POLITECHNIKA WARSZAWSKA

Dziedzina nauk inżynieryjno-technicznych
Informatyka techniczna i telekomunikacja

Rozprawa doktorska

mgr inż. Kazimierz Krosman

Ewaluacja systemów wbudowanych poprzez monitorowanie programowe

Promotor
prof. dr hab. inż. Janusz Sosnowski

Promotor Pomocniczy
dr inż. Piotr Gawkowski

WARSZAWA 2022

Podziękowania

Przede wszystkim dziękuję promotorowi, Panu profesorowi Januszowi Sosnowskiemu za nieograniczoną pomoc oraz wsparcie, które otrzymałem w trakcie tej drogi.

Dziękuję mojej żonie Agnieszce i synowi Stanisławowi za umożliwienie realizacji tej pracy oraz wszelkie wsparcie.

Dziękuję moim rodzicom, bez których praca ta by nie powstała

Streszczenie

Praca poświęcona jest badaniom nad monitorowaniu programowym systemów komputerowych, w szczególności systemów wbudowanych. Rozpatrzono trzy perspektywy monitorowania dotyczące różnych źródeł informacji: i) przebieg wykonania instrukcji (tzw. ślad instrukcji), ii) wybrane sygnały chrakteryzujące pracę systemu, np. wydajność przetwarzania, zużycie energii (prezentowane jako szeregi czasowe), iii) logi programowe. Analiza literatury oraz doświadczenia praktyczne autora pozwoliły stwierdzić potrzebę rozwinięcia efektywnych algorytmów analizy danych uwzględniających specyfikę systemów wbudowanych między innymi: ograniczone zasoby sprzętowe, wymagania czasowe, związki oprogramowania ze sprzętem oraz typowe profile aktywności operacyjnej. Dla rozpatrywanych perspektyw monitorowania opracowano oryginalne modele agregowania danych i dostosowane do nich algorytmy analizy. W ramach pracy powstały trzy zbiory algorytmów wspomagających analizę poprawności pracy systemów oraz ich optymalizację.

Analiza śladów instrukcji ukierunkowana jest na detekcję anomalii i wykorzystuje algorytm DBSCAN wkomponowany w opracowany program. Podejście to zostało zweryfikowane w eksperymentach z urządzeniem monitorującym lokalizację obiektów mobilnych (samochodów ciężarowych). Ślad instrukcji był zbierany przy wykorzystaniu dodatkowego specjalizowanego urządzenia.

Najbardziej zaawansowana jest opracowana metoda analizy szeregów czasowych bazująca na oryginalnym modelu obiektowym. Pozwala ona zidentyfikować charakterystyczne obiekty (zbiory próbek) oraz relacje między nimi. Podział i grupowanie próbek wykorzystują zbiór metryk oparty o dane statystyczne fragmentów szeregu czasowego. Modyfikacje metryk oraz parametryzacja algorytmów umożliwiają dopasowanie ich charakterystyki do różnych typów szeregów czasowych jak i wybranych poziomów abstrakcji. Przedstawiono również metody ułatwiające dobór parametrów algorytmów. Dopełnieniem zaprezentowanego podejścia jest algorytm korelacji logów tekstowych z szeregiem czasowym. Umożliwia on lepszą interpretację obiektów wykrytych w ramach analizy szeregu czasowego. Algorytm identyfikuje zdarzenia z rekordów logów tekstowych i dopasowuje je do obiektów wykrytych podczas analizy szeregów czasowych lub zdarzeń zewnętrznych uzyskanych w inny sposób. Rozpatrzono tutaj również problem korelacji czasowej przy niezależnych zegarach monitora sygnałów oraz monitorowanego systemu. Podejście to zostało zweryfikowane w badaniach poboru prądu dwóch typów komercyjnych holterów.

Słowa kluczowe: monitorowanie programowe, szeregi czasowe, logi zdarzeniowe

Summary

The doctoral thesis is devoted to research on embedded system software monitoring. The author analyses software monitoring from three perspectives based on various data sources: i) executed instructions traces, ii) signals related to system's execution profiles and iii) system logs. The literature review combined with the author's practical experience allows to identify a need to develop effective algorithms that conduct analysis based on listed data sources. The algorithms take into account the properties of embedded systems: limited hardware resources, time requirements, a dependency between software and hardware, or typical operational profiles. The author developed original data models and algorithms for each of the perspectives. As s result, there are three sets of algorithms supporting embedded system behavior analysis and optimization.

Instruction trace analysis is focused on the anomaly detection and utilizes DBSCAN grouping algorithm as a part of the processing sequence. The proposed approach was verified with the device that sends GPS coordinates of mobile objects (HGV- heavy goods vehicles). The instruction trace aggregation required a specialized external device usage.

The most advanced set of algorithms is based on the original time series analysis method that includes an objective data model. The method allows to detect distinctive objects (derived from a time-series) and identify relations between them. The samples division and grouping utilize a set of metrics based on the statistical properties of time-series' segments. The algorithms can be prepared for diverse types of time series and tuned to the desired abstraction layer by metrics or parameters modifications. The thesis proposes methods that ease parameters selection. The algorithm responsible for text logs and time-series correlation completes the presented approach. It enables analysis on a higher level of abstraction and automatically creates descriptions of objects. The algorithm identifies events in a sequence of the text logs and matches them with objects detected in the time series or events originated differently. The algorithm correlates events and time series timestamped by separate and desynchronized clocks. The presented approach was verified during experiments with power consumption analysis of the commercial ECG holters.

Keywords: software monitoring, time series, event logs.

Spis treści

1 Wprowadzenie

1.1 Motywacja tematu badań

Systemy wbudowane [1] charakteryzują się zestawem cech, który odróżnia je od innych systemów komputerowych. Cechy te wpływają na etapy oraz metody projektowania i implementowania oprogramowania dla systemów wbudowanych. Oprogramowanie takiego systemu jest związane ze sprzętem, na którym jest wykonywane. Ograniczenia sprzętowe i fizyczne platformy systemu często utrudniają zastosowanie powszechnie wykorzystywanych metod do usuwania i wykrywania błędów w oprogramowaniu.

Podczas 12 letniej praktyki zawodowej, autor miał możliwość uczestniczyć w rozwoju, w procesach projektowania, specyfikowania, implementacji, weryfikacji, walidacji i wdrażania wielu systemów wbudowanych zarówno dla branż mniej regulowanych (np. urządzenia monitorowania floty- branża automotive) jak i dla branż wymagających certyfikacji każdego systemu u regulatora (np. holter EKG- branża medyczna).

W trakcie pracy zawodowej autor napotykał powtarzające się problemy powstające w szczególności podczas procesu rozwoju oprogramowania. Autor zmagał się m.in. z problemem wykrywania i usuwania błędów, których reprodukcja jest nietrywialna i niedeterministyczna. Na przykład debugując (proces analizy i usuwania błędów w oprogramowaniu) system wbudowany w postaci lokalizatora GPRS/GPS autor poszukiwał przyczyn tymczasowych braków w komunikacji z kartą SD. Kolejną klasą problemów jest analiza i weryfikacja funkcji systemu np. optymalizacja energetyczna złożonego systemu wbudowanego. Reprezentantem takiego systemu jest urządzenie wykorzystujące sieć telefonii komórkowej do zdalnego monitorowania pacjenta poprzez ciągły pomiar EKG zwane holterem. Optymalizacja energetyczna systemu jest złożonym i czasochłonnym procesem. Podczas optymalizacji inżynier wielokrotnie analizuje wykres poboru energii w celu wytworzenia profilu energetycznego pod kątem możliwych poprawek. Cały proces jest utrudniony, ponieważ zmiany w systemie w celu pozyskiwania większej liczby rodzajów danych o stanie systemu również mają wpływ na pobór energii. Ostatnim przykładem problemu motywującego do podjęcia badań jest weryfikacja poprawności wykonania oprogramowania oraz wskazywanie wykrytych anomalii. Podczas rozwoju urządzenia deweloper weryfikuje wpływ swoich rozwiązań na zachowanie systemu. W sytuacji idealnej może to wykonać bez dodatkowych zmian w oprogramowaniu.

Jedną z metod rozwiązywania opisanych problemów jest monitorowanie programowe systemów. Autor wraz z firmą, z którą współpracował, dokonał przeglądu ówcześnie dostępnych systemów pod kątem funkcji takich jak: wspieranie wykrywania i usuwania defektów, detekcja anomalii i weryfikacja poprawności wykonania. Przegląd dostępnych komercyjnie rozwiązań wykazał, że nie istnieje uniwersalne narzędzie. Dostępne komercyjnie systemy skupiają się na części procesu pozyskiwania danych i ich prostej prezentacji (TRACE firmy LAUTERBACH), a ponadto wymagają także integracji w formie samodzielnie przygotowanego komponentu oprogramowania. Takie rozwiązania mają również ograniczenia sprzętowe. Emulator JLINK PRO, który umożliwia pomiar natężenia pobieranego prądu, ma swoje granice częstotliwości próbkowania i pamięci na próbki. Rozwijanie systemów komercyjnych o np. lepsze platformy sprzętowe również jest utrudnione poprzez zamknięte oprogramowanie, które współpracuje tylko z wybranymi platformami sprzętowymi. Zamknięte oprogramowanie zawęża funkcjonalność systemu do operowania tylko na danych, które przewidzieli autorzy oprogramowania. Z kolei automatyzacja monitorowania i analizy systemów wbudowanych wymaga realizowania następujących zadań: 1) agregacji danych i 2) analizy danych mającej na celu stworzenie profilu zachowania urządzenia, wykrycia anomalii czy nawet opisu poszczególnych fragmentów. Zadania te wymagają bardziej skomplikowanych algorytmów. Ponadto takie algorytmy mogą wymagać fuzji danych z różnych źródeł.

