy [28], wymieniając ISO 25010 jako podstawę oceny jakości oprogramowania w przemyśle motoryzacyjnym.

Opis głównych elementów normy

Kategorie oraz elementy normy są przedstawione na rysunku 2.5 i stanowią tłumaczenia autorskie z wersji angielskiej. Dokładne opisy każdego z elementów są dostępne w [4].

Główne kategorie normy są także głównymi priorytetami konsorcjum AUTOSAR [9]. Łatwość ponownego użycia, przenośność, bezpieczeństwo, niezawodność, czy wydajność – te kategorie pozostają kluczowymi powodami rozwoju i używania standardu AUTOSAR. Podczas tworzenia



Rysunek 2.5. Kategorie oraz elementy normy jakości oprogramowania – ISO 25010

oprogramowania elementy normy ISO 25010 są weryfikowane w różny sposób: poprzez inspekcje kodu, statyczną analizę kodu, wielopoziomowe testowanie, automatyczne narzędzia sprawdzające składnię, czy stosowanie zasad nazewnictwa. Norma określa wymagania i ich elementy, a odpowiedzialnością organizacji jest odpowiednie zapewnienie ich implementacji.

Każde z proponowanych przez autora rozwiązań składających się na metodę wspomagającą proces tworzenia oprogramowania zawiera odniesienia do elementów normy, których dotyczy.

2.3.3 ISO 21434

Definicja normy i jej składowe

Norma ISO 21434 [6] adresuje aspekty cyberbezpieczeństwa w systemach E/E w samochodach osobowych. Jej celem jest określenie działań zgodnych ze sztuką w kontekście rosnących wymagań dotyczących cyberbezpieczeństwa. Używanie ISO 21434 pozwala na wprowadzenie polityk i procesów cyberbezpieczeństwa, zarządzanie ryzykami cyberbezpieczeństwa oraz dbanie o kulturę cyberbezpieczeństwa w organizacji. W obecnie produkowanych samochodach, które są stale podłączone do internetu, temat cyberbezpieczeństwa stał się jednym z najważniejszych aspektów. Powszechnym obowiązkiem każdego producenta samochodów lub poddostawcy jest posiadanie dojrzałej polityki i procesu cyberbezpieczeństwa. Nie ogranicza się to tylko do testów penetracyjnych czy audytów cyberbezpieczeństwa. Zgodnie z normą ISO 21434 wymagane jest posiadanie wielowarstwowej struktury dotyczącej cyberbezpieczeństwa. Norma ta wyróżnia zarządzanie na poziomie organizacji, projektu, produktu, konceptu oraz wsparcie po-produkcyjne. Pierwsze wydanie normy zostało opublikowane w 2021 roku, więc większość organizacji jest dopiero w trakcie jej implementowania i realizacji wymagań z nią związanych.

Opis głównych elementów normy

Norma składa się z następujących elementów:

1. Ogólne elementy cyberbezpieczeństwa.
2. Zarządzanie cyberbezpieczeństwem na poziomie organizacji.
3. Zarządzanie cyberbezpieczeństwem na poziomie projektu.
4. Podział aktywności dotyczących cyberbezpieczeństwa.
5. Ciągłe działania dotyczące cyberbezpieczeństwa.
6. Faza koncepcji cyberbezpieczeństwa produktu.
7. Faza wdrażania cyberbezpieczeństwa produktu.
8. Faza walidacji cyberbezpieczeństwa produktu.
9. Faza produkcyjna.
10. Faza operacyjna oraz faza wsparcia cyberbezpieczeństwa produktu.
11. Faza końca wsparcia cyberbezpieczeństwa produktu.
12. Analiza zagrożeń i metody zarządzania ryzykami.

W ramach niniejszej rozprawy opisującej proces tworzenia oprogramowania, najważniejsze części normy, na których skupiono się to: 6, 7, 8, czyli „Faza koncepcji cyberbezpieczeństwa produktu", „Faza wdrażania cyberbezpieczeństwa produktu" oraz „Faza walidacji cyberbezpieczeństwa produktu".

Przeanalizowano normę i przypisano proponowane części metody wspomagającej proces tworzenia oprogramowania do wymagań stawianych przez ISO 21434. Każde z proponowanych rozwiązań zawiera odniesienie do elementów normy, których dotyczy.

Rozdział 3

Identyfikacja problemów występujących w procesie tworzenia oprogramowania

3.1 Problemy wynikające z braku skalowalności oprogramowania AUTOSAR

Z powodu powiększającego się zapotrzebowania na dane używane np. dla systemów wspierania kierowcy Advanced Driver Assistant System (ADAS) lub systemów komunikacyjnych TCU, systemy wbudowane stosowane w samochodach wykorzystują coraz więcej oprogramowania.

3.1.1 Problem P1 – skalowalność ekstraktów diagnostycznych

Istota problemu

Każda jednostka sterująca w samochodzie musi wspierać serwisy diagnostyczne opisane normą międzynarodową ISO 14229 [29]. W zależności od poziomu skomplikowania oraz roli danego sterownika, serwisy diagnostyczne mogą ograniczać się do kilku Diagnostic Trouble Codes (DTC) (potencjalnych błędów, takich jak np. niskie ciśnienie w oponie) oraz kilkunastu Data Identifier (DID) (parametrów diagnostycznych, takich jak np. przebieg samochodu). W przypadku większych systemów mogą to

być dziesiątki DTC i setki DID. Plik opisowy AUTOSAR XML – (ARXML), zawierający zestaw informacji na temat wszystkich danych diagnostycznych (wszystkie DID oraz DTC) jest nazywany „ekstraktem diagnostycznym". Ekstrakt jest przygotowywany przez producenta samochodu i udostępniany dostawcom poszczególnych sterowników, jako podstawa do realizacji funkcji diagnostyki. Każdy serwis określony w ekstrakcie musi być odpowiednio zaimplementowany przez inżynierów pracujących przy danym sterowniku ECU. Im większy ekstrakt diagnostyczny, tym więcej ręcznie pisanego kodu. Ekstrakty diagnostyczne definiujące setki danych diagnostycznych powodują problemy, takie jak niska utrzymywalność i niezawodność kodu oraz długi czas potrzebny na implementację i testowanie.

Zagadnienie badawcze ZB1

Zagadnienie badawcze powiązane ze zidentyfikowanym problemem **P1** brzmi:

ZB1 – Jak rozwiązać problem skalowalności ekstraktów diagnostycznych w projektach AUTOSAR?

Sposób rozwiązania w świetle dotychczasowych badań

Skalowalność jest kluczowym aspektem w każdej dziedzinie oprogramowania, na który wpływają wielowarstwowe czynniki. Zwiększanie skalowalności systemowej w celu minimalizacji kosztu elektroniki zostało przedstawione w [30], gdzie poszukiwano odpowiedniej architektury dla systemów wbudowanych służących do edukacji, które mogłyby być łatwo zastosowane dla dużej liczby stanowisk. Wskazana analiza dotyczyła całego systemu – od mikrokontrolera przez warstwę transportową, aż do serwerów. Rozwiązanie skupiało się jednak na warstwie sprzętowej, co w przypadku zagadnienia badawczego **ZB1** nie stanowi możliwego rozwiązania, gdyż warstwa sprzętowa tylko w pomniejszym stopniu wpływa na problem. Czas integracji ekstraktu nie jest zależny od sprzętu. Badania znacznie bliższe zidentyfikowanemu problemowi prezentuje artykuł [31], w którym autorzy przedstawili dwa zagadnienia badawcze:

- wymagania jakościowe w architekturze oprogramowania dla domeny układu napędowego,
- sposób ciągłego monitorowania jakości architektury oprogramowania AUTOSAR.

Badacze nawiązali do norm ISO 25010 [4] oraz ISO 26262 [5]. Propozycją rozwiązania było podejście oparte na automatycznej generacji i monitorowaniu metryk na poziomie systemu oraz komponentów oprogramowania. Rozwiązanie to jest bardzo użyteczne i zgodne z automatycznym podejściem do tworzenia oprogramowania, ale w kontekście zagadnienia badawczego **ZB1**, może stanowić tylko częściową metrykę opisującą skalę problemu, a nie jego rozwiązanie.

Z powodu wysokiego stopnia skomplikowania oprogramowania w motoryzacji, każde automatyczne podejście przyczynia się do zwiększania jakości i skalowalności, jak na przykład [32], gdzie określono przez autorów sposób automatycznego generowania plików opisowych dla komponentów AUTOSAR powiązanych z architekturą systemów wpływających na bezpieczeństwo kierowcy (zgodnie z normą ISO 26262 [5]). Artykuł miał charakter aplikacyjny i wykorzystywano w nim metodę integrującą wiele narzędzi używanych podczas tworzenia oprogramowania opartego na projekcie realizowanym zgodnie z podejściem „model-based design". Metoda zaprezentowana w [32] wykorzystywała automatyczne generowanie powiązań pomiędzy interfejsami, graficzne przedstawienie zależności oraz wykorzystywała format wymiany danych ARXML. Rozwiązanie to skupiało się na aspektach identyfikowalności (ang. traceability) i potwierdziło zasadność zagadnienia badawczego **ZB1**. Artykuł świetnie przedstawił problemy skalowalności AUTOSAR, lecz zabrakło w nim bezpośredniej relacji do generowania i walidacji kodu źródłowego – skupiony jest on bowiem na architekturze i wymaganiach. W przypadku stosu diagnostycznego, który był głównym elementem zagadnienia badawczego **ZB1**, potrzeba analizy i badań na niższych warstwach abstrakcji.

W kolejnym artykule – [33], podkreślano, że AUTOSAR nie jest wystarczający w zakresie zarządzania zasobami, w szczególności zużycia pamięci. Zaproponowano w nim zastosowanie grupowania jak największej liczby powiązanych elementów we wspólnych komponentach (ang. „component-based approach"). Wyróżniono ponadto 5 specyficznych elementów, dotyczących odpowiednio:

- warstwy komponentu,
- warstwy interfejsu,
- warstwy portu,
- warstwy konektora,
- warstwy platformy.

W celu zastosowania transformacji architektury AUTOSAR, zaproponowano przez autorów pionowe grupowanie warstw (ang. vertical layer slicing). Dotyczy to stosu komunikacji (FlexRay) oraz warstwy aplikacji. Wyniki są obiecujące – zużycie pamięci oraz maksymalny czas wykonania jest znacznie zmniejszony. Główną wadą zaprezentowanego rozwiązania jest manualne podejście do problemu. Stosując tę metodę, każdy projekt musiałby być analizowany osobno przez zespół ekspertów w celu pogrupowania elementów we wspólne komponenty.

Ostatnim artykułem przeanalizowanym w kontekście zagadnienia badawczego **ZB1**, był [34]. Celem jego autorów było skupienie się na automatycznej analizie czasowej przetwarzania sygnałów, opartej na kolejnych zdarzeniach propagowanych w stosie oprogramowania (ang. event chains). Elementem nawiązującym do automatyzacji stosu diagnostycznego jest automatyzacja na poziomie plików opisowych ARXML, które zawierają definicje sygnałów, takich jak np. informacja o stanie stacyjki. We wspomnianym artykule wykorzystano narzędzia do importowania baz sygnałowych i użyto ich do obserwacji czasu propagacji sygnału. Jest to bardzo ciekawe rozwiązanie, pozytywnie wpływające na jakość oprogramowania i ułatwiające proces jego tworzenia. Nie jest jednak możliwe do zastosowania w przypadku problemu **P1**, ponieważ zaproponowane w [34] rozwiązanie dotyczy innej części stosu. Można je zastosować do analiz sygnałów, ale nie pomaga w przypadku problemu powiązanego ze stosem diagnostycznym.

Podsumowując stan obecnej wiedzy można stwierdzić, że problem skalowalności oprogramowania AUTOSAR jest analizowany z wielu perspektyw. Opracowania wykorzystują automatyzacje oparte na plikach ARXML, rozwiązania oparte na metodzie „model-based design", próbujące zwiększyć jakość i bezpieczeństwo oprogramowania w kontekście norm ISO 25010 [4] oraz ISO 26262 [5]. Nie zostały jednak odnalezione prace rozwiązujące problem skalowalności ekstraktów diagnostycznych.

3.2 Problemy sprzętowe

3.2.1 Problem P2 – ograniczone zasoby sprzętowe

Istota problemu

Oprogramowanie stosowane w jednostkach wbudowanych (ang. embedded software), jest w dużym stopniu powiązane z platformą sprzętową, na którą jest przeznaczone. Przemysł motoryzacyjny kładzie nacisk na minimalizację kosztów elektroniki, gdyż w świetle wolumenu produkcyjnego w setkach tysięcy lub milionach, każdy zaoszczędzony cent znacząco wpływa na zyski. Z tego powodu każdy projekt jest realizowany na jak najmniejszym mikrokontrolerze, który spełni wymagania klienta i zrealizuje wszystkie potrzebne funkcje. Zmiana platformy sprzętowej podczas trwania projektu wiąże się z dużymi kosztami i musi zostać bardzo dobrze zaplanowana. Z tego względu oprogramowanie często jest narażone na opisane w niniejszym rozdziale problemy wynikające z ograniczonych zasobów sprzętowych. Należą do nich:

- ograniczona pojemność pamięci Random Access Memory (RAM),
- ograniczona pojemność pamięci Read Only Memory (ROM),
- ograniczona pojemność pamięci Non-Volatile Memory Manager (NvM),
- duże obciążenie jednostki obliczeniowej Central Processing Unit (CPU),
- znaczące obciążenie magistral komunikacyjnych (Ethernet, Serial Peripheral Interface (SPI), Universal asynchronous receiver-transmitter (UART), Controller Area Network (CAN) itp.).

Sam stos AUTOSAR stanowi duże obciążenie dla systemu, a ogromna ilość przetwarzanych danych i realizowanych funkcji, cały czas stanowi wyzwanie dla wybranego mikrokontrolera.

Zagadnienie badawcze ZB2

Zagadnienie badawcze powiązane ze zidentyfikowanym problemem **P2** brzmi:

ZB2 – Jak minimalizować zużycie zasobów sprzętowych wybranych obszarów oprogramowania w projektach AUTOSAR?

Sposób rozwiązania w świetle dotychczasowych badań

Optymalizacja zużycia zasobów mikrokontrolera jest problemem, z którym zmaga się wiele przemysłów używających systemów wbudowanych. W artykule [35] askupiono się na optymalizacji pamięci dla architektury mikrokontrolerów Advanced RISC Machine (ARM), wykonując analizę istniejących technik optymalizacji kodu oraz zaproponowano nową unikalną metodę, która pozwoliła uzyskać rozmiar kodu wynikowego mniejszy o 5% od standardowej kompilacji przy użyciu kompilatora komercyjnego. Jest to ciekawe rozwiązanie z powodu jego uniwersalności. Architektura ARM jest obecnie bardzo popularna również w przemyśle motoryzacyjnym.

Kolejną ciekawą publikacją jest [36], gdzie przeprowadzono badania bateryjnych systemów wbudowanych, w których priorytetem jest ekstremalnie niski pobór energii. Badania te objęły również architekturę ARM, prezentując nowatorską optymalizację kompilatora, która statycznie przesuwa wybrane bloki pamięci z ROM do RAM. Udowodniono w nich, że pamięć RAM ma znacznie niższe zużycie energii. Rezultaty były znaczące – zmniejszono o 41% średni pobór energii, dzięki czemu urządzenie mogło działać o 32% dłużej na baterii. Optymalizacja była kompromisem między prędkością wykonania kodu, która spadła średnio o 19,5% a zużyciem energii. W konkretnych przypadkach przedstawionych przez autorów, sposób ten znalazł zastosowanie. Systemy motoryzacyjne także działające na baterii, również wymagają niskiego poboru energii, aczkolwiek prędkość wykonania kodu stanowi niestety jeszcze większy priorytet.

Przechodząc do analizy publikacji powiązanych konkretnie z optymalizacją systemów realizowanych zgodnie ze standardem AUTOSAR, można wymienić publikację [37], gdzie celem badaczy było skupienie się na stosie komunikacyjnym w kontekście systemów wspomagania kierowcy ADAS, wymaganych obecnie przez prawo. Motywacją dla powstania artykułu była rosnąca popularność skomplikowanych systemów, takich jak np. protokołu komunikacyjnego Vehicle-2-Anything (V2X), przy którym różnego rodzaju czujniki: LIDARy, czujniki podczerwone, czujniki ultradźwiękowe, lasery, produkują ogromne ilości danych, które muszą być przetwarzane i wymieniane między różnymi jednostkami ECU w samochodzie. We wspomnianej publikacji przeanalizowano poszczególne warstwy stosu komunikacyjnego od aplikacji, poprzez warstwę zarządzania sygnałami RTE, stos oprogramowania standardowego BSW, moduły komunikacyjne, aż do warstw sprzętowych dla magistrali CAN i Ethernet.

Propozycją autorów jest implementacja modułu Private Data Adapter (PDA), który jest podzielony na dwie sekcje: CDD i SWC i realizuje komunikację przy użyciu prywatnych kanałów. Moduły te bezpośrednio wymieniają wiadomości z modułem zarządzającym sygnałami komunikacyjnymi Pdu Router – (PduR), używają specjalnych nagłówków oraz realizują dodatkowe funkcje, jak walidacja danych. Rezultatem propozycji badaczy jest znacznie zmniejszony czas wykonania powiązany z transmisją oraz odbiorem danych w porównaniu do standardowego rozwiązania AUTOSAR.

Kolejnym artykułem w kontekście problemu **P2** jest [38], omawiający optymalizację funkcji wykonywanych w systemie przetwarzania równoczesnego. Propozycją autorów, celem minimalizowania ilości pamięci RAM, jest specjalny sposób:

- przypisywania kolejności funkcji w danym zadaniu systemu operacyjnego,
- przypisywania ustawień wywłaszczania dla funkcji,
- realizowania zabezpieczenia zmiennych w kontekście dostępu z wielu wątków.

W tym przypadku propozycja ta dotyczy konkretnie stosu. Wykorzystano w niej specjalne oprogramowanie, aby przy określonych ograniczeniach obliczyć zalecane wartości wymienionych parametrów. Rezultaty są przedstawione jako „optymalne", biorąc pod uwagę funkcję celu minimalizującą ilość wykorzystanego stosu.

Temat optymalizacji czasowej jest bardzo dokładnie omawiany w książce [39], gdzie została podjęta przez autora analiza czasowa oprogramowania w przemyśle motoryzacyjnym. Przedstawiono w niej szczegółowo wpływ różnych czynników na czas wykonywania kodu, zaczynając od procesu budowania oprogramowania, adresowania, architektury mikrokontrolerów, systemów operacyjnych, wielordzeniowych systemów oraz innych krytycznych elementów. Książka ta stanowi monografię dotyczącą pomiarów czasowych. Jej autor to również właściciel firmy oferującej oprogramowanie pomiarowe służące do analizy czasowej systemów opartych na standardzie AUTOSAR, które było używane w projektach komercyjnych opisanych w niniejszej rozprawie doktorskiej. W [39] nie znalazły się nowatorskie elementy naukowe, lecz stanowi ona bardzo dobre

podsumowanie całego obszaru wiedzy powiązanego z obciążeniem mikrokontrolera i pomiarami czasowymi.

Podsumowując przegląd literatury w kontekście problemu **P2** oraz zagadnienia badawczego **ZB2**, istnieje wiele publikacji powiązanych z optymalizacją i analizą zasobów sprzętowych, ale są one głównie powiązane z warstwą aplikacji, systemem operacyjnym lub specyficznymi narzędziami modyfikującymi proces kompilacji kodu. Celem autora niniejszej rozprawy było skupienie się na zbadaniu możliwości optymalizacji w standardowych elementach stosu AUTOSAR w modułach warstwy oprogramowania bazowego BSW. Stanowi to uzupełnienie prac powiązanych z warstwą aplikacji lub z systemem operacyjnym. Dzięki temu, że AUTOSAR jest standardem uznawanym jako aktualny stan techniki w przemyśle motoryzacyjnym, propozycje rozwiązania problemu **P2** mogą być wykorzystane w większości projektów, które używają danej warstwy. W ramach niniejszej pracy skoncentrowano się na tych warstwach, które bezpośrednio wpływają na jakość oraz bezpieczeństwo oprogramowania w kontekście norm o zasięgu międzynarodowym – ISO 26262, ISO 25010 oraz ISO 21434.

3.3 Problemy wynikające ze złożoności oprogramowania

3.3.1 Problem P3 – wysoka złożoność oprogramowania

Istota problemu

Złożoność oprogramowania w pojazdach samochodowych rośnie od lat w zastraszającym tempie. Używanych jest coraz więcej modułów, warstw, bibliotek, funkcji. Producenci samochodów prześcigają się w innowacyjnych rozwiązaniach wspierających kierowców, zabawiających pasażerów, dostarczających komfortu i zwiększających bezpieczeństwo. W momencie cyfryzacji samochodów producenci nie mieli wystarczająco rozwiniętych działów oprogramowania, więc w większości zdali się na poddostawców. Działali jako integratorzy oprogramowania dostarczanego przez wiele firm. AUTOSAR zapewnił bardzo dobre narzędzie w postaci procesu, formatu wymiany danych ARXML oraz innych elementów standardu wspierających ujednolicenie architektury, aczkolwiek dalej oprogramowanie wszystkich

ECU pochodzi od setek firm. Im więcej oprogramowania znajduje się w samochodach, tym bardziej firmy motoryzacyjne rozwijają swoje działy programistyczne. Zgodnie ze słynnym zdaniem Watts S. Humphrey'a, nazywanego często ojcem jakości oprogramowania: „Every business is a software business", każdy biznes obecnie jest biznesem powiązanym z oprogramowaniem. Firmy motoryzacyjne rozpoznają ten trend i starają się samemu rozwijać coraz więcej oprogramowania używanego w ich samochodach, ograniczając liczbę poddostawców, aby zmniejszyć liczbę interfejsów, odrębnych procesów dostaw, odrębnych podsystemów, działających negatywnie na końcową jakość produktu. Nie jest to jednak łatwe, ani możliwe do zrealizowania w szybkim czasie – dalej na całe oprogramowanie w samochodzie składają się setki modułów. Problemy wynikające z ogólnego problemu **P3** zostały przeanalizowane przez autora rozprawy, a główne ich czynniki przedstawiono w tabeli 3.1

Zagadnienie badawcze ZB3

Zagadnienie badawcze powiązane ze zidentyfikowanym problemem **P3** brzmi:

ZB3 – Jak minimalizować negatywne skutki wynikające z wysokiej złożoności oprogramowania?

Sposób rozwiązania w świetle dotychczasowych badań

Złożoność oprogramowania jest problemem, który może być analizowany w różnych warstwach oraz z wielu perspektyw, począwszy od architektury oprogramowania z naciskiem położonym na ciągłe zmiany oprogramowania.

W artykule [40] zaproponowano specjalne metryki dotyczące wielkości kodu oraz jego efektu na poziom skomplikowania oprogramowania.

Publikacja [41] prezentuje nowatorską metodę optymalizacji architektury poprzez wykorzystanie modelowania i analizy eksperymentalnej. Metoda ta najpierw w szybki sposób – przy użyciu modelowania systemu generuje proponowaną architekturę, która w drugim kroku może być dalej optymalizowana w wolniejszy, ale bardziej szczegółowy sposób, przy użyciu analizy działania oprogramowania oraz wiedzy eksperckiej. Dzięki takiemu dwuetapowemu podejściu do optymalizacji architektury systemów wbudowanych, udało się uzyskać zarówno dobre rezultaty w kryteriach wydajności oraz dokładności tych dwóch parametrów, które często są

Czynnik	Opis
Duża liczba interfejsów i podsystemów	Niestety koncept programowania defensywnego, które zawsze nakazuje założenie niewłaściwego użycia modułu, nie zawsze jest stosowany. Rezultatem tego jest fakt, że każdy interfejs stanowi barierę często generującą błędy. Dane przekazywane przez interfejs mogą być źle interpretowane, źle skalowane, wykraczać poza zakres zmiennej itp.
Rozproszenie zespołów inżynieryjnych po całym świecie	Oprogramowanie tworzone w wielu różnych kulturach, strefach czasowych, sposobach analizy problemów przekłada się na wiele stylów pisania, wiele różnych sposobów implementacji algorytmów oraz poziomów zaawansowania i skomplikowania kodu. Zwiększa to poziom skomplikowania oraz zmniejsza łatwość analizy.
Brak wystarczającej liczby specjalistów na rynku	Wysoki poziom skomplikowania rozwiązań wymaga wiedzy i doświadczenia inżynierów biorących udział w procesie rozwoju oprogramowania. Niestety, rynek nie jest w stanie zapewnić zespołów złożonych wyłącznie z osób o wystarczającym doświadczeniu. Dodatkowym aspektem jest tutaj ekonomia – projekt z samymi bardzo doświadczonymi inżynierami mógłby się nie opłacać. Rezultatem pracy niedoświadczonych inżynierów jest dług techniczny (czyli zbiór problemów kumulujących się podczas prac projektowych) oraz niska jakość oprogramowania.
Duża liczba linii kodu	Na całość oprogramowania w samochodzie składają się setki milionów linii kodu. Każdy moduł i każda funkcja muszą być odpowiednio przeanalizowane, przetestowane i zabezpieczone. Im więcej kodu, tym więcej newralgicznych miejsc z potencjalnymi błędami.

Tablica 3.1. Negatywne czynniki problemu wysokiej złożoności oprogramowania.

przeciwstawne. Praca nie odnosi się jednak do systemów opartych na standardzie AUTOSAR, który zapewnia ogólną architekturę systemu, przez co podana przez badaczy metoda nie znajduje tutaj zastosowania.

Próbę zmniejszenia złożoności skomplikowania oprogramowania poprzez automatyczną refaktoryzację kodu źródłowego przedstawiono w artykule [42]. Dokonano w nim analizy metryki, takich jak liczba linii kodu czy złożoność cyklomatyczna [43], lecz oceniono je jako niepasujące do ewaluacji hierarchicznego oprogramowania. Na tej podstawie zaproponowano autorski sposób oceny skomplikowania, który pozwala na stwierdzenie, czy zastosowana przez autorów metoda automatycznej refaktoryzacji przynosi pozytywne rezultaty. Zaproponowaną metodę przetestowano na sterowniku zarządzania baterii BMS. Rezultaty okazały się pozytywne – udokumentowano o 7,51% mniejsze skomplikowanie kodu i o 17,99% mniejsze skomplikowanie połączeń między interfejsami. Rozwiązanie to jednak jest dosyć dyskusyjne, zaproponowana metoda ewaluacji faktycznie może oddawać poziom skomplikowania, ale możliwe było wykazanie związku pomiędzy jakością, bezpieczeństwem, czy łatwością analizy oprogramowania. Nie można jednak jasno wykazać pozytywnego wpływu zaprezentowanej metody na elementy, które są kluczowe dla oprogramowania motoryzacyjnego. Dodatkowo, zasugerowane modyfikacje dotyczące struktury modułów oprogramowania nie są możliwe do zastosowania w systemach opartych na standardzie AUTOSAR Classic, gdyż określa on konkretną architekturę.

Kolejnym ciekawym artykułem jest [44], w którym celem autorów była analiza parametrów oprogramowania powiązanego z niezawodnością i bezpieczeństwem – prawdopodobieństwo wystąpienia awarii w relacji do czasu wykonania. Po dokonaniu analizy badawczej zarzucono, że wymienione parametry są niekompletne i zaproponowano własny sposób oceny niezawodności oparty na skomplikowaniu aplikacji, efektywności testów oraz analizie środowiska operacyjnego. Głównym obszarem badań było skomplikowanie oprogramowania w odniesieniu do jego niezawodności. Poza liczbą linii kodu i złożonością cyklomatyczną [43], wymieniono dodatkowo miarę złożoności Halsteada [45] oraz ilość przepływających danych [46]. W oparciu na pomiarach przy użyciu wymienionych sposobów argumentowano, że z powodu skomplikowania oprogramowania, są potrzebne coraz bardziej obszerne sposoby analizy niezawodności.

Artykuł ten zdecydowanie potwierdza istnienie problemu **P3**, pomimo że został napisany w 2007 roku. Od tego czasu złożoność oprogramowania znacząco się zwiększyła.

Podobną analizę przedstawia praca [47], w której zaprezentowano związek poziomu skomplikowania z liczbą linii kodu. Użyto w tym celu definicji skomplikowania z publikacji [48], jako miary zużycia zasobów systemu wykorzystanych na prowadzenie interakcji z modułem oprogramowania podczas realizowania konkretnego zadania. Wpływ liczby linii kodu został przedstawiony w kilku aspektach:

- zwiększenie rozmiaru oprogramowania,
- zwiększenie liczby potencjalnych błędów,
- zwiększenie kosztów produktu.

Przedstawiono ponadto przegląd literatury i dowiedziono zasadności analizy liczby linii kodu.

Bardziej rozbudowaną listę metod analizy oprogramowania przedstawiono w publikacji [49], gdzie uwzględniono szereg metryk, podkreślając ważność ich stosowania i wskazując praktyczne użycie na przykładowym programie w celu objaśnienia. Niezwykle interesującą tezę zmuszającą do dyskusji zaprezentowano w pracy [50], stwierdzając istnienie negatywnej korelacji pomiędzy poziomem skomplikowania oprogramowania oraz wysiłkiem włożonym w jego rozwój. Udało się dowieść tej nieoczywistej relacji w konkretnych warunkach i odpowiedzieć na zadane zagadnienia badawcze. Odpowiedzi były m.in. powiązane ze zdolnościami inżynierów biorących udział w bardziej skomplikowanych projektach. Im większy projekt, tym bardziej doświadczeni programiści oraz analitycy są potrzebni. Ta relacja neguje wpływ skomplikowania oprogramowania na wysiłek potrzebny do jego realizacji.

Opracowując publikacje powiązane z AUTOSAR i przemysłem motoryzacyjnym, można wymienić artykuł [51], w którym zaprezentowano wzorce dla modułu RTE jako narzędzia do zminimalizowania negatywnych skutków wysokiego stopnia skomplikowania oprogramowania. Głównymi zaletami zaproponowanego rozwiązania jest jego uniwersalność oraz wspomaganie procesu projektowania komponentów oprogramowania. Nie przedstawiono jednak większej liczby elementów pozwalających ocenić zaproponowaną metodę.

W przypadku artykułu [52] skupiono się na metodzie pozwalającej ocenić skomplikowanie ograniczeń czasowych i strukturalnych dla systemów wbudowanych, stosowanych w przemyśle motoryzacyjnym na poziomie ich specyfikacji. Uwzględniono w niej standard AUTOSAR i wykorzystano Unified Modeling Language (UML), aby wesprzeć analityków w określaniu ograniczeń komponentów oprogramowania. Metoda opiera się na dwóch fazach: fazie specyfikacji funkcyjnej oraz fazie specyfikacji komponentów oprogramowania. Ostatnia przeanalizowana praca powiązana z problemem **P3** – [53], przedstawia ciekawe główne zagadnienie badawcze: „Który obszar oprogramowania testować najbardziej w przypadku systemów wbudowanych?". Oparto w niej motywację na rosnącym poziomie skomplikowania i wyróżniono dodatkowe zagadnienia badawcze:

1. Czy moduły z większą liczbą interfejsów są bardziej skomplikowane?

2. Czy liczba interfejsów jest powiązana z liczbą błędów?

Ciekawym faktem jest wyróżnienie interfejsów jako newralgicznych punktów – jest to dobra obserwacja. Udało się dzięki temu dowieść zasadności tego stwierdzenia w dalszej części artykułu i w ten sposób udzielić odpowiedzi na główne zagadnienie badawcze, wskazując konieczność testowania interfejsów.

Negatywne skutki skomplikowania oprogramowania w przemyśle motoryzacyjnym mogą być także minimalizowane już na etapie inżynierii wymagań, zanim powstaną architektura i kod. Przemysł motoryzacyjny nie pozostawia wyboru w kwestii inżynierii wymagań. Odpowiednie zarządzanie i praca z wymaganiami zgodnymi z procesem ASPICE są obligatoryjne.

Według publikacji [54] można stwierdzić, że w pracy z wymaganiami twarde wymagania prawne są spełniane, ale generują negatywny wpływ na tempo powstawania oprogramowania. W celu rozwiązania tego problemu przeprowadzono szereg rozmów z doświadczonymi architektami, liderami technicznymi, managerami oraz skonsultowano się z nimi w kwestii trzech zidentyfikowanych zagadnień badawczych:

1. Które aspekty obecnej pracy z wymaganiami wpływają na tempo rozwoju oprogramowania?

2. Jakie nowe aspekty powinny być rozpatrzone, aby zwiększyć tempo rozwoju oprogramowania?

3. W jakim stopniu podejście zwinne do rozwoju oprogramowania zmieni zidentyfikowane aspekty?

Konkluzją opartą na udzielonych odpowiedziach jest to, że tradycyjne podejście do rozwoju oprogramowania nie wystarczy. Twardo określone procesy mogą znacząco spowalniać rozwój i zmieniające się warunki projektowe. Zwiększenie automatyzacji jest wysoce pożądane, podejście oparte na modelowaniu systemu, bazując na wymaganiach może być odpowiednim kolejnym krokiem w rozwoju oprogramowania.

Podsumowując przegląd literatury w kontekście problemu **P3** i zagadnienia badawczego **ZB3**, można stwierdzić, że jest wiele prac, które podkreślają rosnący poziom skomplikowania oraz wyraźną potrzebę minimalizowania negatywnych skutków tego procesu. Badacze prezentują, co należy testować, jak mierzyć złożoność i jak ją obniżać. Propozycją kontynuacji i uzupełnienia rozwiązania problemu **P3** w świetle badań jest skupienie się na jak największym monitorowaniu i ciągłej analizie oprogramowania zgodnego ze standardem AUTOSAR. Nie jesteśmy w stanie zatrzymać rosnącego stopnia złożoności, więc musimy w jak najlepszy sposób testować oprogramowanie oraz monitorować jego pracę. Im więcej jasnych informacji posiadamy, w tym lepszy sposób możemy zminimalizować negatywne skutki skomplikowania.

3.4 Problemy powiązane z czasem wykonywania kodu

3.4.1 Problem P4 – skomplikowane ograniczenia i zależności czasowe

Istota problemu

Oprogramowanie wbudowane stosowane w przemyśle motoryzacyjnym jest wytwarzane dla wielu różnych sterowników ECU, które realizują szereg wysokopoziomowych funkcji biznesowych – od prostego sterowania podnoszeniem szyb po naciśnięciu przycisku, do bardzo zaawansowanych systemów powiązanych z funkcjami wspomagania kierowcy ADAS. W zależności od typu ECU i jego wpływu na bezpieczeństwo, przed inżynierami stawiane są różne wymagania dotyczące czasu wykonania danych funkcji biznesowych.

Zgodnie z tabelą 2.2 przedstawioną w rozdziale opisującym standard AUTOSAR, platforma Classic jest używana w systemach, dla których obowiązują najbardziej restrykcyjne wymagania czasowe – czasami nawet rzędu mikrosekund. W niniejszej rozprawie skupiono się właśnie na tych projektach. Wymagania czasowe dotyczą wielu różnych aspektów, takich jak:

- maksymalnego czasu inicjalizacji systemu (uruchomienia pojazdu),
- maksymalnego czasu deinicjalizacji systemu (wyłączenia pojazdu),
- maksymalnego czasu odpowiedzi na zapytanie diagnostyczne,
- odpowiedniego określenia podstawy czasowej dla wielu serwisów jednocześnie,
- odpowiedniej podstawy czasowej systemu operacyjnego.

Im większy i bardziej skomplikowany projekt, tym trudniej zarządzić wszystkimi ograniczeniami, skonfigurować planowanie wykonywania zadań przez system operacyjny, przypisać priorytety, zdefiniować wywłaszczalność zadań, określić procedurę inicjalizacji i deinicjalizacji itd.

Zagadnienie badawcze ZB4

Zagadnienie badawcze powiązane ze zidentyfikowanym problemem **P4** brzmi:

ZB4 – Jak zarządzać skomplikowanymi zależnościami i ograniczeniami czasowymi?

Sposób rozwiązania w świetle dotychczasowych badań

Rozwiązanie problemu **P4** nasuwa się samo – należy zastosować w pełni deterministyczny system operacyjny czasu rzeczywistego, zidentyfikować ograniczenia i wymagania, zaprojektować planowanie i wykonać analizę Worst Case Execution Time (WCET).

W tym kontekście pełny determinizm czasowy oznacza wykonywanie każdej funkcji systemu w zaplanowanym czasie. W praktyce takie rozwiązanie jest realizowane poprzez minimalizację wywłaszczeń i przerwań oraz odpowiednie ustawienie priorytetów w systemie operacyjnym czasu

rzeczywistego. Podobne podejście zaproponowano w 2003 roku w pracy [55].

Niestety poziom skomplikowania obecnego oprogramowania, połączony z minimalizacją kosztów elektroniki wykonawczej, skutkuje brakiem możliwości zastosowania takiego rozwiązania. Innymi słowy system ma do zrealizowania zbyt dużą ilość zadań oraz posiada zbyt wiele zależności zewnętrznych, aby móc w pełni świadomy sposób zaplanować jego działanie.

W przypadku nowoczesnych rozwiązań, ogromna liczba realizowanych funkcji praktycznie uniemożliwia w pełni deterministyczną konfigurację systemu operacyjnego, zwłaszcza dla ręcznej konfiguracji ustawień systemu operacyjnego i analizy heurystycznej.

Projekty komercyjne wspomniane w tej pracy, miały setki funkcji planowanych (ang. runnables), dziesiątki tysięcy funkcji oraz dziesiątki zadań (ang. tasks) systemu operacyjnego, do tego były realizowane na mikrokontrolerach wielordzeniowych, z licznymi zależnościami międzyrdzeniowymi.

Ciekawe podejście do takiego problemu zaprezentowano przez badaczy w artykule [56], gdzie starano się zaproponować wydajny sposób mapowania funkcji planowanych w motoryzacyjnych systemach wielordzeniowych, aby zminimalizować koszt komunikacji. Poddano analizie systemy wykorzystujące standard AUTOSAR oraz zastosowano analizę czasową WCET. Rezultatem jest propozycja zawierająca następujący algorytm:

1. Zgrupowanie funkcji planowanych powiązanych ze sprzętem i przypisanie ich do głównego rdzenia. Funkcje powiązane ze sprzętem to na przykład odczyty z przetworników analogowo-cyfrowych, operacje na pamięci, operacje na rejestrach mikrokontrolera.

2. Zgrupowanie reszty niezależnych od sprzętu funkcji planowanych i przypisanie ich do pozostałych rdzeni.

3. Podzielenie zadań zgodnie z planowanymi funkcjami i przypisanie ich do odpowiednich rdzeni. Na przykład wszystkie funkcje powiązane z komunikacją Ethernet powinny się znaleźć w jednym zadaniu, podobnie z innymi grupami funkcji.

4. Obliczenie WCET dla zadań, to znaczy wyznaczenie najdłuższego możliwego czasu wykonania danego zadania, biorąc pod uwagę wszystkie zależności i ograniczenia.

5. Znalezienie najlepszego przypisania grup zadań lub funkcji planowanych poprzez minimalizację funkcji sumy czasu wykonania i ilości danych, wymienianych pomiędzy funkcjami planowanymi. Szczegóły, jak wyznaczyć funkcję sumy dostępne są w artykule [56].

6. Przypisanie zadań i funkcji zgodnie z wynikiem poprzedniego kroku.

Niestety w ujęciu tym nie przedstawiono jasnych metryk pozwalających określić jakość proponowanego rozwiązania oraz w minimalnym stopniu wzięto pod uwagę ograniczenia systemowe (opisując jedynie krótko aspekt „zależności sprzętowych").

Kolejnym artykułem badawczym dotyczącym problemu **P4** jest [57], w którym zadaniem autorów również była próba rozwiązania problemu przypisywania funkcji planowanych do zadań systemu operacyjnego. Zastosowano w tym celu trzy metody: Periodic Solution (PS), Multiple Periodic Solution (MPS) i Arbitrary Periodic Solution (APS), a także wykonano szereg matematycznych dowodów oraz szacunków, wykorzystując analizę WCET oraz prezentując szereg algorytmów wraz z dowodami potwierdzającymi pozytywne rezultaty proponowanych metod. W rezultacie otrzymano o 23,81% lepszy stopień wykorzystania systemu operacyjnego. Jest to wartościowy artykuł. Jedynym brakującym obszarem jest aspekt ograniczeń wynikających z wymaganych relacji międzyfunkcyjnych. Ze strony autorów pojawiła się jednak zapowiedź uzupełnienia tego obszaru w kontynuacji ich pracy.

Podobną pracą jest [58], w której zaprezentowano metodę planowania celem zminimalizowania zużycia stosu. Rezultaty są również obiecujące, aczkolwiek działania minimalizacyjne odnoszą się do jednej wartości – zużycia stosu. Jest to bardzo interesujące podejście, lecz w projektach komercyjnych stos stanowi zaledwie jedno z dziesiątek ograniczeń. W pracy [59] skupiono się natomiast na zużyciu pamięci i czasie komunikacji „end-to-end", a w publikacji [60] na minimalizacji czasu przełączania pomiędzy zadaniami, unikaniu opóźnień czasowych i zapewnieniu spójności danych. Jak widać, sam aspekt planowania i zarządzania systemem operacyjnym jest szeroko badanym tematem analiz naukowych, co wskazuje na aktualność tego problemu badawczego.

Kolejnym newralgicznym obszarem jest procedura inicjalizacji oprogramowania w systemach wbudowanych. Według autorów pracy [61] istotny jest ich sposób optymalizacji w przypadku elektroniki użytkowej – konkretnie telewizorów. Główną argumentacją był czas uruchomienia,

stanowiący jeden z kluczowych aspektów i zbadanie jego pełnego przebiegu dla systemu opartego na systemie operacyjnym Linux, analizując zależności sprzętowe, czas uruchamiania jądra systemu oraz przestrzeń użytkownika. Dzięki zaproponowanej zmianie kolejności przydzielania zasobów podczas inicjalizacji oraz przeplataniu dostępów do wyjść i wejść, możliwe staje się zredukowanie czasu uruchamiania systemu o 35%.

W artykule [62] skoncentrowano się na inicjalizacji systemu wbudowanego dla systemu operacyjnego Android, uzyskując 65% zmniejszenie czasu uruchamiania poprzez modyfikacje jądra i sposobu dostępu do zasobów systemowych. Dodatkowym aspektem poruszanym przez badaczy był wpływ ich metody na zużycie pamięci, próbując uzyskać kompromis pomiędzy prędkością a rozmiarem oprogramowania.

Przechodząc do prac osadzonych w świecie oprogramowania motoryzacyjnego, na łamach publikacji [63] udało się przeprowadzić badania wielordzeniowego systemu wykorzystującego AUTOSAR i metodę analizy Logical Execution Time (LET), aby przez odpowiednie skonfigurowanie systemu operacyjnego zminimalizować czas odpowiedzi funkcji w ujęciu „end-to-end".

Kolejne godne uwagi i obszerne opracowanie przedstawiono w publikacji [64], gdzie argumentując, że z powodu licznych zależności komunikacyjnych, intensywnej wymiany danych i ograniczeń czasowych, trudno jest zapanować nad nowoczesnymi skomplikowanymi systemami, więc zaproponowano metodę zwiększenia przejrzystości analiz czasowych. Stanowi to bardzo interesujące podejście, które nie optymalizuje samego systemu, ale prezentuje narzędzie zapewniające więcej danych na temat przebiegu wykonania oprogramowania. Rozwiązanie to zostało przetestowane na systemie zarządzania silnikiem. Bardzo atrakcyjnym elementem tej pracy jest praktyczne zastosowanie standardu przemysłowego AUTOSAR oraz popularnej platformy sprzętowej AURIX. Proponowana metoda sprowadza się do transformacji modelu planowania wraz z ciągłą obserwacją rezultatów w postaci:

- opóźnień komunikacyjnych,
- czasu wymiany danych „end-to-end",
- czasu wykonania funkcji okresowych,
- wpływu na czas wykonania funkcji połączonych.

Opracowana aplikacja testowa zawiera 1250 funkcji planowanych oraz 21 zadań. Platforma sprzętowa to 4-rdzeniowy mikrokontroler pracujący z częstotliwością 200 MHz. Ta metoda obserwacji pozwala stwierdzić, że wyższa spójność danych zapewniona poprzez synchroniczne przetwarzanie danych wpływa negatywnie na aspekty czasowe, aczkolwiek w przypadku badanego systemu jest to preferowane rozwiązanie, gdyż poprawność funkcyjna systemu jest najważniejsza. Spojrzenie badaczy na ten aspekt jest zatem spójne z wnioskami prezentowanymi w dalszej części rozprawy, wskazując że w krytycznych systemach powiązanych z bezpieczeństwem użytkowników samochodów, jakość jest kluczowa.

Analiza czasowa oprogramowania wbudowanego w przemyśle motoryzacyjnym została opisana także w pracy [65], w której przedstawiono autorskie narzędzie do analizy statycznej kodu, zapewniające skalowalny poziom abstrakcji, który pozwala wykryć błędy występujące podczas wykonania, wyścigi w oprogramowaniu (ang. data races) oraz wzajemne blokady (ang. deadlocks). Wykorzystano w tym celu komputer PC, aby stworzyć oprogramowanie testujące oprogramowanie wbudowane, używające AUTOSAR, sterujące układem hamulcowym dla ciężarówek. Dzięki analizie trwającej ponad 16 godzin, pokryto 75% kodu i znaleziono 876 przypadków niewłaściwego użycia wskaźników, 13 miejsc dzielenia przez zero, 639 sytuacji przepełnienia zmiennych i 50 błędnych wywołań funkcji – wszystkie te zdarzenia, w czasie pracy ECU, mogą powodować trudne do analizy problemy posiadające niedeterministyczne zależności czasowe.

Podsumowując badania literatury, istnieje wiele opracowań potwierdzających wagę problemu **P4**, gdzie skomplikowane zależności i ograniczenia czasowe, powodują poważne problemy. Zagadnienie badawcze **ZB4** stanowi uzupełnienie tych prac, w kontekście skupienia się na jakości oraz bezpieczeństwie oprogramowania. Obecne prace skupiają się na optymalizacji, często ograniczonej do pojedynczych wartości fizycznych. Nie analizują one systemu całościowo oraz w minimalnym stopniu wykazują relacje z międzynarodowymi normami jakości i bezpieczeństwa oprogramowania – ISO 26262, ISO 25010 i ISO 21434.

Rozdział 4

Metodyka badań empirycznych

Rozdział zawiera opis metod badawczych, które umożliwiły opracowanie odpowiedzi na zagadnienia badawcze przedstawione w rozdziale 3.

4.1 Opis stanowiska badawczego

Badania były wykonywane w trzech projektach komercyjnych, przy użyciu układów montowanych w seryjnie produkowanych samochodach oraz narzędzi wykorzystywanych w projekcie. Zapewniło to praktyczny i rzetelny charakter badań. Co ważne, projekty były przeznaczone do trzech różnych modeli samochodów, ale realizowały ten sam typ sterownika. Większość wymagań i główne założenia we wszystkich trzech projektach były wspólne, dlatego autor porównuje je dalej do siebie. Stanowisko badawcze składało się z:

1. Jednostki wbudowanej z mikrokontrolerem w architekturze PowerPC.
2. Jednostki wbudowanej z mikrokontrolerem w architekturze ARM.
3. Debuggera wykorzystującego interfejs Joint Test Action Group (JTAG).
4. Sprzętowego interfejsu komunikacyjnego dla magistrali Local Interconnect Network (LIN) oraz Ethernet.
5. Sterowanego zasilacza 12V.
6. Zestawu przekaźników sterujących procesem programowania układu.

7. Zintegrowanego środowiska programistycznego (Integrated Development Environment (IDE)).
8. Autorskich skryptów pozyskujących dane.
9. Oprogramowania przeprowadzającego analizy czasowe systemu operacyjnego.

Z powodu komercyjnego charakteru badanych obiektów, wszystkie dane klienckie zostały usunięte lub zanonimizowane. Do pracy zostały dołączone wyłącznie autorskie skrypty służące do pozyskiwania danych.

4.1.1 Opis projektów komercyjnych

W pracy są używane następujące odniesienia do projektów komercyjnych:

1. **Projekt 1** – linia ekskluzywnych samochodów osobowych z ECU w architekturze PowerPC (średni wolumen – dziesiątki tysięcy samochodów).
2. **Projekt 2** – linia popularnych samochodów osobowych z ECU w architekturze PowerPC (ogromny wolumen – setki tysięcy samochodów).
3. **Projekt 3** – linia elektrycznych samochodów osobowych z ECU w architekturze ARM (średni wolumen – dziesiątki tysięcy samochodów).

Projekty były realizowane chronologicznie, co oznacza rozpoczęcie przez autora analizy w najstarszym projekcie **Projekt 1**, a następnie stopniowe wprowadzanie proponowanych rozwiązań w czasie rozwoju projektów. Najnowszy **Projekt 3** zawiera wdrożenie wszystkich zaprezentowanych rozwiązań.

4.2 Opis metod badawczych

Metody badawcze zostały wybrane w taki sposób, aby umożliwić obserwację wartości fizycznych oraz uzyskać informacje na temat zidentyfikowanych zagadnień badawczych.

4.2.1 Metoda badawcza MB1 – analiza statyczna kodu

Analiza statyczna kodu to metoda obserwacyjna, w której głównymi parametrami są:

1. Liczba linii kodu (pisanego ręcznie lub generowanego).
2. Liczba komentarzy.
3. Liczba funkcji.
4. Liczba jednostek translacji.

Głównym narzędziem wykorzystanym w metodzie badawczej **MB1** było darmowe oprogramowanie [66] umożliwiające generowanie raportów odnośnie do wymienionych parametrów dotyczących oprogramowania. Dodatkowo, specjalne autorskie skrypty w języku Python posłużyły do analizy szczególnych fragmentów kodu. Umożliwiły one m.in. zlokalizowanie liczby wywołań konkretnych funkcji lub odczytanie i zmianę parametrów oprogramowania. Użycie oprogramowania [66] zostało oparte na komendach w wierszu poleceń. Narzędzie może analizować pojedyncze pliki, zestawy plików, a nawet całe katalogi. W celu wykonania pomiaru, należy uruchomić wiersz poleceń w odpowiednim miejscu w strukturze projektu i wywołać komendę „cloc" z odpowiednim parametrem. Dokumentacja oprogramowania jest dostępna na stronie [66].

Autorskie skrypty były specjalnie przygotowywane do konkretnych zestawów danych. Instrukcje używania skryptów są dołączone do ich kodów źródłowych.

4.2.2 Metoda badawcza MB2 – badania czasu wykonania kodu

Metoda badań czasu wykonania kodu to metoda eksperymentalna, przy użyciu której badano pomiar czasu wykonania konkretnych fragmentów oprogramowania. W zależności od potrzebnej dokładności i czasu pomiaru zostały wykorzystane metody przedstawione w tabeli 4.1.

Pomiar czasu przy użyciu debuggera jest zależny od producenta sprzętu. Większość dostępnych na rynku urządzeń umożliwia pomiary czasowe. Należy pamiętać o rozdzielczości pomiarów i przeanalizować instrukcję.

Oznaczenie	Metoda pomiaru	Rozdzielczość	Komentarz
MB2.1	Pomiar przy użyciu debuggera	1 ms (1 kHz)	+ brak potrzeby modyfikacji kodu w celu pomiaru – niska rozdzielczość
MB2.2	Pomiar przy użyciu makra OS_GetTimestamp	125 ns (8 MHz)	+ brak potrzeby konfiguracji – wyłączenie przerwań mikrokontrolera zatrzymuje pomiar
MB2.3	Pomiar przy użyciu timera sprzętowego	41 ns (24 MHz)	+ bardzo wysoka rozdzielczość – potrzeba konfiguracji dla każdego użycia
MB2.4	Pomiar przy użyciu oscyloskopu	1 ns (1 GHz)	+ najwyższa rozdzielczość – potrzeba lutowania punktów testowych

Tablica 4.1. Metody badawcze pomiaru czasu wykonania kodu

Pomiar przy użyciu oscyloskopu wymaga przylutowania punktów testowych na elektronice wykonawczej. Należy zgodnie ze schematem zlokalizować piny mikrokontrolera, które można skonfigurować jako wyjścia cyfrowe oraz przylutować do nich przewody pomiarowe, które umożliwią przymocowanie sond oscyloskopu. Następnie należy skonfigurować piny wyjściowe mikrokontrolera i zaplanować pomiar, w taki sposób, aby zarówno na początku, jak i na końcu badanej procedury pin zmienił swój status. Oscyloskop pozwoli w ten sposób zmierzyć czas pomiędzy dwoma zmianami sygnału.

Pomiar przy użyciu timera sprzętowego wymaga skonfigurowania modułu General Purpose Timer (GPT), zgodnie z dokumentacją [67] lub odniesienia się bezpośrednio do rejestrów mikrokontrolera. Na początku i końcu badanej procedury należy odczytać obecne wartości timera oraz zgodnie z jego rozdzielczością obliczyć różnicę czasową.

Pomiar przy użyciu makra OS_GetTimestamp jest najprostszy, bez

jakiejkolwiek konfiguracji wystarczy wywołać funkcję przed i po badanej procedurze oraz przeanalizować różnicę. W konfiguracji systemu operacyjnego należy sprawdzić rozdzielczość, zgodnie z [68]. W przypadku projektów komercyjnych badanych przez autora, makro zapewniało rozdzielczość 125 ns.

4.2.3 Metoda badawcza MB3 – badania obciążenia CPU

Metoda badania obciążenia CPU to metoda eksperymentalna oparta na wykorzystaniu algorytmów pomiaru obciążenia mikrokontrolera. Bardzo ważnym elementem jest odniesienie do przedziału czasowego pomiaru. Obciążenie CPU musi zawsze być mierzone w danym przedziale czasu, np. 10 lub 100 milisekundach. Wykorzystano dwa główne sposoby pomiaru:

1. **MB3.1** Użycie interfejsów systemu operacyjnego Open Systems and their Interfaces for the Electronics in Motor Vehicles (OSEK) – wykorzystanie zapewnionych przez stos metod pomiaru. Polegają one na implementacji kodu pomiarowego w makrach Pre-task hook i Post-task hook. Są to makra zdefiniowane w [68], które są wywoływane przez system operacyjny przed i po uruchomieniu każdego zadania. Umożliwiają one pomiar tzw. „Idle task", czyli zadania wykonywanego przez system po wykonaniu wszystkich zadań w danym przedziale czasu.

2. **MB3.2** Użycie oprogramowania komercyjnego umożliwiającego profilowanie systemu operacyjnego. Oprogramowanie tego typu było dostępne w projekcie komercyjnym **Projekt 3**. Umożliwia ono dokładną analizę czasu wykonania każdego zadania systemu operacyjnego oraz kolejność przechodzenia pomiędzy zadaniami. Zaletą tego typu narzędzia jest możliwość graficznego przedstawienia wywłaszczeń, przerwań, przekroczeń budżetu czasowego zadań itp.

4.2.4 Metoda badawcza MB4 – analiza dokumentów

Metoda badania dokumentów polega na analizie dokumentacji powiązanej z danym problemem. Głównymi kategoriami są:

1. Dokumentacja sprzętowa – dokumenty powiązane z mikrokontrolerem, pamięciami, transceiverami oraz innymi elementami elektronicznymi.

2. Dokumentacja standardu AUTOSAR – szerzej opisana w rozdziale 2.2.1. Ogólna dokumentacja dostępna na stronie www.autosar.org.

3. Dokumentacja używanego stosu AUTOSAR – powiązana z implementacją stosu AUTOSAR przez konkretnego dostawcę. Jest ona dostarczana przy dostawie zakupionego oprogramowania. Musi być zgodna z ogólną dokumentacją AUTOSAR, wszystkie odstępstwa od standardu muszą być udokumentowane.

4. Normy ISO – w zależności od obszaru badań, normy ISO stanowią bazę realizacji danych funkcji.

5. Dokumentacja klienta – wymagania stawiane przez producentów samochodów.

4.2.5 Metoda badawcza MB5 – analiza przebiegu wykonania kodu

Metoda badań przebiegu wykonania kodu to metoda eksperymentalna, przy użyciu której badano przebieg wykonania konkretnych fragmentów oprogramowania, wykorzystując debuggera i analizę logów systemowych. W zależności od obserwowanej funkcjonalności, sprawdzano różne zestawy danych. Przegląd przebiegu wykonania kodu jest najczęściej wykorzystywaną metodą podczas analiz błędów. Metoda polega na badaniu korelacji czasowych oraz inspekcji danych systemowych. Przykładem takiej analizy jest zestawienie zmian sygnału stacyjki samochodu, inicjalizacji poszczególnych modułów systemowych, stanów magistrali komunikacyjnych oraz zmiennych szczegółowych, aby potwierdzić odpowiedni proces inicjalizacji sterownika. Bardzo często system zachowuje się inaczej podczas działania w samochodzie, gdzie komunikacja dziesiątek sterowników wpływa na sposób wykonania danej funkcji, która działała inaczej podczas symulacji lub działania tylko z jednym czy dwoma sterownikami testowymi.

4.2.6 Metoda badawcza MB6 – analiza przy użyciu obserwatora systemu

Metoda badawcza opracowana przez autora niniejszej pracy łączy metodę eksperymentalną i obserwacyjną, która polega na użyciu zaproponowanego „Obserwatora systemu" opisanego w rozdziale 5.4. Sprowadza się ona do analizy krytycznych faz systemu, generowania raportów z wielokrotnych iteracji wykonywania testów, a bazując na wynikach – do ciągłej optymalizacji systemu oraz eliminacji błędów. Metoda **MB6** stanowi rozszerzenie manualnej metody eksperymentalnej **MB5**, poprzez dodanie automatycznego generowania raportów systemowych oraz dodaje korelacje pomiędzy modułami i czynnikami zewnętrznymi. Przykładowo realizuje pomiary czasowe zapisów do pamięci lub czasu trwania faz inicjalizacji, zapisuje statystyki ilości wysłanych i odebranych danych, podsumowuje ilość błędów, które wystąpiły podczas działania systemu itp. Szczegóły dotyczące modułu obserwatora systemu są przedstawione w rozdziale 5.4.

4.3 Metody badawcze a cele badawcze

Zrealizowanie celów badawczych było możliwe dzięki wprowadzeniu metod **MB1** do **MB5**, wybranych na podstawie doświadczeń autora z pracy w przemyśle. Dodatkowo opracowano autorską metodę badawczą **MB6**, jako rozwiązanie pozwalające na uzupełniającą analizę poruszanych zagadnień. Cele badawcze **ZB1** do **ZB4** sformułowano jako otwarte tematy, w ramach których można realizować wiele propozycji ich rozwiązania. Ponadto przedstawiono ramy do określonych zagadnień w postaci konkretnych modułów standardu AUTOSAR oraz opisanych obszarów analiz. Zaproponowane metody (zawierające konkretne autorskie rozwiązania) zostały wdrożone w projektach przemysłowych, co dowodzi, że założone cele badawcze zostały osiągnięte.

Rozdział 5

Pakiet rozwiązań poprawiających jakość i bezpieczeństwo oprogramowania i jego walidacja

5.1 Automatyzacja stosu diagnostycznego

5.1.1 Nawiązanie do zagadnienia badawczego

Rozwiązanie 1 – automatyzacja stosu diagnostycznego, jest zaproponowanym rozwiązaniem zidentyfikowanego problemu **P1** opisanego w rozdziale 3.1.1. W celu zrealizowania badań dotyczących zagadnienia badawczego **ZB1**, zostały wykorzystane metody badawcze **MB1** oraz **MB4**. Pierwszym etapem była analiza dokumentów AUTOSAR:

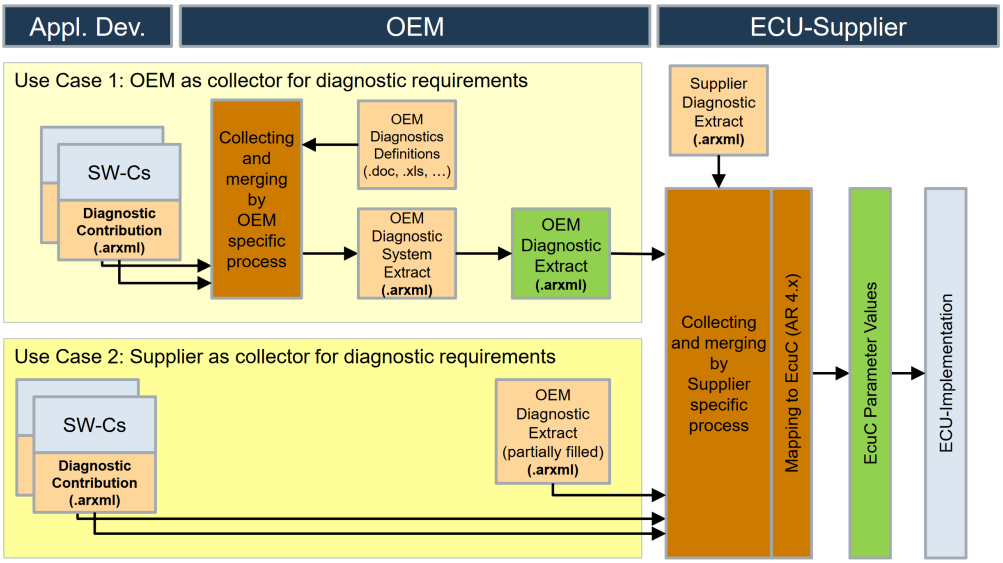
1. specyfikacji modułu managera zdarzeń Diagnostic Event Manager – (Dem) [69],
2. specyfikacji modułu managera zapytań diagnostycznych Diagnostic Communication Manager – (Dcm) [70],
3. wzorca ekstraktu diagnostycznego (Diagnostic Extract Template) [71].

Standard AUTOSAR definiuje generyczny opis procesu oraz procedur, ale każda implementacja stosu ma swoją unikalną realizację. Właściwymi dokumentami definiującymi strukturę kodu i proces integracji jest dokumentacja używanego stosu AUTOSAR. Dokumentacja ta nie jest

publiczna, zostaje ona dostarczona wraz ze stosem. Podczas badań i analiz wykorzystano stos jednego z wiodących producentów narzędzi AUTOSAR.

Analiza procesu integracji ekstraktu diagnostycznego

Procedura integracji ekstraktu diagnostycznego jest ogólnie zdefiniowana przez [71], proces przetwarzania został zaś przedstawiony na diagramie 5.1. Producent samochodu (Original Equipment Manufacturer (OEM)) po uzgodnieniu z dostawcą i zebraniu wymagań opracowuje ekstrakt w formacie ARXML. Dostawca przechodzi przez podobny proces – określa swoje własne wymagania i opracowuje ekstrakt w formacie ARXML. Obydwa pliki są łączone w jeden całościowy opis stosu diagnostycznego i importowane do narzędzi generujących implementację. Główne kategorie serwisów przetwarzanych na potrzeby diagnostyki są



Rysunek 5.1. Proces przetwarzania ekstraktu diagnostycznego (Źródło [71])

oparte na kategoriach standardu ISO 14229 [29]:

1. Dane diagnostyczne (DID) – dane przetwarzane przez serwisy ReadDataByIdentifier (0x22) oraz WriteDataByIdentifier (0x2E). Są to zestawy informacji, które zapisuje się w pamięci nieulotnej w samochodzie. Najczęściej podzielone są na odpowiednie kategorie,

takie jak dane produkcyjne, dane kalibracyjne itp. W zależności od typu danych, mogą być zapisywane tylko raz podczas produkcji (np. numer VIN) lub okresowo (np. licznik samochodu).

2. Rutyny diagnostyczne – funkcje wywoływane przez serwis RoutineControl (0x31). Są to funkcje, które tester diagnostyczny może wywołać na żądanie, np. uruchomienie testu wycieraczek.

3. Serwisy powiązane z DTC – ReadDTCInformation (0x19), ClearDiagnosticInformation (0x14). Są to procedury powiązane z przetwarzaniem błędów wykrywanych podczas działania samochodu.

4. Serwisy powiązane z aktualizacją oprogramowania – TransferData (0x35), RequestDownload (0x34).

5. Serwisy powiązane z zabezpieczeniami – SecurityAccess (0x27), SecuredDataTransmission (0x84).

Wymienione serwisy są głównymi elementami przetwarzanymi przez stos diagnostyczny, które były przeanalizowane w niniejszej pracy, ponieważ są używane w największym zakresie i generują największą ilość kodu. Dokładny opis wszystkich serwisów określa norma [29]. Ekstrakty diagnostyczne przedstawione na rysunku 5.1 są zmieniane podczas czasu trwania projektu, co wymaga wielokrotnego procesu ich integracji. Pierwsza integracja ekstraktu jest najdłuższa, polega ona na użyciu dostarczonego pliku ARXML, skonfigurowaniu stosu AUTOSAR i ręcznej implementacji wszystkich wymaganych przez producenta samochodu serwisów oraz danych diagnostycznych. Jako przykład można podać błąd anteny GPS, czyli DTC. Ekstrakt diagnostyczny zawiera informacje na temat wymaganego czasu wykrywania, że na przykład w ciągu 8 sekund sterownik musi wskazać zwarcie na antenie GPS. Dodatkowo ekstrakt definiuje warunki wykrywania, że stacyjka musi być w pozycji działania silnika samochodu. Po stronie inżyniera, który integruje ekstrakt jest konfiguracja parametrów takiego błędu anteny oraz ręczna implementacja odczytów wartości elektrycznych przez mikrokontroler, aby określić czy wystąpiło zwarcie.

Kolejne aktualizacje ekstraktów polegają na wprowadzeniu zmian, takich jak dodanie czy usunięcie konkretnych serwisów. Na przykład producent samochodu może dodać wymaganie, że w przypadku wykrycia zwarcia anteny GPS należy do raportu błędu dodać informacje o obecnym stanie licznika

oraz ostatnią znaną pozycję GPS. W zależności od wielkości projektu, proces aktualizacji ekstraktu bez automatyzacji, może trwać nawet do 2 tygodni.

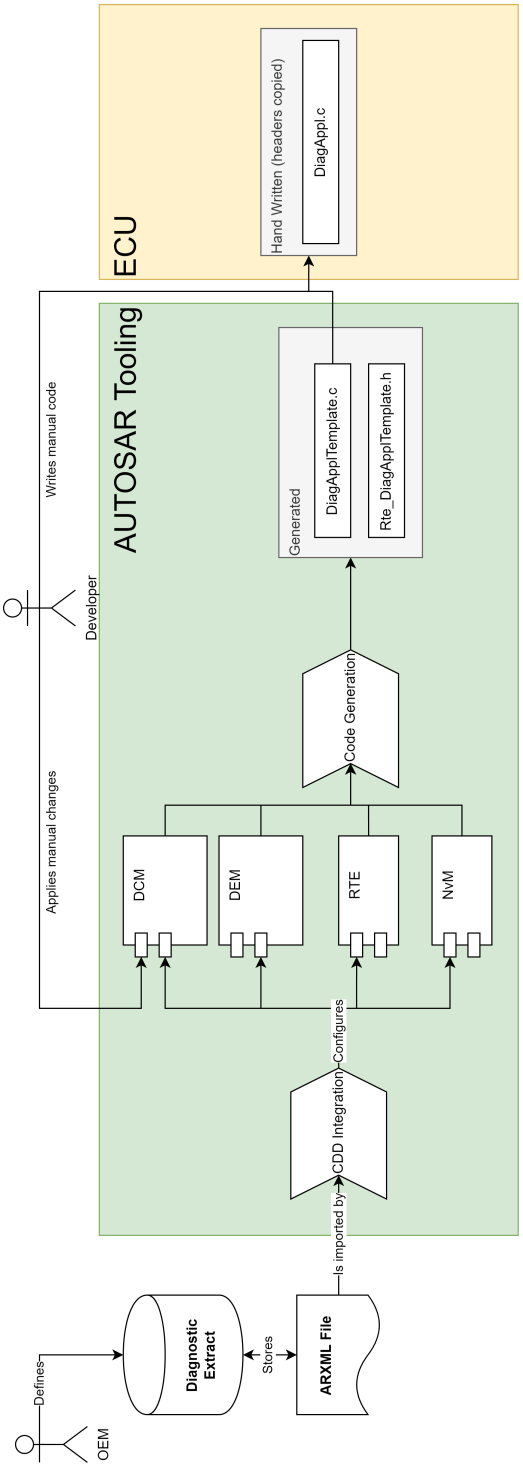
W projektach **Projekt 1** i **Projekt 2** proces trwał około 15 dni roboczych i został przedstawiony na rysunku 5.2, gdzie widoczny jest początek przepływu informacji w momencie, gdy producent samochodu (OEM) definiuje ekstrakt diagnostyczny w formacie ARXML. Integracja ekstraktu jest wykonywana ręcznie przez inżyniera, który musi sam weryfikować wszystkie zmiany oraz manualnie implementować każdy serwis diagnostyczny. Czynności, które musiał wykonać integrator to:

1. Analiza zmian ekstraktu – usunięte, zmienione oraz dodane serwisy.
2. Import ekstraktu w programie obsługującym stos AUTOSAR.
3. Ręczne rozwiązanie konfliktów.
4. Generacja kodu.
5. Ręczne usunięcie zbędnych serwisów oraz dodanie nowych.
6. Implementacja obsługi nowych serwisów.
7. Kompilacja.

Analiza procesu integracji ekstraktu diagnostycznego wykazała, że istnieje kilka obszarów, gdzie widać potencjał do wprowadzenia automatyzacji, które przyczynią się do rozwiązania zidentyfikowanego problemu **P1**.

Analiza statyczna

Kolejny krok to analiza statyczna kodu (metoda badawcza **MB5**), a więc ile tysięcy linii kodu źródłowego realizuje implementację stosu diagnostycznego. Został w tym obrębie uwzględniony tylko kod specyficzny dla projektu. Sterowniki i warstwy pomocnicze nie zostały wzięte pod uwagę. Stanowią one osobną część oprogramowania, która jest dostarczana i nie wolno jej modyfikować z powodu potencjalnej utraty certyfikacji. Podsumowanie analizy dla projektów **Projekt 1** oraz **Projekt 2** zostało przedstawione w tablicy 5.1. Wolumen oprogramowania w modułach diagnostycznych to około 30 tysięcy linii kodu, z czego aż 25 tysięcy pisanych ręcznie. Komentarze stanowią zaledwie 2800 linii. Taka duża ilość ręcznie pisanego kodu, stanowi znaczące ryzyko wystąpienia wielu błędów i



Rysunek 5.2. Ręczny proces integracji ekstraktu diagnostycznego.