Na podstawie doświadczeń jak i przeglądu rozwiązań autor podjął decyzję o rozpoczęciu działalności naukowo-badawczej na Politechnice Warszawskiej w Instytucie Informatyki/Zakładzie Oprogramowania i Architektury Komputerów w celu badania możliwości monitorowania oprogramowania i sprzętu systemów wbudowanych. Początkowym celem badań było znalezienie metod rozwiązujących opisane problemy (w szczególności poszukiwania anomalii i błędów funkcji oprogramowania). Wiele wyników prowadzonych badań zostało opublikowanych w [2] [3] [4] [5]. W pracy zostały one rozwinięte, uogólnione i usystematyzowane. Między innymi istotnie rozszerzono opisy algorytmów, interpretację analiz oraz eksperymentów. Załączona została również specyfikacja oraz możliwości funkcjonalne opracowanych narzędzi. Powstały także artykuły i raporty omawiające problematykę awarii w systemach wbudowanych [6] [5]. Kierunek podjętych badań wynikał z przeprowadzonej analizy literatury oraz doświadczeń z praktyki zawodowej w zakresie różnych poziomów monitorowania systemów wbudowanych. W tym procesie pomocne były również wyniki wcześniejszych prac prowadzonych w Instytucie Informatyki dotyczące systemów wbudowanych [7] [8], analizy logów wydajnościowych i zdarzeniowych [9] [10] [11] [12] [13] [14].

Pierwszy z prezentowanych tematów badawczych dotyczył monitorowania systemu z wykorzystaniem niskopoziomowego śladu instrukcji wykonania procesora. Jednym z wniosków z badań była potrzeba zweryfikowania pomysłów monitorowania systemu korzystając z danych na innych poziomach abstrakcji, które w sposób pośredni lub bezpośredni generuje analizowany system. Podczas praktyki zawodowej autor pracował nad optymalizacją poboru energii przez urządzenie medyczne zasilane bateryjnie. Znajomość problematyki poboru energii przez urządzenia o twardych wymaganiach energetycznych wraz z wnioskami z poprzednich badań umożliwiły wyznaczenie nowych celów badawczych w formie: analizy systemów z wykorzystaniem szeregów czasowych sygnałów (m.in. poboru prądu) oraz korelacji wyników analizy z logami zdarzeniowymi. W ramach badań powstały algorytmy i systemy, które w dużym stopniu się uzupełniają i pozwalają na wielopoziomowe monitorowanie systemów wbudowanych.

1.2 Przegląd literatury

Podczas rozwoju i utrzymania oprogramowania zespół deweloperski przeznacza nawet 25-40% czasu i zasobów na prace związane z testowaniem, weryfikacją i walidacją tworzonego oprogramowania [15]. Koszty te znacząco wzrastają przy systemach o wysokiej niezawodności. Do weryfikacji i analizy funkcji projektowanego oprogramowania stosuje się różne metody monitorowania systemu. Metody te mogą stanowić narzędzia zarówno ułatwiające procesy testowania, weryfikacji lub walidacji jak i poprawiające jakość wyników będących rezultatem tych procesów [16] [17] [18] [19].

Monitorowanie programowe może skupiać się na obserwacjach całych systemów i poszczególnych komponentów jak np. dyski twarde [9]. Pracę systemów wbudowanych można monitorować na różnych poziomach. W przeprowadzonych badaniach rozpatrzono trzy poziomy monitorowania: 1) monitorowanie przebiegu wykonania instrukcji (tzw. ślad instrukcji), 2) monitorowanie sygnałowe (obejmujące wybrane sygnały chrakteryzujące wydajność przetwarzania, efektywność sterowania, itp.) oraz 3) monitorowanie zdarzeń (logi zdarzeniowe). Monitorowanie śladów instrukcji jest zwykle realizowane ręcznie z ewentualnym wsparciem specjalizowanych narzędzi (rozdz. 1.1). Tematyka ta jest często traktowana powierzchownie w literaturze, stąd też możliwości tego podejścia wymagały przebadania i rozwinięcia (rozdz. 2). Ślad instrukcji jest wykorzystywany w metodach detekcji wykonania złośliwego oprogramowania opisanych w [20] [21]. W [22] autorzy używają sieci neuronowej do detekcji anomalii na podstawie śladu instrukcji. W literaturze opisane są rozwiązania techniczne ułatwiające agregacje śladu instrukcji z różnych źródeł [23] [24]. Dane

z monitorowania sygnałowego mogą przyjąć formę szeregów czasowych wartości próbek pochodzących z różnych źródeł [25]. Analiza szeregów czasowych jest bardzo obiecująca i szeroko opisywana w literaturze, stąd celowe było przeprowadzenie przeglądu proponowanych metod i zakresu ich zastosowań w celu ewentualnego wykorzystania w planowanych badaniach.

Na podstawie zebranych danych tworzone są profile opisujące zachowanie systemu. Literatura tematu proponuje wiele metod analiz bazujących na różnych źródłach przedstawiając realne zastosowania w konkretnych aplikacjach takich jak analiza EKG [26] [27], rozpoznawanie aktywności ludzkiej [28], analiza poboru mocy [29] [30] [31], ekonometria, bioinformatyka, analiza ruchu drogowego [32], a także kontrola jakości [33]. Z przykładów aplikacji w literaturze można wyszczególnić trzy główne typy zadań dla opisywanego monitorowania: a) detekcja anomalii np. [34], b) tworzenie profili behawioralnych systemów [11] i c) predykcja przyszłych zachowań [35]. Detekcja anomalii w szeregach czasowych jest zwykle określana poprzez amplitudę oraz charakterystykę tymczasowego kształtu przebiegu sygnału [36]. Bardziej zaawansowane metody wykorzystują podział na segmenty [37] jak i detekcję sekwencji symboli [38]. Symbole są generowane poprzez segmentacje szeregu czasowego i określenia ich parametrów, a następnie wykorzystanie różnych metod grupowania lub porównywania. Sekwencje symboli tworzą maszynę stanową, która definiuje model behawioralny wykorzystywany następnie np. do rozpoznawania mowy. Innymi aplikacjami wykorzystującymi sekwencje symboli są kontrola jakości, fintech [39], ekonometria. W tych aplikacjach ważna jest detekcja anomalii w formie wykrywania grup lub punktów zmian. W [36] przedstawiona jest technika dekompozycji szeregów czasowych oraz redukcji zbioru danych w celu wykrywania punktów nagłych zmian. Metody dekompozycji i przetwarzania danych z szeregów czasowych mogą być dopasowywane do cech charakterystycznych dla źródeł pochodzenia sygnału np. EKG [40] [41]. Ten ostatni jest bogato opisany w literaturze. Ze względu na specyfikę i ustandaryzowane metody opisu fragmentów sygnału EKG (puls, załamki QRS) zaproponowane zostały liczne metody do automatyzacji procesu analizy EKG na podstawie zależności czasowych i amplitudowych. W [42] sześć załamków PQRSTU jest oznaczane jako segmenty z wykorzystaniem algorytmów uczenia maszynowego w celu znalezienia arytmii i innych anormalnych symptomów.

Innym podejściem proponowanym w literaturze jest oznaczanie trendów oraz periodycznych właściwości [43]. W [44] zaproponowane zostały różne metody dekompozycji szeregu czasowego opartego na dziedzinie spektralnej np. transformata Fouriera lub falkowa. Transformaty te dekomponują sygnał do formy współczynników liczbowych.

Wykrywanie fragmentów periodycznych, regularnych, a także trendów zaproponowane zostało w [45] dla aplikacji predykcji pogody. Model dzieli zdekomponowany sygnał na część liniową (wykorzystując model MapReduce oparty o ARIMA- ang. autoregressive integrated moving average) oraz nieliniową opartą na modelu M-KNN (ang. Modified K-Nearest Neighbor). Właściwości sygnału mogą być określane na podstawie algorytmów bazujących na modelu okna przesuwnego [46]. Takie podejście wymaga zastosowania metod grupowania. Przegląd różnych metod grupowania wykorzystywanych na szeregach czasowych w celu wydobycia wiedzy zawarty jest w [47] (porównane są m.in. algorytmy k-Means, k-Nedoids, PAM, CLARA). W artykule [35] wykorzystywany jest autoregresyjny model umożliwiający przewidywanie kolejnych wartości próbek w szeregu czasowym. Predykcje te są podstawą do wykrywania anomalii w obserwowanym szeregu czasowym. Wykrywanie anomalii w szeregu czasowym może być również realizowane z wykorzystaniem uczenia maszynowego np. za pomocą głębokich sieci neuronowych [48] [49] [50].

Metody oparte na algorytmach klasyfikacji implementują metryki podobieństwa między szeregami czasowymi. Metryka umożliwia ustalenie stopnia relacji podobieństwa pomiędzy różnymi fragmentami szeregów czasowych. Rozróżniane są trzy kategorie metryk podobieństwa: a) w domenie czasu (oparte na np. wartości korelacji), b) zmiany sygnału (oparte na autokorelacji), c) kształtu sygnału [51]. W przypadku klasyfikacji opartej na metryce kształtu sygnału często stosowane są struktury typu shapelet. Shapelet jest podciągiem szeregu czasowego, który jest reprezentatywny dla danej klasy kształtów szeregów czasowych [52]. Metryka podobieństwa używana jest jako narzędzie dyskryminacyjne podczas klasyfikacji szeregów czasowych. Inne metryki podobieństwa mogą być oparte na wyznaczonych i obliczonych cechach bazując na korelacji w czasie, parametrach zmian sygnału czy też kształcie sygnału. W [53] autorzy proponują algorytm wyznaczania wektora cech na podstawie szeregu czasowego. Następnie wektor cech stanowi bazę dla algorytmu uczenia maszynowego dedykowanego klasyfikacji szeregów. Klasyfikacja szeregów czasowych znajduje zastosowanie w aplikacjach takich jak rozpoznawanie profili poboru energii w celu predykcji zapotrzebowania na energię [54], detekcji arytmii serca lub też w celach odkrywania wiedzy [55]. W [55] przedstawiona jest metodyka klasyfikacji szeregów czasowych bazująca na cechach takich jak: pochodne, całki oznaczone czy widmo spektralne.