Moduły diagnostyczne				
Projekt	Linie kodu	Linie komentarzy	Funkcje	Linie pisane ręcznie
Projekt 1	30758	2783	1965	24863
Projekt 2	31568	2854	2059	25391

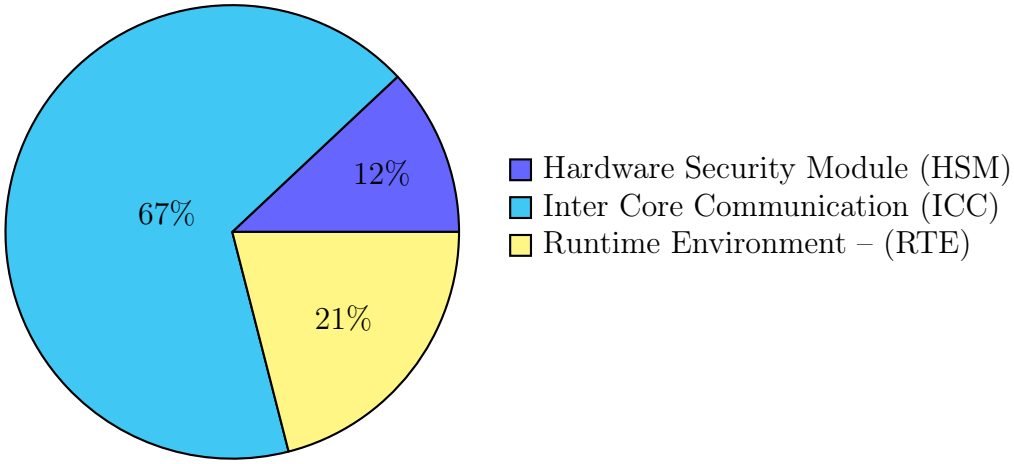
Tablica 5.1. Ilość kodu w modułach diagnostycznych w projektach przed wprowadzeniem automatyzacji

jest negatywnie oceniana w kontekście normy ISO 25010 w kategoriach użyteczności, łatwości utrzymania, przenośności oraz niezawodności. Analiza wykazała także, że kod jest wysoce powtarzalny.

Wykres 5.3 przedstawia źródła pochodzenia danych diagnostycznych, gdzie 12% stanowią dane powiązane z cyberbezpieczeństwem, uzyskiwane z jednostki przetwarzającej dane kryptograficzne – Hardware Security Module (HSM), na przykład certyfikaty, klucze, sumy kontrolne. Z kolei 21% to dane pochodzące z warstwy RTE, czyli najczęściej dane z warstwy aplikacyjnej, takie jak na przykład pozycja GPS samochodu, stan baterii, przebieg. Największą część, czyli 67% stanowią dane z komunikacji międzyrdzeniowej – Inter Core Communication (ICC), czyli dane przetwarzane przez inne rdzenie sterownika, takie jak na przykład informacje na temat połączenia Global System for Mobile Communications (GSM), status połączenia z serwerami producenta samochodu, czas globalny samochodu itp. Zarówno pozyskiwanie danych z RTE, jak i danych z komunikacji międzyrdzeniowej ICC, jest możliwe do zautomatyzowania. Na bazie pliku konfiguracyjnego można wygenerować kod obsługujący interfejsy pozyskujące dane. W świetle tych obserwacji, decyzją autora było postanowienie kontynuowania prac nad automatyzacją procesu integracji ekstraktów diagnostycznych w celu realizacji zagadnienia badawczego **ZB1**. Głównymi założeniami było zminimalizowanie ilości ręcznie pisanego kodu, zwiększenie utrzymywalności oraz skrócenie czasu procesu integracji ekstraktów diagnostycznych.

5.1.2 Sposób rozwiązania problemu badawczego

Po fazie analizy określono ogólną architekturę skryptu automatyzującego, widoczną na rysunku 5.4, gdzie przedstawiony został początek przepływu informacji w momencie, gdy producent samochodu



Rysunek 5.3. Pochodzenie danych diagnostycznych

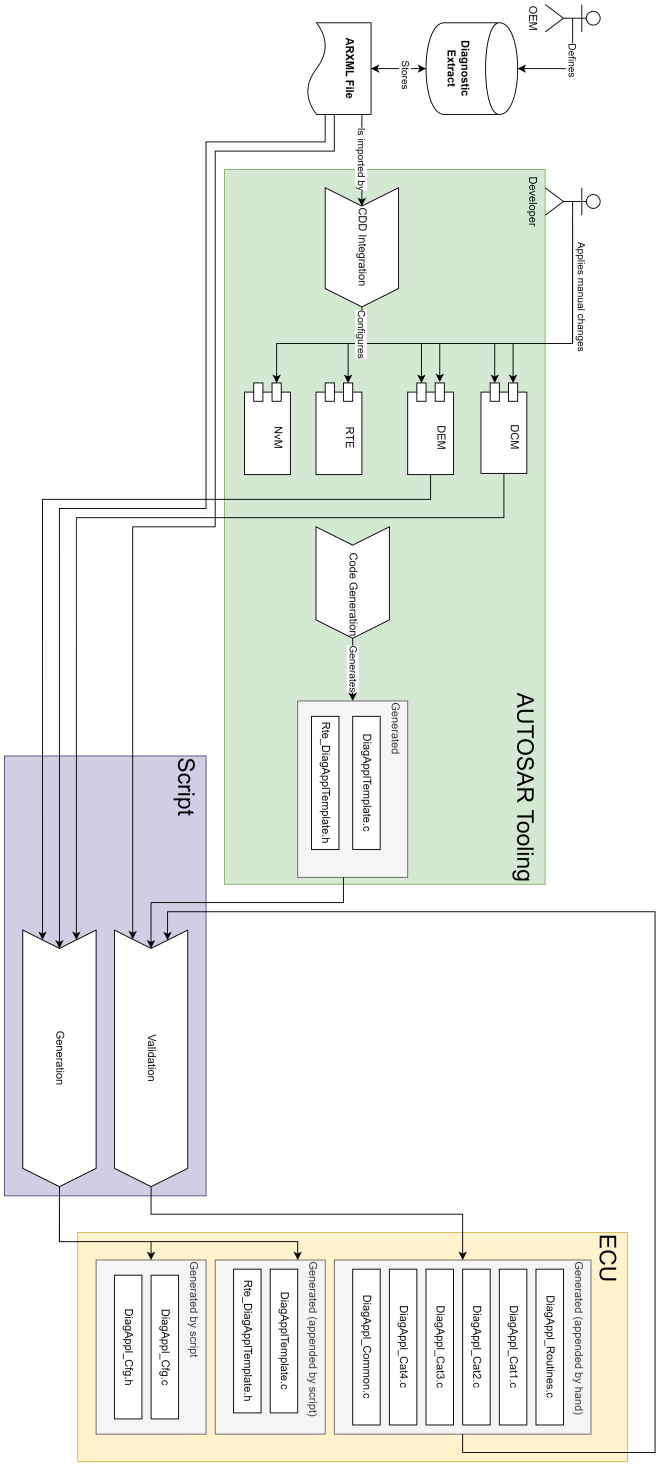
(OEM) definiuje ekstrakt diagnostyczny w formacie ARXML. Integracja ekstraktu ogranicza się do wprowadzenia drobnych ręcznych modyfikacji, większość zmian jest realizowana przez autorski skrypt, którego dwa główne cele to: weryfikacje danych i generowanie kodu.

Weryfikacja danych

Weryfikacja opiera się na odczytaniu wszystkich serwisów diagnostycznych z ekstraktu w formacie ARXML, wraz z ich parametrami (długościami sygnałów, kategoriami, powiązaniami), odczytaniu aktualnej konfiguracji narzędzia stosu AUTOSAR oraz porównaniu tych dwóch źródeł. Rezultatem tej operacji jest lista serwisów, które są w jakikolwiek sposób zmodyfikowane (dodane, usunięte, zmieniono ich długość lub kategorię itp.). Drugą możliwością jest porównywanie dwóch ekstraktów w formacie ARXML. Funkcja ta jest używana w fazie analizy zmian między dostarczonymi przez producenta ekstraktami.

Generowanie kodu

Generowanie polega na przetworzeniu wszystkich zaimportowanych serwisów, podzieleniu ich na poszczególne kategorie, przydzieleniu odpowiednich wzorców kodu oraz wygenerowaniu plików źródłowych i



Rysunek 5.4. Automatyczny proces integracji ekstraktu diagnostycznego

nagłówkowych.

W celu określenia, w jaki sposób dany serwis ma być przetwarzany, został użyty plik konfiguracyjny, w którym integrator określa, jakie ma być źródło danych (ICC/RTE/Inne) oraz czy ma zostać wygenerowany automatyczny kod lub czy serwis powinien być obsługiwany ręcznie.

Przy standardowym procesie integracji, narzędzie stosu generuje pliki nagłówkowe oraz puste pliki źródłowe. Przykład wygenerowanego zestawu funkcji do odczytu i zapisu danej diagnostycznej o nazwie „NazwaDid" jest zaprezentowany w kodzie źródłowym 5.1.

```
1  FUNC(Std_ReturnType, RTE_CODE) DiagAppl_NazwaDid_ReadData (Dcm_OpStatusType
       OpStatus, P2VAR(uint8, AUTOMATIC, RTE_APPL_DATA) Data)
2  {
3  /* suppress compiler warnings about unused arguments */
4    (void)OpStatus;
5    (void)Data;
6    return RTE_E_OK;
7  }
8
9  FUNC(Std_ReturnType, RTE_CODE) DiagAppl_NazwaDid_WriteData (P2CONST(uint8,
       AUTOMATIC, RTE_APPL_DATA) Data, Dcm_OpStatusType OpStatus, P2VAR(
       Dcm_NegativeResponseCodeType, AUTOMATIC, RTE_APPL_DATA) ErrorCode)
10 {
11 /* suppress compiler warnings about unused arguments */
12   (void)Data;
13   (void)OpStatus;
14   (void)*ErrorCode;
15   return RTE_E_OK;
16 }
17
18 FUNC(Std_ReturnType, RTE_CODE) DiagAppl_NazwaDid_ConditionCheckRead (
       Dcm_OpStatusType OpStatus, P2VAR(Dcm_NegativeResponseCodeType, AUTOMATIC
       , RTE_APPL_DATA) ErrorCode)
19 {
20 /* suppress compiler warnings about unused arguments */
21   (void)OpStatus;
22   (void)*ErrorCode;
23   return RTE_E_OK;
24 }
```

Kod źródłowy 5.1. Przykład wygenerowanych funkcji stosu diagnostycznego

Wnętrza funkcji są wypełnione pustym kodem, który musi zostać zamieniony na właściwą implementację. Każdy serwis diagnostyczny składa się z wielu zestawów zaprezentowanych funkcji – trzech dla sygnału zapisywanego i odczytywanego, dwóch dla sygnału tylko odczytywanego. W przypadku ekstraktu diagnostycznego z projektu **Projekt 3**, posiada on 196 serwisów oraz aż 1059 sygnałów. Oznacza to, że w przypadku pełni manualnego procesu, integratorzy muszą zaimplementować 1059 zestawów funkcji zaprezentowanych w kodzie źródłowym 5.1.

Kod źródłowy 5.2 przedstawia te same funkcje, po przetworzeniu przez skrypt automatyzujący przygotowany przez autora. Z powodu tajności projektów, prawdziwa implementacja została zastąpiona pseudokodem.

```
1  /* DID Name: NazwaSerwisu*/
2  /* DID Signals Count: 5*/
3  /* DID Targets: Rdzen 1, Rdzen 2*/
4  /* DID Category: Kategoria Dane GSM*/
5  /* This signal: "NazwaSynalu" belongs to DID: "NazwaSerwisu" and is 1 out of
       5*/
6  FUNC(Std_ReturnType, RTE_CODE) DiagAppl_NazwaSerwisu_ConditionCheckRead (
       Dcm_OpStatusType OpStatus, P2VAR(Dcm_NegativeResponseCodeType, AUTOMATIC
       , RTE_APPL_DATA) ErrorCode)
7  {
8      return UzyskajDane(OpStatus, ErrorCode, IdentyfikatorDanych);
9  }
10
11 /* Signal Length: SIGNAL_NAZWASYGNALU_LENGTH (4000)*/
12 /* Signal Offset: SIGNAL_NAZWASYGNALU_OFFSET (0)*/
13 FUNC(Std_ReturnType, RTE_CODE) DiagAppl_NazwaSerwisu_ReadData (
       Dcm_OpStatusType OpStatus, P2VAR(uint8, AUTOMATIC, RTE_APPL_DATA) Data)
14 {
15     SkopiujWynikiUzyskaniaDanych();
16     ZwolnijBufor();
17
18     return RTE_E_OK;
19 }
20
21 /* Signal Length: SIGNAL_NAZWASYGNALU_LENGTH (4000)*/
22 /* Signal Offset: SIGNAL_NAZWASYGNALU_OFFSET (0)*/
23 FUNC(Std_ReturnType, RTE_CODE) DiagAppl_NazwaSygnalu_WriteData (P2CONST(
       uint8, AUTOMATIC, RTE_APPL_DATA) Data, Dcm_OpStatusType OpStatus, P2VAR(
       Dcm_NegativeResponseCodeType, AUTOMATIC, RTE_APPL_DATA) ErrorCode)
24 {
25     *ErrorCode = WyslijDane(Data, IdentyfikatorDanych);
26     return = RTE_E_OK;
27 }
```

Kod źródłowy 5.2. Przykład wygenerowanych funkcji stosu diagnostycznego po przetworzeniu przez skrypt automatyzujący

W przypadku projektu komercyjnego, znajdowała się tam docelowa, całkowicie wygenerowana implementacja obsługi serwisów diagnostycznych. Polega ona na zrealizowaniu trzech funkcji: „ConditionCheckRead", czyli sprawdzenia, czy dany serwis diagnostyczny może być obecnie zrealizowany, potem w zależności od typu serwisu – odczyt danych przez „ReadData" lub zapis danych przez „WriteData". Sam sposób odczytywania i zapisywania danych diagnostycznych jest całkowicie zależny od konkretnego projektu, nie jest to ważne w kontekście proponowanego rozwiązania. Najważniejszy jest fakt, że każdy serwis oraz każdy sygnał może być wygenerowany wraz z zawartością. Skrypt ma informacje, jaka funkcja jest przypisana do jakiego

serwisu, jaką ma długość, nazwę, przesunięcie bitowe, typ danych. Jak widać w komentarzach dla funkcji, skrypt przygotowuje także dostępne definicje wraz z ich wartościami (SIGNAL_NAZWASYGNALU_LENGTH, SIGNAL_NAZWASYGNALU_OFFSET), ułatwiają one analizę i zwiększają czytelność. Dzięki temu zespół po integracji skryptu może łatwo zdefiniować zawartość funkcji i dostosować je do dostępnej architektury systemu, poprzez implementację ogólnych funkcji wymienionych w kodzie 5.2, czyli:

- UzyskajDane() – (tylko dla odczytu danych) funkcja, która sprawdza, czy dany serwis jest dostępny i przygotowuje dane do przetworzenia. Na przykład funkcja może wysyłać zapytanie po magistrali komunikacyjnej do innego rdzenia, po czym dostać dane na temat wież bazowych sieci komórkowej.

- SkopiujWynikiUzyskaniaDanych() – (tylko dla odczytu danych) funkcja, która kopiuje dane z bufora tymczasowego, do buforów modułu diagnostycznego Dcm.

- ZwolnijBufor() – (tylko dla odczytu danych) funkcja, która zgłasza buforowi aplikacyjnemu, że może być gotowy na odczytanie kolejnych danych, ponieważ poprzednie zapytanie znajduje się już w buforze modułu diagnostycznego Dcm.

- WyslijDane() – (tylko dla zapisu danych) funkcja, która kopiuje dane z bufora modułu diagnostycznego Dcm do docelowego modułu aplikacyjnego. Na przykład funkcja może wysyłać zlecenie po magistrali komunikacyjnej – zapisz nowy zestaw informacji na temat numerów seryjnych modemu.

Nie każdy serwis był możliwy do zautomatyzowania. Wiele danych diagnostycznych musiało dalej być zaimplementowane w specyficzny, unikalny sposób. Rozgraniczenie stanowiło pochodzenie danych, jeżeli pochodziły z innego rdzenia i były odczytywane właściwym protokołem, lub były odczytywane z pamięci nieulotnej, to mogły być całkowicie generowane. Jeżeli jednak specyficzna dana musiała być ręcznie zakodowana lub pochodziła z niestandardowego interfejsu, kod był unikalny. Przykładowo wszystkie zapytania do modułu kryptograficznego HSM musiały być realizowane ręcznie. Aby umieścić ręcznie pisane

fragmenty kodu w generowane pliki, wprowadzono tak zwanych „wartowników” (ang. sentinels), tj. odpowiednie łańcuchy znaków, które były umieszczane przed i po każdym ręcznie pisanym fragmencie kodu. Skrypt, generując nowy plik, odnajdywał wszystkie fragmenty kodu, które były otoczone wartownikami, zapisywał zawartość i umieszczał z powrotem po generacji kodu. Wartownicy zostali wprowadzeni na dwóch poziomach:

1. Na poziomie jednostek translacji – wartownicy modułowi. Wartownicy modułowi pozwalali na umieszczenie kodu w generowanych plikach źródłowych. Użytkownik mógł definiować swoje funkcje, zmienne, definicje itp. Po ręcznie pisanym fragmencie, skrypt umieszczał resztę generowanego kodu. Przykład takiego kodu został zaprezentowany jako fragment kodu źródłowego 5.3. Dotyczy on jednostki translacji realizującej serwisy powiązane z danymi produkcyjnymi samochodu. Szereg funkcji i zależności modułu jest pisana ręcznie, jednak znacząca większość jest generowana przez autorski skrypt.

2. Na poziomie funkcji – wartownicy funkcyjni. Wartownicy funkcyjni pozwalali na umieszczenie kodu wewnątrz generowanych funkcji, osobno dla każdego serwisu diagnostycznego. Nagłówek funkcji i dane pomocnice były generowane, ale sama zawartość funkcji była realizowana ręcznie. Przykład takiego kodu został zaprezentowany jako fragment kodu źródłowego 5.4. Dotyczy on implementacji specyficznej funkcji, gdzie kod musiał być pisany ręcznie, na przykład wykonanie zmiany unikalnego numeru seryjnego samochodu Vehicle Identification Number (VIN). Z powodu bardzo dużego stopnia skomplikowania tej procedury, musi ona być zaimplementowana ręcznie.

```
#include "Include1.h"
#include "Include2.h"
(...)
/* MODULE USER CODE START SENTINEL — DO NOT REMOVE. */
#define DEFINICJA_UZYTKOWNIKA (1u)
void SpecjalnaFunkcjaUzytkownika(void)
{
    WykonajKod();
}
(...)
/* MODULE USER CODE END SENTINEL — DO NOT REMOVE. */
```

Kod źródłowy 5.3. Wartownik modułowy dla pojedynczej jednostki translacji

```
/* Routine Name: Rutyna */
/* Routine Target: 0 */
/* Routine Category: Kategoria rutyny cyberbezpieczenstwa */
FUNC(Std_ReturnType, RTE_CODE) DiagAppl_RoutineServices_Rutyna_Start (
    Dcm_OpStatusType OpStatus, P2VAR(Dcm_NegativeResponseCodeType, AUTOMATIC
    , RTE_APPL_DATA) ErrorCode)
{
    /* This is a start function -- routine name: Rutyna */
    /* FUNCTION USER CODE START SENTINEL -- DO NOT REMOVE. */
    WykonajSpecyficznaRutyne();
    /* FUNCTION USER CODE END SENTINEL -- DO NOT REMOVE. */
}
```

Kod źródłowy 5.4. Wartownik modułowy dla pojedynczej funkcji

Warto wspomnieć, że nawet w przypadku ręcznie pisanego kodu, skrypt automatyzujący generował definicje dla użytkownika, które znacząco poprawiały identyfikowalność kodu (ang. traceability) w kontekście normy ISO 26262. Fragment kodu źródłowego 5.5 przedstawia definicje dla przykładowego jednego serwisu oraz jednego sygnału.

```
#define NAZWA_SERWISU_INDEX 0u
#define NAZWA_SERWISU_DID 0x185u
#define NAZWA_SERWISU_LENGTH 164u
#define NAZWA_SERWISU_CATEGORY KATEGORIA_DANE_GSM

#define NAZWA_SYGNALU_OFFSET 12u
#define NAZWA_SYGNALU_LENGTH 1u
```

Kod źródłowy 5.5. Przykład wygenerowanych definicji przez skrypt automatyzujący

Wartości definicji pochodzą bezpośrednio z pliku ARXML, przez co nie jest możliwe popełnienie błędu. Przy imporcie ekstraktu, wartości są aktualizowane. Kod źródłowy zawiera definicje długości, identyfikatora, kategorii serwisu oraz długość i pozycję sygnału diagnostycznego. Serwisem może być na przykład „Dane GSM" a sygnałem „Siła sygnału GSM".

Kolejnym elementem generowanego kodu, jest możliwość wprowadzenia logowania, zgodnie ze standardem AUTOSAR, przy użyciu modułu Diagnostic Log and Trace – (DLT). W zależności od wymagań projektowych, możliwe są dwie opcje:

1. Użycie globalnych funkcji „ManufacturerNotificatio" oraz „SupplierNotification". Są to dwie funkcje uruchamiane przed przetworzeniem każdego serwisu diagnostycznego. Jeżeli obciążenie CPU nie jest problemem, to można wprowadzić globalne logowanie każdego serwisu.

2. Użycie logowania w funkcjach dla specyficznych serwisów. Rozwiązanie to jest dużo bardziej dopasowane i umożliwia śledzenie całego przebiegu funkcji obsługującej dany serwis.

W rozdziale 5.2 przedstawiono sposób, w jaki zrealizować logowanie DLT, żeby nawet przy dużych ekstraktach diagnostycznych z setkami funkcji nie stanowiło to problemu dla zasobów systemowych.

Ostatnim opisanym elementem, który wyróżnia proponowane autorskie rozwiązanie, jest możliwość wprowadzenia analizy przebiegów czasowych. W każdej funkcji obsługującej serwisy diagnostyczne, można umieścić zapisywanie czasu otrzymania zapytania, długość i status przetwarzania oraz moment zwrócenia rezultatu. W połączeniu z logowaniem danych, znacząco skraca to czas analizy potencjalnych problemów. Wygenerowana funkcja, która odwzorowuje rozwiązanie opracowane przez autora i wdrożone w projekcie **P3**, w pełni obsługująca wszystkie opisane elementy, jest zaprezentowana na fragmencie kodu źródłowego 5.6.

```
1  /* DID Name: NazwaSerwisu*/
2  /* DID Signals Count: 2*/
3  /* DID Targets: Rdzen 1*/
4  /* DID Category: Kategoria Dane GSM*/
5  /* This signal: "NazwaSygnalu" belongs to DID: "NazwaSerwisu" and is 1 out
       of 2*/
6  FUNC(Std_ReturnType, RTE_CODE) DiagAppl_NazwaSerwisu_Read (Dcm_OpStatusType
       OpStatus, P2VAR(Dcm_NegativeResponseCodeType, AUTOMATIC, RTE_APPL_DATA)
       ErrorCode)
7  {
8      WyslijLog("Prosba odczytu serwisu 'NazwaSerwisu' w czasie xxxxxx");
9      PozyskajDane();
10     rezultat = ZwrocDane();
11
12     return rezultat;
13 }
```

Kod źródłowy 5.6. Przykład wygenerowanej funkcji odczytu z logami oraz zapisaniem czasu

Przykładowy kod zawiera wygenerowany komentarz ze szczegółami na temat serwisu diagnostycznego („DID Name”„,DID Signals” itd.). W środku prezentowanej funkcji widać funkcje obsługujące pozyskanie danych diagnostycznych i zwrócenie ich do stosu w celu dalszego przetworzenia. W celu dostosowania funkcji do innego projektu, należy podmienić zawartość funkcji na implementację zależną od sposobu pozyskiwania danych (np. magistrala komunikacyjna, pamięć nieulotna, porty RTE itp.).

Reasumując, metoda rozwiązania problemu badawczego zakłada generowanie automatyczne następujących elementów:

- Plików źródłowych w języku C, zawierających implementacje obsługi całych kategorii serwisów diagnostycznych, takich jak dane produkcyjne czy dane kierowcy.
- Plików nagłówkowych w języku C, zawierających makra i definicje, które ułatwiają programiście implementację pisanych ręcznie serwisów. Dzięki używaniu wygenerowanych elementów, zmniejsza się ryzyko popełnienia błędu oraz zwiększa łatwość aktualizacji wartości. Jako przykład można tu podać indeksy, długości oraz kategorie serwisów diagnostycznych.
- Funkcji w pełni obsługujących serwisy diagnostyczne dotyczące danych oraz procedur. Na przykład obsługa przetwarzania numeru seryjnego sterownika, czy procedury zapisania danych dotyczących działania silnika.
- Komentarzy w kodzie, pozwalających na wyśledzenie, z jakiego wymagania klienta wynika dany serwis diagnostyczny. Zawartość komentarzy jest generowana na bazie informacji odczytanych z ekstraktu diagnostycznego, dostarczanego przez producenta samochodu.
- Funkcji wysyłających logi na temat wykonania danego serwisu diagnostycznego wraz ze znacznikiem czasowym. Na przykład o godzinie 15:30:45 zostało odebrane i przetworzone zapytanie na temat aktualnego przebiegu samochodu.
- Raportów weryfikujących kod wynikowy, które podsumowują, czy wszystkie serwisy zawarte w ekstrakcie diagnostycznym są zaimplementowane i obsługiwane.

5.1.3 Walidacja i ocena rozwiązania

Walidacja rozwiązania, była oparta na kilku aspektach:

1. Porównaniu ilości ręcznie pisanego kodu.
2. Porównaniu czasu integracji ekstraktu diagnostycznego.
3. Ocenie parametrów jakości kodu opartych na międzynarodowej normie ISO 25010 [4].

4. Ocenie wpływu na bezpieczeństwo oparte na wymaganiach międzynarodowej normy ISO 26262 [5].

5. Ocenie wpływu na cyberbezpieczeństwo oparte na wymaganiach międzynarodowej normy ISO 21434 [6].

Tabela 5.2 przedstawia porównanie projektów przed i po automatyzacji – projekty **Projekt 1** oraz **Projekt 2** były realizowane bez zaproponowanych rozwiązań. W projekcie **Projekt 3** zostały natomiast wprowadzone autorskie rozwiązania przedstawione w niniejszym rozdziale. Co ważne, wszystkie trzy projekty realizowały ten sam typ sterownika, więc większość funkcji i wymagań była taka sama. Ekstrakty diagnostyczne, czyli dane wejściowe określające, jakie serwisy diagnostyczne są wymagane, były wspólne. W czasie prac rozwojowych, producent samochodów aktualizował ekstrakty, dodając dodatkowe serwisy (takie jak na przykład dodatkowe zestawy danych powiązane z technologią najnowszych sieci komórkowych 5G). Najstarszy **Projekt 1** zawiera 292 serwisów diagnostycznych, a najnowszy **Projekt 3** aż 343. Biorąc pod uwagę wspólne dane wejściowe, autor porównuje stosy diagnostyczne tych trzech projektów.

Z powodu wysokiej powtarzalności kodu, nawet pomimo większego ekstraktu diagnostycznego, rezultaty są bardzo zadowalające. Przy użyciu metody badawczej **MB1** i oprogramowania liczącego linie kodu [66] przeprowadzono porównanie bazy kodów źródłowych dla projektów bez automatyzacji **Projekt 1** i **Projekt 2** z projektem **Projekt 3**, gdzie wdrożono proponowane autorskie rozwiązanie.

Porównując moduły diagnostyczne dla projektów bez automatyzacji z projektem po automatyzacji (dane widoczne w tabeli 5.2):

- ilość ręcznie pisanego kodu została zmniejszona o ponad 91%, ponad 24 tysiące linii kodu w projektach **Projekt 1** i **Projekt 2** niezawierających automatyzacji oraz zaledwie 2 tysiące linii kodu w projekcie **Projekt 3**,

- ilość komentarzy została zwiększona o około 250%, w projektach **Projekt 1** i **Projekt 2** znalazło się około 2800 linii komentarzy i prawie 10 tysięcy w projekcie **Projekt 3**,

- ilość całkowitego kodu implementującego stos diagnostyczny została

Moduły diagnostyczne							
Projekt	Linie kodu	Linie komentarzy[a]	Funkcje	Linie pisane ręcznie[b]	DID[c]	RID [d]	Suma[e]
Projekt 1	30758	2783	1965	24863	249	43	292
Projekt 2	31568	2854	2059	25391	245	62	307
Projekt 3	24910	9942	2678	2053	276	67	343

Projekt 3 vs **Projekt 2**							
Różnica	-21,09%	248,35%	30,06%	-91,91%	12,65%	8,06%	11,73%

Projekt 3 vs **Projekt 1**							
Różnica	-19,01%	257,24%	36,28%	-91,74%	10,84%	55,81%	17,47%

Tablica 5.2. Porównanie projektów przed wprowadzeniem automatyzacji – **Projekt 1**, **Projekt 2** z projektem **Projekt 3**, który już zawierał autorskie rozwiązania

[a]Więcej – lepiej
[b]Mniej – lepiej
[c]Data Identifier – Dana diagnostyczna
[d]Routine Identifier – Procedura diagnostyczna
[e]Suma DID oraz RID, tj. liczba najważniejszych przetwarzanych serwisów diagnostycznych

zmniejszona o około 20%, ponad 30 tysięcy linii kodu w projektach **Projekt 1** i **Projekt 2** oraz 25 tysięcy w projekcie **Projekt 3**.

Generowany kod jest oparty na parametrach odczytanych z ekstraktu, więc nie ma możliwości błędu we wprowadzaniu danych. Liczba komentarzy została zwiększona o prawie 250%, a ich zawartość jest automatycznie importowana z ekstraktu diagnostycznego, co przyczynia się do lepszej czytelności kodu i łatwiejszej analizy potencjalnych błędów.

Z powodu wysokiego stopnia automatyzacji czas integracji zmniejszył się z 15 do 3 dni roboczych. Dodatkowo poprzez generowanie większości kodu, jest możliwe dopracowanie wzorców, które są używane do generacji, przez co można usunąć tzw. „zapachy kodu" (ang. bad smells) [72], czyli cechy kodu źródłowego, które są sygnałami do potrzeby refaktoryzacji. W publikacji [73] dowiedziono, że usuwanie takich elementów pozytywnie wpływa zarówno na jakość, jak i cyberbezpieczeństwo kodu.

Proces integracji nowego ekstraktu diagnostycznego jest znacznie uproszczony – wystarczy załadować go do opisanego automatycznego skryptu i uruchomić proces przetwarzania. Skrypt wykonuje porównanie obecnych plików oraz nowego ekstraktu, dodaje wszystkie nowe serwisy, usuwa wszystkie nieistniejące oraz wyświetla listę zmian. W każdym nowym serwisie zostaje umieszczona instrukcja przedstawiona na kodzie źródłowym 5.7, dzięki któremu użytkownik nie zapomni zaimplementować nowego serwisu – w przeciwnym razie kod się nie skompiluje.

```
1 #error Nastepujacy sygnal: NazwaSygnalu, nie jest zaimplementowany, prosze
    dodac kod uzytkownika.
```

Kod źródłowy 5.7. Kod zabezpieczający przed brakiem nowego sygnału

5.1.4 Odniesienie do norm jakości, bezpieczeństwa i zabezpieczeń

ISO 25010

Przedstawione rozwiązanie problemu **P1** zrealizowane poprzez wprowadzenie automatyzacji stosu diagnostycznego zwiększa jakość oprogramowania wpływając na elementy jakościowe normy ISO 25010 [4]. Tabela 5.3 opisuje kategorie, w ramach których nastąpiła poprawa jakości. Wszystkie kategorie jakości opisywane przez normę były wcześniej przedstawione na rysunku 2.5.

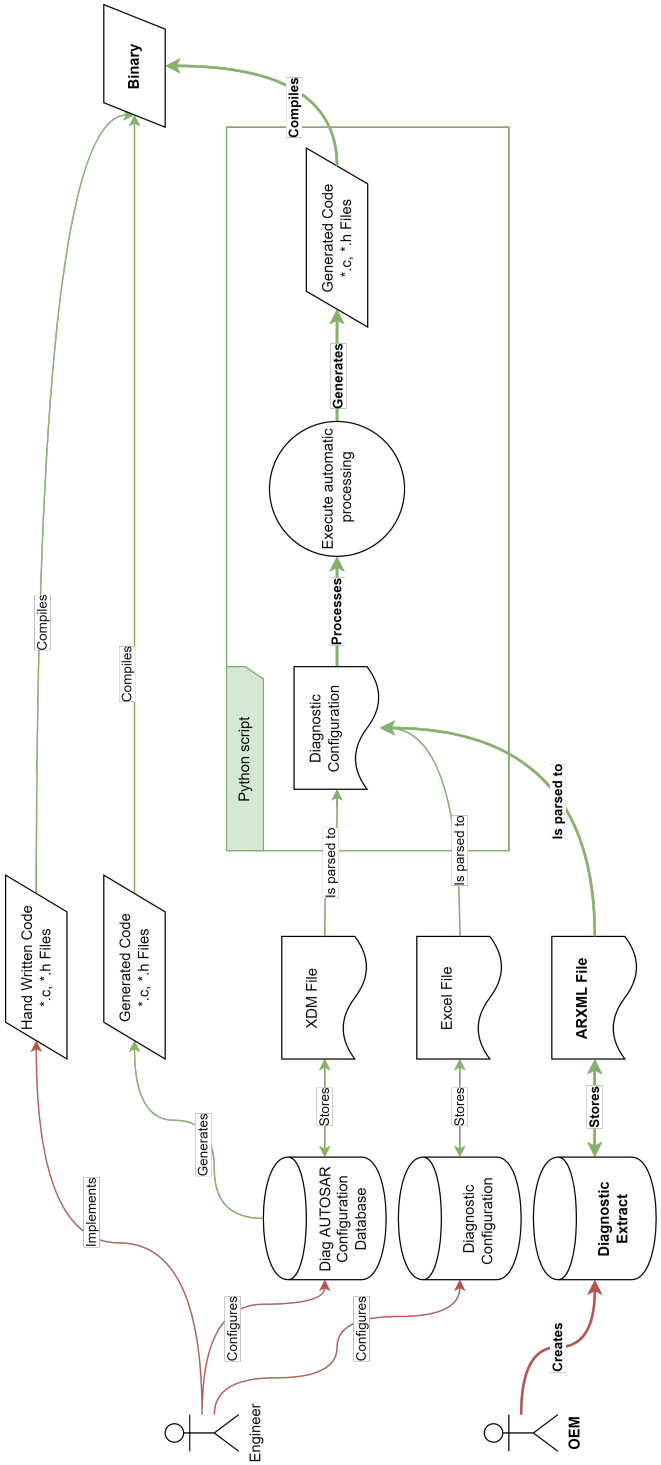
Relacja automatyzacji stosu diagnostycznego do normy ISO 25010		
Kategoria	Parametr	Opis w kontekście automatyzacji
Funkcjonalne dopasowanie	Funkcjonalna kompletność	Generowany kod jest oparty na ekstrakcie diagnostycznym, więc zapewnia, że wszystkie określone w nim serwisy będą obsłużone.
Kompatybilność	Interoperacyjność	Powiązanie systemu generowania kodu AUTOSAR i skryptu generującego kod, zwiększa interoperacyjność.
Użyteczność	Rozpoznawalność zastosowania	Faza weryfikacji zapewnia raport dla użytkownika, czy system implementuje wszystkie serwisy.
	Łatwość operowania	Cały proces generacji ogranicza się do zapewnienia ekstraktu i uruchomienia skryptu.
	Ochrona użytkownika przed błędami	Jakiekolwiek rozbieżności pomiędzy ekstraktem a generowanym kodem, są wyświetlane i przerywają proces.
Łatwość utrzymania	Modułowość	Podzielenie kodów źródłowych na kategorie serwisów zwiększa modułowość.
	Łatwość ponownego użycia	Generowany kod jest bardzo łatwo przenoszony na inne systemy.
	Łatwość analizy	Dzięki logowaniu danych oraz czasów wykonania, analiza jest znacznie uproszczona.
	Łatwość modyfikowania	Wszelkie modyfikacje muszą być wykonane tylko jednokrotnie, skrypt przeniesie je automatycznie na wszystkie generowane serwisy.
	Łatwość testowania	Tylko wzorce generowanego kodu muszą być przetestowane, ich użycie jest przy każdej funkcji takie samo. Zmniejsza to potrzebną liczbę testów.

Tablica 5.3. Odniesienie automatyzacji stosu diagnostycznego do normy ISO 25010

ISO 26262

W kontekście normy ISO 26262, zgodnie z zaleceniem **ISO 26262 7.4.2** wymagana jest dwukierunkowa zdolność wyśledzenia (ang. traceability). Jest to jeden z najważniejszych elementów procesu tworzenia oprogramowania w przemyśle motoryzacyjnym.

Proponowane rozwiązanie **Rozwiązanie 1** znacznie zwiększa zdolność wyśledzenia w kontekście wymagań dotyczących obszaru diagnostyki. Dzięki autorskiemu automatycznemu podejściu i generowaniu kodu bezpośrednio z danych odczytanych z ekstraktu diagnostycznego, praktycznie eliminuje się możliwość popełnienia błędu przy implementacji. Dodatkowo dzięki temu, że skrypt generuje dokumentację w formie komentarzy do funkcji (ang. fine specification) i umieszcza w nich nazwy sygnałów oraz serwisów, tworzy to zdolność wyśledzenia prowadzącego do ekstraktu diagnostycznego. Cała ścieżka zależności i śledzenia jest przedstawiona na diagramie 5.5. Bezpośrednia relacja wymagań z kodem wynikowym została przedstawiona przez zieloną wytłuszczoną ścieżkę. Zarówno ekstrakt diagnostyczny w formacie ARXML tworzony przez producenta samochodu, jak i pliki konfiguracyjne są razem przetwarzane i walidowane przez autorski skrypt. Dzięki automatycznej walidacji każdy serwis, taki jak na przykład pozycja GPS, może być powiązany bezpośrednio z wymaganiem klienta (producenta samochodu). Zgodnie z wnioskami artykułu badawczego [54], zdolność wyśledzenia jest jednym z wysoce pożądanych elementów w rozwoju oprogramowania.



Rysunek 5.5. Zdolność wyśledzenia elementów stosu diagnostycznego

ISO 21434

Automatyzacja stosu diagnostycznego przyczynia się do realizacji następujących zaleceń normy ISO 21434:

1. Zalecenie **[ISO 21434 RQ-09-08]** – Elementy oprogramowania powiązane z cyberbezpieczeństwem powinny być opisane, biorąc pod uwagę przejrzyste przedstawienie relacji między modułami i funkcjami wpływającymi na cele cyberbezpieczeństwa. Weryfikacja kodu oraz bezpośrednie powiązanie generowanego kodu z wymaganiami producenta samochodu stanowi szczegółową dokumentację. Komentarze są automatycznie generowane po zmianach wymagań producenta, więc są zawsze aktualne.

2. Zalecenie **[ISO 21434 RQ-10-04]** – Zwiększenie modułowości, poziomu abstrakcji i enkapsulacji, poprzez zastosowanie generowanego kodu opartego na ustalonych wzorcach. Poprawia to możliwości powtórnego wykorzystania artefaktów projektowych. Dodatkowo przez generowanie instrukcji pre-procesora „#error" w niezaimplementowanych serwisach diagnostycznych, zapewniona jest odporność na potencjalne podatności, wynikające z przeoczenia danego serwisu diagnostycznego (kod źródłowy 5.7).

3. Zalecenie **[ISO 21434 RQ-10-10]** – Realizowanie weryfikacji kodu powiązanego z celami zabezpieczeń poprzez statyczną analizę źródeł danych (ekstraktów diagnostycznych) oraz generowanego kodu wynikowego. Innymi słowy integracja nowego ekstraktu diagnostycznego wygeneruje w fazie weryfikacji danych raport, który podsumuje zmiany wymagań producenta.

5.1.5 Dostarczenie pakietu wiedzy i wzorców projektowych dla inżynierów oprogramowania

Ogólna architektura rozwiązania

Opisane zagadnienia zostały wdrożone w projekcie komercyjnym **Projekt 3**. Kody źródłowe są własnością intelektualną klienta, ale w tym rozdziale autor dostarcza wzorców oraz koncepcji architektury, które umożliwiają użycie rozwiązania w dowolnym projekcie korzystającym ze standardu AUTOSAR.

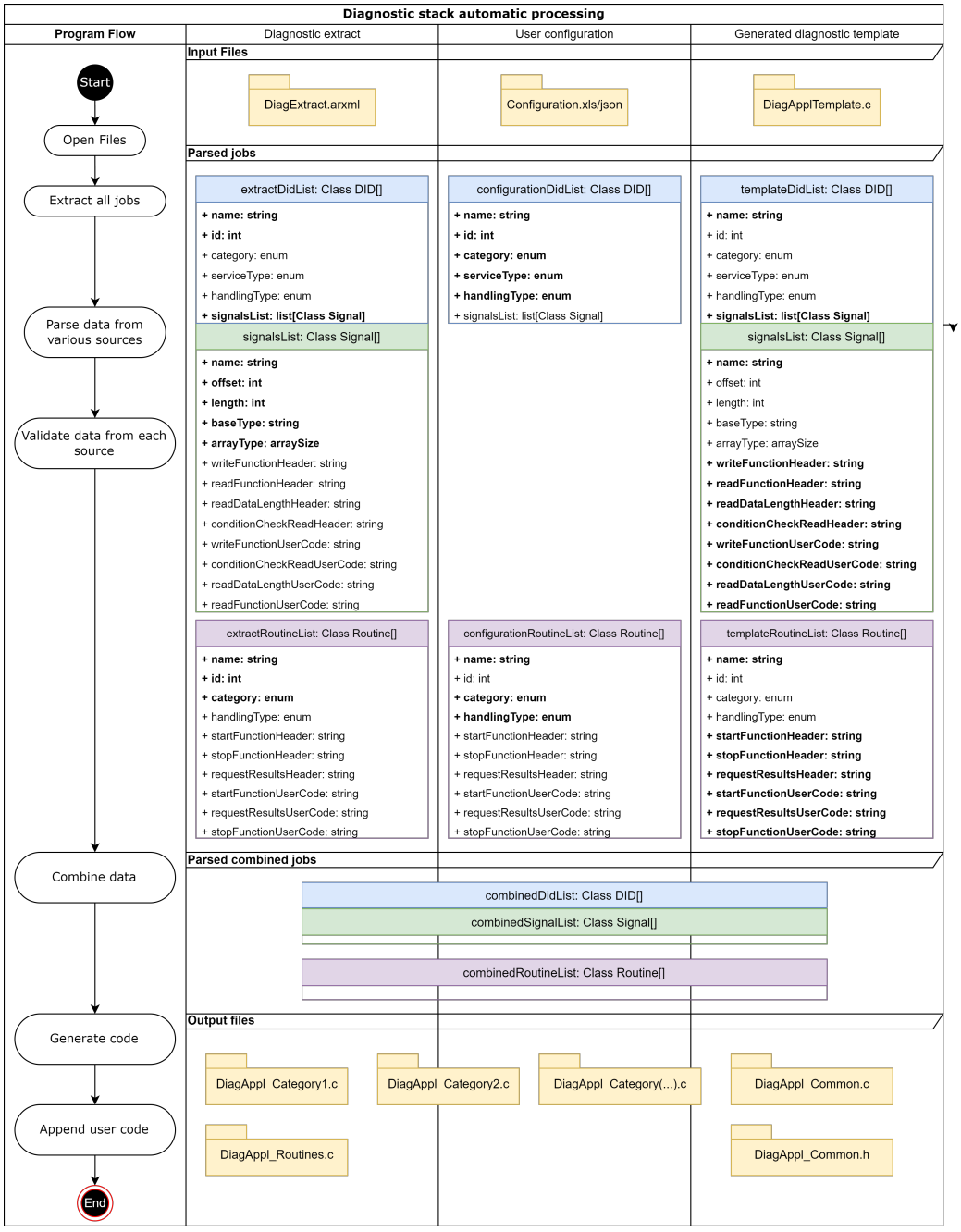
Ogólna architektura rozwiązania jest przedstawiona na diagramie 5.6, który został podzielony na kilka części widocznych po lewej stronie, w sekcji „Program Flow". W nawiasach umieszczono odniesienia do nazw angielskich na rysunku:

1. Załadowanie i odczytywanie plików wejściowych („Open Files") – ekstraktu diagnostycznego (DiagExtract.arxml), pliku konfiguracyjnego (Configuration.xls/json) oraz wygenerowanego przez narzędzie stosu AUTOSAR wzorca komponentu diagnostycznego (DiagApplTemplate.c).

2. Odczytanie danych z plików wejściowych („Extract all jobs"). Każdy z plików ma inny format danych, ale wspólnym elementem, który pozwala na porównywanie i grupowanie danych jest nazwa procedury lub danej diagnostycznej. Na diagramie w każdej liście zostały pogrubione elementy, które są odczytywane dla danego źródła. Kod źródłowy 5.8 przedstawia przykładowy serwis diagnostyczny o nazwie „NazwaSygnalu". Pola widoczne w kodzie źródłowym są opisane w tabeli 5.4. Odpowiednio plik konfiguracyjny oraz plik wzorcowy powinny mieć serwis o tej samej nazwie. Każdy plik wejściowy zawiera inny zestaw danych dla konkretnego serwisu. Celem fazy odczytania danych jest zebranie wszystkich informacji na temat każdego z sygnałów.

3. Przetworzenie i walidacja danych („Parse/Validate data from each source"), listy z danymi z każdego źródła są porównywane, wszelkie nieścisłości (np. brak danego sygnału w którymś ze źródeł) są wyświetlane użytkownikowi.

4. Agregacja danych („Combine data"), listy z danymi łączone są w jedną wspólną listę, która zawiera wszystkie potrzebne pola. Na podstawie tej agregowanej listy będzie generowany kod wynikowy. Przykładowo z ekstraktu używane są dane na temat identyfikatora serwisu, listy sygnałów, typu danych, długości oraz kategorii. Z pliku konfiguracyjnego jest używany sposób przetwarzania: ręcznie/automatycznie, kategoria, typ serwisu. Ze wzorca komponentu diagnostycznego używane są nagłówki funkcji, parametry wejściowe oraz listy sygnałów. Wszystkie te dane muszą być przypisane do każdego z serwisów.



Rysunek 5.6. Algorytm przetwarzania ekstraktu diagnostycznego

5. Generowanie plików wynikowych („Generate code”). Pliki źródłowe oraz pliki nagłówkowe zostają wygenerowane. Zawierają one wszystkie odczytane serwisy oraz procedury diagnostyczne. Generowanie wykorzystuje wszystkie wcześniej zagregowane dane.

6. Dodawanie kodu użytkownika („Append user code”), do wygenerowanych plików zostają dodane ręcznie pisane fragmenty kodu.

```
<DIAGNOSTIC-DATA-IDENTIFIER>
    <SHORT-NAME>NazwaDid</SHORT-NAME>
    <ID>8260</ID>
    <DATA-ELEMENTS>
        <DIAGNOSTIC-PARAMETER>
            <BIT-OFFSET>0</BIT-OFFSET>
            <DATA-ELEMENTS>
                <DIAGNOSTIC-DATA-ELEMENT>
                    <SHORT-NAME>NazwaSygnalu</SHORT-NAME>
                    <ARRAY-SIZE-SEMANTICS>FIXED-SIZE</ARRAY-SIZE-SEMANTICS>
                    <MAX-NUMBER-OF-ELEMENTS>4000</MAX-NUMBER-OF-ELEMENTS>
                    <SW-DATA-DEF-PROPS>
                        <SW-DATA-DEF-PROPS-VARIANTS>
                            <SW-DATA-DEF-PROPS-CONDITIONAL>
                                <BASE-TYPE-REF DEST="SW-BASE-TYPE">/Diagnostics/
    Common/BaseTypes/uint8_opaq</BASE-TYPE-REF>
                                <COMPU-METHOD-REF DEST="COMPU-METHOD">/Diagnostics/
    Common/CompuMethods/NazwaDid_4000_byte</COMPU-METHOD-REF>
                            </SW-DATA-DEF-PROPS-CONDITIONAL>
                        </SW-DATA-DEF-PROPS-VARIANTS>
                    </SW-DATA-DEF-PROPS>
                </DIAGNOSTIC-DATA-ELEMENT>
            </DATA-ELEMENTS>
        </DIAGNOSTIC-PARAMETER>
    </DATA-ELEMENTS>
</DIAGNOSTIC-DATA-IDENTIFIER>
```

Kod źródłowy 5.8. Przykładowy fragment ekstraktu diagnostycznego przedstawiający jeden serwis diagnostyczny (DID)

Instrukcja przetwarzania danych

Kluczowym elementem automatyzacji jest odpowiednie przetwarzanie danych wejściowych: ekstraktu diagnostycznego i plików wygenerowanych przez stos. W niniejszym rozdziale przedstawiona jest instrukcja określającą, jakich znaczników oraz fragmentów kodu używać w procesie przetwarzania danych wejściowych. Biorąc pod uwagę, że standard AUTOSAR definiuje strukturę plików ARXML – instrukcja jest

uniwersalna i będzie odpowiednia dla każdego projektu zgodnego z AUTOSAR.

Ekstrakt ARXML – Zgodnie z dokumentacją dotyczącą ekstraktów diagnostycznych – [71], głównym elementem ważnym dla prezentowanej metody jest „Diagnostic Data Identifier", który musi posiadać przynajmniej jeden „Data Element". W kodzie źródłowym 5.8 został przedstawiony przykładowy element DID, wraz z jego parametrami. Przykładowy DID zawiera tylko jeden sygnał o długości 4000 bajtów.

Tabela 5.4 przedstawia wszystkie potrzebne parametry ekstraktu, dzięki którym można zidentyfikować serwis diagnostyczny dla usług ReadDataByIdentified oraz WriteDataByIdentifier.

Wygenerowany plik wzorcowy – Plik wzorcowy (ang. „Template") jest plikiem generowanym przez narzędzia stosu AUTOSAR dla każdego komponentu oprogramowania (ang. Software Component) i sterownika CDD. Pliki wzorcowe są wygenerowane w celu ułatwienia implementacji danej funkcji. W przypadku manualnej implementacji, integrator kopiuje wygenerowaną funkcję do swojego komponentu i zastępuje jej ciało właściwymi instrukcjami. W przypadku automatycznie generowanego kodu, skrypt musi odczytać wygenerowane funkcje, usunąć ich zawartość, przekopiować do docelowego kompilowanego pliku oraz wygenerować zawartość. W zależności od producenta i wersji narzędzi stosu AUTOSAR, pliki wzorcowe mogą mieć różny format. Przykładowa wygenerowana funkcja dla stosu diagnostycznego została przedstawiona we wcześniejszym rozdziale we fragmencie kodu źródłowego 5.1.

Konfiguracja użytkownika – Odczytując ekstrakt ARXML i plik wzorcowy, skrypt pozyska informacje na temat serwisów diagnostycznych wraz z ich parametrami. Brakującym elementem jest informacja, w jaki sposób serwisy diagnostyczne powinny być przetwarzane – czy mają być ręcznie obsługiwane, czy powinny wysyłać zapytanie do innego rdzenia, czy powinny korzystać z konkretnego interfejsu RTE lub odnieść się do pamięci nieulotnej. O obsłudze każdego serwisu musi zadecydować integrator.

W przypadku projektu komercyjnego **Projekt 3**, konfiguracja użytkownika była plikiem w formacie *.xls, ale można równie dobrze wykorzystać tutaj na przykład format *.json lub inny. Konfiguracja

Obiekt AUTOSAR	Nazwa parametru	Znacznik parametru ARXML	Znaczenie
Diagnostic Identifier Data	shortName	<SHORT-NAME>	Nazwa DID
Diagnostic Identifier Data	id	<ID>	Identyfikator DID
Diagnostic Identifier Data	dataElement	<DATA-ELEMENT>	Sygnał należący do DID
Diagnostic Element Data	shortName	<SHORT-NAME>	Nazwa sygnału
Diagnostic Element Data	arraySizeSemantics	<ARRAY-SIZE-SEMANTICS>	Typ rozmiaru sygnału
Diagnostic Element Data	maxNumberOfElements	<MAX-NUMBER-OF-ELEMENTS>	Maksymalny rozmiar sygnału

Tablica 5.4. Opis parametrów ekstraktu diagnostycznego ARXML

użytkownika na pewno musi zawierać nazwy serwisów, aby skrypt mógł w automatyczny sposób zweryfikować, czy każdy serwis został skonfigurowany i wygenerować implementację. Pozostałe elementy konfiguracji są zależne od konkretnego projektu i jego architektury.

Przekazywanie informacji do obserwatora systemu

Dzięki temu, że obsługa wszystkich serwisów diagnostycznych jest generowana, bardzo łatwo jest zintegrować przetwarzanie stosu diagnostycznego z autorskim modułem obserwatora systemu opisanym w rozdziale 5.4. Dla wszystkich serwisów (lub wybranych przez integratora w pliku konfiguracyjnym), można wygenerować obsługę logu DLT, wygenerowanie znacznika czasowego, lub dodać punkt kontrolny dla watchdoga. W przypadku krytycznych serwisów diagnostycznych, których przetwarzanie jest skomplikowane i rozbite na wiele faz, obserwator systemu może znacząco pomóc w weryfikowaniu poprawności wykonania.

5.2 Automatyzacja i architektura DLT

5.2.1 Nawiązanie do zagadnienia badawczego

Moduł DLT jest zdefiniowany przez standard AUTOSAR oraz określa sposób logowania danych. Standard definiuje szereg ustawień i możliwości zbierania informacji z różnych warstw oprogramowania. DLT opisuje format transmisji w ramkach zawierających nagłówek główny, opcjonalny nagłówek rozszerzony oraz dane. Nagłówki zawierają informacje pozwalające na przetwarzanie i filtrowanie wiadomości, takie jak np. identyfikator sterownika, poziom logowania, identyfikator aplikacji czy kontekstu. Przykładowo wiadomość może być zdefiniowana w następujący sposób:

- Identyfikator sterownika – „BMS1”, czyli Battery Management System, sterownik odpowiedzialny za zarządzanie zasilaniem samochodu.
- Identyfikator aplikacji – „DIAG”, czyli logi pochodzące z diagnostyki.
- Identyfikator kontekstu – „PROD”, czyli log dotyczący danych diagnostycznych powiązanych z procesem produkcji samochodu.

- Znacznik czasowy – moment wystąpienia loga, generowany przez zegar systemowy sterownika.
- Poziom logowania – „WARN", czyli ostrzeżenie. Taki log nie będzie wyświetlany, gdy użytkownik zdecyduje, że chce widzieć tylko błędy.

Opisane parametry są powiązane z konkretną wiadomością o danym identyfikatorze, która jest transmitowana w określonych momentach działania systemu. Ilość takich wiadomości zależy od specyfiki działania sterownika. Konkretny log może być wyświetlany tylko raz podczas startu samochodu, lub np. co 100 ms, raportując wartość napięcia na klemie akumulatora.

Dokumentacja DLT nie zapewnia jednak zalecanej struktury użycia modułu. Aspekt ten leży w zakresie odpowiedzialności integratora stosu i architekta systemu. **Rozwiązanie 2** – automatyzacja oraz architektura DLT, jest zaproponowanym rozwiązaniem zidentyfikowanych problemów **P2** oraz **P3**, opisanych w rozdziałach 3.2.1 oraz 3.3.1. W celu zrealizowania badań dotyczących zagadnień badawczych **ZB2** i **ZB3**, zostały wykorzystane metody badawcze **MB1**, **MB2**, **MB3**.

Analizy wykonane w ramach artykułu badawczego

Pierwszy etap analiz został przeprowadzony przez autora w ramach artykułu badawczego [74]. Badania i analizy zostały wykonane na projekcie komercyjnym **Projekt 1**. Wykorzystując metodę badawczą **MB1** określono, że w projekcie większość wysyłanych logów, to były wiadomości statyczne, tzn. niezmienne wiadomości oznaczające, że oprogramowanie wykonało jakąś procedurę. Brak wiadomości dynamicznych, a więc posiadających transmitowane wartości zmiennych, oznaczał duży potencjał na wprowadzenie binarnego trybu transmisji „non-verbose". Tryb ten oznacza, że zamiast wysyłać wiadomość z logami, zawierającą wszystkie parametry i informacje, np.

„Sygnał stacyjki został zmieniony z wartości OFF na wartość IGNITION, o godzinie 12:48:02",

można wysłać sam identyfikator wiadomości, wraz ze zmiennymi. Przykładowa wiadomość wyglądałaby następująco:

01 00 05 12 48 02,

gdzie 01 to identyfikator wiadomości o zmianie sygnału stacyjki, 00 to wartość sygnału „OFF", 05 to wartość sygnału „IGNITION" a 12, 48, 02 to odpowiednio godziny, minuty oraz sekundy. Binarny tryb „non-verbose" pozwolił wysłać tę samą wiadomość, używając tylko 6 bajtów zamiast 88 bajtów w przypadku pierwszej wiadomości w standardowym trybie „verbose", wysyłającym dane jako łańcuch znaków.

Dodatkowe analizy przeprowadzone przez autora za pomocą metod badawczych **MB2** oraz **MB3**, pozwoliły finalnie określić architekturę logowania dla projektu **Projekt 1** i uzyskać znaczącą poprawę parametrów wydajności systemu:

1. Zajętość pamięci flash została zmniejszona o 86,66%.
2. Średnie obciążenie CPU wynikające z operacji modułu DLT zostało zmniejszone o 50,15%.
3. Średnie obciążenie magistrali komunikacyjnej zostało zmniejszone o 70,3%.

Zidentyfikowano także kolejne kroki dla analiz powiązanych z DLT: automatyzację generowania logów, rozwijanie trybu „non-verbose" oraz dalsze wykorzystanie i automatyzację generowania plików opisowych Field Bus Exchange Format – (FIBEX). Pliki FIBEX są oparte na formacie XML, stosowanym w przemyśle motoryzacyjnym dla opisywania struktur magistral komunikacyjnych oraz logowania danych. W przypadku użycia dla logowania, pliki zawierają definicje i opisy wiadomości z logami. Zgodnie ze zidentyfikowanymi celami, autor kontynuował analizy i badania w projektach komercyjnych **Projekt 2** oraz **Projekt 3**.

Analiza statyczna

Projekty komercyjne **Projekt 2** oraz **Projekt 3** miały inną strukturę logowania, ale autor wykorzystując badania wykonane w ramach artykułu badawczego, mógł wykorzystać zaproponowaną architekturę logowania, aby także uzyskać lepszą wydajność systemu. Autor zastosował metodę badawczą **MB1**, aby określić zajętość pamięci flash dla wybranych komponentów oprogramowania, które posiadały największą liczbę logów. Wyniki analizy zostały przedstawione w tabeli 5.5. Celem analizy było stwierdzenie, jak dużo danych jest transmitowanych podczas logowania i

	Moduł 1	**Moduł 2**	**Moduł 3**
Zajętość pamięci flash [bajty]	3385	2210	1984
Liczba wiadomości	86	58	54

Tablica 5.5. Liczba logów w trzech największych modułach aplikacyjnych

czy wprowadzenie optymalizacji jest wymagane. Wyniki powyżej 50 wiadomości na komponent wskazują na duży potencjał na optymalizację.

Dodatkowym aspektem, zidentyfikowanym podczas analizy statycznej, był niski poziom łatwości analizy, łatwości testowania i używalności w kontekście normy ISO 25010. W wyniku braku wykorzystania odpowiedniej ilości kontekstów, jawnego podziału aplikacji oraz poziomów logowania, logi były mało czytelne – każdy moduł produkował dużą ilość logów, które nie były możliwe do łatwego filtrowania lub kategoryzacji. Rezultaty analizy statycznej w kontekście problemów **P2** oraz **P3**, wykazały, że architektura modułu DLT znacząco wpływa na jakość i bezpieczeństwo oprogramowania.

5.2.2 Sposób rozwiązania problemu badawczego

Architektura

Sposoby realizacji architektury i konfiguracji modułu DLT opisano przez autora w artykule [74], główne elementy zaprezentowanych rozwiązań zostały zaś podsumowane w tabeli 5.6. Opracowane sposoby zostały również wdrożone w projekcie komercyjnym **Projekt 3**, gdzie z powodu wykorzystania innego mikrokontrolera obciążenie CPU było bardzo wysokie, dlatego optymalizacja była pożądana. Proces wprowadzania opisanych elementów architektury i konfiguracji do istniejących modułów, okazał się czasochłonny i w dużym stopniu powtarzalny, co naprowadziło na kolejny element rozwiązania problemów badawczych **P2** oraz **P3** – automatyzację procesu integracji modułu DLT dla komponentów oprogramowania.

Automatyzacja

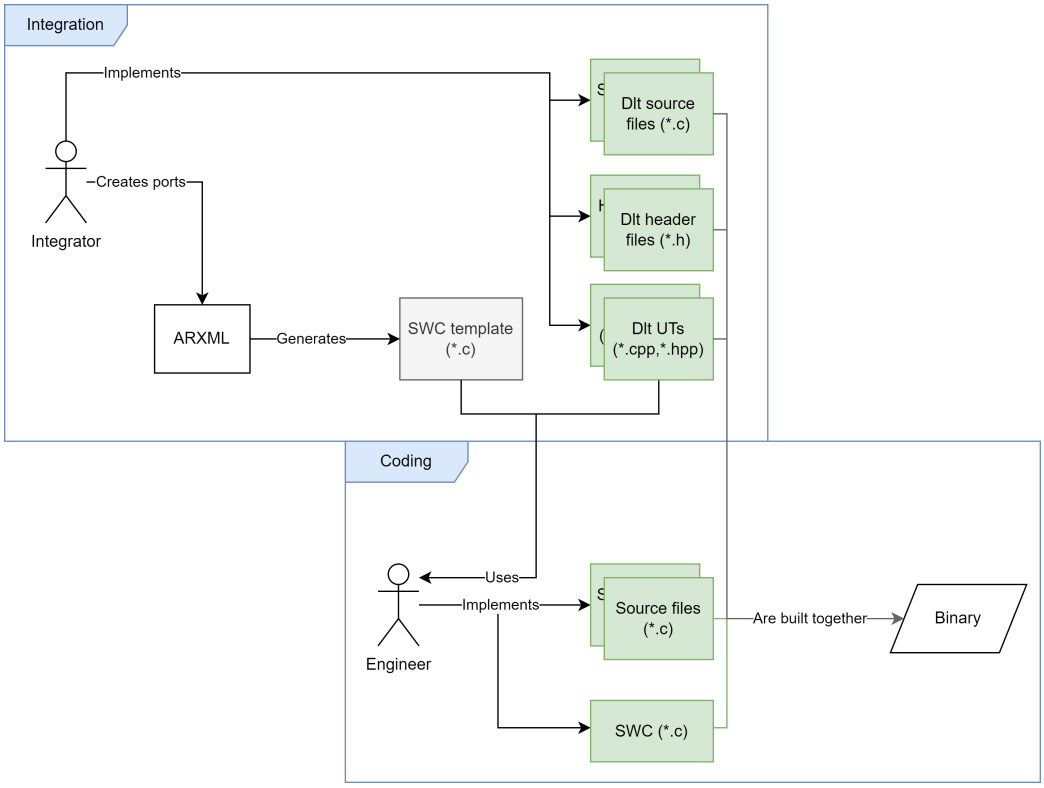
Wykorzystanie trybu „non-verbose” oraz logowania binarnego wymaga zwiększonego wysiłku podczas konfiguracji. W przypadku wykorzystania trybu „verbose”, jeden wspólny moduł nadawczy może być wykorzystywany

Kategoria	**Opis**	**Powiązany zasób**
Architektura oraz konfiguracja	Wykorzystanie trybu „non-verbose" i użycie plików FIBEX wraz z powiązaną aplikacją dekodującą wiadomości binarne.	Zajętość pamięci, obciążenie magistrali komunikacyjnej oraz mikrokontrolera.
Architektura	Łączenie wielu logów w jedną binarną ramkę, gdzie tylko to możliwe.	Obciążenie magistrali komunikacyjnej.
Architektura	Przetwarzanie poziomów logowania w każdym komponencie osobno, zamiast centralnie przez sterownik DLT.	Obciążenie mikrokontrolera.
Konfiguracja	Odpowiednie określanie poziomów logowania, w celu minimalizacji wysyłania wszystkich wiadomości przez cały okres działania systemu.	Obciążenie magistrali komunikacyjnej oraz mikrokontrolera.
Konfiguracja	Minimalizacja nagłówka DLT i wykorzystanie tylko potrzebnych pól.	Obciążenie magistrali komunikacyjnej.

Tablica 5.6. Główne obszary optymalizacji architektury oraz konfiguracji modułu DLT

dla wielu modułów, pliki FIBEX nie są wymagane, więc wprowadzenie nowych logów jest szybsze. W przypadku manualnej konfiguracji logowania dla wielu komponentów aplikacyjnych, tryb „non-verbose" może być uciążliwy i powodować występowanie błędów oraz rozbieżności wynikających z wymagania tworzenia plików FIBEX dla każdego komponentu. Analizie poddano trzy opisane przypadki.

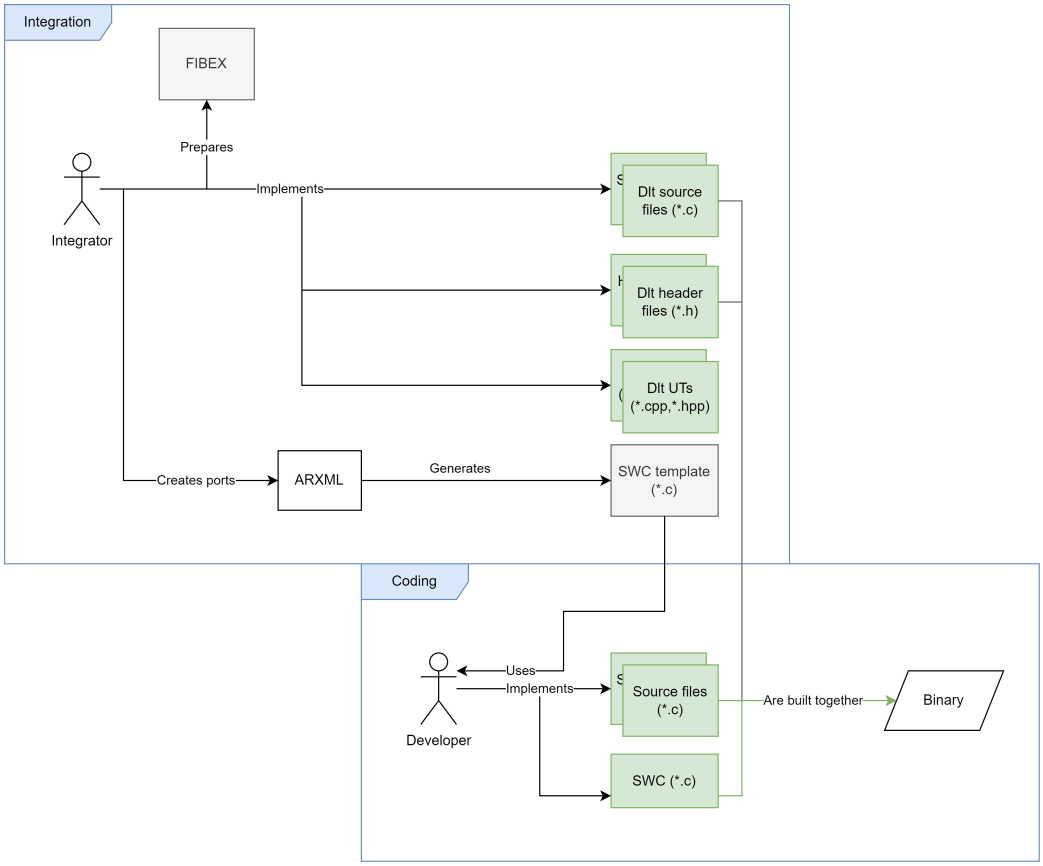
Diagram 5.7 przedstawia manualną konfigurację w trybie „verbose". Integrator opcjonalnie tworzy porty dla warstwy aplikacji, na których podstawie jest generowany plik wzorcowy dla danego komponentu („SWC Template"). Wszystkie działania powiązane z logowaniem, leżą po stronie programisty: musi on zaimplementować osadzenie modułu DLT w danym komponencie, zarejestrować kontekst, stworzyć zawartość logów, przygotować wiadomości i osadzić wiadomości w odpowiednich miejscach logiki aplikacyjnej (wszystkie zielone bloki zaznaczone na rysunku 5.7).



Rysunek 5.7. Proces manualnej integracji logowania w trybie „verbose”

Podejście to jest najbardziej uniwersalne, ponieważ jeden moduł pośredniczący z komunikacją ze sterownikiem DLT, może być wykorzystywany dla większej liczby komponentów. Niestety ogromną wadą trybu „verbose” jest znacząco większe obciążenie mikrokontrolera oraz pamięci, co dowiedziono przez autora w artykule [74].

Drugi przypadek – wykorzystanie trybu „non-verbose”, poza wymienionymi zaletami dotyczącymi zasobów, wymaga stworzenia pliku opisowego FIBEX. Diagram 5.8 przedstawia manualną integrację logowania w tym trybie. Integrator musi przygotować plik opisowy FIBEX, zdefiniować wszystkie potrzebne wiadomości wraz z ich strukturą i danymi, podobnie jak w trybie „verbose” przygotować osadzenie modułu DLT w danym komponencie („Dlt source files” oraz „Dlt header files”) oraz przygotować porty komunikacyjne w pliku ARXML. Programista może później wykorzystać przygotowaną konfigurację DLT dla konkretnego



Rysunek 5.8. Proces manualnej integracji logowania w trybie „non-verbose”

komponentu SWC i użyć interfejsów logowania w logice biznesowej. Podejście to pozwala na optymalizację zasobów, lecz dalej wymaga dużego nakładu ręcznej pracy integratora, który może popełnić wiele błędów z powodu manualnej realizacji opisanych kroków.

Z powodu problemów obecnych w dwóch poprzednich przypadkach, metoda wdrożona w projekcie komercyjnym **Projekt 3** została zmodyfikowana poprzez automatyzację większości powtarzalnych czynności i oparciu się na jednym źródle danych – pliku konfiguracyjnym przygotowywanym przez integratora. Integrator przygotowuje jeden plik konfiguracyjny zawierający definicję wiadomości DLT dla konkretnego komponentu aplikacyjnego. W przypadku projektu komercyjnego **Projekt 3** był to plik w formacie JavaScript Object Notation (JSON).

Na bazie opisanego pliku, specjalnie przygotowany skrypt generuje:

1. Plik konfiguracyjny FIBEX.
2. Pliki źródłowe osadzające moduł DLT w danym komponencie („Dlt source files").
3. Pliki nagłówkowe osadzające moduł DLT w danym komponencie („Dlt header files").
4. Testy jednostkowe w języku C++, które testują osadzenie modułu DLT w komponencie („Dlt UTs").

Dzięki automatycznej generacji wymienionych plików, uzyskano wszystkie korzyści płynące z trybu „non-verbose", takie jak zmniejszona zajętość pamięci flash, mniej danych obciążających magistralę komunikacyjną, jednocześnie unikając głównej wady – dużego nakładu pracy, wymaganego do powtarzalnej czynności osadzenia modułu DLT w każdym komponencie, który potrzebuje transmitować logi.