Zagadnieniem poruszanym w literaturze związanej z monitorowaniem systemów komputerowych jest problematyka doboru scenariuszy testowych. Scenariusze determinują parametry działania obserwowanego systemu, w szczególności określają rodzaj i wzmocnienie generowanego obciążenia [9]. Kolejnym parametrem scenariuszy testowych jest możliwość

ich automatyzacji opisana między innymi w [56]. Scenariusze testowe mają wpływ na jakość wyników testów (w tym i monitorowania oprogramowania) wykorzystujących techniki wstrzykiwania uszkodzeń (ang. fault injection) [7]. Wstrzykiwanie uszkodzeń [8] jest techniką wykorzystywaną m.in. do weryfikacji zaimplementowanych technik tolerowania uszkodzeń (ang. fault tolerance) [4] [57] oraz określania odporności systemu na błędy.

Ważnym aspektem, który należy uwzględnić przy analizie szeregów czasowych generowanych przez systemy wbudowane jest charakterystyka systemów wbudowanych. Analiza systemów wbudowanych poprzez literaturę jak i też bazując na doświadczeniu zawodowym autora pozwala na sklasyfikowanie szeregów czasowych sygnałów z uwzględnieniem modelu behawioralnego systemów wbudowanych. Szeregi czasowe sygnałów generowanych przez systemy wbudowane są przypisywane do jednej z 4 klas: 1) proste okresowe (ang. simple regular time series- SRT), 2) złożone okresowe (ang. composed regular time series- CRT), 3) nieokresowe (ang. irregular time series-IRT), niedeterministyczne (ang. indeterministic time series- IDT). Sygnały typu SRT i CRT odróżnia poziom złożoności związany z atrybutami sygnału np. kształtem. Przykładem SRT może byś pojedynczy impuls EKG. Systemy wbudowane często charakteryzują się profilem zachowania, który można podzielić na czynności wykonywane regularnie okresowo oraz na czynności będące reakcją na nieregularne zdarzenia zewnętrzne. Przykładem systemu wbudowanego wpisującego się przedstawiony model może być terminal monitoringu floty (terminal GPRS/GPS), który w trybie domyślnym jest uśpiony, a wybudza się okresowo w celu określenia pozycji i wysłania jej do centrum monitoringu. Ponadto system reaguje na różne zdarzenia zewnętrzne jak np. wykrycie zmiany stanu linii zapłonu. Szereg czasowy sygnału będącego pochodną zachowania systemu wbudowanego może odpowiadać jego profilowi behawioralnemu. Z tego powodu taki szereg czasowy zawiera zarówno fragmenty typu SRT/CRT jak i IRT/IDT. Sygnały typu IRT występują nieregularnie, jednak ich kształt i parametry są związane ze zdarzeniem, które spowodowało wygenerowanie danego podciągu szeregu czasowego. Sygnały, których atrybuty związane są z profilem wykonania systemu wbudowanego, odwzorowują zarówno stany systemu wbudowanego określane jako regularne będące modelowo wynikiem działania procesów tła jak i nieregularne- reakcyjne względem zdarzeń zewnętrznych.

Ze względu na złożoność wykonywanych procesów przez systemy wbudowane szeregi czasowe (np. zagregowanych próbek chwilowego poboru prądu) mogą przyjmować charakterystyki, które z bliskiej perspektywy wyglądają na nieregularne i niedeterministyczne- niezwiązane z działaniem systemu. W celu analizy takiego sygnału dane są agregowane i tworzony jest hierarchiczny model opisujący szereg czasowy reprezentujący zachowanie

systemu wbudowanego. Przykład takiego podejścia jest zawarty w [11], w którym autorzy analizują anomalie generowane przez złożone systemy serwerowe. Algorytmy z [11] przygotowane są dla szeregów czasowych okresowych i nie uwzględniają charakterystyki systemów wbudowanych. W przypadku systemów wbudowanych przy rozważaniach rozwiązań analizy sygnałów należy uwzględnić ograniczenia w zasobach charakterystyczne dla tego typu systemów. Ta cecha jest w szczególności ważna dla systemów mobilnych, które polegają na zmagazynowanej energii w baterii [58] [59] [29]. W [60] przedstawiona jest analiza profilów poboru energii systemów opartych o mikrokontroler. Na tworzony profil energetyczny wpływ miały takie czynniki jak częstotliwość sygnałów zegarowych, częstotliwość próbkowania, wykorzystanie FPU, itp. W [30] zaproponowano inne podejście do analizy poboru prądu bazujące na trzech blokach funkcjonalnych: akwizycji danych, przetwarzania danych oraz komunikacji. Metoda ta umożliwia estymację poziomu prądu w zależności od różnych wartości parametrów technicznych powyższych procesów. Pozyskanie odpowiednich szeregów czasowych danych w celu analizy poboru prądu wymaga instalacji w badanym systemie specjalnych komponentów pomiarowych, których zastosowanie w rzeczywistym komercyjnym systemie byłoby wymagające i może mieć wpływ na ostateczne wyniki.

Zaawansowane analizy praktycznych problemów wymagają także wykorzystania innych źródeł informacji podczas monitorowania np. logów zdarzeniowych [61] [62]. Większość literatury poświęconej analizie logów zdarzeniowych zgłębia tematykę przetwarzania logów [63], ich klasyfikacji i detekcji sytuacji wyjątkowych [64]. Wykorzystywanie wielu źródeł danych różnego typu stwarza wymóg ich korelacji np. w dziedzinie czasu. Cechą systemów wbudowanych związaną z ograniczonymi zasobami sprzętowymi, jest niemożliwa lub ograniczona synchronizacja czasu zegarów ze światem zewnętrznym. Z tego powodu temat korelacji logów generowanych przez system wbudowany z szeregami czasowymi jest rozważany w pracy. W wielu artykułach opisane są różne metody korelacji między szeregami czasowymi [65] [66]. Bardziej zaawansowane metody korelacji między szeregami czasowymi wykrywające nawet bardzo niewielkie różnice w czasie są opisane w [67] [68]. Klasyczne podejście zakłada wykorzystanie metryki korelacyjnej Pearson [69]. Inne metody oparte są na wyznaczaniu wartości średnich czy trendów szeregów czasowych oraz określeniu metryki podobieństwa na potrzeby klasyfikacji [70]. Cześć źródeł skupia się na metodach ekstrakcji cech charakterystycznych oraz wykrytych anomaliach odpowiednio: 1) z logów [71] lub 2) z szeregów czasowych [72] w celu dokonania dopasowania. Metody korelacji można wykorzystywać także w celu wykrycia okresowości zdarzeń/sygnału oraz wartości tego

okresu [73] [74]. Problem dopasowania szeregów czasowych do innej skali czasowej jest rozważany w [75]. Dopasowanie wspierane jest poprzez zdefiniowanie algebry operacji na szeregach czasowych. Między innymi zdefiniowany jest operator pochodnej generujący rodzinę szeregów czasowych przesuniętych o wszystkie możliwe wartości przesunięcia w ramach zadanej dziedziny. Artykuł nie przedstawia metod wykonywania operacji, natomiast proponuje wysokopoziomowy język zapytań korzystający z zaproponowanych operatorów. Przykład metod dekompozycji szeregów czasowych wraz z korelacją zależną od skal czasowych (wykorzystując metodę korelacji Pearson) jest opisywany w [76]. Metodyka przedstawiona w artykule aplikowalna jest dla danych, które są szeregami czasowymi, co wyklucza zastosowanie w problematyce rozważanej w tym paragrafie. W [77] zaproponowana została metoda wykrywania skokowej zmiany korelacji wielowymiarowych sygnałów. Metoda może mieć zastosowanie przy wykrywaniu anomalii względem innego szeregu czasowego. W raportach [12] [11] zawarta jest problematyka korelacji logów zdarzeniowych oparta o znajdywanie kolejnych wystąpień różnych typów zdarzeń. W artykule [78] opisana jest metoda wspomagająca korelacje między rekordami logów w celu eliminacji nadmiarowych rekordów (szumu lub powtórzeń). Przegląd literatury wykazał, że problem korelacji dwóch różnych typów danych (jakie tworzą szereg: czasowy sygnału oraz zbiór rekordów logów) jest rzadko poruszany.

W [79] autorzy proponują metodę korelacji sekwencji zdarzeń z szeregiem czasowym. Metoda oparta jest na trzech procesach: 1) wykryciu wystąpień zależności między zdarzeniem a fragmentem szeregu czasowego, 2) określeniu kolejności zależności w celu wyznaczenia kierunku korelacji oraz 3) wykryciu monotoniczności efektu powstałego w wyniku wystąpienia przyczyny- np. im większe dane wejściowe programu generującego zdarzenie tym dłuższe bądź większe obciążenie CPU będącego źródłem szeregu czasowego. Procesy te zostały opracowane z wykorzystaniem algorytmów odkrywania wiedzy. Algorytmy zakładają, że zegary używane do znakowania zdarzeń jak i szeregów czasowych pozwalają na poprawne określenie monotoniczności pomiędzy próbkami, a wystąpieniami zdarzeń. Inna metoda bazująca na tym samym założeniu jest przedstawiona w [80]. W [81] przedstawione jest narzędzie do interaktywnego wyszukiwania skorelowanych zdarzeń z punktami szeregu czasowego. Artykuł [82] opisuje przypadek korelacji szeregów czasowych związanych z danymi meteorologicznymi o kryzysowych zdarzeniach pogodowych. W tym przypadku analiza czasowa jest uproszczona ze względu na powolne zmiany monitorowanych sygnałów w porównaniu ze zjawiskami zachodzącymi w typowych urządzeniach wbudowanych. W literaturze można znaleźć wiele algorytmów synchronizacji czasów między dwoma jak

i wieloma węzłami. Algorytmy te oparte są o wymianę wiadomości synchronizacyjnych [83] lub o schematy kompensacji czasu [84]. Przedstawiona powyżej literatura nie porusza jednak problematyki, w której szereg czasowy sygnału jak i rekordy zdarzeń oznaczane są innymi i niezsynchronizowanymi zegarami. Ponadto ograniczenia zasobów systemów wbudowanych wraz z funkcją agregowanych sygnałów uniemożliwiają zastosowanie znanych algorytmów wykorzystujących wstrzykiwane markery do rekordów logów i sygnałów.