5.2.3 Walidacja i ocena rozwiązania

Wyniki dla największych komponentów oprogramowania są przedstawione w tabeli 5.7. Dzięki wprowadzonym zmianom, znacznie

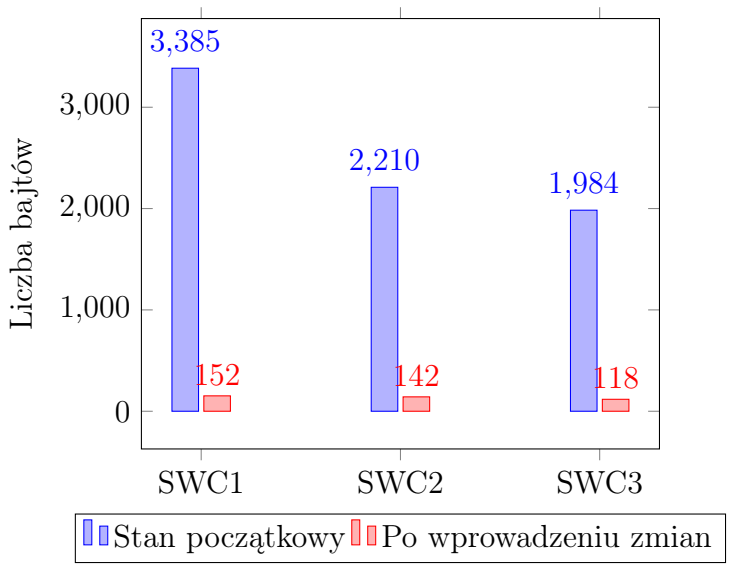
	SWC1	**SWC2**	**SWC3**
Zajętość pamięci flash początkowa	3385	2210	1984
Zajętość pamięci flash po zmianach	152	142	118
Procentowe zmniejszenie pamięci flash	95,51%	93,57%	94,05%
Liczba wiadomości początkowa	86	58	54
Liczba wiadomości po zmianach	76	71	59
Ilość logów (tekstu) w pliku FIBEX	4371	2272	2878
Procentowe zwiększenie liczby logów	22,56%	2,73%	31,06%
Liczba logów (wiadomości) w pliku FIBEX	192	166	117
Procentowe zwiększenie liczby wiadomości	55,21%	65,06%	53,85%

Tablica 5.7. Efekty wprowadzenia trybu „non-verbose" oraz logowania binarnego dla projektu **P3**, na przykładzie danych z trzech największych komponentów oprogramowania: SWC1, SWC2, SWC3

zmniejszono wymaganą pojemność pamięci flash – dla wszystkich przedstawionych komponentów o ponad 90%. Wynika to z przeniesienia treści wiadomości do plików opisowych FIBEX oraz transmisji ramek DLT z samym identyfikatorem wiadomości (ang. MessageId) i dynamicznymi parametrami.

Dodatkowym atutem tego rozwiązania jest możliwość znacznie bogatszego opisu wiadomości. Plik FIBEX jest przechowywany na odbiorczym komputerze PC, więc praktycznie nie ma ograniczeń dotyczących ilości pamięci na wiadomości. Ilość tekstu w logach dla komponentu SWC3 zwiększyła się o ponad 30%. Przedstawione rezultaty dowodzą, że wprowadzone optymalizacje w przypadku badanych komponentów wykazały znaczące zmniejszenie wymaganej pamięci flash, pomimo zawarcia większej ilości informacji w transmitowanych wiadomościach.

Wykres 5.9 przedstawia porównanie wymaganej pamięci flash dla trzech największych komponentów oprogramowania w projekcie komercyjnym **Projekt 3**, przed i po wprowadzeniu prezentowanych zmian w architekturze oraz automatyzacji.



Rysunek 5.9. Ilość danych przed i po optymalizacji – w trzech komponentach oprogramowania (SWC1, SWC2, SWC3)

5.2.4 Odniesienie do norm jakości, bezpieczeństwa i zabezpieczeń

ISO 25010

Przedstawione metody przyczyniające się do rozwiązania problemów **P2** oraz **P3**, zrealizowane poprzez wprowadzenie odpowiedniej architektury i automatyzacji konfiguracji modułu DLT poprawiają jakość oprogramowania, wpływając na elementy normy ISO 25010. Tabela 5.8 opisuje kategorie, w ramach których nastąpiła poprawa jakości. Wszystkie kategorie jakości opisywane przez normę były wcześniej przedstawione na rysunku 2.5.

Relacja automatyzacji stosu diagnostycznego do normy ISO 25010		
Kategoria	Parametr	Opis w kontekście automatyzacji
Wydajność	Charakterystyka czasowa	Zgodnie z wynikami badań z [74], czas transmisji danych został zmniejszony.
	Zużycie zasobów	Zgodnie z wynikami badań, zajętość pamięci flash została zmniejszona.
	Oczekiwana wydajność	Zgodnie z wynikami badań z [74], obciążenie CPU zostało zmniejszone.
Przenośność	Łatwość adaptacji	Dzięki generycznemu podejściu, skrypt może być łatwo zaadaptowany w innym projekcie.
	Łatwość zamiany	DLT i zaproponowana automatyzacja jest niezależna od używanej magistrali komunikacyjnej oraz narzędzi przetwarzających logi.
Użyteczność	Rozpoznawalność zastosowania	Dzięki bardziej rozległemu logowaniu, użytkownik może dobrze rozpoznać poprawność zastosowania.
	Łatwość operowania	Proces generacji ogranicza się do zdefiniowania wiadomości i utworzenia portów.
	Ochrona użytkownika przed błędami	Proces automatyzacji eliminuje potencjalne błędy ludzkie przy ręcznym pisaniu funkcji realizujących osadzenie DLT.
Łatwość utrzymania	Modułowość	Podzielenie kodów źródłowych na kategorie serwisów zwiększa modułowość.
	Łatwość ponownego użycia	Generowany kod jest bardzo łatwo przenoszony na inne systemy.
	Łatwość analizy	Dzięki większej ilości logowanych danych, analiza jest znacznie uproszczona.
	Łatwość modyfikowania	Wszelkie modyfikacje muszą być wykonane tylko w pliku konfiguracyjnym i skrypcie, inne moduły są generowane.
	Łatwość testowania	Testy są generowane automatycznie i wszędzie w ten sam sposób.

Tablica 5.8. Odniesienie architektury DLT do normy ISO 25010

ISO 26262

W kontekście normy ISO 26262, zwiększona liczba logów i ilość informacji w nich zawartych powoduje wzrost niezawodności oprogramowania i umożliwia realizację weryfikacji wielu parametrów, powiązanych z bezpieczeństwem funkcyjnym, takich jak: spójność interfejsów, łatwość zrozumienia kodu, prostota kodu, odpowiednia kolejność wykonania kodu (zalecenie **ISO 26262 8.4.4**). Dodatkowo zgodnie z zaleceniem **ISO 26262 7.4.17** proponowana architektura znacząco zmniejsza zużycie zasobów, co pozytywnie wpływa na wydajność systemu. Kolejnym aspektem zaprezentowanej architektury i automatyzacji jest poprawa zdolności wyśledzenia wymagań, rozszerzone logi mogą bowiem zawierać odniesienie do wymagań, zapisane w plikach FIBEX (zalecenie **ISO 26262 7.4.2**).

ISO 21434

Automatyzacja oraz architektura DLT przyczynia się do realizacji następujących zaleceń normy ISO 21434:

1. Zalecenia **[ISO 21434 RQ-08-05]** oraz **[ISO 21434 RQ-09-02]** – zwiększona ilość logów i informacji na temat systemu pozwala na analizę potencjalnych błędów stanowiących podatności cyberbezpieczeństwa. Zalecenia określają, że elementy oprogramowania ważne dla celów cyberbezpieczeństwa muszą być odpowiednio udokumentowane. Pliki FIBEX pozwalają na obszerne opisy zdarzeń systemowych bez obciążania zasobów mikrokontrolera (pliki FIBEX znajdują się na komputerze i są interpretowane przez aplikacje PC – nie zajmują pamięci mikrokontrolera).

2. Zalecenie **[ISO 21434 RQ-09-07]** – norma wymaga odpowiedniego zarządzania ryzykami cyberbezpieczeństwa. Automatycznie generowane logi pozwalają na weryfikację, czy np. zabezpieczona ramka, zawierająca sygnał stacyjki zapłonu samochodu, została potwierdzona przez moduł kryptograficzny HSM.

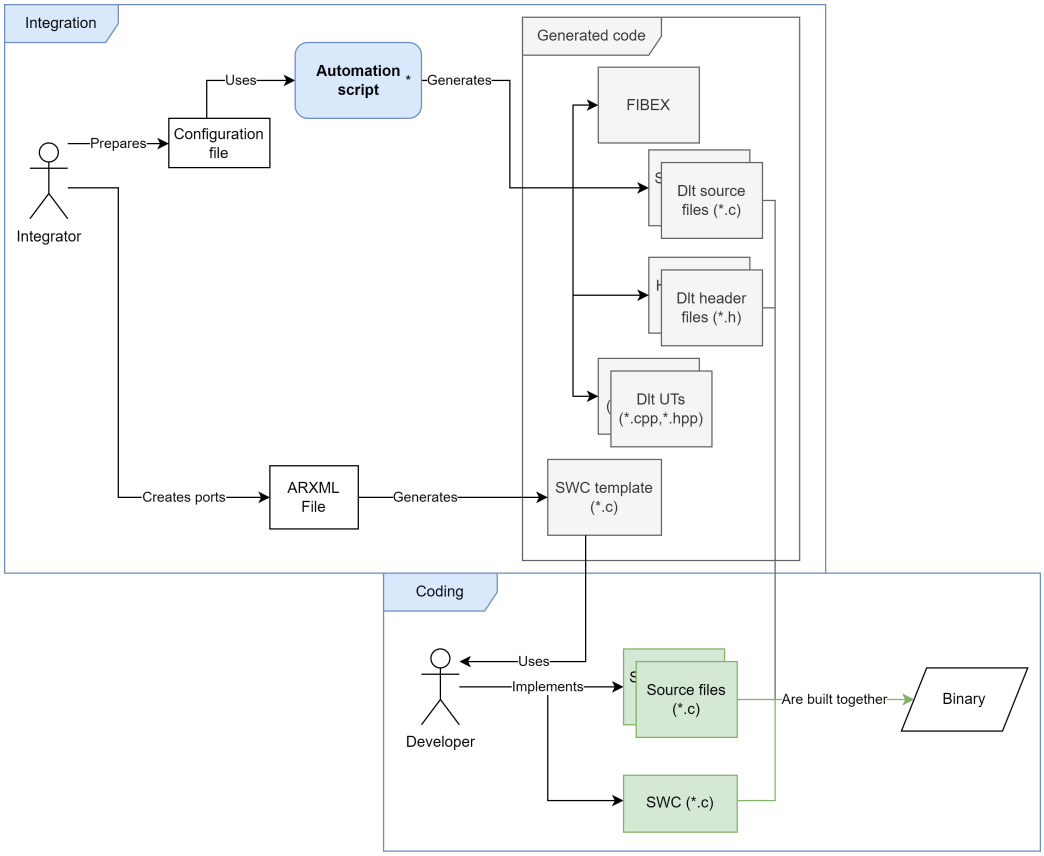
5.2.5 Dostarczenie pakietu i wzorców projektowych dla inżynierów oprogramowania

Opisane zagadnienia zostały wdrożone w projekcie komercyjnym **Projekt 3**. Kody źródłowe są własnością intelektualną klienta, ale w niniejszym rozdziale dostarczane są wzorce i koncepcje architektury, które umożliwiają użycie rozwiązania w dowolnym projekcie korzystającym ze standardu AUTOSAR.

Ogólna architektura rozwiązania

Zgodnie z określonymi problemami badawczymi **P2**, **P3**, celem zaproponowanej metody stosowania modułu DLT jest maksymalizacja ilości informacji na temat systemu poprzez transmitowanie dużej liczby logów, minimalizacja obciążenia zasobów mikrokontrolera i minimalizacja czasu potrzebnego na przygotowanie logów. Diagram 5.10 przedstawia zaproponowaną i wdrożoną metodę. Rysunek przedstawia fazę integracji modułu („Integration") DLT dla danego komponentu oprogramowania, w której praca inżyniera ogranicza się do określenia wszystkich potrzebnych logów w pliku konfiguracyjnym („Configuration file") oraz przygotowanie portów w pliku ARXML. Druga faza przedstawiona na rysunku to faza implementacji („Coding"), w której inżynier wykorzystuje wygenerowany kod oraz implementuje logikę biznesową w danym komponencie oprogramowania. Wysyłanie logów ogranicza się do wywołania wygenerowanych funkcji DLT przygotowanych w fazie integracji. Kluczowym elementem rozwiązania jest skrypt („Automation script") automatyzujący generację plików konfiguracyjnych DLT dla wszystkich modułów oprogramowania, gdzie potrzebne są logi („Generated Code"). Skrypt powinien realizować następujące kroki:

1. Skrypt powinien posiadać plik konfiguracyjny, gdzie użytkownik będzie mógł zdefiniować konfigurację DLT, potrzebne identyfikatory kontekstów, identyfikatory aplikacji, poziomy logowania, parametry i treść logów. Przykładowy fragment pliku konfiguracyjnego jest przedstawiony w kodzie źródłowym 5.9. Zawiera on przykład jednego loga, każda kolejna wiadomość powinna być zdefiniowana w podobny sposób. W celu dostosowania kodu do innego modułu należy zmienić „ApplicationId" oraz „MessageBaseId". W celu dodania kolejnych



Rysunek 5.10. Proces automatycznej integracji logowania w trybie „non-verbose”

wiadomości wystarczy skopiować wiadomość znajdującą się w polu „Messages”.

2. Skrypt powinien na bazie pliku konfiguracyjnego wygenerować: plik FIBEX, pliki źródłowe wraz z wymaganymi interfejsami DLT, pliki nagłówkowe oraz opcjonalnie testy jednostkowe.

```
"ApplicationId": "APP1",
"SessionId": 1000,
"MessageBaseId": 10000,
"Description": "This configuration contains\gls{dlt} message definitions for
    ComponentX",
"ComponentName": "ComponentX",
"Contexts": [
    "ContextId": "CTX1",
```

```
8       "Description": "Traces related to general operation of ComponentX",
9       "LogLevel": "DLT_LOG_INFO",
10      "Messages": [
11          "MessageId": "INFO_MESSAGE",
12          "Description": "Occurrence of info message.",
13          "LogLevel": "DLT_LOG_INFO",
14          "Arguments": [
15              "ArgumentType": "Static",
16              "Text": "Code has reached here."
17          ]
18      ]
19  ]
```

Kod źródłowy 5.9. Przykładowa konfiguracja pliku opisowego DLT dla pojedynczego komponentu oprogramowania, zawierająca jeden log

W przypadku wdrożonego skryptu, plik konfiguracyjny wykorzystywał format JSON, powodem tego wyboru była popularność tego formatu i znacznie prostsza składnia, niż w przypadku plików konfiguracyjnych FIBEX. Ponadto JSON pozwalał na dodawanie dodatkowych funkcji, takich jak generowanie większych fragmentów kodu.

Kolejnym ważnym aspektem jest implementacja przetwarzania logów podczas dwóch kluczowych momentów działania systemu – procedury inicjalizacji oraz procedury wyłączania. Wymienione procedury są bardzo skomplikowane i przedstawiają wiele zależności w systemach wbudowanych opartych na AUTOSAR, a co za tym idzie, są narażone na błędy. Stąd potrzeba jak najdokładniejszego logowania tych procedur. Niestety w obydwu przypadkach, powstaje problem inicjalizacji i deinicjalizacji samego modułu DLT oraz powiązanej z nim magistrali komunikacyjnej.

Dokumentacja modułu DLT [75] proponuje rozwiązanie fazy inicjalizacji – wystarczająco duży bufor, wykorzystywany do przechowywania logów zebranych zanim DLT zostanie zainicjalizowany. W przypadku procedury wyłączania systemu, moment deinicjalizacji DLT i transmisji końcowych logów musi zostać uwzględniony w architekturze monitorowania krytycznych faz systemu.

Przekazywanie informacji do obserwatora systemu

Zaproponowana architektura DLT wraz z automatyzacją generowania plików konfiguracyjnych wpływa pozytywnie na ilość informacji dotyczących działania systemu. Autorski moduł obserwatora systemu opisywany w rozdziale 5.4 wykorzystuje w dużej mierze moduł DLT do transmisji zebranych informacji. Metoda badawcza **MB6** opiera się na

monitorowaniu krytycznych faz systemu przy użyciu modułu obserwatora systemu oraz szczegółowych logów DLT. Dzięki zaproponowanemu rozwiązaniu, które minimalizuje obciążenie zasobów systemowych powiązanych z logowaniem, ilość transmitowanych danych jest minimalna, a ilość informacji, które są powiązane z każdym logiem pozwalają na analizę skomplikowanych problemów i zależności.

5.3 Automatyzacja i architektura stosu NvM

5.3.1 Nawiązanie do zagadnienia badawczego

Rozwiązanie 3 – automatyzacja oraz architektura stosu NvM, jest zaproponowanym rozwiązaniem zidentyfikowanych problemów **P2**, **P3** oraz **P4**, opisanych w rozdziałach 3.2.1 oraz 3.4.1. W celu zrealizowania badań dotyczących zagadnień badawczych **ZB2**, **ZB3** oraz **ZB4**, wykorzystano metody badawcze **MB1**, **MB2**, **MB4**, **MB5** i **MB6**. W kontekście stosu pamięci nieulotnej problem **P2** – ograniczeń zasobów sprzętowych dotyczy ilości oraz żywotności pamięci, problem **P3** – wysokiej złożoności oprogramowania dotyczy skomplikowanej architektury NvM, która powoduje bezpośrednio problem **P4** – skomplikowanych ograniczeń i zależności czasowych opisanych w niniejszym rozdziale.

Analizy wykonane w ramach artykułu badawczego

Dokonano analizy stosu NvM w ramach artykułu [8]. Celem badań było opracowanie algorytmu pozwalającego na estymację żywotności pamięci nieulotnej Flash Emulated EEPROM (FEE). Badania i analizy zostały wykonane na projekcie komercyjnym **Projekt 3**.

Przy użyciu metod badawczych **MB1**, **MB4** i **MB5** określono, jakie czynniki wpływają na częstotliwość przeprowadzania procesu czyszczenia sekcji pamięci i jej degradacji. Każda pamięć nieulotna ma skończoną, zdefiniowaną przez producenta liczbę zapisów i wyczyszczeń. Po ich wykorzystaniu system jest narażony na błędy i przekłamania danych.

Moduł AUTOSAR FEE odpowiada m.in. za realizację procesu balansowania pamięci, to znaczy odpowiedniego wykorzystania każdej komórki, aby nie zapisywać ciągle najczęściej używanych obszarów, pozostawiając inne nietknięte. Algorytm balansowania pamięci („wear-leveling") jest realizowany w następujący sposób: każda aktualizacja

wartości w pamięci nieulotnej nie jest nadpisywana w jej obecnej lokalizacji, ale zapisywana jako nowa instancja na pierwszej wolnej komórce. W pamięci może się znajdować jedna wartość numeru seryjnego pojazdu, ponieważ nie jest zmieniana po opuszczeniu fabryki, ale dziesiątki wartości przebiegu, ponieważ przy każdym wyłączeniu silnika jest zapisywana nowa wartość. Dopiero po zapełnieniu całego sektora pamięci danymi, sterownik wykonuje operację skopiowania wszystkich aktualnych wartości (czyli bieżący numer seryjny, bieżący przebieg itd.) do nowego pustego sektora i czyści obecny. W ten sposób pamięć jest równomiernie zużywana. Algorytm ten opisano dokładnie przez autora w artykule badawczym [8].

Artykuł skupiony był głównie na zagadnieniach badawczych **ZB1** i **ZB2**. Implementacja algorytmu w projekcie **Projekt 3** pozwoliła na ciągłe monitorowanie zasobu sprzętowego – pamięci, która jest wymieniana jako jeden z elementów powiązanych z bezpieczeństwem funkcyjnym, zgodnie z [5], jako jedna z metod testowania jednostek oprogramowania – testowania zużycia zasobów sprzętowych.

Analiza statyczna

Analiza statyczna polegała na identyfikacji miejsc w kodzie, gdzie stos NvM jest wywoływany ręcznie poza standardowo skonfigurowanymi zadaniami. Funkcje stosu NvM muszą być zaplanowane w systemie operacyjnym z określonym priorytetem oraz częstotliwością wykonania. Dokładnie chodzi o:

1. NvM_MainFunction() – zarządzającą blokami pamięci nieulotnej,
2. Fee_MainFunction() – zarządzającą wirtualnymi sektorami oraz algorytmem równoważenia zużycia pamięci,
3. Fls_MainFunction() – zarządzającą fizycznymi dostępami do komórek pamięci.

W większości implementacji stosu, wymienione funkcje nie są typu „reentrant", to znaczy nie mogą zostać przerwane i wywołane ponownie z innego kontekstu (na przykład innego rdzenia lub funkcji o wyższym priorytecie). Stanowi to problem w przypadku użycia ich wielokrotnie w kodzie w różnych przypadkach. Zaobserwowanym problemem jest przerywanie procedury zapisu lub odczytu danych przez nagłe

synchroniczne żądania wykorzystania stosu. Kod źródłowy 5.10 prezentuje przykładową implementację zadania systemu operacyjnego realizującego funkcje stosu NvM.

```
TASK(MemStack)
{
  Rte_Task_Dispatch(MemStack);
  {
    /* BswSchedulableEntity NvM_MainFunction */
    SchM_Schedulable_NvM_MainFunction_Start();
    NvM_MainFunction();
    SchM_Schedulable_NvM_MainFunction_Return();

    /* BswSchedulableEntity Fee_MainFunction */
    SchM_Schedulable_Fee_MainFunction_Start();
    Fee_MainFunction();
    SchM_Schedulable_Fee_MainFunction_Return();

    /* BswSchedulableEntity Fls_MainFunction */
    SchM_Schedulable_Fls_MainFunction_Start();
    Fls_MainFunction();
    SchM_Schedulable_Fls_MainFunction_Return();
  }

  (void)TerminateTask();

} /* TASK(MemStack) */
```

Kod źródłowy 5.10. Przykład zadania systemu operacyjnego realizującego funkcje stosu NvM

Najpierw wykonuje się NvM_MainFunction, która zleca operację na bloku logicznym, jako druga funkcja wprowadzana jest Fee_MainFunction(), która znajduje blok logiczny NvM w wirtualnej mapie pamięci i zleca fizyczną operację dla sterownika pamięci. Jako ostatnia uruchamiana jest Fls_MainFunction(), która realizuje fizyczny dostęp do danego adresu w pamięci. W tym przypadku można założyć, że priorytet zadania jest równy 100. Analizowany problem wystąpi, gdy inne zadanie z wyższym priorytetem będzie potrzebować nagłego synchronicznego zapisu do pamięci nieulotnej (kod źródłowy realizujący takie żądanie jest przedstawiony na 5.11) i przerwie wykonywane zadanie. W takim przypadku funkcje stosu mają niezdefiniowane zachowanie, co w zależności od architektury pamięci może doprowadzić nawet do zawieszenia systemu. W dalszej części pracy przedstawiono możliwe rozwiązania tego problemu.

```
void FunctionReadMemStack(void)
{
    NvM_RequestResultType tBlockStatus = NVM_REQ_NOT_OK;

    do
```

```
6       {
7           NvM_MainFunction();
8           Fee_MainFunction();
9           Fls_MainFunction();
10
11          (void)NvM_GetErrorStatus(NVM_BLOCK_MULTI, &tBlockStatus);
12
13      } while (NVM_REQ_PENDING == tBlockStatus);
14  }
```

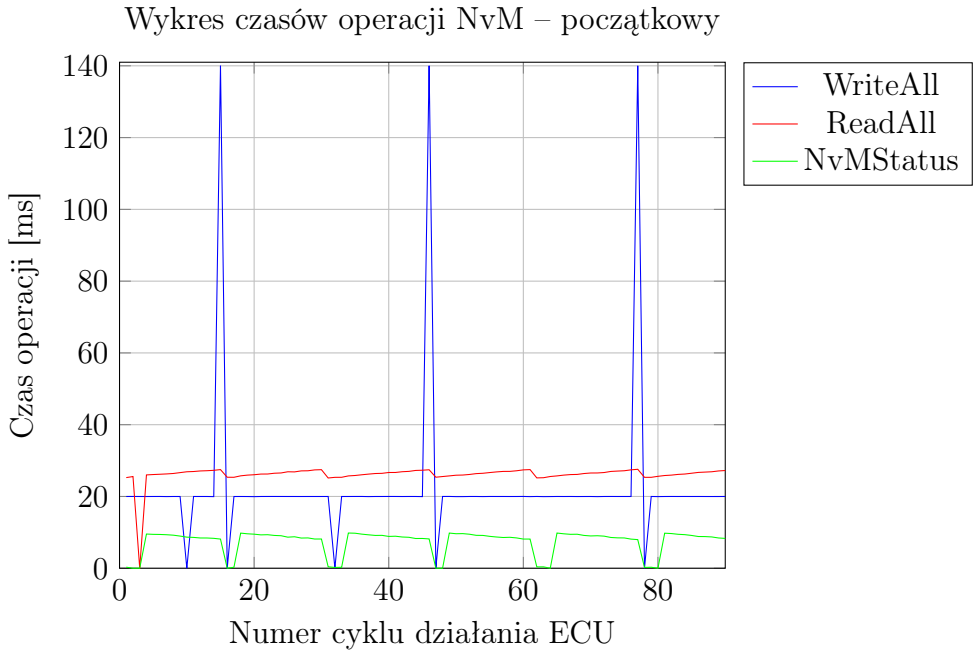
Kod źródłowy 5.11. Przykład synchronicznej funkcji realizującej funkcje stosu NvM

Analiza czasu wykonania kodu

Analiza czasu wykonania kodu, czyli metoda badawcza **MB2** polegała na przeprowadzeniu badań, jak długo trwa zapis i odczyt do pamięci nieulotnej – w szczególności w połączeniu z metodą badawczą **MB6**, czyli analizą przebiegu krytycznych faz systemu – uruchamiania się i deinicjalizacji.

Specjalny komponent obserwatora systemu opisany w rozdziale 5.4 został wykorzystany do zgromadzenia danych. Stos pamięci nieulotnej jest krytycznym elementem tych dwóch faz. Podczas uruchamiania systemu jest wykonywany odczyt pamięci, oznaczony jako NvM_ReadAll, a podczas procedury wyłączania systemu wykonywany jest zapis pamięci oznaczonej jako NvM_WriteAll. Obydwie te fazy realizują operacje na wybranych blokach pamięci. Zgodnie z dokumentacją AUTOSAR [76] jest to opcjonalny parametr. Integrator może zdecydować, że pamięć zostanie ręcznie odczytana po uruchomieniu systemu, aczkolwiek taki wybór implikuje znaczne opóźnienie w dostępności danych.

Wyniki badań czasu wykonywania kodu są przedstawione na wykresie 5.11. Jeden cykl działania ECU oznacza czas od włączenia do wyłączenia stacyjki zapłonu. Wyniki badań to rezultaty czasów operacji na pamięci w konkretnym cyklu działania sterownika. Stanowisko pomiarowe automatycznie zbierało dane z każdego cyklu, który trwał około 3 minut. Pomiary w sumie zajęły ponad 5 godzin (ponad 100 cykli, każdy po 3 minuty). Czasy odczytu oscylują około 25ms. Bardzo widoczne są wystąpienia procedury zmiany sektora pamięci powiązane z balansowaniem. Czasy odczytu wówczas rosną, gdyż dane ulokowane są w coraz dalszych adresach, aż dochodzą do końca sektora. Inicjalizowana jest wtenczas zmiana sektora, po czym czas odczytu spada do wartości początkowej.



Rysunek 5.11. Czasy operacji zapisu, odczytu oraz raportowania statusu bloków pamięci NvM zebrane podczas ponad 100 cykli działania sterownika ECU

Procedury zmiany sektora w niektórych przypadkach są wykonywane podczas działania systemu, a w trzech cyklach są wyraźnie uchwycone poprzez pomiar fazy zapisu. Widać wtedy znaczne zwiększenie czasu zapisu z 20 ms do ponad 140 ms. Niepokojącymi obserwacjami są wielokrotne wystąpienia pomiarów równych zeru – wskazuje to na poważny błąd systemu, który sprawia, że pomiar nie zostaje wykonany.

Dzięki tej analizie, udało się uchwycić poważny błąd związany z wielowątkowym dostępem do stosu pamięci nieulotnej. W dalszych rozdziałach jest opisany sposób rozwiązania opisanego problemu.

Analiza dokumentów

Metoda badawcza **MB4** została wykorzystana do określenia dokładnego sposobu implementacji procedur zarządzania pamięcią nieulotną i jej parametrów.

Dokumenty [77], [76], [78] i [79] określają generalną architekturę stosu pamięci nieulotnej. Są one wymagane w każdym projekcie zgodnym ze standardem AUTOSAR, ale nie regulują niektórych aspektów, takich jak na przykład podejście do procedury balansowania pamięci (ang. wear leveling) opisanego w artykule [8].

Artykuły badawcze poruszające temat balansowania pamięci: [80], [81], [82] zostały przeanalizowane w celu sprawdzenia, jak inne przemysły zajmują się tematyką żywotności pamięci. Temat ten jest bardzo dobrze opisany, w szczególności w kontekście przemysłu elektroniki użytkowej – dysków Solid State Drive (SSD), pamięci wbudowanych w telefonach komórkowych itp. Implementacje zgodne ze standardem AUTOSAR stosowane w motoryzacji są uproszczone i zarządzają mniejszymi pamięciami, ale zasady działania są podobne. Opracowano także przeanalizowane artykuły badawcze z obszaru motoryzacji – [83] i [84], gdzie zostały przedstawione ciekawe koncepcje powiązane z pamięcią nieulotną w systemach używających AUTOSAR, ale nie rozwiązywały one zidentyfikowanych problemów **P2**, **P3**, **P4**.

5.3.2 Sposób rozwiązania problemu badawczego

Tabela 5.9 przedstawia listę problemów powiązanych ze stosem pamięci nieulotnej NvM i proponowane w pracy sposoby ich rozwiązania.

Problem	**Opis**	**Sposób rozwiązania**
P2	Ograniczona pojemność i żywotność pamięci	Algorytm do obliczania żywotności
P3	Wysoka złożoność architektury NvM	Obserwator systemu
P4	Czas trwania operacji i konflikty dostępu do pamięci	Metoda zarządzania dostępami do pamięci

Tablica 5.9. Lista problemów badawczych stosu NvM

Algorytm do obliczania żywotności

Algorytm do obliczania żywotności został opracowany na bazie badań i analiz wykonanych w ramach artykułu badawczego [8]. Badania opisane w

artykule były wykonane w projekcie **Projekt 3**, gdzie została wprowadzona modyfikacja częstotliwości zapisu do pamięci, wykonywanym co 5,2 sekundy. Celem tej zmiany była walidacja proponowanego algorytmu obliczania żywotności.

Dzięki częstym dostępom do pamięci, zmiany sektorów zmniejszające żywotność były wykonywane znacznie częściej, niż w standardowych warunkach w samochodzie.

Kolejnym krokiem wykonanym jako kontynuacja prac opisanych w artykule, była implementacja specjalnych skryptów obliczających żywotność pamięci na bazie konfiguracji stosu NvM z projektu **Projekt 3**. Pierwszy skrypt służył do eksportu konfiguracji NvM z narzędzia EB Tresos [85], używanego do generacji stosu AUTOSAR. Drugi skrypt napisany w języku Python używał eksportowanej konfiguracji, przetwarzał dane, obliczał, ile bloków jest zapisywane podczas procedur NvM_ReadAll i Nvm_WriteAll, zakładał częstotliwość uruchamiania samochodu na poziomie 4 uruchomień samochodu dziennie, a na bazie tych parametrów estymował oczekiwaną żywotność pamięci. Dodatkowo skrypt przeszukiwał cały kod źródłowy i listował ręczne wywołania stosu pamięci nieulotnej, które są powiązane z problemami **P3** oraz **P4**. Parametry użyte przez skrypt i wyniki estymacji żywotności są przedstawione w tabeli 5.10. Wynik około 59 lat działania

Parametr	**Wartość**
Rozmiar sekcji pamięci	65535 bajtów
Suma zajmowanej pamięci	30820 bajtów
Ilość danych odczytywanych podczas podczas inicjalizacji systemu	23812 bajtów
Ilość danych odczytywanych podczas działania systemu	7008 bajtów
Ilość danych zapisywanych podczas działania systemu	22748 bajtów
Ilość danych zapisywanych podczas deinicjalizacji systemu	8072 bajtów
Estymowana żywotność pamięci	**58,93 lat**

Tablica 5.10. Parametry badanego stosu pamięci NvM

systemu jest estymacją, gdyż dla dokładnego obliczenia żywotności, należałoby znać dokładną częstotliwość zapisywania bloków do pamięci. W przypadku przygotowanego skryptu, autor założył 4 uruchomienia samochodu dziennie, przez 365 dni w roku. W przypadku badanego projektu, okres żywotności jest duży, natomiast należy pamiętać, że często

samochody są wykorzystywane w bardzo intensywny sposób, mogą służyć jako narzędzie pracy, codziennie mogą być wielokrotnie włączane i faktycznie używane przez cały rok.

Metoda zarządzania dostępami do pamięci

Analiza statyczna kodu oraz dokumentacji wykazały, że znaczącym problemem stosu pamięci nieulotnej jest brak wielobieżności kodu implementującego dostęp do różnych warstw stosu pamięci nieulotnej. Jest to powiązane z problemem **P4**, którego proponowaną metodą rozwiązania jest odpowiednie zarządzanie dostępu do pamięci i ciągłe monitorowanie kodu źródłowego, przy wykorzystaniu specjalnego oprogramowania przygotowanego przez autora. Zgodnie z dokumentacją implementacji stosu AUTOSAR oraz samym standardem – [77], możliwe są następujące zlecenia operacji na pamięci NvM:

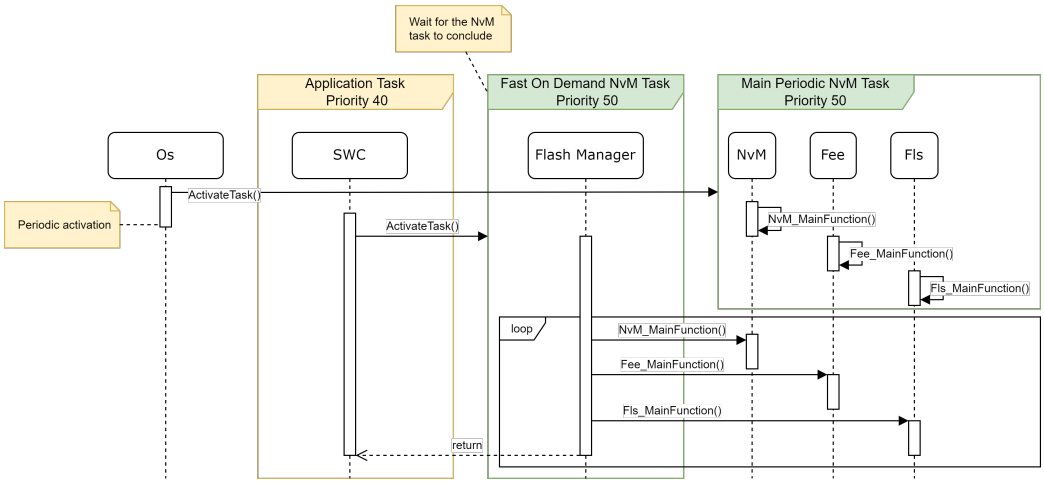
1. Odczyt podczas inicjalizacji systemu – NvM_ReadAll.
2. Odczyt podczas działania systemu – NvM_ReadBlock.
3. Zapis podczas deinicjalizacji systemu – NvM_WriteAll.
4. Zapis podczas działania systemu – NvM_WriteBlock.