W literaturze modele klasyfikacji i dekompozycji szeregów czasowych były prezentowane w odniesieniu do dobrze określonych własności szeregów wynikających z ich specyficznych zastosowań np. przebiegi EKG. Brakuje modeli oraz algorytmów identyfikujących charakterystyczne segmenty szeregów czasowych oraz ich relacji na wyższym poziomie (struktury hierarchiczne). Do tego nadają się modele obiektowe nierozpatrywane w literaturze, które autor rozwinął w pracy. Takie modele stwarzają również większe możliwości badań korelacyjnych

1.3 Teza i cel pracy

Na postawie doświadczenia praktycznego oraz analizy literatury sformułowałem następującą tezę pracy:

Analiza i monitorowanie pracy systemów wbudowanych wymaga stworzenia odpowiednich modeli dekompozycji, agregacji i korelacji danych z procesów monitorowania.

Zakres monitorowania określony został na wszystkie warstwy stosu oprogramowania systemów wbudowanych od warstwy instrukcji sprzętowych (procesora) do warstwy aplikacyjnej. Efektywna analiza zebranych danych wymaga stworzenia odpowiednich ich modeli poprzez wyodrębnienie obiektów o różnym stopniu agregacji. Biorąc pod uwagę różne źródła monitorowanych danych celowe jest badanie ich korelacji. W pracy wyróżniono następujące cele badawcze:

- Identyfikacja problemów testowania i weryfikacji systemów wbudowanych oraz ich powiązanie z różnymi perspektywami obserwacji (monitorowania).
- Analiza specyfiki monitorowanych danych oraz opracowanie obiektowych modeli ich reprezentacji.
- Opracowanie algorytmów eksploracji oraz korelacji danych bazujących na stworzonych modelach.
- Opracowanie narzędzi wspomagających tworzenie modeli i ich analizę.

- Weryfikacja opracowanej metodyki badań na przykładzie zrealizowanych systemów wbudowanych.

Sformułowanie problemu badawczego jest wynikiem analizy doświadczeń praktycznych autora zdobytych podczas projektowania, testowania i rozwijania specjalizowanych systemów wbudowanych wdrożonych do produkcji. W procesach tych coraz większego znaczenia nabierały techniki monitorowania bazujące na wewnętrznych mechanizmach programowych oraz zewnętrznych urządzeniach. Niestety klasyczne analizy tak zbieranych danych nie dawały zadowalających wyników. Na przykład, podczas optymalizacji energetycznej holtera EKG, w celu pozyskania informacji o stanie różnych komponentów programowych systemu wykorzystano podsystem jądra Linux: *FTrace*. Agregacja dużej liczby danych z systemu *FTrace* miała wpływ na wyniki pomiarów poboru prądu. Z kolei analiza wykresu poboru energii wymagała każdorazowej ręcznej analizy porównawczej wraz z wygenerowanymi logami systemowymi. Potrzeba bardziej efektywnego rozwiązania zaowocowała opracowaniem nowego podejścia obejmującego różne perspektywy monitorowania i wielopoziomową analizę zgodnie z opracowanymi oryginalnymi modelami danych. Praca łączy w sobie aspekty teoretyczne i wdrożeniowe, te ostatnie dotyczą opracowania specjalizowanych narzędzi programowych.

W klasycznym podejściu do testowania i usuwania defektów oprogramowania, deweloper analizuje zachowanie systemu poprzez obserwacje zachowania systemu z perspektywy zewnętrznej (testowanie czarnej skrzynki ang. black box) np. poprzez agregacje wyników działania systemu albo wewnętrznej (testowanie białej skrzynki ang. white box) np. poprzez użycie sprzętowego lub programowego narzędzia typu debugger. Następnie zbiór obserwacji pozwala utworzyć model opisujący stany i ich tranzycje dla różnych komponentów oprogramowania [85]. Model może przybrać bezpośrednią formę maszyny stanów lub inną pośrednią opisującą sytuacje/zachowanie za pomocą innych pojęć. Modele te są tworzone wprost w sposób formalny lub też są abstrakcyjnym bytem, którym posługuje się deweloper. W przypadku braku dokładnej, szczegółowej i drobnoziarnistej specyfikacji komponentów deweloper często dokonuje analizy porównawczej- powtarzając kilkukrotnie wykonanie oprogramowania w różnych warunkach i tworząc model z każdego przebiegu. Na podstawie utworzonych modeli lub analizy porównawczej określane są konkretne komponenty, a nawet fragmenty kodu zawierające defekty oprogramowania i powodujące anomalie w wykonaniu. Opisywana analiza może być realizowana na różnych poziomach abstrakcji oprogramowania. Współczesne systemy wbudowane posiadają wiele warstw abstrakcji, z których każda wyższa warstwa może realizować bardziej skomplikowaną logikę.

Wraz ze wzrostem złożoności logiki wrasta też złożoność tworzonych modeli. W przypadku wyszukiwania anomalii lub poszukiwania błędów przemijających/tymczasowych oraz błędów typu heisenbug opisywana metoda jest w praktyce pracochłonna oraz wymaga powtarzania całej procedury dla każdej iteracji. Pojęcie heisenbug jest definiowane jako błąd, który zmienia charakter obserwowanych awarii lub nawet przemija pod wpływem ingerencji w celu obserwacji np. zmian kodu niezwiązanych z komponentem generującym błędy [86]. Autor również doświadczał w swojej praktyce zawodowej wyzwań związanych z opisywaną problematyką. Na bazie doświadczeń opracowane zostały koncepcje algorytmów, które między innymi automatyzują w znaczącej części opisywany proces. Metody prezentowane w pracy wykorzystują dane pochodzące z monitorowania sygnałów zewnętrznych systemu oraz rekordów logów. Wykorzystanie tych źródeł nie generuje wymagania ingerencji w parametry czasowe wykonania i modyfikację kodu oprogramowania badanego systemu. Taka cecha pozwala także na zaaplikowanie proponowanych rozwiązań do usuwania błędów typu heisenbug.

Środkowym aspektem cyklu życia urządzenia/systemu wbudowanego jest jego produkcja i utrzymanie. W tych etapach monitorowanie stanu urządzenia jest narzędziem umożliwiającym weryfikację poprawności działania systemu jako części systemu kontroli jakości (np. test końcowy małoseryjnej linii produkcyjnej). Wielowymiarowe monitorowanie programowe również jest narzędziem pomocnym dla zespołów serwisowych, które na podstawie różnych profili zachowań badanych systemów mogą stwierdzać stan techniczny urządzeń bądź wskazywać obszary, które uległy awarii. Nieingerencyjne monitorowanie umożliwia wgląd w stan urządzenia na wielu poziomach abstrakcji, nawet w przypadku urządzeń, które uniemożliwiają zmianę oprogramowania ze względu na uszkodzenia lub też architekturę systemu.

Efektywna analiza tak zebranych danych wymaga opracowania odpowiednich modeli ich reprezentacji i algorytmów analizy, co stanowi główny cel pracy. W tym procesie istotnym elementem jest dobór odpowiednich/reprezentatywnych scenariuszy testowych, które zależą od aplikacji systemu, to zagadnienie jest poza zakresem pracy Pewne aspekty były poruszane we wcześniej prowadzonych badaniach w Instytucie np. [9] [56] [7]. Wnioski z opracowania i przetestowania algorytmu dla początkowego typu monitorowania wykazały, że niski poziom perspektywy analizy (ślad instrukcji) nie pozwala na proste powiązanie wyników z oprogramowaniem na wyższych poziomach abstrakcji. Wyniki prac nad początkowym monitorowaniem ukierunkowały prace nad drugim typem monitorowania, które wykorzystuje bardziej złożone algorytmy i pozwala na wielopoziomową, obiektową i hierarchiczną analizę

zależną od cech używanego szeregu czasowego. Monitorowanie oparte o szereg czasowy pozwala na wykrywanie różnych stanów, zdarzeń i anomalii. Natomiast opisanie i określenie pozostaje w zakresie prac użytkownika. Jeśli system wbudowany udostępnia logi tekstowe, to trzeci zbiór algorytmów umożliwia korelację logów z wynikami działania algorytmów drugiego typu monitorowania, nawet w przypadku braku synchronizacji między znacznikami czasowymi. W ten sposób monitorowania drugie i trzecie uzupełniają się, tworząc system, który nie tylko wykrywa zdarzenia, stany, anomalie, ale jest w stanie je opisywać odkrywając ich znaczenie.

1.4 Struktura pracy

W niniejszej pracy przedstawiony jest proces oraz rezultaty rozważań nad tezą i realizacją celów rozprawy doktorskiej. Dysertacja została podzielona na rozdziały. W kolejnych paragrafach scharakteryzowane są poszczególne rozdziały.

W rozdziale 2 przedstawiono wyniki badań nad możliwościami monitorowania systemów z wykorzystaniem sprzętowego śladu instrukcji. Pozwoliły one na opracowanie autorskiego algorytmu detekcji anomalii na podstawie wielokrotnego przebiegu kolejnych iteracji testów badanego systemu w kontrolowanych warunkach. Algorytm oparty jest na metodach klasyfikacji i grupowania oraz na autorskiej metryce określającej odległości między wykrytymi grupami. Algorytm został zaimplementowany w języku programowania C++ i przetestowany na prostych jak i złożonych (terminal GPS/GPRS) systemach wbudowanych. Wyniki zostały przedstawione i opisane. W opisie zawarto informacje o możliwościach jakie oferuje przedstawiane podejście do monitorowania systemów.

W rozdziale 3 zawarto wyniki prac nad autorskim algorytmem umożliwiającym wielopoziomowe monitorowanie systemów wbudowanych. Przedstawiony autorski algorytm obiektowej dekompozycji szeregów czasowych jest m.in. rezultatem doświadczeń autora z pracy nad komercyjnymi systemami wbudowanymi. Algorytm na podstawie szeregów czasowych sygnału tworzy obiektowy i hierarchiczny model danych reprezentujący zmiany stanu badanego urządzenia. W rozdziale przedstawiono doświadczenie autora pochodzące z pracy zawodowej w formie opisu problemów optymalizacji energii złożonego systemu wbudowanego. Rozdział przedstawia założenia algorytmu oraz prezentuje model danych wykorzystywany przez algorytm. W ramach modelu danych zaprezentowano autorską metodę oraz metryki pozwalające na porównywanie obiektów pochodzących z szeregu czasowego. Algorytm opisany jest za pomocą pseudokodu z podziałem na kilkanaście procedur. W rozdziale przeprowadzona jest dyskusja dobierania parametrów algorytmu, ocena

złożoności, a także przedstawienie właściwości algorytmu. Na koniec rozdziału przedstawiony jest przykład działania algorytmu na danych hipotetycznych wraz z ich wizualizacją.

Algorytm opisany w rozdziale 4 jest konsekwencją wyników badań nad algorytmem z rozdziału 3. Obiekty wykryte i wyeksponowane przez algorytm dekompozycji szeregów czasowych w części zastosowań wymagają dodatkowych adnotacji nadających znaczenie. Takimi adnotacjami są np. wybrane rekordy logów tekstowych. Algorytm zaprezentowany w rozdziale 4 pozwala na dopasowanie poprzez korelację logów tekstowych badanego urządzenia do wykrytych zdarzeń. Opisane, zgrupowane, odfiltrowane i sklasyfikowane rekordy logów są przypisywane do zdarzeń tworząc w ten sposób opisy tychże zdarzeń. W rozdziale przedstawiony został model danych generowany z logów tekstowych oraz metody korelowania logów ze zdarzeniami. Podczas wytwarzania modelu danych algorytm korzysta z autorskiej metryki porównującej rekordy logów tekstowych. Opisane zostały właściwości algorytmu, a także przedstawiono przykład działania algorytmu dla reprezentatywnego zbioru danych wejściowych.

Algorytmy zostały zaimplementowane w postaci dwóch rzeczywistych systemów monitorowania programowego. System *StateEventAnalyzer* został przetestowany podczas monitorowania rzeczywistych komercyjnych systemów wbudowanych (różnych typów holterów EKG). Wyniki przeprowadzonych analiz wraz z ich opisami zawarte są w kolejnych podrozdziałach rozdziału 5.

Na zakończenie, w rozdziale 6 podsumowano wykonaną pracę. Autor dokonał zbiorczej oceny zaprezentowanych algorytmów oraz systemów w kontekście tezy i celu rozprawy. Wskazane zostały także obszary możliwego rozwoju prac badawczych oraz usprawnień algorytmów.

Uzupełnieniem pracy są dwa dodane załączniki (rozdział 8), w których opisano opracowane narzędzia programowe wykorzystywane w badaniach śladów instrukcji oraz szeregów czasowych. Rozdział 8.1 zawiera prezentację autorskiego systemu *TraceAnalyzer* implementującego algorytm detekcji anomalii oparty o analizę śladu instrukcji wykonania. W rozdziale 8.2 przedstawiony jest autorski system *StateEventAnalyzer*, który m.in. implementuje opisane w rozdziałach 3 i 4 algorytmy. Rozdział opisuje system z perspektywy użytkownika przedstawiając wszystkie dostępne polecenia oraz przykłady ich wykorzystania wraz z wynikami pochodzącymi z operacji na rzeczywistych systemach wbudowanych. Rozdział skupia się również na architekturze oprogramowania prezentowanego systemu.

2 Analiza śladu instrukcji

Rozdział poświęcony jest autorskiemu algorytmowi, który wykorzystuje ślad instrukcji sprzętowych wykonywanych przez procesor. Ślad instrukcji jest sekwencją danych pozwalających odtworzyć kolejno wykonane instrukcje (np. poprzez adresy instrukcji). W rozdziale 2.2 opisany jest algorytm. Oczekiwanym rezultatem wykonania algorytmu są wykryte anomalie na podstawie analizy porównawczej śladu instrukcji z wielu wykonań badanego systemu w kontrolowanych warunkach. Przedstawiony w kolejnych podrozdziałach model działania algorytmu oparty jest o podział sekwencji śladu instrukcji oraz o grupowanie z wykorzystaniem autorskiej metryki podobieństwa grup instrukcji. Algorytm został zaimplementowany w formie rzeczywistego systemu *TraceAnalyzer* wykorzystującego ślad instrukcji generowany przez moduł sprzętowy ETM procesorów z rdzeniem CORTEX-M/A (załącznik 8.1). Algorytm został przetestowany i opisany na przykładzie badania systemu monitorowania floty- terminala GPS/GPRS (rozdział 2.3). Wnioski zawarte zostały w rozdziale 2.4.

2.1 Problemy monitorowania przebiegu wykonania instrukcji

Rozwój współczesnego oprogramowania systemów wbudowanych wymaga długotrwałego i wieloetapowego procesu usuwania błędów. W przypadku systemów wbudowanych na błędy oprogramowania podatne są w szczególności komponenty oprogramowania odpowiedzialne za konfigurację, kontrolę i obsługę sprzętu. Komponenty sprzętowe systemu zwykle modelowane są z poziomu oprogramowania jako nieawaryjne lub deterministycznie awaryjne urządzenia, które zachowują się zawsze zgodnie ze specyfikacją czy dokumentacją dostarczoną przez producenta. Doświadczenie komercyjne jak i literatura [23] [87] [88] wskazują, że w rzeczywistych systemach nieprzewidywalne czy też niedeterministyczne defekty sprzętowe występują powszechnie prowadząc do błędów oprogramowania. Powszechnie publikowane są noty katalogowe czy erraty[1][2][3] zawierające informacje o błędnym zachowaniu sprzętu i przypadkach, w których sprzęt realizuje zadania w sposób niezgodny z oryginalną dokumentacją. Defekty sprzętu mogą manifestować się w oprogramowaniu oraz w zachowaniu systemu lub mogą być maskowane poprzez właściwości implementacji oprogramowania. Przypadki, w których defekt objawia się zawsze

[1] https://www.st.com/resource/en/errata_sheet/es0206-stm32f427437-and-stm32f429439-line-limitations-stmicroelectronics.pdf [dostęp 01.04.2022]
[2] http://www.ti.com/lit/pdf/SPRZ318 [dostęp 01.04.2022]
[3] https://ww1.microchip.com/downloads/en/DeviceDoc/80000588K.pdf [dostęp 01.04.2022]

w formie błędu oprogramowania, są łatwiejsze do namierzenia i reprodukcji, a w związku z tym prawdopodobieństwo ich wyeliminowania podczas procesu tworzenia i testowania oprogramowania jest większe. Defekty należące do klasy defektów sprzętu, które są maskowane (nie objawiają się jako błąd oprogramowania) są znacznie trudniejsza do wykrycia. Ponadto na skomplikowanie i trudność przygotowania oprogramowania odpornego na defekty tej klasy wpływ mają inne czynniki takie jak: wielowarstwowość abstrakcji oprogramowania czy ograniczone informacje o stanie wykonania oprogramowania dostępne dla programisty. Każda warstwa abstrakcji dodaje własną obsługę błędów, prowadząc do zwiększenia prawdopodobieństwa niecelowego zamaskowania defektu. Kolejne zamaskowane błędy mogą skutkować stałą zmianą stanu oprogramowania prowadząc do awarii systemu. Podczas analizy poawaryjnej programista ma typowo dostęp tylko do stanu systemu w momencie wystąpienia awarii w postaci zrzutu pamięci i rejestrów. Dokładne zrozumienie bezpośrednich przyczyn awarii oraz wykrycie defektów wymaga poznania historii zmian stanu oprogramowania w określonym czasie przed wystąpieniem awarii.

Znanych jest kilka rodzajów dostępnych informacji o stanie systemu czy zdarzeniach sprzed awarii np. logi systemowe, liczniki sprzętowe i programowe czy też ślad stosu. Wykorzystanie tych metod wymaga uprzedniego przygotowania np. odpowiednio gęstego wstawienia instrukcji generowania logów do oprogramowania. Ponadto metody te zwykle pozwalają na wykrycie przyczyny awarii pod warunkiem, że jej źródło znajduje się w wyższych/pośrednich warstwach oprogramowania. Jeśli błąd wystąpi w niższych warstwach oprogramowania np. w sterowniku sprzętu w jądrze systemu, a manifestacja może być obserwowana dopiero w warstwie aplikacyjnej, to analiza stosu procesu nie wskaże bezpośredniej przyczyny i wymagana jest osobna analiza stosu wywołań na poziomie systemu operacyjnego. W sytuacji, gdy bezpośrednią przyczyną awarii jest niedeterministyczny i przemijający defekt sprzętu, zarówno logi, liczniki jak i ślad stosu mogą nie zawierać informacji niezbędnych do poznania mechanizmu prowadzącego do awarii. Opisywana klasa defektów nie powoduje natychmiastowej manifestacji w postaci błędów oprogramowania czy nawet awarii, ale modyfikuje stan systemu w stopniu, w którym oprogramowanie nie jest w stanie jej wykryć lub w którym defekt jest maskowany. Dopiero koincydencja tego typu defektów w połączeniu ze zdarzeniem, które wywoła odpowiedni fragment oprogramowania może skutkować awarią całego systemu. Przy opisywanej złożoności i mnogości czynników prowadzących do tego typu awarii, zadanie wykrycia koincydencji zdarzeń wymaga przeprowadzenia serii wielokrotnych testów w kontrolowanych warunkach oraz późniejszej

analizy kontroli przepływu wykonania oprogramowania. Celem takiej analizy jest wykrycie anomalii w wykonaniu oprogramowania.