Wszystkie wymienione zlecenia, są realizowane przez stos przy użyciu przetwarzania w funkcjach NvM_MainFunction, Fee_MainFunction, Fls_MainFunction (w przypadku pamięci Electrically Erasable Programmable Read-Only Memory (EEPROM), są to odpowiednio Ea_MainFunction i Eep_MainFunction). Wszystkie te funkcje muszą zostać zaplanowane w odpowiednich zadaniach systemu operacyjnego. Musi im zostać przydzielony priorytet oraz okres wywoływania (zgodnie z [77]).

W przypadku, gdy stos NvM jest wykonywany tylko z jednego miejsca, problemy braku wielobieżności nie występują, ale z powodu wymagań projektowych, powiązanych z czasem wykonania operacji na pamięci (wymaganych natychmiastowych zapisów do pamięci danych podczas sytuacji kryzysowej – wypadku samochodu, wystąpienia krytycznego błędu, zaniku zasilania itp.) W większości przypadków stos pamięci nieulotnej jest wywoływany z wielu miejsc, bardzo często w sposób blokujący przedstawiony wcześniej, w kodzie źródłowym 5.11.

Przedstawiony kod w pętli „do... while()", realizuje wielokrotne wykonanie funkcji stosu NvM w blokujący sposób. Kod skończy się dopiero, gdy operacja na pamięci zostanie ukończona. Wywoływanie stosu w ten sposób powoduje potencjalnie bardzo poważne problemy zależności czasowych, naruszenia spójności danych oraz brak stabilności systemu. W przypadku, gdy jakakolwiek warstwa stosu zostanie wywłaszczona przez inną próbę dostępu do pamięci, oprogramowanie może wywołać krytyczny błąd systemu i zachowywać się w sposób nieprzewidywalny. Jest to szczególnie niebezpieczne w kontekście opisanych w [8] zmian sekcji wywołanych przez procedurę balansowania pamięci, które trwają znacznie dłużej niż standardowe zapisy do pamięci.

Proponowanym rozwiązaniem opisanego problemu jest zarządzanie przetwarzaniem pamięci w scentralizowany sposób – przy użyciu specjalnej biblioteki realizującej dostęp do stosu pamięci. Diagram blokowy proponowanego rozwiązania jest przedstawiony na rysunku 5.12. Można



Rysunek 5.12. Centralne synchroniczne zarządzanie dostępami do pamięci NvM

założyć dla przykładu, że komponent oprogramowania SWC potrzebuje zapisać dane na temat przebiegu samochodu do pamięci, a w tym samym czasie stos pamięci nieulotnej jest zajęty przetwarzaniem innej zleconej operacji realizującej zapis danych powiązanych z pozycją GPS. W takim przypadku komponent SWC nie wywołuje bezpośrednio funkcji, lecz aktywuje zadanie systemu operacyjnego, które ma taki sam priorytet jak

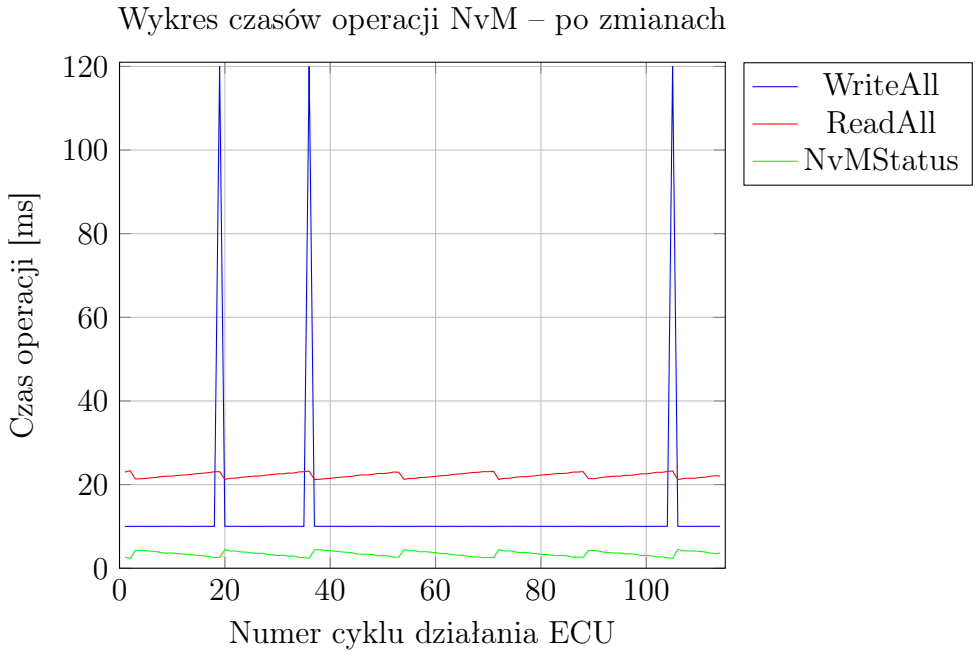
zadanie uruchamiające okresowo funkcje stosu NvM. Dzięki temu, aktywowane zadanie czeka, aż okresowo wywoływane funkcje skończą realizację zapisów i odczytów, po czym zgłasza kolejne żądanie. Nawet jeżeli komponent SWC ma wyższy priorytet, to dzięki izolacji żądania przez aktywację zadania, a nie przez wywołanie funkcji, nie wystąpi problem wielokrotnego dostępu. Rezultatem takiego sposobu zarządzania dostępem do pamięci jest brak wzajemnego wywłaszczania funkcji powiązanych z dostępem do stosu NvM. Innymi słowy zapis licznika samochodu poczeka, aż dane powiązane z GPS zostaną przetworzone. Nasuwa się pytanie, dlaczego nie użyć kolejki dla żądań operacji na pamięci NvM? Dokumentacja AUTOSAR NvM wymaga obsługi kolejek, aczkolwiek należy pamiętać, że opisywana kwestia dotyczy wielu nagłych żądań, które wymagają natychmiastowego przetworzenia. W przypadku użycia standardowego algorytmu kolejkowania funkcje stosu są wykonywane okresowo, np. co 10 ms. Zakładając, że do zapisu danego bloku potrzeba 1000 iteracji stosu, zaproponowane synchroniczne wykonanie operacji przy użyciu pętli będzie znacznie szybsze, niż asynchroniczne wykonywanie okresowe.

Kolejnym elementem proponowanej konfiguracji jest konieczność wyrównania priorytetów wszystkich zadań realizujących funkcje stosu NvM lub wykorzystanie zasobów systemu operacyjnego (ang. OS resources) opisanych w [68]. W celu utrzymania takiego zarządzania pamięcią, przygotowano specjalny autorski skrypt, który przeszukuje cały wykorzystywany kod źródłowy i weryfikuje, czy nigdzie bezpośrednio nie są wywoływane funkcje stosu NvM.

5.3.3 Walidacja i ocena rozwiązania

Walidacja proponowanych sposobów rozwiązania zidentyfikowanych problemów badawczych **P2**, **P3** oraz **P4** sprowadza się do kilku aspektów.

Pierwszym z nich jest porównanie wyników badań po wprowadzeniu centralnej metody zarządzania dostępem do pamięci, z wynikami uzyskanymi przed jego wprowadzeniem. Dzięki modułowi obserwatora systemu, wykonano badania kontrolne po wprowadzeniu zmian. Widoczne są one na wykresie 5.13. Wykres przedstawia czas operacji funkcji odczytującej pamięć NvM_ReadAll, zapisującej pamięć NvM_WriteAll oraz raportującej status komórek pamięci NvM_Status. Widoczny powtarzalny trend z okresem około 20 cykli jest powiązany z momentami



Rysunek 5.13. Czasy operacji zapisu, odczytu oraz raportowania statusu bloków pamięci NvM po zmianach usuwających problem wielowątkowego dostępu do stosu, zebrane podczas ponad 100 cykli działania sterownika ECU

zmian sektorów pamięci.

Na początku pustego sektora pamięci odczyt trwa najszybciej, gdyż dane są umieszczone na początku sektora. Im więcej danych znajduje się w sektorze, tym na dalszych adresach są umieszczane, czego rezultatem jest zwiększenie czasu odczytu. Sterownik musi iteracyjnie odnaleźć dane na końcu sektora. Czas zapisów jest w większości przypadków stały, poza momentami zmiany sektora. W tych momentach czas zapisu jest znacząco zwiększony, ponieważ sterownik musi skopiować wszystkie dane ze starego do nowego sektora, zanim zapisze konkretną aktualizowaną wartość bieżącej operacji. Jest to widoczne na wykresie 3 razy.

Najważniejszym rezultatem jest usunięcie występowania braku pomiarów, widoczne w poprzednich wynikach (rysunek 5.11) jako zerowe czasy trwania zapisu, a więc momentów, gdzie niestabilność systemu wywołana przez wielobieżny dostęp do stosu NvM powodowała krytyczne błędy przebiegu faz systemu. Jak widać na wykresie, nie występują już

pomiary z zerowym czasem. Dodatkowym atutem proponowanego i wdrożonego rozwiązania jest zmniejszenie czasu realizacji operacji odczytu oraz zapisu. W początkowych pomiarach operacje odczytu oscylowały około 25 ms, zapisu 20 ms (rysunek 5.11). Dla pomiarów po wprowadzeniu centralnej metody zarządzania dostępami, operacje odczytu zajmują średnio 22 ms, zapisu 12 ms.

Ostatnim aspektem walidacji proponowanych sposobów rozwiązania problemów badawczych są widoczne na wykresie procedury zmiany sektora, czyli momenty, w których pomiary przedstawiają znacznie zwiększony czas zapisu – około 120 ms. Algorytm do obliczania żywotności pamięci pozwala w automatyczny sposób określić, jak często występuje zmiana sektora i weryfikować, czy wysoka częstotliwość tej procedury nie zagraża żywotności komórek pamięci. Zgodnie z użyciem skryptu i wynikami przedstawionymi w tabeli 5.10 pamięć projektu komercyjnego **Projekt 3** wystarczy na ponad 50 lat ciągłego używania.

5.3.4 Odniesienie do norm jakości, bezpieczeństwa i zabezpieczeń

ISO 25010

Przedstawione metody przyczyniające się do rozwiązania problemów **P2**, **P3** i **P4**, zrealizowane poprzez wprowadzenie odpowiedniej architektury oraz automatyzacji konfiguracji stosu NvM, zwiększają jakość oprogramowania, wpływając na elementy normy ISO 25010. Tabela 5.11 opisuje, w jaki sposób rozwiązanie przyczyniło się do poprawy jakości dla wybranych kategorii normy. Wszystkie kategorie jakości opisywane przez normę były wcześniej przedstawione na rysunku 2.5.

ISO 26262

Proponowany algorytm do obliczenia żywotności pamięci realizuje zalecenie **ISO 26262 7.4.17**, czyli wyznacza estymację limitu zużycia zasobu pamięci NvM. Dodatkowo, zgodnie z **ISO 26262 8.4.4** zwiększa niezawodność systemu, łatwość analizy i ułatwia modyfikację oprogramowania. Metoda zarządzania dostępami do pamięci realizuje zalecenie **ISO 26262 D.2.2**, czyli zabezpiecza przed wzajemną blokadą wywołania funkcji stosu NvM i błędami synchronizacji pomiędzy modułami

Relacja architektury stosu pamięci nieulotnej do normy ISO 25010		
Kategoria	Parametr	Opis w kontekście stosu NvM
Wydajność	Charakterystyka czasowa Oczekiwana wydajność	Zgodnie z przedstawionymi wynikami badań, czas realizacji odczytów i zapisów został zmniejszony. Dzięki zmniejszonym czasom operacji, start systemu mógł spełnić wymagania klienta.
Kompatybilność	Współistnienie	Zaproponowany centralny sposób zarządzania dostępem do stosu zapewnił poprawne działanie nawet przy wielu jednoczesnych żądaniach.
Użyteczność	Ochrona użytkownika przed błędami	Zaproponowana automatyzacja oparta na skryptach oraz użycie obserwatora systemu eliminuje potencjalne błędy oraz pozwala na proces ciągłej weryfikacji.
Łatwość utrzymania	Łatwość analizy Łatwość modyfikowania Łatwość testowania	Dzięki większej ilości logowanych danych, analiza jest znacznie uproszczona. Wszelkie modyfikacje są przetwarzane przez skrypt i od razu weryfikowane w sposób automatyczny. Skrypty i obserwator systemu zapewniają pierwszy poziom testów.

Tablica 5.11. Odniesienie architektury NvM do normy ISO 25010

korzystającymi z pamięci nieulotnej.

ISO 21434

Architektura pamięci nieulotnej jest krytyczna, zgodnie z rezultatami analizy Threat Assessment and Remediation Analysis (TARA), ponieważ artefakty cyberbezpieczeństwa, takie jak certyfikaty, klucze czy sumy kontrolne są przechowywane w pamięci nieulotnej. Z tego powodu zaproponowana architektura i automatyzacja bezpośrednio wpływają na obniżenie ryzyka, wynikającego z wystąpienia błędów danych powiązanych z cyberbezpieczeństwem. Skrypt obliczający żywotność pamięci bezpośrednio realizuje cele **ISO 21434 [10.2]**, czyli identyfikuje wrażliwości systemu. Przedstawia dowody, że rezultat implementacji i integracji komponentów jest zgodny ze specyfikacją cyberbezpieczeństwa oraz z **ISO 21434 [RQ-10-10]**. Rozwiązanie stanowi metodę ewaluacji stopnia zużycia zasobów systemowych.

5.3.5 Dostarczenie pakietu i wzorców projektowych dla inżynierów oprogramowania

Opisane zagadnienia zostały wdrożone w projekcie komercyjnym **Projekt 3**. Skrypty do obliczania żywotności i eksportowania parametrów stosu zostały dołączone do pracy. W niniejszym rozdziale dostarczone są przez autora wzorce i koncepcje architektury, które umożliwiają użycie rozwiązania w dowolnym projekcie korzystającym ze standardu AUTOSAR i używającym pamięci FEE.

Skorzystano w tym celu ze stosu jednej z dwóch najpopularniejszych firm na rynku, gdzie konfiguracja może być przetwarzana przy użyciu specjalnego języka skryptowego stosowanego w tzw. „code templates” (opisanego w [86]). W zależności od stosowanego stosu, inżynier może w różny sposób eksportować dane ustawień pamięci NvM, aby użyć ich w skrypcie do obliczania żywotności. Metoda zarządzania dostępami do pamięci wymaga zrealizowania kilku kroków. Szczegóły, jak zastosować zaproponowane metody i architekturę opisano w niniejszym rozdziale.

Ogólna architektura rozwiązania

Automatyzacja – W celu wdrożenia opisanej metody obliczania żywotności oraz użycia zaprezentowanego skryptu, należy przygotować dane wejściowe dotyczące konfiguracji stosu NvM. W zależności od implementacji stosu, może to być zrealizowane w różny sposób.

W przypadku stosu firmy Elektrobit, można wykorzystać napisaną przez autora procedurę przedstawioną w kodzie źródłowym 5.12

```
1  Name,BlockId,Priority,Length,NumberOfCopies,BlockType,CRC,ReadAllEnabled,
       WriteAllEnabled,NvMExtraBlockChecks,NvMInitBlockCallback,
       NvMResistantToChangedSw
2  [!LOOP "/AUTOSAR/TOP-LEVEL-PACKAGES/NvM/ELEMENTS/NvM/NvMBlockDescriptor/*"!]
3  [!"name(.)"!],[!"./NvMNvramBlockIdentifier"!],[!"./NvMBlockJobPriority"
       !],[!"./NvMNvBlockLength"!],[!"./NvMNvBlockNum"!],[!"./
       NvMBlockManagementType"!],
4  [!IF "node:exists(./NvMBlockCrcType)"!]
5  [!"./NvMBlockCrcType"!]
6  [!ELSE!]NOCRC[!ENDIF!],
7
8  [!"./NvMSelectBlockForReadAll"!],
9  [!IF "node:exists(./NvMSelectBlockForWriteAll)"!][!"./
       NvMSelectBlockForWriteAll"!]
10 [!ELSE!]ERROR[!ENDIF!],
11
12 [!"./NvMExtraBlockChecks"!],
13
14 [!IF "node:exists(./NvMInitBlockCallback)"!][!"./NvMInitBlockCallback"!]
15 [!ELSE!]NOCALLBACK[!ENDIF!],[!"./NvMResistantToChangedSw"!]
16 [!ENDLOOP!]
```

Kod źródłowy 5.12. Funkcja eksportująca wszystkie bloki NvM z konfiguracji w narzędziu EB Tresos wraz z ich parametrami

Przygotowana funkcja iteruje po wszystkich blokach NvM i wyświetla w konsoli ich nazwy oraz wybrane parametry, takie jak identyfikator, priorytet, długość, liczba kopii, typ itp. W przypadku, gdy niektóre ustawienia są nieobecne, tak jak na przykład obecność sumy kontrolnej Cyclic redundancy check (CRC), funkcja wypełni je predefiniowanym tekstem. W przypadku chęci użycia skryptu do eksportowania innych parametrów, wystarczy zmienić ścieżki oraz identyfikatory. Po ukończeniu działania, dane oddzielone od siebie przecinkami są gotowe do importu w postaci Comma-separated values (CSV).

Skrypt do obliczania żywotności potrzebuje następujących danych: nazwa, identyfikator bloku, priorytet, długość, liczba kopii, typ bloku, ustawienie sumy kontrolnej, ustawienie „ReadAll”, ustawienie „WriteAll”. Gdy wszystkie opisane dane są gotowe w pliku CSV, skrypt potrzebuje

jeszcze ścieżki do folderu z kodami źródłowymi, w celu znalezienia wszystkich wywołań stosu, aby zmodyfikować je celem zastosowania metody zarządzania dostępami do pamięci.

Przykładowy rezultat działania skryptu jest przedstawiony w kodzie źródłowym 5.13.

```
Total size in bytes: 30820
Total size of writeAll blocks in bytes: 8072
Total size of non-writeAll blocks in bytes: 22748
Section Switch Time for all blocks is 0.0029467858057419787 years.
Memory durability is 58.93571611483957 years.

NvM WriteAll occurences:
Line: 871  Plik1.c
Line: 553  Plik1.c
Line: 1201 Plik1.c
Line: 1686 Plik1.c
Line: 1423 Plik2.c
Line: 187  Plik2.c
Line: 1242 Plik3.c
Line: 1727 Plik4.c

NvM Write occurences:
Line: 181 Plik1.c
Line: 404 Plik1.c
Line: 566 Plik2.c
Line: 581 Plik3.c
Line: 187 Plik4.c
Line: 365 Plik5.c
Line: 525 Plik5.c
```

Kod źródłowy 5.13. Przykładowy rezultat działania skryptu do obliczania żywotności

Faktyczne nazwy plików z wystąpieniami dostępów do stosu NvM zostały zastąpione nazwami „plik.c” z powodu tajności projektu komercyjnego. Kod przedstawia ilość danych stosu NvM, podzieloną na bloki zapisywane podczas startu systemu („WriteAll”), np. przebieg samochodu i bloki zapisywane w specyficznych momentach, np. podczas wypadku. Skrypt oblicza żywotność pamięci w latach i wyświetla użycia funkcji zapisu, aby użytkownik mógł przeanalizować, czy nie występuje problem jednoczesnego wielokrotnego dostępu do pamięci.

Ustawienia stosu – Zgodnie z zaprezentowaną metodą zarządzania dostępami do pamięci, po zidentyfikowaniu wszystkich funkcji wykonujących dostęp do stosu NvM należy upewnić się, że wszystkie funkcje realizujące dostęp do NvM powinny mieć równy priorytet lub wprowadzoną sekcję krytyczną, aby uniemożliwić wielokrotne jednoczesne

wywołanie z różnych kontekstów. Należy pamiętać o starcie i deinicjalizacji systemu oraz regułach modułu Basic Software Manager (BswM).

Po wprowadzeniu jednej synchronicznej funkcji, dobrą praktyką jest dodanie informacji na temat jej wykonania i czasu trwania. Zrealizowano przez autora logowanie stosu przy użyciu zaproponowanego obserwatora systemu.

Przekazywanie informacji do obserwatora systemu

Moduł obserwatora systemu został dokładnie opisany w rozdziale 5.4, ale zgodnie z tabelą 5.9, przy jego użyciu zminimalizowano problem **P3** w kontekście stosu NvM. Autorski moduł obserwatora pozwolił zebrać dane badawcze z wielu uruchomień systemu, uzyskać statystyki czasowe operacji stosu NvM i w sposób ciągły monitorować wystąpienia zmian sekcji oraz problemów przetwarzania danych nieulotnych. Wyniki obserwacji są przedstawione na wykresach 5.11 oraz 5.13.

Proponowanym rozwiązaniem problemu badawczego **P3** jest umieszczenie w obserwatorze systemu specjalnych funkcji odpytujących system o czas zapisów i odczytów danych ze stosu pamięci nieulotnej. W przeciwieństwie do badań wykonanych w ramach artykułu [8], zaproponowana architektura nie wymaga modyfikacji certyfikowanego stosu, a wykorzystuje koncept funkcji wywołanych zwrotnie podczas inicjalizacji oraz deinicjalizacji systemu. Szczegóły są opisane w rozdziale 5.4 oraz przedstawione na diagramie 5.17.

5.4 Obserwator systemu

5.4.1 Nawiązanie do zagadnienia badawczego

Rozwiązanie 4 – obserwator systemu jest zaproponowanym rozwiązaniem zidentyfikowanych problemów **P3** oraz **P4**, opisanych w rozdziałach 3.3.1 oraz 3.4.1. W celu zrealizowania badań dotyczących zagadnień badawczych **ZB3** oraz **ZB4**, zostały wykorzystane metody badawcze **MB1**, **MB2**, **MB4**, **MB6**, **MB5**.

Analizy wykonane w ramach artykułu badawczego

Autor dokonał analizy przebiegu oraz czasu wykonania fazy inicjalizacji systemów wbudowanych w ramach artykułu badawczego [87] przy użyciu metod badawczych **MB1**, **MB2**, **MB4**, **MB5** i **MB6**. Celem artykułu było rozwiązanie zidentyfikowanego problemu zbyt długiej inicjalizacji systemu oraz spełnienie wymagań przemysłu motoryzacyjnego odnośnie do czasu realizacji pierwszej odpowiedzi komunikacyjnej po dostarczeniu zasilania. Przeanalizowano procedurę inicjalizacji, sprawdzono czas wykonania poszczególnych faz startu systemu, zidentyfikowano potencjalne miejsce optymalizacji w postaci funkcji inicjalizacji pamięci Block Started by Symbol (BSS) oraz zaproponowano algorytm minimalizujący czas wykonania.

Badania oraz walidacja zaproponowanego algorytmu zostały przeprowadzone w projektach komercyjnych **Projekt 2** oraz **Projekt 3**. Rezultatem artykułu było zmniejszenie czasu inicjalizacji pamięci w obydwu projektach o około 90%. Zaletą algorytmu jest jego uniwersalność. Nie jest ograniczony tylko do przemysłu motoryzacyjnego i może zostać wykorzystany w różnych systemach wbudowanych. Zaproponowane rozwiązanie zostało wdrożone. Artykuł bezpośrednio wpływa na zidentyfikowany w tej pracy problem badawczy **P4**. Ręczne analizy przeprowadzone w ramach artykułu badawczego były inspiracją do modułu obserwatora systemu, który w sposób ciągły prezentuje zależności i pomiary czasowe, tak aby inżynierowie widzieli, jakie charakterystyki czasowe ma system po wprowadzanych zmianach.

Analiza statyczna

Analiza statyczna polegała na wyszukiwaniu miejsc w kodzie źródłowym, gdzie można osadzić punkty kontrolne przebiegu działania systemu. Dla modułów aplikacyjnych SWC oraz CDD nie stanowiło to problemu, aczkolwiek w przypadku certyfikowanych części stosu AUTOSAR, należało odwoływać się do konfiguracji funkcji wywoływanych zwrotnie. Przykładem były funkcje wywoływane po ukończeniu operacji zapisu danych do pamięci nieulotnej lub po wysłaniu konkretnej ramki komunikacyjnej. Nie wszystkie moduły mają możliwość włączenia takich funkcji.

Podczas analizy statycznej zidentyfikowano miejsca, gdzie bez naruszania certyfikacji oprogramowania, można umieścić procedurę zapisu informacji na temat punktu kontrolnego.

Analiza przebiegu krytycznych faz systemu

Analiza przebiegu krytycznych faz systemu jest bardzo często wykorzystywana podczas szukania przyczyn defektów w samochodzie. Z powodu dużej liczby ECU i wielu zależności pomiędzy nimi, fazy inicjalizacji oraz deinicjalizacji są szczególnie narażone na występowanie problemów (zgodnie z opisanymi problemami **P3** i **P4**).

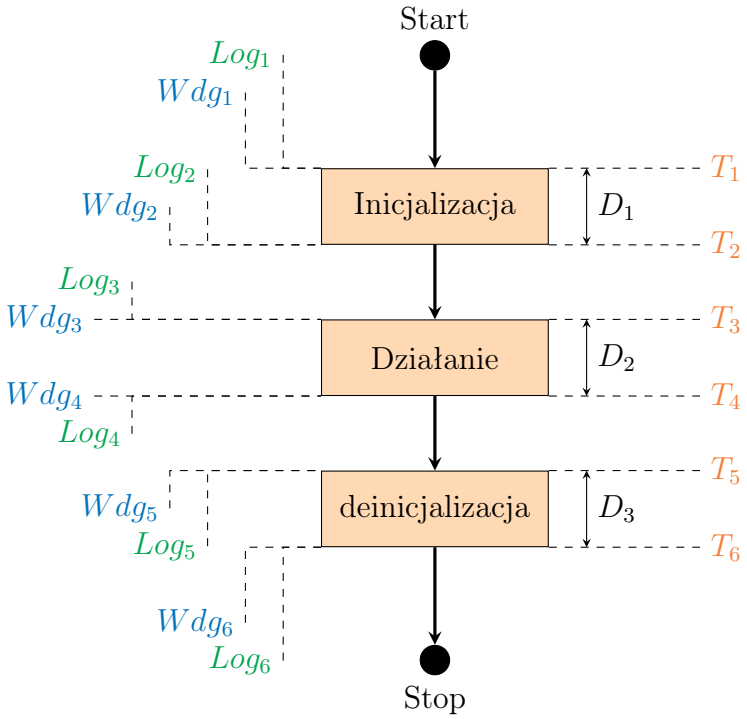
Analiza wykonana przez autora skupiała się na badaniu przebiegu krytycznych faz systemu, aby jak najlepiej zdefiniować punkty kontrolne obserwatora systemu. Bazując na dokumentacji [88] i [89] przygotowano listę kroków pomiarowych, które są bazą dla obserwatora systemu.

5.4.2 Sposób rozwiązania problemu badawczego

Zidentyfikowane zagadnienia badawcze **ZB3** oraz **ZB4** są powiązane z wysokim stopniem złożoności oprogramowania oraz negatywnymi skutkami tej złożoności. Proponowanym przez autora rozwiązaniem problemów badawczych jest użycie modułu obserwatora systemu, który pozwala na ciągłe monitorowanie zachowania oprogramowania wraz z ich zależnościami czasowymi. Punkty kontrolne są przedstawione na rysunku 5.14. Trzy podstawowe fazy systemu to inicjalizacja, działanie oraz deinicjalizacja. Na rysunku są widoczne:

- punkty pomiarowe czasu działania – T,
- czasy trwania poszczególnych faz – D,
- punkty wysłania logów – Log,
- punkty kontrolne modułu watchdog – Wdg.

Taką ogólną formę architektury obserwatora systemu można zastosować w większości projektów systemów wbudowanych. Liczba punktów pomiarowych może być znacznie większa. Każdą z faz można podzielić na wiele mniejszych, w zależności od wymagań projektowych. We wdrożeniu modułu obserwatora systemu w projekcie **P3**, zdefiniowano 15 faz,



Rysunek 5.14. Podstawowe fazy punktów kontrolnych obserwatora systemu

skupiając się na fazach inicjalizacji i deinicjalizacji – zostały one podzielone na kroki. Dzięki temu można było analizować dokładnie czasy wykonywania poszczególnych etapów oraz identyfikować newralgiczne miejsca. Kolejnym krokiem jest osadzenie modułu w kontekście standardu AUTOSAR.

Logowanie

Logowanie zdarzeń to główna funkcja obserwatora systemu. Dzięki odpowiedniej architekturze zaproponowanej w rozdziale 5.2, logowanie binarne „non-verbose" dostarcza możliwość wysyłania dużej ilości informacji, jednocześnie nie obciążając zasobów systemu. W celu zminimalizowania negatywnych skutków problemów badawczych **P3** i **P4**, jest potrzebna jak największa ilość informacji na temat działania systemu oraz zależności pomiędzy modułami. Podstawowe miejsca logowania osadzone w kontekście standardu AUTOSAR są przedstawione w tabeli 5.12.

Moduł	**Opis**
BswM, Electronic Control Unit Manager (EcuM)	Logowanie faz przebiegu inicjalizacji i deinicjalizacji systemu.
Dcm, Dem	Logowanie zdarzeń warstwy diagnostyki, w szczególności wystąpień błędów (DTC) i procedur powiązanych z aktualizacją oprogramowania (Software Loading (SWL)).
NvM	Logowanie zdarzeń stosu pamięci nieulotnej, statusów bloków itp.
Communication Manager – (ComM), SPI	Logowanie zdarzeń powiązanych z komunikacją, zmian trybu komunikacji „FULL-COM", „NO-COM" itp.
Warstwa aplikacji	Logowanie zdarzeń powiązanych z logiką biznesową, jak na przykład stanu sygnału stacyjki (ang. Ignition).

Tablica 5.12. Podstawowe miejsca logowania w oprogramowaniu wykorzystującym standard AUTOSAR

Wymienione punkty są dostępne w każdym projekcie zgodnym z AUTOSAR. Poza nimi, można dodawać specyficzne punkty charakterystyczne dla danego projektu. Moduł obserwatora systemu ma agregować najważniejsze informacje wraz ze znacznikami czasowymi. Zgodnie z 5.2 należy przygotować plik FIBEX, zawierający opis wszystkich transmitowanych wiadomości w trybie „non-verbose". Kolejnym newralgicznym aspektem jest logowanie informacji na temat startu systemu, kiedy to jeszcze moduł DLT może być niedostępny. Proponowanym rozwiązaniem jest utworzenie bufora zapisującego konkretną liczbę zdarzeń, które są wyświetlane okresowo. Przykładowe logi wdrożone w projekcie komercyjnym **Projekt 3** są przedstawione w kodzie źródłowym 5.14.

```
1 SMON|IGNI Ignition changed from SNA -> LOCK, timestamp: 2401451[us], change
      number: 1
```

Kod źródłowy 5.14. Przykładowy log przedstawiający informację na temat zmiany sygnału stacyjki ze stanu nieokreślonego Signal Not Available (SNA) na stan LOCK czyli samochód zamknięty, kluczyk nieobecny

Na początku loga widoczne są ApplicationId – „SMON” oraz ContextId – „IGNI”, które pozwalają na łatwe filtrowanie wiadomości i monitorowanie zmian stacyjki przez cały okres działania systemu, a także w relacji z innymi zdarzeniami. Dodatkowo wyświetlony jest znacznik czasowy, który pozwala określić, kiedy dokładnie dany sygnał się zmienił i „change number”, czyli licznik, który przy każdej zmianie sygnału zwiększa swoją wartość o 1.

Pomiary czasowe

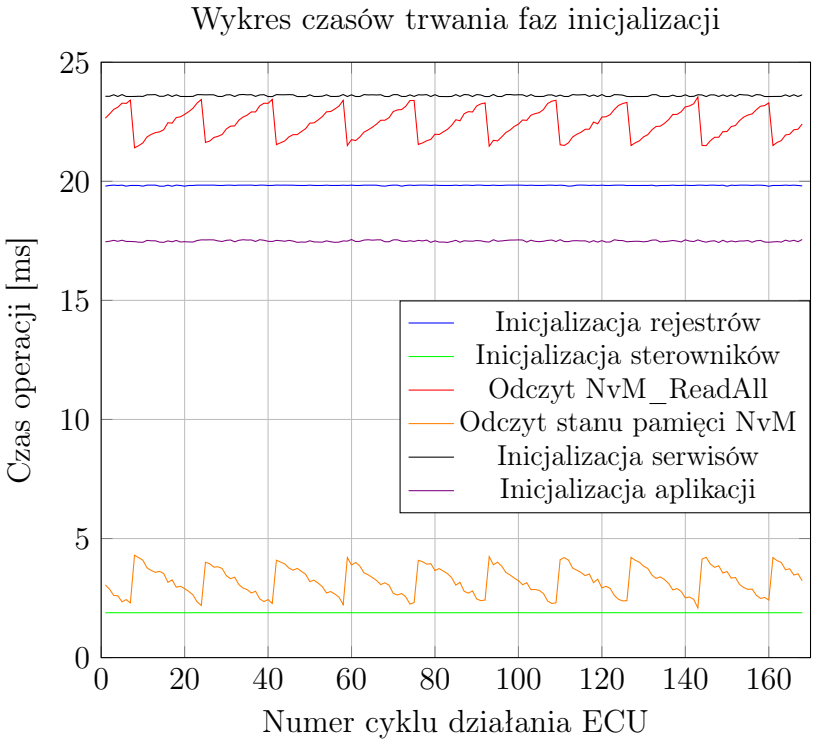
Proces ciągłego monitorowania i mierzenia zależności czasowych jest kolejnym aspektem minimalizującym negatywne skutki problemów badawczych **P3** i **P4**. Dzięki prostemu dostępowi do wspólnego źródła pomiarów czasowych, moduł obserwatora systemu może wyświetlać długości trwania poszczególnych faz systemu, zależności czasowe pomiędzy różnymi funkcjami oraz zapisywać do pamięci nieulotnej krytyczne zdarzenia powiązane z czasem wykonania. Źródło czasowe musi zostać skonfigurowane przez integratora. W przypadku wdrożenia w projekcie komercyjnym **Projekt 3**, został wykorzystany asynchroniczny 56-bitowy licznik sprzętowy liczący w mikrosekundach, który pozwala na wspieranie nawet bardzo długich cykli działania samochodu bez ryzyka przepełnienia.

Automatyzacja

Moduł obserwatora generuje duże ilości istotnych danych. Inżynierowe mogą je analizować podczas prac projektowych, skupiając się na pojedynczych przebiegach i sytuacjach. Drugim ważnym scenariuszem jest zbieranie danych z wielu cykli działania podczas testowania, w szczególności podczas długotrwałych testów wytrzymałościowych (ang. endurance tests). W celu wizualizacji danych zebranych podczas wielogodzinnych testów, przygotowano autorski skrypt parsujący logi, wycinający wiadomości z obserwatora systemu i przygotowujący statystyki czasowe dla wybranych elementów. Po przetworzeniu zapisanych logów, skrypt przygotowuje dane gotowe do wizualizacji, na przykład w programie Microsoft Excel.