Monitorowanie śladu wykonania jest jedną z metod wykorzystywaną podczas rozwoju oprogramowania w celu wykrywania i usuwania błędów. Ślad wykonania może mieć zastosowanie w ewaluacji i weryfikacji oprogramowania czy sprzętu. Monitorowanie śladu polega na agregacji oraz analizie instrukcji, które zostały wykonane na rdzeniu mikroprocesora lub mikrokontrolera. Ślad wykonania wraz z kodem maszynowym analizowanego oprogramowania umożliwia odtworzenie w czasie historii wykonania oprogramowania z dokładnością do pojedynczych instrukcji. Ślad można agregować w czasie rzeczywistym podczas wykonania oprogramowania. Ślad może być również zapisywany w buforze w pamięci operacyjnej systemu dostępnej w momencie wystąpienia awarii. Ślad wykonania w formie użytecznej może zajmować duży obszar pamięci. Uwzględniając fakt, że analiza przyczyn awarii potrzebuje odpowiednio dużego okna historii wykonania, agregacja wymaga interfejsów o dużej przepustowości lub poświęcenia pamięci operacyjnej oprogramowania. Współczesne mikrokontrolery udostępniają metody pozyskiwania śladu np. mikrokontrolery z rdzeniem CORTEX-M lub CORTEX-A umożliwiają monitorowanie wykonania zarówno w czasie rzeczywistym za pomocą interfejsu równoległego dla modułów ETM/ITM (ang. Embedded Trace Macrocell/Instruction Trace Macrocell) jak i zapis śladu do bufora cyklicznego w pamięci RAM/SRAM mikrokontrolera. Rozdział 8.1 prezentuje autorski system pozyskiwania śladu podczas testowania oprogramowania systemu wbudowanego oraz powiązania go z obrazem kodu maszynowego oprogramowania. System umożliwia również zautomatyzowaną analizę śladu poprzez autorski algorytm wykrywający anomalie w kontroli przepływu wykonania badanego systemu.

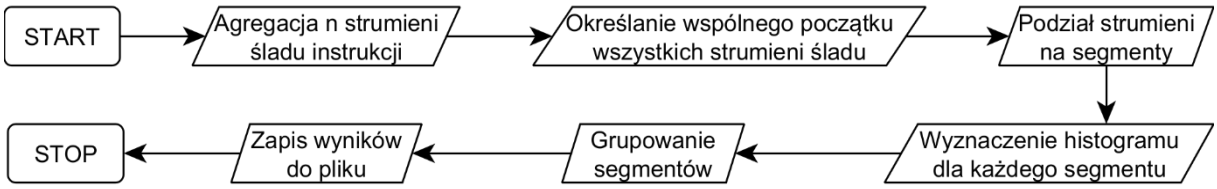
Ślad instrukcji pozwala na wykonywanie skomplikowanych analiz oprogramowania czy też sprzętu. Informacja o wykonanych instrukcjach jest pozyskiwana, przetwarzana oraz analizowana przez odpowiednie narzędzia. Dane te są natywnie generowane w formie skompresowanej i nieczytelnej dla człowieka. Ponadto nawet krótki ślad wykonania po dekompresji potrafi zajmować znaczący obszar pamięci. Istnieją narzędzia, które umożliwiają i ułatwiają konfigurację generowania śladu, jego zapisywanie a nawet przetwarzanie. Narzędziem typu open source jest interfejs *debugfs* udostępniany przez jądro systemu operacyjnego Linux. Interfejs *debugfs* pozwala na konfigurację generowania śladu ETM jak i ITM poprzez wystawienie do przestrzeni użytkownika pośrednio rejestrów konfiguracyjnych czy też portu ITM. Ponadto *debugfs* pozwala na odczyt ETB (ang. Embedded Trace Buffer) umożliwiając gromadzenie śladu instrukcji na badanej maszynie. Za pomocą

debugfs można odczytać ślad ETM, jednak aby go przetworzyć i zwizualizować potrzebne jest dodatkowe oprogramowanie. W celu agregacji śladu w czasie rzeczywistym lub śladu z badanego urządzenia za pomocą interfejsu TRACE należy skorzystać z urządzenia, które jest w stanie odbierać dane zgodne z ETM. Najczęściej taki sprzęt jest elementem wybranych emulatorów JTAG używanych standardowo do debugowania kodu. Komercyjne rozwiązania takie jak: Lauterbach PowerTrace wraz z Trace32 czy też Segger JLink Pro Trace wraz z oprogramowaniem pozwalają na agregację śladu, przetwarzanie, wizualizację czy też eksport danych do własnej analizy. Wizualizacja w tego typu narzędziach ogranicza się najczęściej do wskazania instrukcji na osi czasu, skorelowania instrukcji z kodem źródłowym oprogramowania lub wyznaczenia wybranych metryk. Bardziej zaawansowana analiza musi być zrealizowana przez inne rozwiązania na wyeksportowanych danych śladu instrukcji.

Opisane powyżej ograniczenia dostępnych rozwiązań w momencie tworzenia systemu do analizy śladu ETM oznaczają, że nie istnieje zintegrowane otwarte oprogramowanie oraz sprzęt pozwalające na: (1) agregacje śladu ETM, (2) dekompresję i wstępne przetwarzanie oraz (3) zaawansowaną analizę śladu instrukcji. Innym ważnym czynnikiem wpływającym na powstanie opisywanego systemu analizy śladu instrukcji jest wymaganie detekcji anomalii podczas testowania oprogramowania wbudowanego. Defekty oprogramowania i sprzętu mogą propagować się w postaci błędów przez wiele różnych modułów oprogramowania, generując awarie w znacznych odstępach czasowych od momentu natrafienia na faktyczny defekt. Wykrycie anomalii rozumianej w tym kontekście jako błędnego przepływu kontroli sterowania przez blok, który nie został zaplanowany przez programistę może ułatwić odnalezienie defektu i jego usunięcie. Wystąpienie niektórych defektów, może być maskowane przez pozostałe elementy oprogramowania/sprzętu. Część defektów nie generuje awarii pod wpływem większości danych wejściowych, jednak przy odpowiedniej koincydencji danych wejściowych i stanu systemu możliwe jest wystąpienie awarii systemu. Wczesna detekcja anomalii na poziomie instrukcji może wskazywać na fragmenty kodu podatne na tego typu defekty. System analizy śladu instrukcji może również wspomagać wykrywanie błędów przemijających, trudnych do odtworzenia w inny sposób niż regularne powtarzanie wykonania oprogramowania celem detekcji anomalii. Opisywana detekcja anomalii jest główną funkcją systemu przedstawionego w kolejnych podrozdziałach.

2.2 Algorytm detekcji anomalii

System analizy śladu instrukcji na podstawie wielokrotnej iteracji testu realizowanego w powtarzalnych i kontrolowanych warunkach przeprowadza detekcję anomalii na poziomie pojedynczych instrukcji. Rysunek 1 przedstawia sześć etapów algorytmu.



Rysunek 1. Struktura algorytmu detekcji anomalii

Wejście:
instr_streams - Tablica ciągów adresów kolejnych wykonywanych instrukcji. Każdy ciąg jest wygenerowany z pojedynczej iteracji testów.
Wyjście:
clustering_results - zbiór wykrytych anomalii w formie zgrupowanych segmentów ciągów instrukcji

```
1: function algorithm (instr_streams[1..N])
2:      streams = trim_streams(instr_streams, N, M)
3:      segments_bags = new SegmentsBags
4:      foreach (stream in  streams) do
5:          segments = new Segments(stream, S)
6:          index = 0
7:          foreach (segment in streams) do
8:              segment.calculate_histogram()
9:              segemnts_bags[index].add_segment(segment)
10:             index = index + 1
11:         end foreach
12:     end foreach
13:     clustering_results = new ClustersBags
14:     foreach (segemnts_bag in segments_bags.get_segments()) do
15:         clustering_results.add(DBSCAN(segemnts_bag, params))
16:     end foreach
17:     return clustering_results
18:end function
```

Algorytm 2-1. Algorytm detekcji anomalii

Pierwszym etapem jest agregacja śladu instrukcji z N iteracji testu. Z każdej iteracji testu wygenerowany zostaje jeden strumień śladu instrukcji. Strumienie (ciągi) instrukcji zbierane są w tablice strumieni *instr_streams*. Następnie wykonywane jest wstępne przetwarzanie na każdym ze strumieni, które polega na m.in znalezieniu identycznych fragmentów instrukcji z pierwszych M instrukcji (funkcja *trim_streams*). Etap ten pozwala na ewentualne usunięcie pierwszych instrukcji z każdego strumienia tak, aby jak najwięcej strumieni miało wspólny

początek. Systemy wbudowane podczas uruchamiania/restartu wykonują kod, który jest deterministyczny- najczęściej jest to inicjalizacja sprzętu i pamięci. Korzystając z tego faktu system analizy stara się dopasować początki każdego ze strumieni, zwiększając precyzję algorytmu w kolejnych krokach. Istnieje opcja konfiguracyjna algorytmu pozwalająca na usunięcie ze śladu wybranych fragmentów np. wszystkich funkcji systemu operacyjnego. Taka funkcja systemu może być wykorzystana w przypadku, gdy celem analizy jest badanie tylko warstwy aplikacyjnej oprogramowania. Po wstępnym przetwarzaniu algorytm dzieli każdy strumień na odpowiadające sobie segmenty o równej liczbie instrukcji (linie 4-12). Segment zawiera sekwencję adresów wykonanych instrukcji. Każdy segment posiada swój indeks, który jest rosnący wraz z każdym następującym segmentem. Segment o indeksie *x* w strumieniu *A* odpowiada segmentowi o indexie *x* w strumieniu *B* itd. Relacja ta oznacza, że instrukcje z obu segmentów wykonały się w podobnym czasie od rozpoczęcia iteracji, a także współdzielą podobny kontekst, gdyż algorytm zakłada, że każdy strumień śladu instrukcji został wygenerowany w identycznych warunkach. Po wykonaniu kroków 4-12 algorytm w zmiennej *segments_bags* przechowuje segmenty każdego ze strumieni dostępne pod kolejnymi indeksami. Każdy segment jest traktowany jako nieposortowany zbiór adresów instrukcji. Dla każdego segmentu jest wyznaczany histogram wykonanych instrukcji na przykład, dla segmentu o 8 adresach instrukcji: {0x01, 0x02, 0x03, 0x01, 0x03, 0x04, 0x05, 0x01} histogram przyjmie kształt: 0x01: 3, 0x03:2, 0x02:1, 0x04: 1, 0x05: 1. Wyliczone wartości histogramów umożliwiają wydajne wykonanie kolejnych etapów algorytmu.

W piątym etapie (linie 13-16) dla każdego zbioru segmentów o indeksie równym *x*, gdzie *x* = <1, N> wykonywany jest algorytm DBSCAN. DBSCAN jest algorytmem grupowania, który wykorzystuje miarę odległości między grupowanymi obiektami. W tym przypadku obiektami grupowanymi są segmenty. W ramach prac zaproponowana została metryka do pomiaru odległości między segmentami. Odległość między dwoma segmentami A i B zdefiniowana została jako moc zbioru będącego różnicą symetryczną zbiorów A i B. (równanie 2.1)

$$d(A,B) = |(A/B) \cup (B/A)| = |A| + |B| - 2|A \cap B| \qquad (2.1)$$

gdzie A i B są zbiorami liczb będących adresami instrukcji przypisanych do segmentów A i B o tym samym indeksie. Jeśli zdefiniujemy zbiór X jako zbiór zbiorów instrukcji, to metryka $d(A,B): X \times X \longrightarrow [0,+\infty)$ ponieważ $\forall A,B \in X;\ |A| \geq |A \cap B|\ i\ |B| \geq |A \cap B|$, a więc $d(A,B) \geq 0$.

Tak zdefiniowana funkcja spełnia wszystkie trzy warunki stanowienia metryki dla algorytmu DBSCAN:

1. Identyczność:

$$d(A,A) = |A|+|A|-2|A| = 0 \quad (2.2)$$

2. Symetria:

$$d(A,B) = d(B,A) \quad (2.3)$$

$$d(A,B) = |A| + |B| - 2|A \cap B| = |B| + |A| - 2|B \cap A| = d(B,A)\ bo \quad (2.4)$$

$$|B \cap A| = |A \cap B| \quad (2.5)$$

3. Nierówność trójkąta

$$d(A,B) \leq d(A,C) + d(B,C) \quad (2.6)$$

Załóżmy, że:

$$T\colon |A| + |B| - 2|A \cap B| \leq |A| + |C| - 2|A \cap C| + |B| + |C| - 2|B \cap C| \quad (2.7)$$

$$|A| + |B| - 2|A \cap B| \leq |A| + |C| - 2|A \cap C| + |B| + |C| - 2|B \cap C| \quad (2.8)$$

$$-2|A \cap B| \leq 2|C| - 2|A \cap C| - 2|B \cap C| \quad (2.9)$$

$$|A \cap C| + |B \cap C| \leq |C| + |A \cap B| \quad (2.10)$$

$$|A \cap C| = |(A \cap C)/B| + |A \cap B \cap C| \quad (2.11)$$

$$|(A \cap C)/B| + |A \cap B \cap C| + |B \cap C| \leq |C| + |A \cap B| \quad (2.12)$$

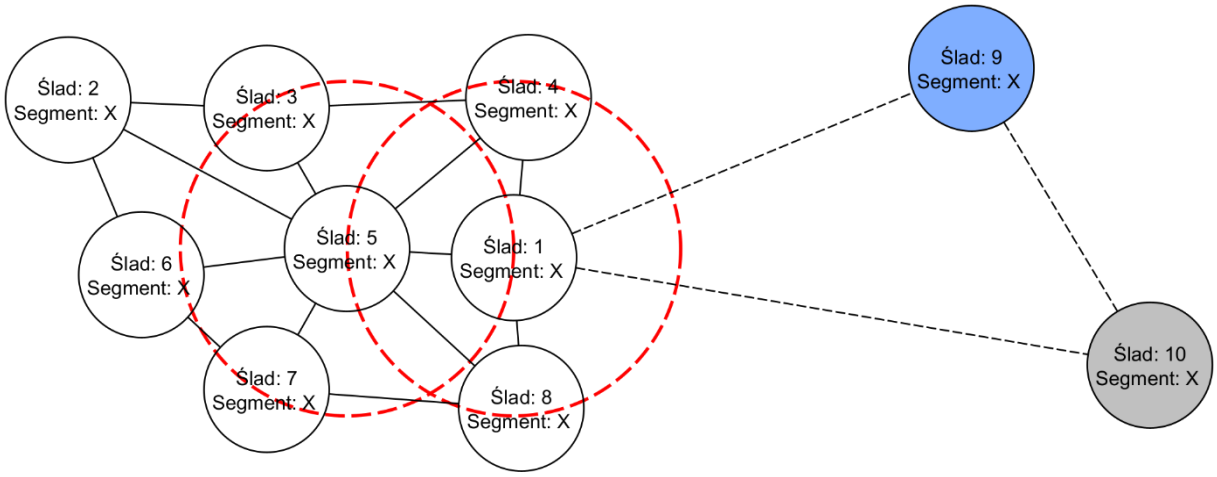
$$|(A \cap C)/B| + |B \cap C| \leq |C| + |A \cap B| - |A \cap B \cap C| \quad (2.13)$$

Wnioski z równań (2.7) do (2.13):

1. $|A \cap B| - |A \cap B \cap C| \geq 0, gdyż\ (A \cap B \cap C) \subseteq (A \cap B)$
2. $((A \cap C)/B) \cap (B \cap C) = A/B \cap C/B \cap (B \cap C) \subseteq \emptyset$
3. Z punktu 2 wynika, że $|(A \cap C)/B| + |B \cap C| \leq |C|$, gdyż oba lewostronne czynniki są wartościami mocy zbiorów rozłącznych i będących częścią zbioru C
4. Z punktów 1 i 3 wynika, że $\forall(A,B,C),\ |(A \cap C)/B| + |B \cap C| \leq |C| + |A \cap B| - |A \cap B \cap C|$ co potwierdza, tezę z (2.14). W przypadku założenia tezy przeciwnej wszystkie operacje są analogiczne (dodawanie i odejmowanie stronami czynników) i prowadzą do wniosku o fałszu tezy przeciwnej.

DBSCAN jest algorytmem grupowania bazującym na gęstości grupy [89] [90] [91]. Parametrami algorytmu są ε- promień dostępności- maksymalna odległość między obiektami, które należą do tego samego sąsiedztwa, MinPts- minimalna liczba elementów, która może utworzyć grupę. W opisywanym algorytmie elementami są segmenty traktowane jako zbiory adresów. Jeśli któryś z segmentów nie spełnia kryteriów przypisania do jednej z grup, jest on uznawany jako szum, a w kontekście algorytmu detekcji anomalii jako anomalia. Rysunek 2 przedstawia ilustrację działania algorytmu DBSCAN na 10 segmentach o indeksie X. Czerwone okręgi oznaczają wartości parametru ε. Segment z strumienia 9 i 10

(kolor niebieski i szary) są zbyt odległe, aby zostać sklasyfikowane jako członkowie grupy czerwonej. Segmenty zostaną oznaczone jako anomalie i zwrócone w pliku wynikowym.



Rysunek 2. Wizualizacja przykładu grupowania z wykorzystaniem algorytmu DBSCAN

Wynikiem działania algorytmu jest ślad instrukcji podzielony na odpowiadające sobie segmenty sklasyfikowane do oznaczonych numerami grup lub oznaczonych jako anomalie w obrębie segmentów o danym indeksie. Wyniki są zapisywane do pliku opisanego w załączniku 8.1. Rezultat działania algorytmu w dużym stopniu zależy od parametrów konfiguracyjnych: 1) rozmiaru segmentu, 2) promienia dostępności, 3) minimalnej liczby segmentów tworzących grupę. Wzrost wartości parametru rozmiaru segmentu skutkuje spadkiem prawdopodobieństwa wykrycia fałszywej anomalii, ale równocześnie powoduje, że relatywnie małe i krótkie anomalie mogą zostać niewykryte. Zbyt duża wartość promienia skutkować może sklejaniem się rozdzielonych grup ze sobą, a nawet klasyfikowaniu anomalii jako zwykłych segmentów. Wartość promienia może być zdefiniowana jako część względna rozmiaru segmentu. Rozmiar segmentu jest stałą wartością przez cały czas działania algorytmu. Ponadto rozmiar segmentu bezpośrednio wpływa na rząd wielkości odległości między segmentami mierzonymi zaproponowaną metryką.