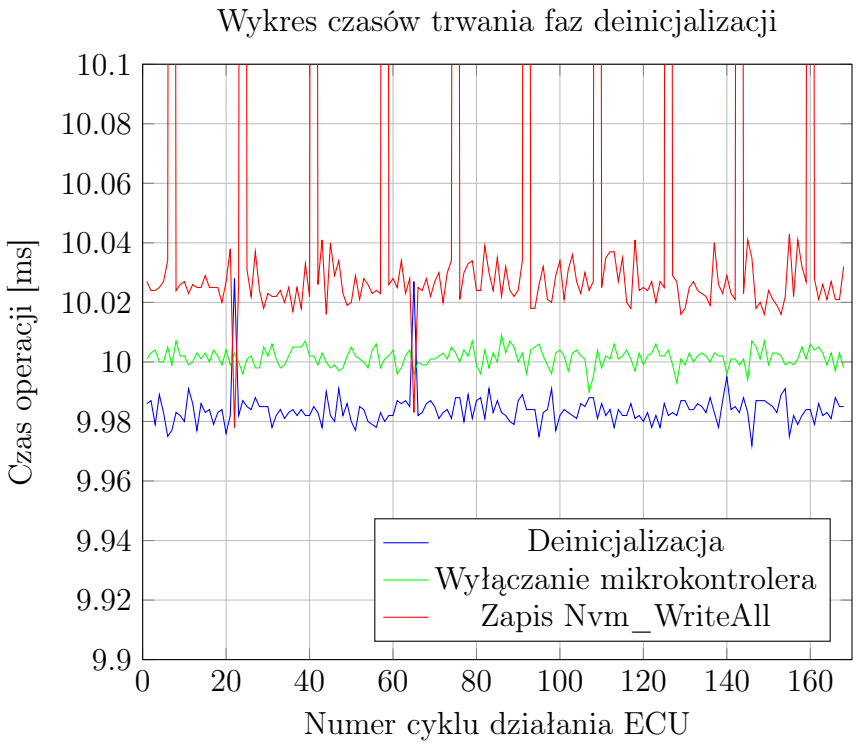
5.4.3 Walidacja i ocena rozwiązania

Rezultatem wdrożenia obserwatora systemu możliwa była analiza skomplikowanych błędów oraz zależności.



Rysunek 5.15. Czasy trwania poszczególnych faz inicjalizacji systemu zebrane podczas ponad 160 cykli działania sterownika

Dzięki jego wprowadzeniu w projekcie komercyjnym **Projekt 3** możliwe było zaobserwowanie problemu wielokrotnych jednoczesnych dostępów do stosu NvM, przedłużone działanie funkcji WriteAll oraz ReadAll. Dodatkowo, moduł jest łatwo rozszerzalny, można tam umieścić dane, które będą potrzebne w konkretnym projekcie, np. dane powiązane ze zdalną diagnostyką lub komunikacją międzyrdzeniową. Dzięki automatycznemu skryptowi do parsowania danych, można realizować automatyczne testy zależności, powiązań oraz czasów trwania. Wyniki przykładowego testowania czasu trwania poszczególnych faz inicjalizacji systemu są przedstawione na wykresie 5.15. Wielokrotne testowanie procesu inicjalizacji pozwala łatwiej wychwytywać rzadko występujące problemy, które w przypadku braku informacji mogłyby być ominięte i zidentyfikowane dopiero przez użytkowników samochodów. Zgodnie z wnioskami artykułu badawczego [54] automatyzacja testowania



Rysunek 5.16. Czasy trwania poszczególnych faz deinicjalizacji systemu zebrane podczas ponad 160 cykli działania sterownika

zintegrowana z procesem „continous integration", jest wysoce pożądana. Wyniki procesu deinicjalizacji (wyłączania systemu) są przedstawione na wykresie 5.16. Zmienność czasowa fazy deinicjalizacji jest znacznie mniejsza niż w przypadku inicjalizacji, dodatkowo widoczne są momenty zmiany sektora pamięci FEE (opisane przez autora w rozdziale 5.3 oraz dokładnie przebadane w artykule [8]) – w tych przypadkach czas wykonania procedury WriteAll zajmuje znacząco więcej czasu – około 120 ms.

Obydwie fazy – inicjalizacji i deinicjalizacji są oparte na zalecanych procedurach kolejności, odnoszących się do ustawionych reguł modułów EcuM oraz BswM. W większości projektów powinny one zatem bazować na dokumentacji AUTOSAR [88], [90] oraz [91]. Zgodnie z wymienioną dokumentacją, integrator systemu powinien opracować odpowiedni zestaw reguł, które często są powiązane ze specyficznymi wymaganiami producenta samochodów i procedurami zabezpieczeń (szyfrowaniem,

zapisywaniem/odczytywaniem kluczy, sprawdzaniem certyfikatów itp.). Dzięki obserwatorowi systemu, konfiguracja reguł może być na bieżąco weryfikowana – czy żadna reguła nie wywołuje się wiele razy, czy kolejność procedur jest odpowiednia oraz czy funkcje stosu zarządzania trybami ECU są realizowane z odpowiednią częstotliwością. Pozostałe monitorowane elementy, które zostały wdrożone to:

1. Zmiany sesji diagnostycznych.
2. Zmiany stanów wirtualnych sieci lokalnych Virtual Local Area Network (VLAN).
3. Informacje na temat procedury aktualizacji oprogramowania SWL.
4. Cykliczne raportowanie czasu cyklu życia systemu.
5. Statystyki komunikacyjne – ile ramek zostało wysłanych/odebranych w różnych magistralach komunikacji.
6. Zmiany sygnału stacyjki.
7. Raportowanie stanu bloków pamięci nieulotnej.
8. Raportowanie wystąpień kodów błędów działania sterownika DTC.

5.4.4 Odniesienie do norm jakości, bezpieczeństwa i zabezpieczeń

ISO 25010

Przedstawione metody przyczyniające się do rozwiązania problemów **P3** i **P4**, zrealizowane poprzez wprowadzenie modułu obserwatora systemu zwiększają jakość oprogramowania, wpływając na elementy normy ISO 25010. Tabela 5.13 opisuje, w jaki sposób rozwiązanie przyczyniło się do poprawy jakości dla wybranych kategorii normy. Wszystkie kategorie jakości opisywane przez normę były wcześniej przedstawione na rysunku 2.5.

Relacja modułu obserwatora systemu do normy ISO25010		
Kategoria	Parametr	Opis w kontekście modułu obserwatora systemu.
Użyteczność	Rozpoznawalność zastosowania	Ciągłe monitorowanie umożliwia weryfikację wymagań i zapewnia możliwość wyśledzenia kodu źródłowego do wymagania producenta samochodu.
	Ochrona użytkownika przed błędami	Zaproponowany zakres automatyzacji w postaci skryptów oraz użycie obserwatora systemu zmniejsza ryzyko wystąpienia potencjalnych błędów oraz pozwala na proces ciągłej weryfikacji oprogramowania.
Łatwość utrzymania	Łatwość ponownego użycia	Moduł obserwatora systemu korzysta z narzędzi stosu AUTOSAR, więc może być łatwo użyty w każdym projekcie wykorzystującym ten standard.
	Łatwość analizy	Dzięki większej ilości logowanych danych, analiza jest znacznie uproszczona.
	Łatwość modyfikowania	Wszelkie modyfikacje są przetwarzane przez skrypt i od razu weryfikowane w sposób automatyczny.
	Łatwość testowania	Skrypty i obserwator systemu zapewniają pierwszy poziom testów. Testerzy zbierają informacje z obserwatora systemu i umieszczają je w raportach o błędach.

Tablica 5.13. Odniesienie modułu obserwatora systemu do normy ISO25010

ISO 26262

Obserwator systemu może realizować weryfikację specyficznych celów bezpieczeństwa funkcjonalnego i ryzyka zidentyfikowanego podczas analiz Hazard Analysis and Risk Assessment (HARA) oraz TARA. Zgodnie z zaleceniem **ISO 26262 7.4.14** proponowane rozwiązanie zapewnia sposób zewnętrznego monitorowania działania krytycznych modułów, weryfikuje poprawność kolejności wykonywania krytycznych funkcji systemu, a także wykrywa potencjalne błędy przepływu i poprawności danych. Dodatkowo zgodnie z **ISO 26262 7.4.17** moduł zapewnia weryfikację estymacji czasu trwania faz systemu, zgodnie z **ISO 26262 8.4.4** ułatwia analizę, zapewnia prostotę oraz ułatwia modyfikację oprogramowania.

ISO 21434

Proponowany moduł obserwatora systemu zapewnia ciągłość monitorowania krytycznych funkcji powiązanych bezpośrednio z celami i artefaktami cyberbezpieczeństwa. Dzięki automatyzacji zbierania wyników obserwatora z setek cykli działania oraz zapisywania potencjalnych błędów w pamięci nieulotnej, realizowana jest analiza wrażliwości na różnych etapach działania produktu w fazach rozwoju, produkcji, operacji oraz utrzymania.

Zgodnie z zaleceniem **ISO21434 [RQ-05-08]** zidentyfikowane podatności mogą zostać dodane jako obserwowane zdarzenia systemowe oraz używane w wielu projektach, stanowiąc część procesu stałej optymalizacji (ang. continous improvement process). Weryfikacja kluczowych faz działania systemu – inicjalizacji i deinicjalizacji, spełnia zalecenia **ISO21434 [RQ-10-10]**, czyli weryfikację dynamiczną, weryfikację przepływu danych oraz kolejności algorytmów kontrolnych.

5.4.5 Dostarczenie pakietu i wzorców projektowych dla inżynierów oprogramowania

Ogólna architektura rozwiązania

W niniejszym rozdziale dostarczone przez autora wzorce i koncepcje architektury umożliwiają użycie rozwiązania w dowolnym projekcie korzystającym ze standardu AUTOSAR. W celu wdrożenia modułu obserwatora systemu należy wykorzystać architekturę przedstawioną na

rysunku 5.17. Na rysunku jest przedstawiony „System Monitor CDD", czyli przygotowany przez autora moduł obserwatora. Komunikuje się on z warstwą oprogramowania standardowego BSW, które powinno być skonfigurowane zgodnie z instrukcjami autora. Architektura i konfiguracja logowania DLT jest opisana w artykule badawczym [74] oraz w rozdziale 5.2. Ponadto należy zapewnić podstawę pomiarów czasowych, na przykład z modułu zegara GPT i skonfigurować zapisywanie do pamięci nieulotnej NvM, zgodnie z instrukcjami przedstawionymi w rozdziale 5.3. W trakcie konfiguracji stosu NvM warto także użyć algorytmu obliczającego żywotność pamięci, przedstawionego w artykule [8]. Dodatkowo na rysunku widnieje warstwa aplikacji oraz warstwa sterowników Microcontroller Abstraction Layer – (MCAL), których konfiguracja i implementacja jest zależna od wymagań projektowych.

Opis proponowanej struktury zdarzenia systemowego i proponowana struktura plików jest przedstawiona na rysunku 5.18. Zdarzenie posiada unikalny identyfikator, znacznik czasowy, licznik cyklu życia ECU (powinien być on iterowany na bieżąco i zapisywany w pamięci nieulotnej) oraz zestaw akcji wraz z wskaźnikami na funkcje. Licznik cykli życia pozwoli w łatwy sposób zidentyfikować, kiedy dane zdarzenie się zapisało. W przypadku testów wytrzymałościowych lub analiz błędów występujących u klienta jest to kluczowa informacja. Zestaw akcji pozwala na dodanie specyficznych zachowań systemu w przypadku wystąpienia danego zdarzenia. Mogą być one rozszerzane i uzupełniane w zależności od potrzeb projektowych. Podstawowym zestawem zaproponowanym przez autora są:

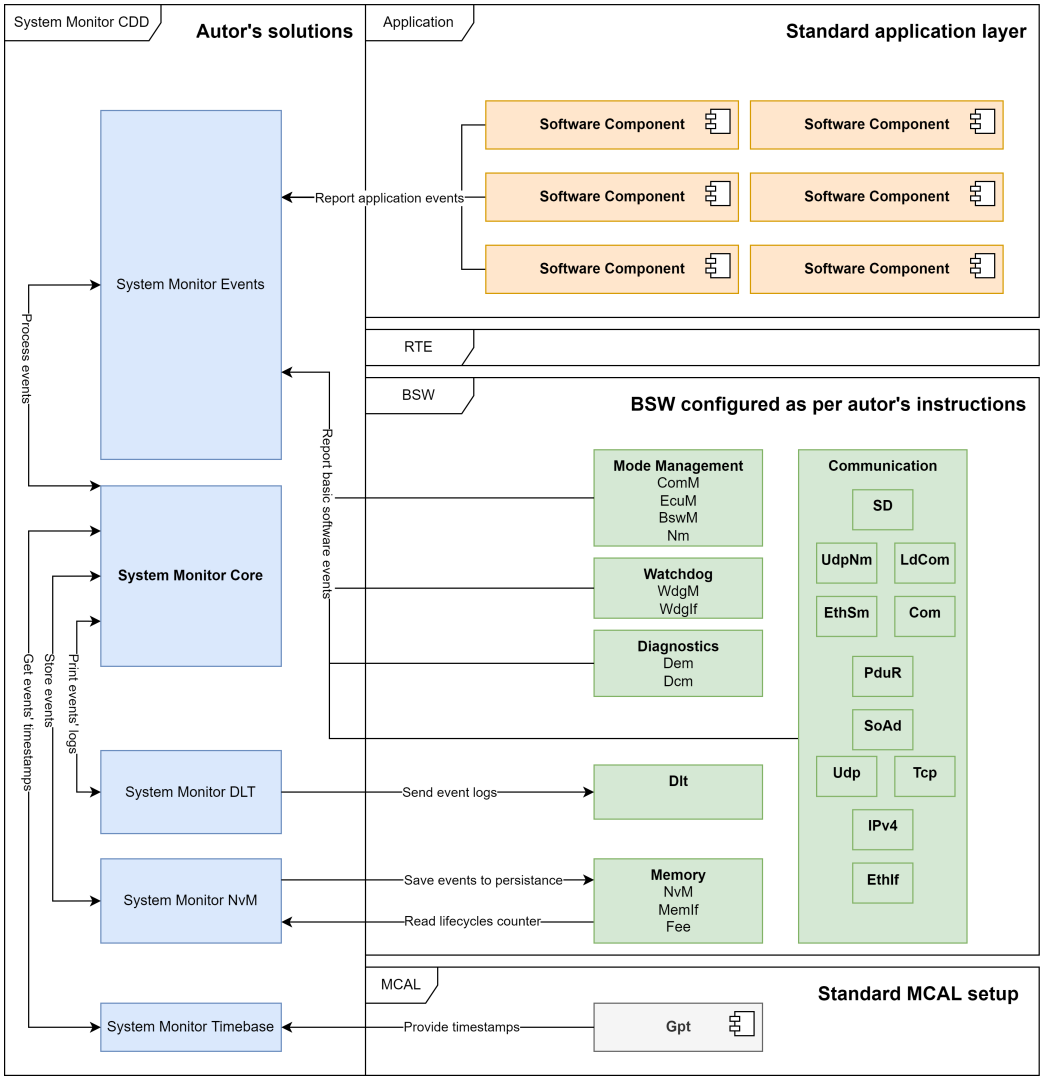
1. Akcja DLT – wiadomość wyświetlana w przypadku wystąpienia zdarzenia.

2. Akcja NvM – zapis zdarzenia do pamięci nieulotnej.

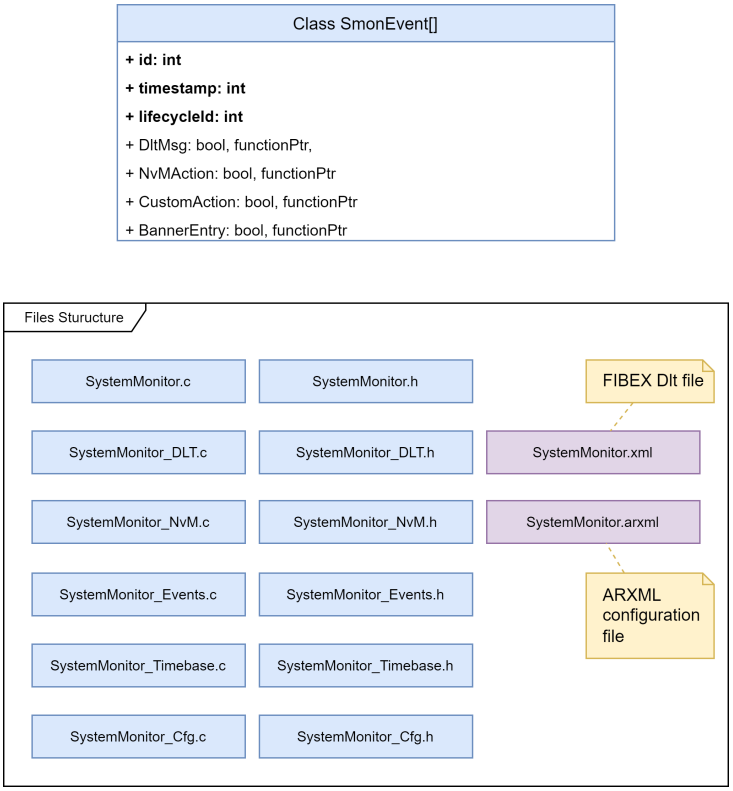
3. Akcja specyficzna – funkcja wywoływana w przypadku wystąpienia zdarzenia.

4. Akcja banner – funkcja definiująca, czy zdarzenie będzie wyświetlane w banerze informacyjnym.

Wszystkie akcje mają pierwszy parametr w postaci flagi, która definiuje czy dana funkcjonalność jest aktywna – w ten sposób można dowolnie konfigurować zachowania każdego zdarzenia.



Rysunek 5.17. Ogólna architektura modułu obserwatora systemu w systemie używającym standardu AUTOSAR



Rysunek 5.18. Proponowana struktura modułu obserwatora systemu w systemie używającym standardu AUTOSAR

W przypadku użycia obserwatora systemu w systemach opartych na językach obiektowych, naturalnym podejściem byłoby zdefiniowanie zdarzenia jako klasy bazowej i dziedziczenia jej dla odpowiednich zdarzeń opartych na wymaganiach systemowych.

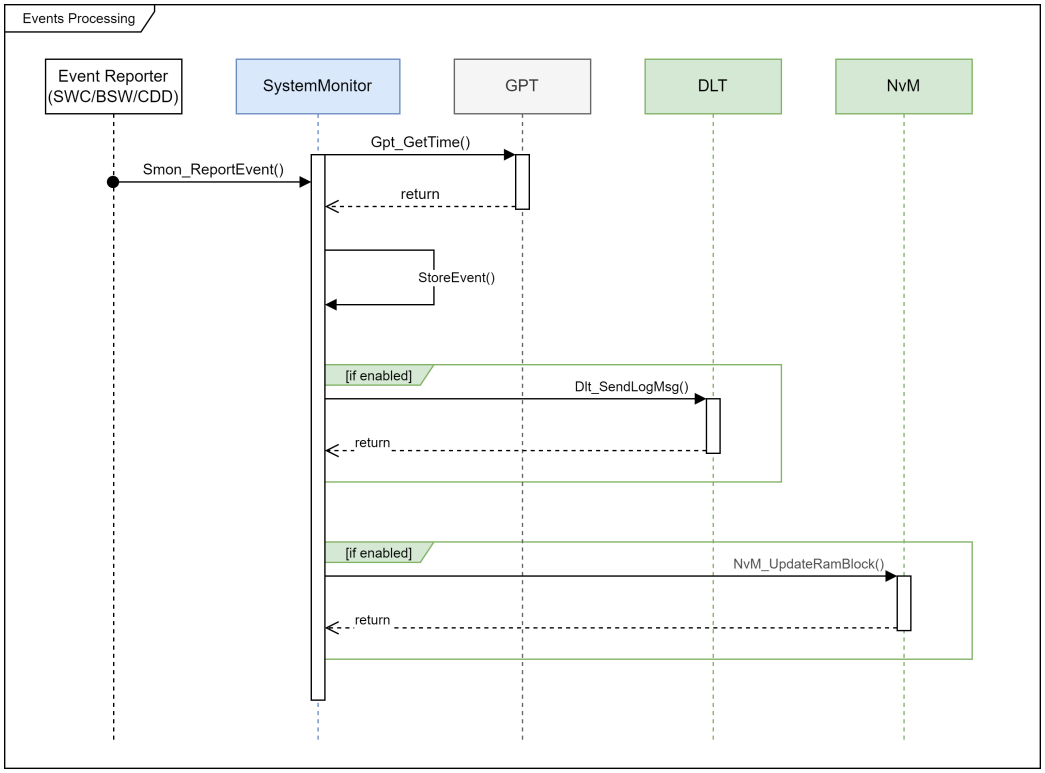
W przypadku systemów AUTOSAR Classic, język C wymaga określenia statycznych struktur. Podstawowe wykorzystywane moduły to:

1. DLT w celu transmisji logów,
2. Watchdog – (WDG) w celu konfigurowania punktów kontrolnych,
3. GPT w celu zbierania znaczników czasowych,
4. NvM w celu zapisywania i odczytywania statystyk oraz krytycznych informacji.

W przypadku wdrożenia w projekcie był wykorzystany stos komunikacyjny Ethernet, ale DLT może być wykorzystane z jakimkolwiek protokołem komunikacyjnym. Zgodnie z [92] moduł, który prowadzi interakcje ze stosem oprogramowania bazowego BSW powinien być zdefiniowany jako sterownik typu CDD oraz posiadać plik konfiguracyjny w formacie ARXML. Opisane wcześniej zestawy informacji powinny być skonfigurowane jako porty Sender-Receiver (S-R) lub Client-Server (C-S) w zależności od modułu. Porty S-R używane są do przekazywania danych, więc na przykład znacznik czasowy lub wartość napięcia na akumulatorze powinny być skonfigurowane jako port S-R.

Porty C-S używane są do wywoływania funkcji, czyli np. akcji zapisu do pamięci nieulotnej lub wywołanie procedury diagnostycznej powinny być skonfigurowane jako port C-S. Dla zgodności ze standardowym sposobem implementacji modułów CDD powinny zostać skonfigurowane dwie funkcje wykonywalne (ang. runnables): funkcja inicjalizacyjna i okresowa funkcja główna (ang. MainFunction). Dodatkowo moduł obserwatora systemu może zostać używany jako centralne miejsce na funkcje niezdefiniowane przez AUTOSAR, ale używane przez konkretne moduły – jak na przykład odczytywanie czasu systemowego lub udostępnianie informacji kontekstowych dla modułów diagnostycznych.

Sposób raportowania zdarzeń systemowych zaproponowany przez autora jest przedstawiony na diagramie 5.19. Przedstawiono na nim tylko dwie funkcje: wyświetlenie wiadomości DLT i zapisanie zdarzenia do pamięci nieulotnej – obydwie zależne od konfiguracji flag. W ten sam sposób będą przetwarzane inne akcje w zależności od potrzeb projektowych. Może to być na przykład wywołanie funkcji naprawczej w przypadku wystąpienia błędu, zapisanie kodu błędu diagnostycznego DTC, a nawet wywołanie resetu całego ECU.



Rysunek 5.19. Przekazywanie informacji do modułu obserwatora w systemie używającym standardu AUTOSAR

Rozdział 6

Wnioski i uwagi końcowe

6.1 Podsumowanie prac badawczych

Zgodnie z opisanymi w rozdziale 1.1 celami pracy, przedstawiono przez autora pakiet rozwiązań, stanowiący metodę wspomagającą proces tworzenia oprogramowania.

Rozwiązania te nawiązują do problemów – **P1**, **P2**, **P3**, **P4**, które zostały zidentyfikowane podczas analiz procesu tworzenia oprogramowania. Skorzystano z 10-letniego doświadczenia komercyjnego autora, aby opisana metoda miała charakter wdrożeniowy, a zaproponowany pakiet rozwiązań mógł być użyty w wielu innych projektach komercyjnych.

Dzięki osadzeniu badań i analiz w kontekście norm o zasięgu międzynarodowym: [4], [5], [6] oraz standardu AUTOSAR, które są uznawane jako „zgodne ze sztuką" w przemyśle motoryzacyjnym, elementy metody można wykorzystać w systemach wbudowanych stosowanych w każdym seryjnie produkowanym samochodzie.

W celu weryfikacji zidentyfikowanych problemów oraz zagadnień badawczych: **ZB1**, **ZB2**, **ZB3**, **ZB4**, przeprowadzono autorskie badania i analizy w trzech projektach komercyjnych: **Projekt 1**, **Projekt 2** oraz **Projekt 3**, opisanych w rozdziale 4.1.1. Badania zostały wykonane przy użyciu metod badawczych: **MB1**, **MB2**, **MB3**, **MB4**, **MB6**, **MB5**, opisanych w rozdziale 4. W przypadku części badań przygotowano przez autora specjalne oprogramowanie, którego celem była analiza danych lub modyfikacja kodów źródłowych, aby zaobserwować działanie konkretnych obszarów systemu. Oprogramowanie napisane przez autora jest dołączone

do pracy, ale z powodu przeprowadzania badań w projektach komercyjnych objętych klauzulami poufności, wszystkie dane komercyjne są zanonimizowane, a dokładne nazwy czy fragmenty kodów źródłowych nie mogły być dołączone do pracy. Dla każdego rozwiązania przedstawiono instrukcje wdrożeniowe, jak wykorzystać dane rozwiązanie w dowolnym projekcie komercyjnym. Weryfikacja każdego elementu metody wspomagającej proces tworzenia oprogramowania polegała na prezentacji wyników badań oraz zaprezentowaniu wpływu na normę jakości oprogramowania ISO 25010, normę bezpieczeństwa funkcyjnego ISO 26262 oraz normę cyberbezpieczeństwa ISO 21434.

Listy przedstawione poniżej podsumowują wszystkie zidentyfikowane problemy, zagadnienia badawcze, metody badawcze oraz rozwiązania. Tabela 6.1 przedstawia podsumowanie relacji pomiędzy wymienionymi elementami badań. Zidentyfikowane problemy:

Problemy	Zagadnienie badawcze	Metody badawcze	Rozwiązanie
P1	**ZB1**	**MB1**, **MB4**	**Rozwiązanie 1**
P2, **P3**	**ZB2**, **ZB3**	**MB1**, **MB2**, **MB3**	**Rozwiązanie 2**
P2, **P3**, **P4**	**ZB2**, **ZB3**, **ZB4**	**MB1**, **MB2**, **MB4**, **MB6**, **MB5**	**Rozwiązanie 3**
P3, **P4**	**ZB3**,**ZB4**	**MB1**, **MB2**, **MB4**, **MB6**, **MB5**	**Rozwiązanie 4**

Tablica 6.1. Podsumowanie relacji problemów, metod oraz zagadnień badawczych wraz z elementami rozwiązań

1. **P1** – skalowalność ekstraktów diagnostycznych.
2. **P2** – ograniczeń zasobów sprzętowych.
3. **P3** – wysokiej złożoności oprogramowania.
4. **P4** – skomplikowanych ograniczeń i zależności czasowych.

Postawione zagadnienia badawcze:

1. **ZB1** – Jak rozwiązać problem skalowalności ekstraktów diagnostycznych w projektach AUTOSAR?

2. **ZB2** – Jak minimalizować zużycie zasobów sprzętowych wybranych obszarów oprogramowania w projektach AUTOSAR?
3. **ZB3** – Jak minimalizować negatywne skutki wynikające z wysokiego skomplikowania oprogramowania?
4. **ZB4** – Jak zarządzać skomplikowanymi zależnościami i ograniczeniami czasowymi?

Zastosowane metody badawcze:

1. **MB1** – analiza statyczna kodu.
2. **MB2** – badania czasu wykonania kodu.
3. **MB3** – badania obciążenia CPU.
4. **MB4** – analiza dokumentów.
5. **MB5** – analiza przebiegu wykonania kodu.
6. **MB6** – analiza przy użyciu obserwatora systemu.

Zaproponowane rozwiązania:

1. **Rozwiązanie 1** – automatyzacja stosu diagnostycznego.
2. **Rozwiązanie 2** – automatyzacja oraz architektura DLT.
3. **Rozwiązanie 3** – automatyzacja oraz architektura stosu NvM.
4. **Rozwiązanie 4** – obserwator systemu.

6.2 Podsumowanie pakietu rozwiązań

W rozdziale 5.1 zaproponowano metodę automatyzacji oprogramowania przetwarzającego dane diagnostyczne w systemach zgodnych ze standardem AUTOSAR. Przygotowana przez autora metoda polega na automatycznym generowaniu kodu obsługującego serwisy diagnostyczne wymagane przez producenta samochodu. Skrypt generujący kod wynikowy jest oparty na ekstraktach diagnostycznych w formacie ARXML, dzięki czemu może być zastosowany w większości projektów komercyjnych.

Jednocześnie, metoda przewiduje weryfikację danych diagnostycznych, porównując ekstrakt ARXML z wynikowym kodem źródłowym. Dzięki temu można uniknąć błędów, takich jak rozbieżność danych – w przypadku, gdy producent samochodu zmieni format jakiegoś serwisu (na przykład format daty czy godziny, lub dokładność lokalizacji GPS). W takich sytuacjach skrypt automatycznie zmieni wymagane długości i pozycje danych diagnostycznych.

Uniwersalność przedstawionej metody polega na tym, że badania mogą być kontynuowane przez zastosowanie przyjętego rozwiązania do przetwarzania ekstraktów komunikacyjnych, które także są definiowane w formacie ARXML, ale dotyczą sygnałów komunikacyjnych, a nie diagnostycznych. Integracja ekstraktów komunikacyjnych jest bardzo czasochłonna i skomplikowana, więc możliwość poprawy efektywności procesu poprzez wdrażanie innowacyjnych rozwiązań jest wysoce pożądana.

Kolejne zaproponowane przez autora rozwiązanie opisane w rozdziale 5.2 to automatyzacja tworzenia infrastruktury logowania danych w komponentach oprogramowania, umożliwiająca szybkie tworzenie kompleksowych zestawów informacji na temat działania systemu. Dodatkowo przedstawiono przez autora propozycję architektury systemu logowania opartego na module AUTOSAR DLT, która uzupełnia istniejące rozwiązanie, biorąc pod uwagę potrzebę minimalizacji zużycia zasobów systemowych oraz maksymalizacji ilości informacji generowanych przez komponenty. Innymi słowy, propozycja autora określa jak optymalnie zrealizować system logowania danych w sterownikach samochodowych, spełniając następujące założenia:

1. krótki czas integracji systemu logowania w komponentach oprogramowania,
2. niskie wykorzystanie zasobów mikrokontrolera (obciążenie CPU, wykorzystanie pamięci itd.),
3. duża ilość transmitowanych informacji na temat działania systemu.

Myślą przewodnią proponowanego rozwiązania jest efektywne radzenie sobie z wysokim poziomem skomplikowania oprogramowania w pojazdach samochodowych, a co za tym idzie, możliwość ograniczenia dużej ilości coraz trudniejszych do analizy błędów. Dzięki opisanemu rozwiązaniu, komponenty oprogramowania mogą transmitować na bieżąco bardzo dużo informacji na temat działania sterownika ECU, np.:

1. dotyczących zmian sygnałów komunikacyjnych (sygnału stacyjki, stanu naładowania baterii, pozycji GPS samochodu, statusu podsystemów itp.),

2. interakcji z warstwą sprzętową mikrokontrolera (operacji na pamięci, przetwornicach, jednostkach kryptograficznych itp.),

3. raportowania stanu najważniejszych komponentów aplikacji (ewentualnych kodów błędów, nieobsłużonych wyjątków, niewłaściwych wyników walidacji danych itp.).

Ogromne możliwości daje ewentualna praca nad automatyczną analizą i prezentowaniem danych zaraportowanych przez sterowniki w samochodzie. Automatyczny sposób korelacji danych z wielu sterowników może znacznie ułatwić proces analizy zachowań i potencjalnych błędów. Graficzne przedstawienie inicjalizacji wszystkich podsystemów w samochodzie zaprezentowane na osi czasu niewątpliwie ułatwiłoby analizę złożonych zależności.

Kolejnym analizowanym obszarem jest stos pamięci nieulotnej NvM. W rozdziale 5.3 prezentowany automatyczny sposób obliczania żywotności pamięci nieulotnej oraz architektura zarządzania dostępami do pamięci rozwiązują problem wielowątkowego dostępu. Podobnie jak w całej pracy, celem jest umożliwienie uzyskania jak największej ilości informacji na temat działania systemu. W przeciwieństwie do logów DLT, które są transmitowane na bieżąco, pamięć NvM umożliwia utrwalanie informacji. Dzięki zaproponowanym przez autora rozwiązaniom, możliwa jest ciągła analiza ilości wykorzystanej pamięci przy uwzględnieniu wielokrotnych zapisów i algorytmu równoważenia zapisów (znanego głównie z dysków i pamięci typu SSD). Ponadto, zapisy do NvM mogą być realizowane z wielu komponentów jednocześnie, bez obawy o problemy wynikające ze współbieżnego zapisu. Innymi słowy, np. w czasie wypadku samochodowego, duża ilość danych może być natychmiastowo zapisana (dane z poduszek powietrznych, lokalizacja i prędkość pojazdu, stan pasów bezpieczeństwa, ilość pasażerów itp.).

Kontynuacją badań w tym obszarze może być analiza i minimalizacja prędkości zapisu i odczytu danych, w celu możliwości zapisania jak największej ilości informacji, nawet po awaryjnej utracie zasilania. Dodatkowo standaryzacja zestawów danych zapisywanych przez różne sterowniki również mogłaby stanowić ciekawą, nową perspektywę badawczą.

Umożliwiłoby to syntezę zestawów konkretnych danych z różnych sterowników, np. jaki przebieg jest raportowany przez główny komputer, a jaki przez komputer odpowiedzialny za przeciwblokujący system hamulcowy Anti-lock Braking System (ABS).

Ostatnim zaprezentowanym rozwiązaniem, które powstało na bazie poprzednich oraz stanowi zwieńczenie badań autora jest opisany w rozdziale 5.4 moduł obserwatora systemu. Ponownie głównym motywem było zapewnienie maksymalnej ilości informacji na temat działania systemu wraz z automatyczną możliwością agregacji danych. Moduł obserwatora systemu wykorzystuje logowanie danych przy użyciu protokołu DLT, zapisywanie danych do pamięci nieulotnej oraz generuje raporty podsumowujące konkretny cykl działania sterownika (zazwyczaj zaczynający się w momencie otwarcia samochodu, uruchomienia silnika, a kończący się po wyłączeniu silnika i zamknięciu samochodu). Obserwator systemu ma prezentować i zapisywać informacje, takie jak na przykład:

1. pomiary czasowe faz działania systemu (kolejnych etapów inicjalizacji, trybu czuwania, kolejnych etapów wyłączania itd.),

2. statystyki dotyczące ilości błędów, przekroczeń budżetów czasowych zadań systemu operacyjnego itp.,

3. statystyki dotyczące poziomu obciążenia CPU, zajętości stosu, ilości wywłaszczeń procesów itp.,

4. raporty czasowe najważniejszych zdarzeń systemowych (otrzymanie zmiany sygnału stacyjki, odczyt/zapis pamięci nieulotnej, krytycznych błąd systemu operacyjnego, wystąpienie resetu mikrokontrolera itd.).

Obserwator systemu to narzędzie badawcze, które umożliwia analizę skomplikowanych zależności modułów oprogramowania wbudowanego stosowanego w pojazdach samochodowych używających standardu AUTOSAR.

Kontynuacją badań w tym obszarze może być na pewno analiza danych zebranych przez obserwatora systemu podczas testów w samochodach. Dodatkowo, podobnie jak w przypadku prac powiązanych ze stosem pamięci nieulotnej, standaryzacja zestawów danych zbieranych przez moduł obserwatora systemu stanowi bardzo interesujący kierunek dalszych prac. Każde użycie obserwatora systemu może wykorzystywać standardowy

zestaw danych oraz dodatkowe specyficzne zestawy zależne od wymagań projektowych.

Wszystkie przedstawione rozwiązania stanowią propozycję realizacji celów pracy określonych w rozdziale 1.1. Dokonano autorskiego opracowania elementów znacząco wpływających na jakość oprogramowania, zidentyfikowano obszary, które posiadały potencjalne wady lub luki oraz przeprowadzono analizy prowadzące do efektywnego zminimalizowania możliwości wystąpienia błędów, istotnego zwiększenia jakości i bezpieczeństwa całego systemu. Analiza wykazała, że zwiększona ilość informacji na temat systemu w formie logów DLT, informacji zapisanych w pamięci NvM i zdarzeń systemowych zbieranych przez obserwatora systemu pozwala zwiększyć jakość, bezpieczeństwo oraz cyberbezpieczeństwo systemu w rozumieniu norm międzynarodowych: ISO 25010, ISO 26262, ISO 21434. Zaproponowano także metody automatyzacji części składowych procesu tworzenia oprogramowania, w celu minimalizacji ryzyk wystąpienia błędów oraz zmniejszenia czasu potrzebnego na implementację. Wykazano przez autora pracy, że rozwiązania **Rozwiązanie 1** i **Rozwiązanie 2** minimalizują zużycie zasobów sprzętowych, dzięki czemu wpisują się w rygorystyczne zasady środowiska przemysłu motoryzacyjnego.

6.3 Konkluzje i dyskusja

Każdy zidentyfikowany problem został osadzony w kontekście istniejących prac naukowych, które także starają się go rozwiązać lub przedstawiają powiązane rozważania. Warto wspomnieć, że zaprezentowane problemy są złożone i ogólne. Każde zaproponowane rozwiązanie przyczynia się do zmniejszenia negatywnych skutków wynikających z problemów, ale nie eliminuje ich w całości.

Poziom skomplikowania oprogramowania cały czas rośnie w zastraszającym tempie, stąd ciągła potrzeba jego analizy, zwiększania jakości, cyberbezpieczeństwa oraz bezpieczeństwa funkcjonalnego. Każda nowa funkcja dla użytkowników końcowych to tysiące linii kodu. W ramach podsumowania pracy należy podkreślić, że jakość powinna być kluczowym elementem tego skomplikowanego świata oprogramowania. Potwierdza to fakt wprowadzania nowych norm cyberbezpieczeństwa, takich jak ISO 21434, które m.in. mają wymusić odpowiedzialne podejście do potencjalnych luk i błędów w oprogramowaniu w przemyśle

motoryzacyjnym.

Im więcej samochodów jest stale podłączonych do internetu i chmur, tym większe znaczenie ma jakość. Skala problemów wynikających z błędów w oprogramowaniu rośnie drastycznie wraz z rozwojem usług o zasięgu światowym. Przemysł w bardzo dobry sposób odpowiada poprzez stosowanie standardów, norm, metryk, ale nie zawsze jest to wystarczające. Aspekt ludzki jest bardzo ważnym elementem, który generuje znaczącą ilość problemów. Bardzo ciekawą tezę przedstawia artykuł [93]: „The coming software apocalypse", gdzie zdaniem autora obecne metody tworzenia oprogramowania nie nadążają za jego skomplikowaniem. W artykule cytowana jest także Nancy Levenson – ekspertka od bezpieczeństwa oprogramowania z uniwersytetu Massachusetts Institute of Technology (MIT), według której inżynierowie oprogramowania często nie rozumieją problemu, który starają się rozwiązać lub nie wykazują żadnego zainteresowania w tej kwestii. Skupiają się oni na rezultacie i fakcie, czy oprogramowanie działa, a nie na tym, czy zadany problem został rozwiązany w odpowiedni sposób.

Rozwiązania zaprezentowane w rozprawie są wynikiem odmiennego podejścia do problemów – nie dotyczą bezpośrednio tego, czy dana funkcjonalność działa, a skupiają się na tym **„jak"** działa oraz dążą do optymalizacji procesu i zwiększenia jakości.

Zgodnie z autorską wiedzą opartą na analizie istniejącej literatury przedstawionej w rozdziale 3, wszystkie przedstawione w niniejszej rozprawie rozwiązania są nowatorskie oraz stanowią oryginalny wkład w obecny stan wiedzy. Dla każdego problemu przeanalizowano istniejące rozwiązania w przeglądzie literatury naukowej oraz opisano jak niniejsza praca je uzupełnia. Osadzenie analiz w kontekście jednostek wbudowanych używających standardu AUTOSAR pozwoliło zacieśnić obszar badań i prac oraz nakreślić im ramy. W przypadku każdego problemu analizowano także sposób, w jaki inne przemysły go rozwiązują. Metody badawcze zostały wybrane pod kątem charakteru rozwiązywanych problemów i określone na podstawie doświadczenia komercyjnego autora oraz analizy naukowej. Dzięki dostępności komercyjnych środowisk uruchomieniowych, badania mogły być wykonywane na realnych projektach stosowanych w przemyśle.

W każdym z opisanych rozwiązań starano się zaprezentować, w jaki sposób można je rozszerzać oraz kontynuować prace badawcze i wdrożenia przemysłowe. Rozwiązanie **Rozwiązanie 1** może zostać także zastosowane do innych źródeł danych w formacie ARXML (np. ekstraktów

komunikacyjnych), moduł obserwatora systemu, czyli **Rozwiązanie 4** może zostać rozszerzony o wiele elementów w zależności od potrzeb projektowych (np. o monitorowanie i pomiary dotyczące przetwarzania magistrali komunikacyjnych).

Koncepcje architektury i automatyzacji przedstawione w ramach rozwiązań **Rozwiązanie 2** i **Rozwiązanie 3** mogą być uzupełniane zgodnie z dokumentacją [75], [76] oraz wymaganiami projektowymi (np. integracji pomiarów z systemami ciągłej integracji oprogramowania, jak Jenkins). Powiązanie metod badawczych pozwala na kontynuowanie badań w przypadku wprowadzania nowych elementów metody wspomagającej proces tworzenia oprogramowania. Odniesienia przedstawione w tabeli 6.1 mogą być wykorzystane w przypadku chęci realizacji dalszych prac.

Opracowane rozwiązania w formie metody wspomagającej proces tworzenia oprogramowania zostały udostępnione dla architektów innych systemów w formie wzorców i instrukcji, co wzmacnia charakter wdrożeniowy pracy. Dzięki osadzeniu metody w kontekście standardu AUTOSAR, używanego w prawie każdym seryjnie produkowanym pojeździe samochodowym, rozwiązanie może być szeroko upowszechniane wśród architektów systemów wbudowanych.

Reasumując, w pracy zidentyfikowano problemy badawcze dotyczące: skalowalności ekstraktów diagnostycznych (**P1**), ograniczeń zasobów sprzętowych (**P2**), wysokiej złożoności oprogramowania (**P3**) oraz skomplikowanych ograniczeń i zależności czasowych (**P4**). Do ich rozwiązania opracowano stanowisko badawcze oraz dobrano metody badawcze, takie jak: analiza statyczna kodu (**MB1**), badania czasu wykonania kodu (**MB2**), badania obciążenia CPU (**MB3**), analiza dokumentów (**MB4**), analiza przebiegu wykonania kodu (**MB5**) oraz analiza przy użyciu obserwatora systemu (**MB6**). Wynikiem było opracowanie pakietu rozwiązań stanowiącego metodę wspomagającą proces tworzenia oprogramowania. Walidacja rozwiązań wykazała następujące usprawnienia: skrócenie czasu wykonywania operacji na pamięci, zmniejszenie zajętości pamięci, zmniejszenie obciążenia mikrokontrolera, skrócenie czasu integracji oprogramowania, zwiększenie czytelności kodu przez generowanie komentarzy, zwiększenie ilości automatycznie generowanego kodu źródłowego. Rezultaty oceniono także z punktu widzenia kryteriów zawartych w normach o zasięgu międzynarodowym.

Uważa się, że postawiona teza:

Poprzez wykorzystanie zaproponowanego w pracy pakietu rozwiązań stanowiącego metodę wspomagającą proces tworzenia oprogramowania, istnieje możliwość poprawy jakości, bezpieczeństwa funkcyjnego i cyberbezpieczeństwa oprogramowania w rozumieniu norm o zasięgu międzynarodowym – ISO 25010, ISO 26262 i ISO 21434.

została wykazana.

Program doktoratów wdrożeniowych pozwolił autorowi „wybudować most" pomiędzy naukowym oraz komercyjnym podejściem do inżynierii oprogramowania. Wszystkie opracowane metody zostały wdrożone w opisanych projektach komercyjnych. Wgląd w świat naukowej analizy problemów w bardzo dużej mierze pomógł autorowi w pracy komercyjnej oraz otworzył oczy na wiele kluczowych aspektów, na które wcześniej nie zwracał uwagi. Można przypuszczać, że zaprezentowane przez autora analizy i rozwiązania będą mogły być wykorzystane przez innych inżynierów oraz naukowców w celu dalszego rozwoju współpracy przemysłu z nauką. Zdaniem autora jest to szczególnie potrzebne, chcąc się odnaleźć we współczesnym, skomplikowanym i fascynującym świecie.

Spis rysunków

2.1 Model referencyjny procesu ASPICE 15
2.2 Warstwowa walidacja dla modelu V 18
2.3 Partnerzy konsorcjum AUTOSAR (Źródło [22]) 20
2.4 Stos AUTOSAR Classic 21
2.5 Kategorie oraz elementy normy jakości oprogramowania – ISO 25010 27

5.1 Proces przetwarzania ekstraktu diagnostycznego (Źródło [71]) 60
5.2 Ręczny proces integracji ekstraktu diagnostycznego. 63
5.3 Pochodzenie danych diagnostycznych 65
5.4 Automatyczny proces integracji ekstraktu diagnostycznego . . 66
5.5 Zdolność wyśledzenia elementów stosu diagnostycznego 79
5.6 Algorytm przetwarzania ekstraktu diagnostycznego 82
5.7 Proces manualnej integracji logowania w trybie „verbose” . . . 91
5.8 Proces manualnej integracji logowania w trybie „non-verbose” 92
5.9 Ilość danych przed i po optymalizacji – w trzech komponentach oprogramowania (SWC1, SWC2, SWC3) 94
5.10 Proces automatycznej integracji logowania w trybie „non-verbose” 99
5.11 Czasy operacji zapisu, odczytu oraz raportowania statusu bloków pamięci NvM zebrane podczas ponad 100 cykli działania sterownika ECU 105
5.12 Centralne synchroniczne zarządzanie dostępami do pamięci NvM 109
5.13 Czasy operacji zapisu, odczytu oraz raportowania statusu bloków pamięci NvM po zmianach usuwających problem wielowątkowego dostępu do stosu, zebrane podczas ponad 100 cykli działania sterownika ECU 111
5.14 Podstawowe fazy punktów kontrolnych obserwatora systemu . 120

5.15 Czasy trwania poszczególnych faz inicjalizacji systemu zebrane podczas ponad 160 cykli działania sterownika 123
5.16 Czasy trwania poszczególnych faz deinicjalizacji systemu zebrane podczas ponad 160 cykli działania sterownika 124
5.17 Ogólna architektura modułu obserwatora systemu w systemie używającym standardu AUTOSAR 129
5.18 Proponowana struktura modułu obserwatora systemu w systemie używającym standardu AUTOSAR 130
5.19 Przekazywanie informacji do modułu obserwatora w systemie używającym standardu AUTOSAR 132

Lista kodów źródłowych

5.1 Przykład wygenerowanych funkcji stosu diagnostycznego . . . 67
5.2 Przykład wygenerowanych funkcji stosu diagnostycznego po przetworzeniu przez skrypt automatyzujący 68
5.3 Wartownik modułowy dla pojedynczej jednostki translacji . . 70
5.4 Wartownik modułowy dla pojedynczej funkcji 71
5.5 Przykład wygenerowanych definicji przez skrypt automatyzujący 71
5.6 Przykład wygenerowanej funkcji odczytu z logami oraz zapisaniem czasu 72
5.7 Kod zabezpieczający przed brakiem nowego sygnału 76
5.8 Przykładowy fragment ekstraktu diagnostycznego przedstawiający jeden serwis diagnostyczny (DID) 83
5.9 Przykładowa konfiguracja pliku opisowego DLT dla pojedynczego komponentu oprogramowania, zawierająca jeden log 99
5.10 Przykład zadania systemu operacyjnego realizującego funkcje stosu NvM 103
5.11 Przykład synchronicznej funkcji realizującej funkcje stosu NvM 103
5.12 Funkcja eksportująca wszystkie bloki NvM z konfiguracji w narzędziu EB Tresos wraz z ich parametrami 115
5.13 Przykładowy rezultat działania skryptu do obliczania żywotności 116
5.14 Przykładowy log przedstawiający informację na temat zmiany sygnału stacyjki ze stanu nieokreślonego SNA na stan LOCK czyli samochód zamknięty, kluczyk nieobecny 121

Słownik skrótów ze skorowidzem

ABS Anti-lock Braking System. 138

ADAS Advanced Driver Assistant System. 31, 36, 44

APS Arbitrary Periodic Solution. 47

ARM Advanced RISC Machine. 36, 51, 52

ARXML AUTOSAR XML –. 32–34, 38, 60–62, 65, 71, 78, 83–85, 91, 98, 131, 135, 136, 140, 147, *Słownik terminów:* AUTOSAR XML –

ASIL Automotive Safety Integrity Level. 19, 20, 25

ASPICE Automotive Software Process Improvement and Capability Determination. 14–19, 43, 143

AUTOSAR AUTomotive Open System ARchitecture. 8, 10–12, 14, 18–24, 26, 31–38, 41–46, 48, 49, 56, 57, 59–62, 65, 71, 77, 80, 81, 83–86, 98, 100, 101, 104, 106–108, 110, 114, 118, 120, 121, 124, 126, 127, 129–136, 138, 140, 141, 143, 144

BCM Body Control Module. 10

BMS Battery Management System. 10, 41

BSS Block Started by Symbol. 118

BSW Basic Software –. 22, 23, 36, 38, 128, 131, 147, *Słownik terminów:* Basic Software –

BswM Basic Software Manager. 117, 121, 124

C-S Client-Server. 131

CAN Controller Area Network. 35, 36

CDD Complex Device Driver –. 23, 37, 84, 118, 131, 148, *Słownik terminów:* Complex Device Driver –

Com Communication –. 148, *Słownik terminów:* Communication –

ComM Communication Manager –. 121, 148, *Słownik terminów:* Communication Manager –

CPU Central Processing Unit. 35, 55, 71, 88, 89, 96, 136, 138

CRC Cyclic redundancy check. 115

CSV Comma-separated values. 115

Dcm Diagnostic Communication Manager –. 59, 69, 121, 148, *Słownik terminów:* Diagnostic Communication Manager –

Dem Diagnostic Event Manager –. 59, 121, 148, *Słownik terminów:* Diagnostic Event Manager –

DID Data Identifier. 31, 32, 60, 75, 83–85, 145

DLT Diagnostic Log and Trace –. 71, 72, 86–101, 121, 128, 130, 131, 136–139, 145, 148, *Słownik terminów:* Diagnostic Log and Trace –

DMIP Dhrystone Million Instructions Per Second –. 20, 148, *Słownik terminów:* Dhrystone Million Instructions Per Second –

DTC Diagnostic Trouble Codes. 31, 32, 61, 121, 125, 131

E/E Electrical/Electronic. 25, 28

ECU Electronic Control Unit –. 7, 22, 32, 36, 39, 44, 49, 52, 104, 105, 111, 119, 123–125, 128, 131, 136, 143, 148, *Słownik terminów:* Electronic Control Unit –

EcuM Electronic Control Unit Manager. 121, 124

EEPROM Electrically Erasable Programmable Read-Only Memory. 108

FEE Flash Emulated EEPROM. 101, 114, 124

FIBEX Field Bus Exchange Format –. 88, 90, 91, 93, 94, 97, 99, 100, 121, 149, *Słownik terminów:* Field Bus Exchange Format –

GPT General Purpose Timer. 54, 128, 130

GSM Global System for Mobile Communications. 64

HARA Hazard Analysis and Risk Assessment. 127

HIS Hersteller Initiative Software. 14, 24

HSM Hardware Security Module. 64, 65, 69, 97

ICC Inter Core Communication. 64, 65, 67

IDE Integrated Development Environment. 52

JSON JavaScript Object Notation. 92, 100

JTAG Joint Test Action Group. 51

LET Logical Execution Time. 48

LIN Local Interconnect Network. 51

MCAL Microcontroller Abstraction Layer –. 128, 149, *Słownik terminów:* Microcontroller Abstraction Layer –

MISRA Motor Industry Software Reliability Association. 13, 24, 25

MIT Massachusetts Institute of Technology. 140

MPS Multiple Periodic Solution. 47

NvM Non-Volatile Memory Manager. 35, 101–117, 121, 123, 128, 130, 137, 139, 143, 145

OEM Original Equipment Manufacturer. 60, 62, 65

OSEK Open Systems and their Interfaces for the Electronics in Motor Vehicles. 55

PDA Private Data Adapter. 37

PduR Pdu Router –. 37, 150, *Słownik terminów:* Pdu Router –

PS Periodic Solution. 47

QM Quality Management. 20, 25

RAM Random Access Memory. 35–37

ROM Read Only Memory. 35, 36

RTE Runtime Environment –. 22, 36, 42, 64, 65, 67, 72, 84, 150, *Słownik terminów:* Runtime Environment –

S-R Sender-Receiver. 131

SNA Signal Not Available. 121, 145

SPI Serial Peripheral Interface. 35, 121

SSD Solid State Drive. 106, 137

SWC Software Component –. 21, 22, 37, 92, 109, 110, 118, 150, *Słownik terminów:* Software Component –

SWE Software Engineering. 15

SWL Software Loading. 121, 125

TARA Threat Assessment and Remediation Analysis. 114, 127

TCU Telematic Control Unit. 10, 31

UART Universal asynchronous receiver-transmitter. 35

UML Unified Modeling Language. 43

V2X Vehicle-2-Anything. 36

VFB Virtual Function Bus. 23

VIN Vehicle Identification Number. 70

VLAN Virtual Local Area Network. 125

WCET Worst Case Execution Time. 45–47

WDG Watchdog –. 130, 151, *Słownik terminów:* Watchdog –

Słownik terminów ze skorowidzem

AUTOSAR XML – format plików określony przez AUTOSAR, zawierający opisy ustawień modułów, sygnałów oraz innych części projektów zgodnych ze standardem. 32, 147

Basic Software – warstwa oprogramowania zawierająca wszystkie moduły określone standardem AUTOSAR. Znajdują się w nim m.in. system operacyjny, moduły magistrali komunikacyjnych, moduły powiązane z cyberbezpieczeńswem, moduły pamięci nieulotnej itp. 22, 147

Communication – moduł AUTOSAR Classic odpowiedzialny za przetwarzanie sygnałów wszystkich magistrali komunikacyjnych, pomiędzy logiką biznesową a modułem PduR. 148

Communication Manager – moduł AUTOSAR Classic odpowiedzialny za zarządzanie stanami magistrali komunikacyjnych, np. wybudzaniem przy wykryciu ruchu, ustawianiem w stan uśpienia przy braku aktywności itp. 121, 148

Complex Device Driver – sterowniki używane w projektach zgodnych z AUTOSAR Classic niebędące częścią standardowych modułów, np. sterowniki zewnętrznych układów scalonych lub specyficznych magistrali komunikacyjnych. 23, 148

Dhrystone Million Instructions Per Second – miara wydajności procesora określająca liczbę milionów operacji całkowitoliczbowych wykonywanych w ciągu jednej sekundy. 148

Diagnostic Communication Manager – moduł AUTOSAR Classic odpowiedzialny za obsługę zapytań diagnostycznych określonych

normą ISO 14229, np. zwrócenie błędów zapisanych w sterowniku. 59, 148

Diagnostic Event Manager – moduł AUTOSAR Classic odpowiedzialny za przetwarzanie wydarzeń diagnostycznych, np. wykrycia braku oleju silnikowego przez sterownik. 59, 148

Diagnostic Log and Trace – moduł AUTOSAR Classic odpowiedzialny za zarządzanie logowaniem informacji na temat działania systemu. 71, 148

Electronic Control Unit – elektroniczny sterownik wbudowany odpowiadający za daną część funkcji w samochodzie, np. sterownik silnika, sterownik baterii, sterownik świateł itp. 7, 148

Field Bus Exchange Format – format plików opisowych oparty na XML stosowany w przemyśle motoryzacyjnym dla opisywania struktur magistrali komunikacyjnych oraz logowania danych. W przypadku użycia dla logowania, plik zawiera definicje i opisy wiadomości z logami. 88, 149

Microcontroller Abstraction Layer – warstwa oprogramowania zawierająca moduły określone standardem AUTOSAR. Warstwa ta realizuje bezpośrednie operacje na sprzęcie, rejestrach mikrokontrolera czy urządzeniach zewnętrznych. Przykładami sterowników tej warstwy jest na przykład sterownik zegara, sterownik watchdoga, sterownik pamięci nieulotnej. 128, 149

Pdu Router – moduł AUTOSAR Classic odpowiedzialny za zarządzanie wszystkimi wiadomościami komunikacyjnymi, niezależnie od magistrali. Może na przykład przekazywać ruch lub jego część z magistrali Ethernet na CAN. 37, 150

Runtime Environment – warstwa AUTOSAR Classic, której głównym celem jest zapewnienie abstrakcji logiki biznesowej od stosu oprogramowania standardowego. 22, 65, 150

Software Component – komponent oprogramowania na warstwie aplikacyjnej, wszystkie moduły realizujące specyficzne potrzeby biznesowe konkretnego projektu. Na przykład moduły algorytmów balansowania baterii, sterowania silnikiem, otwierania drzwi, przetwarzania sygnałów itp. 21, 150

Watchdog – układ elektroniczny lub moduł oprogramowania, który umożliwia wywołanie resetu mikrokontrolera, w przypadku gdy ten przestaje odpowiadać. 130, 151

Bibliografia

[1] Alex Davies. „Bored Charging Your Tesla? Play a Videogame Right in Your Car". W: *Wired* (19 czer. 2019).

[2] Vard Antinyan. „Revealing the complexity of automotive software". W: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, list. 2020.

[3] Philippe Cuenot i in. „Managing Complexity of Automotive Electronics Using the EAST-ADL". W: *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*. IEEE, lip. 2007.

[4] *ISO25010: Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. Standard. Geneva, CH: International Organization for Standardization, 2011.

[5] *ISO26262: Road vehicles – Functional safety*. Standard. Geneva, CH: International Organization for Standardization, 2018.

[6] *ISO21434: Road vehicles – Cybersecurity engineering*. Standard. Geneva, CH: International Organization for Standardization, 2021.

[7] AUTOSAR. *AUTOSAR Glossary*. 2019.

[8] Patryk Pankiewicz. *Determination of non-volatile memory lifetime in AUTOSAR based systems*. 2022.

[9] AUTOSAR. *AUTOSAR Introduction*. 2020.

[10] Prof. Phil Koopman. *A Case Study of Toyota Unintended Acceleration and Software Safety*. Wrz. 2014.

[11] *ISO33061: Information technology – Process assessment – Process assessment model for software life cycle processes*. Standard. Geneva, CH: International Organization for Standardization, 2021.

[12] Helmar Kuder. *HIS Source Code Metrics*. 2008.

[13] Motor Industry Software Reliability Association. *MISRA C: 2012 Guidelines for the use of the C language in critical systems*. 2013.

[14] VDA QMC Working Group 13 / Automotive SIG. *Automotive SPICE Process Assessment / Reference Model*. 2017.

[15] VDA QMC Project Group 13. *Automotive SPICE® for Cybersecurity Process Reference and Assessment Model*. 2021.

[16] W. W. Royce. „Managing the Development of Large Software Systems: Concepts and Techniques". W: *Proceedings of the 9th International Conference on Software Engineering*. ICSE '87. Monterey, California, USA: IEEE Computer Society Press, 1987, s. 328–338.

[17] Apoorva Srivastava, Sukriti Bhardwaj i Shipra Saraswat. „SCRUM model for agile methodology". W: *2017 International Conference on Computing, Communication and Automation (ICCCA)*. 2017, s. 864–869.

[18] Patryk Pankiewicz. „Metodyki zwinne w przemyśle Automotive". W: *Wybrane zagadnienia zarządzania projektami w przedsiębiorstwach informatycznych, Praca zbiorowa pod redakcją Jana Werewki, VII edycja SPZPI Kraków 2016*. Paź. 2016, s. 312–324.

[19] Manish Kumar, Jonghun Yoo i Seongsoo Hong. „Enhancing AUTOSAR methodology to a cotsbased development process via mapping to V-Model". W: *2009 IEEE International Symposium on Industrial Embedded Systems*. 2009, s. 50–53.

[20] Bohan Liu, He Zhang i Saichun Zhu. „An Incremental V-Model Process for Automotive Development". W: *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. 2016, s. 225–232.

[21] Iris Graessler i Julian Hentze. „The new V-Model of VDI 2206 and its validation". W: *at - Automatisierungstechnik* 68.5 (kw. 2020), s. 312–324.

[22] AUTOSAR. *AUTOSAR Layered Software Architecture*. 2017.

[23] AUTOSAR. *General Specification of Basic Software Modules*. 2017.

[24] Martin Vogel i in. „Metrics in automotive software development: A systematic literature review". W: *Journal of Software: Evolution and Process* 33.2 (sierp. 2020).

[25] Jelena Vlaovic i in. „Application lifecycle management while developing consumer electronics software using A-SPICE". W: *2018 Zooming Innovation in Consumer Technologies Conference (ZINC)*. Novi Sad: IEEE, maj 2018.

[26] *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems*. Standard. Geneva, CH: International Organization for Standardization, 2010.

[27] Yann Argotti i in. „Quality Quantification Applied to Automotive Embedded Systems and Software Advances with qualimetry science". W: sty. 2020.

[28] Egbert Touw. „Multi-faceted Reliability Assessment Techniques: An Industrial Case Study". W: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, kw. 2017.

[29] *ISO14229: Road vehicles – Unified diagnostic services (UDS)*. Standard. Geneva, CH: International Organization for Standardization, 2013.

[30] Aitor Villar-Martinez i in. „Improving the Scalability and Replicability of Embedded Systems Remote Laboratories Through a Cost-Effective Architecture". W: *IEEE Access* 7 (2019), s. 164164–164185.

[31] Hariharan Venkitachalam i in. „Automated Continuous Evaluation of AUTOSAR Software Architecture for Complex Powertrain Systems". W: *INFORMATIK 2017*. Red. Maximilian Eibl i Martin Gaedke. Gesellschaft für Informatik, Bonn, 2017, s. 1563–1574.

[32] Georg Macher, Eric Armengaud i Christian Kreiner. „Automated generation of AUTOSAR description file for safety-critical software architectures". W: *Informatik 2014*. Red. E. Plödereder i in. Bonn: Gesellschaft für Informatik e.V., 2014, s. 2145–2156.

[33] Thomas M. Galla i in. „Refactoring an Automotive Embedded Software Stack using the Component-Based Paradigm". W: *2007 International Symposium on Industrial Embedded Systems*. IEEE, lip. 2007.

[34] Padma Iyenghar, Lars Huning i Elke Pulvermüller. „Automated End-to-End Timing Analysis of AUTOSAR-based Causal Event Chains." W: *ENASE*. 2020, s. 477–489.

[35] Sendhilraj Thangaraj, Sudhakar Gummadi i Shanmugasundaram Radhakrishnan. „Enhancement in ARM Code Optimization for Memory Constrained Embedded Systems". W: *2006 International Conference on Advanced Computing and Communications*. IEEE, grud. 2006.

[36] James Pallister, Kerstin Eder i Simon J. Hollis. „Optimizing the flash-RAM energy trade-off in deeply embedded systems". W: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, lut. 2015.

[37] Răzvan Bogdan i in. „Optimization of AUTOSAR Communication Stack in the Context of Advanced Driver Assistance Systems". W: *Sensors* 21.13 (lip. 2021), s. 4561.

[38] Haibo Zeng i Marco Di Natale. „Efficient implementation of AUTOSAR components with minimal memory usage". W: *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. IEEE, czer. 2012.

[39] Peter Gliwa. *Embedded Software Timing*. Springer International Publishing, 2021.

[40] Darko Durisic i in. „Measuring the impact of changes to the complexity and coupling properties of automotive software systems". W: *Journal of Systems and Software* 86.5 (maj 2013), s. 1275–1293.

[41] Federico Ciccozzi i in. „Architecture optimization: speed or accuracy? both!" W: *Software Quality Journal* 26.2 (list. 2016), s. 661–684.

[42] Siteng Cao, Yongxin Zhao i Ling Shi. „Software Complexity Reduction by Automated Refactoring Schema". W: *2019 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, lip. 2019.

[43] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. 2nd. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[44] Yanming Chu i Shiyi Xu. „Exploration of complexity in software reliability". W: *Tsinghua Science and Technology* 12.S1 (lip. 2007), s. 266–269.

[45] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. USA: Elsevier Science Inc., 1977.

[46] T.J. McCabe. „A Complexity Measure". W: *IEEE Transactions on Software Engineering* SE-2.4 (grud. 1976), s. 308–320.

[47] Sonam Bhatia i Jyoteesh Malhotra. „A survey on impact of lines of code on software complexity". W: *2014 International Conference on Advances in Engineering Technology Research (ICAETR - 2014)*. IEEE, sierp. 2014.

[48] Ilja Heitlager, Tobias Kuipers i Joost Visser. „A Practical Model for Measuring Maintainability". W: *6th International Conference on the Quality of Information and Communications Technology (2007)*. IEEE, wrz. 2007.

[49] Tu Honglei, Sun Wei i Zhang Yanan. „The Research on Software Metrics and Software Complexity Metrics". W: *2009 International Forum on Computer Science-Technology and Applications*. IEEE, 2009.

[50] Zheng Li, Liam O'Brien i Ye Yang. „Impact of Product Complexity on Actual Effort in Software Developments: An Empirical Investigation". W: *2014 23rd Australian Software Engineering Conference*. IEEE, kw. 2014.

[51] Hyun Chul Jo i in. „RTE Template Structure for AUTOSAR based Embedded Software Platform". W: *2008 IEEE/ASME International Conference on Mechtronic and Embedded Systems and Applications*. IEEE, paź. 2008.

[52] Mohit Garg i Richard Lai. „Measuring the constraint complexity of automotive embedded software systems". W: *2014 International Conference on Data and Software Engineering (ICODSE)*. IEEE, list. 2014.

[53] Jooyoung Seo i in. „Which Spot Should I Test for Effective Embedded Software Testing?" W: *2008 Second International Conference on Secure System Integration and Reliability Improvement*. IEEE, lip. 2008.

[54] S. Magnus Ågren i in. „The impact of requirements on systems development speed: a multiple-case study in automotive". W: *Requirements Engineering* 24.3 (lip. 2019), s. 315–340.

[55] P. Puschner i R. Kirner. „Avoiding timing problems in real-time software". W: *Proceedings IEEE Workshop on Software Technologies for Future Embedded Systems. WSTFES 2003*. IEEE Comput. Soc.

[56] Priyanshi Gupta, N. P. Singh i Geetha Srinivasan. „An Efficient Approach For Mapping AUTOSAR Runnables in Multi-core Automotive systems to Minimize Communication Cost". W: *2019 Innovations in Power and Advanced Computing Technologies (i-PACT)*. IEEE, mar. 2019.

[57] Fouad Khenfri, Khaled Chaaban i Maryline Chetto. „Efficient mapping of runnables to tasks for embedded AUTOSAR applications". W: *Journal of Systems Architecture* 110 (list. 2020), s. 101800.

[58] Qingling Zhao, Zonghua Gu i Haibo Zeng. „Design optimization for AUTOSAR models with preemption thresholds and mixed-criticality scheduling". W: *Journal of Systems Architecture* 72 (sty. 2017), s. 61–68.

[59] Ernest Wozniak i in. „An optimization approach for the synthesis of AUTOSAR architectures". W: *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE, wrz. 2013.

[60] Rongshen Long i in. „An Approach to Optimize Intra-ECU Communication Based on Mapping of AUTOSAR Runnable Entities". W: *2009 International Conference on Embedded Software and Systems*. IEEE, 2009.

[61] Heeseung Jo i in. „Optimizing the startup time of embedded systems: a case study of digital TV". W: *IEEE Transactions on Consumer Electronics* 55.4 (list. 2009), s. 2242–2247.

[62] Gaurav Singh, Kumar Bipin i Rohit Dhawan. „Optimizing the boot time of Android on embedded system". W: *2011 IEEE 15th International Symposium on Consumer Electronics (ISCE)*. IEEE, czer. 2011.

[63] Alessandro Biondi i in. „Logical Execution Time Implementation and Memory Optimization Issues in AUTOSAR Applications for Multicores". W: czer. 2017.

[64] Arne Hamann i in. „Communication Centric Design in Complex Automotive Embedded Systems". en. W: (2017).

[65] Daniel Kaestner i in. „Finding All Potential Run-Time Errors and Data Races in Automotive Software". W: *SAE Technical Paper Series*. SAE International, mar. 2017.

[66] Al Danial. *cloc - Count Lines of Code*. Wer. 1.92. 5 grud. 2021. URL: `https://github.com/AlDanial/cloc`.

[67] AUTOSAR. *Specification of GPT Driver*. 2017.

[68] AUTOSAR. *Specification of Operating System*. 2017.

[69] AUTOSAR. *Specification of Diagnostic Event Manager*. 2017.

[70] AUTOSAR. *Specification of Diagnostic Communication Manager*. 2017.

[71] AUTOSAR. *Diagnostic Extract Template AUTOSAR*. 2017.

[72] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[73] Haris Mumtaz i in. „An empirical study to improve software security through the application of code refactoring". W: *Information and Software Technology* 96 (kw. 2018), s. 112–125.

[74] Patryk Pankiewicz. „Influence of Various DLT Architectures on the CPU Resources". W: *Theory and Engineering of Dependable Computer Systems and Networks*. Springer International Publishing, 2021, s. 339–348.

[75] AUTOSAR. *Specification of Diagnostic Log and Trace*. 2017.

[76] AUTOSAR. *Specification of Specification of NVRAM Manager*. 2017.

[77] AUTOSAR. *NV Data Handling Guideline*. 2017.

[78] AUTOSAR. *Specification of Specification of Flash EEPROM Emulation*. 2017.

[79] AUTOSAR. *Specification of Specification of Flash Driver*. 2017.

[80] Yuan-Hao Chang, Jen-Wei Hsieh i Tei-Wei Kuo. „Improving Flash Wear-Leveling by Proactively Moving Static Data". W: *IEEE Transactions on Computers* 59.1 (sty. 2010), s. 53–65.

[81] Yuan-Hao Chang, Jen-Wei Hsieh i Tei-Wei Kuo. „Endurance enhancement of flash-memory storage systems". W: *Proceedings of the 44th annual conference on Design automation - DAC '07*. ACM Press, 2007.

[82] Ming-Chang Yang i in. „Garbage collection and wear leveling for flash memory: Past and future". W: *2014 International Conference on Smart Computing*. IEEE, list. 2014.

[83] Dragan Radanovic i in. „One solution for persistent data storage in automotive industry". W: *2017 Zooming Innovation in Consumer Electronics International Conference (ZINC)*. IEEE, maj 2017.

[84] Hyun-Yong Hwang, Jeong-Hwan Lee i Tae-Man Han. „Design of new AUTOSAR memory stack for diagnostic message". W: *2012 International Conference on ICT Convergence (ICTC)*. IEEE, paź. 2012.

[85] Elektrobit. *Tresos Studio*. 1 wrz. 2022.

[86] Elektrobit. *EB Tresos Studio for ACG8 developer's guide.* 2020.

[87] Patryk Pankiewicz. „Embedded Systems' Startup Code Optimization". W: *New Advances in Dependability of Networks and Systems*. Springer International Publishing, 2022, s. 197–205.

[88] AUTOSAR. *Guide to Mode Management*. 2017.

[89] Elektrobit. *EB Tresos AutoCore Generic 8 Mode Management documentation.* 2020.

[90] AUTOSAR. *Specification of ECU State Manager*. 2017.

[91] AUTOSAR. *Specification of Basic Software Mode Manager*. 2017.

[92] AUTOSAR. *Complex Driver design and integration guideline*. 2017.

[93] James Somers. *The coming software apocalypse*. List. 2017.