2.3 Badania eksperymentalne

W celu przetestowania systemu wykorzystano system wbudowany: terminal GPS/GPRS. Terminal GPS/GPRS jest urządzeniem wykorzystywanym do monitoringu pozycji obiektów takich jak pojazdy naziemne czy wodne oraz transportowane towary. Ponadto w przypadku pojazdów terminal posiada funkcje monitorowania stanu linii sygnałowych generowanych przez obiekt jak i czytania stanów komponentów pojazdu za pośrednictwem magistrali CAN.

Zarówno pozycja jak i szeroko rozumiany stan pojazdu jest transmitowany na zdalny serwer za pośrednictwem sieci 2G. Testowany system bazuje na mikrokontrolerze z pojedynczym rdzeniem ARM Cortex-M3 z 128KB pamięci operacyjnej SRAM i 512KB nieulotnej pamięci *flash*. Terminal posiada modem GSM/GPRS umożliwiający transfer danych z wykorzystaniem protokołów TCP/IP na zdalny serwer. Badany system posiada także moduł GPS, który umożliwia określenie pozycji w formie współrzędnych geograficznych pojazdu na podstawie sygnału z satelitów GPS i GLONASS. Ponadto system wyposażony jest w akcelerometr i moduł wejścia/wyjścia do monitorowania stanu systemu. Oprogramowanie układowe bazuje na systemie operacyjnym czasu rzeczywistego *FreeRTOS* w wersji 6.0.1. W ramach oprogramowania system operacyjny zarządza czasem wykonywania pięciu wątków: (1) *GPS service,* (2) *NetworkConnection service,* (3) *Aggregation service,* (4) *Input/Output service,* (5) *Watchdog*. Wątek *GPS* regularnie odpytuje moduł GPS z żądaniem podania statusu i ewentualnej aktualnie dostępnej pozycji. Komunikacja realizowana jest z wykorzystaniem portu UART, tak samo jak w przypadku komunikacji z modemem GSM. Wątek *NetworkConnection* odpowiada za udostępnianie usług komunikacji przez sieć Internet. Wątek *Aggregation* jest odpowiedzialny za: fuzję wszystkich danych z czujników oraz stanu samego urządzenia, opakowanie danych zgodnie z protokołem komunikacyjnym, przygotowanie ich do wysłania przez sieć Internet, a także zapisanie ich w zewnętrznej pamięci *nand* (archiwum). *Input/Output* jest wątkiem, w którym realizowana jest obsługa synchroniczna i asynchroniczna odczytu stanu czujników i sygnałów wejścia oraz wyjścia m.in: odczyt wartości przeciążeń z akcelerometru za pośrednictwem magistrali I2C, analiza stanu linii wejściowych, czy też wysterowanie sygnałów wyjściowych zgodnie ze specyfikacją. Ostatni wątek nadzoruje stabilność systemu sprawdzając okresowo responsywność każdego z wątków oraz obsługując sprzętowy układ *watchdog*.

Terminal GPS/GPRS pracuje w dwóch trybach poboru energii: a) aktywny- włączany, gdy sygnał włączenia zapłonu pojazdu jest w stanie wysokim, b) obniżonego poboru energii- włączany, gdy linia sygnałowa stanu stacyjki przechodzi w stan niski. W stanie obniżonego poboru energii wyłączonych jest wiele urządzeń peryferyjnych takich jak modem 2G, pamięć *flash* itd. W tym stanie zmniejszona jest także częstotliwość próbkowania pozycji geograficznej oraz niezapisywane są rekordy z danymi, brak jest komunikacji z serwerem. W trybie obniżonego poboru energii system działa głównie reaktywnie reagując na zmiany sygnałów linii wejściowych, itp. Z powodu zmniejszonej liczby zadań w tym trybie mikrokontroler przebywa głównie w stanie uśpienia, aby zminimalizować pobór prądu. W stanie aktywnym, przy włączonym zapłonie oprogramowanie rzadziej wchodzi w stan uśpienia,

gdyż mikrokontroler realizuje zadania związane z: obsługą włączonego modemu, komunikacją ze zdalnym serwerem, agregacją danych z sensorów z podwyższoną częstotliwością i zapisem danych do pamięci NAND. Więcej szczegółów o badanym systemie zawarte jest w [4] oraz [6].

Na potrzeby weryfikacji działania algorytmu z rozdziału 2.2 przeprowadzone zostały trzy zestawy testów. Podczas każdego testu monitorowany był system GPS/GPRS opisany w paragrafach powyżej. Terminal wykonywał swój domyślny program. Każda z iteracji testu skupiała się na innych aspektach algorytmu jak i systemu detekcji anomalii (rozdział 8.1). Pierwszy test skupia się na stochastycznej analizie wydajnościowej poprzez próbkowanie wykonywanych procedur. Drugi test wyznacza liczbę instrukcji faktycznie wykonanych w ramach wykonywanego programu. Trzeci test wykorzystuje prezentowany algorytm w celu wykrycia anomalii zarówno prawdziwych jak i sztucznie generowanych.

W pierwszym teście system analizy śladu został skonfigurowany do przeprowadzenia pierwszej iteracji testu w celu wygenerowania profilu wydajnościowego. Jako emulator JTAG wykorzystany został J-Link Pro. W pliku konfiguracyjnym podane zostały skrypty, które włączają próbkowanie wartości rejestru licznika rozkazów (PC- ang. program counter) co 512 instrukcji. Odczyt śladu instrukcji realizowany jest przez analizator stanów logicznych Salea8 podłączony do linii TRACESWO mikrokontrolera. Symulator środowiska podłączony do badanego systemu jest skonfigurowany do utrzymywania linii sygnału zapłonu w stanie niskim przez czas trwania iteracji. Czas trwania iteracji został ustawiony na 45s. Po 45 sekundach system generuje profil wydajnościowy i zapisuje go do pliku wynikowego. Tabela 1 prezentuje profil wydajnościowy uzyskany podczas testu. Pierwsza kolumna zawiera nazwę funkcji, a druga procentowy udział próbkowanych instrukcji danej funkcji w ogóle wszystkich zagregowanych instrukcji.

Nazwa funkcji	Odsetek wystąpień [%]
vApplicationIdleHook	39.88692
FTFI_lowService	17.2696
IWDG_ReloadCounter	14.03463
prvIdleTask	12.90529
I2C_CheckEvent	7.87593
prvMMA75xxI2C_readBytes	4.78462
prvMMA75xxI2C_initializeTransaction	1.7938
memcmp	0.24678
SysTick_timersService	0.17177
xTaskIncrementTcik	0.12441
vTaskSwitchContext	0.10527
vApplicationTickHook	0.07793

Tabela 1. Profil wydajnościowy oprogramowania terminala GPS/GPRS

Profil wydajnościowy pokazuje, że system najwięcej czasu spędza na wykonywaniu kodu funkcji *vApplicationIdleHook*. Jest to funkcja wołana w momencie, gdy system operacyjny podczas szeregowania zadań wykrywa stan zawieszenia się wszystkich wątków w oczekiwaniu na zdarzenie zewnętrzne lub wewnętrzne. Funkcja ta wprowadza procesor w tryb uśpienia. Funkcja *vApplicationIdleHook* wywoływana jest przez *prvIdleTask*. Z profilu wydajnościowego można oszacować, że w trybie obniżonego poboru prądu system usypia się lub budzi się przez ponad 53% czasu procesora. Zasoby czasu procesora przeznaczane na obsługę systemu operacyjnego można wyliczyć sumując wartości histogramu dla funkcji odpowiedzialnych za obsługę przerwania *SysTick*: *SysTick_timersService, xTaskIncrementTick, vTaskSwitchContext, vApplicationTickHook*. System przeznacza ponad 0.4% czasu procesora na obsługę systemu operacyjnego, czyli realizację przerwania licznika czasu oraz przełączania kontekstu wątków w trybie obniżonego poboru prądu. Funkcje, które absorbują w sumie 32% czasu CPU (*IWDG_ReloadCounter*, *FTFI_lowService*), odpowiedzialne są za funkcje systemu związane z weryfikacją poprawności działania oprogramowania. Funkcje te są wywoływane z wątku o najniższym priorytecie i nie mają wpływu na wyszukiwanie funkcji generujących zbyt duże obciążenie, ponieważ każda inna funkcja wywłaszcza wątek *watchdog*. Pozostały czas CPU jest wykorzystywany

na obsługę komunikacji po interfejsie I2C oraz na odczyt wartości z akcelerometru (ok. 15%). Podczas pomiarów częstotliwość sygnału zegarowego rdzenia mikrokontrolera wynosiła 8MHz, a przerwania systemowe generowane były z częstotliwością 1KHz. Czas poświęcany na odczyt akcelerometru jest relatywnie duży. Wynika to ze sposobu implementacji funkcji odczytu. Odczyt nie jest realizowany przy użyciu przerwań dedykowanych przez mikrokontroler STM32f103 do interfejsu I2C, natomiast implementacja opiera się o ciągły odczyt w pętli rejestru zdarzeń I2C. Niemożność wykorzystania trybu przerwaniowego wynika z udokumentowanego błędu sprzętowego kontrolera I2C, który powoduje losowe pomijanie zdarzeń I2C, skutkując tym, że cała transmisja staje się nierzetelna, a czas jej zakończenia jest niedeterministyczny.

Drugi test został wykonany w trybie generowania przebiegu wykonanych instrukcji. Wszelkie parametry są identyczne jak w teście pierwszym z wyjątkiem skryptu OpenOCD, który włącza generowanie pełnego śladu instrukcji. Na podstawie śladu system wygenerował przebieg wykonania instrukcji znormalizowany do funkcji na poziomie języka C. W tabeli 2 przedstawiony jest wybrany krótki fragment, który został zarejestrowany podczas przełączania kontekstu wątków pod wypływem przerwania licznika systemu operacyjnego.

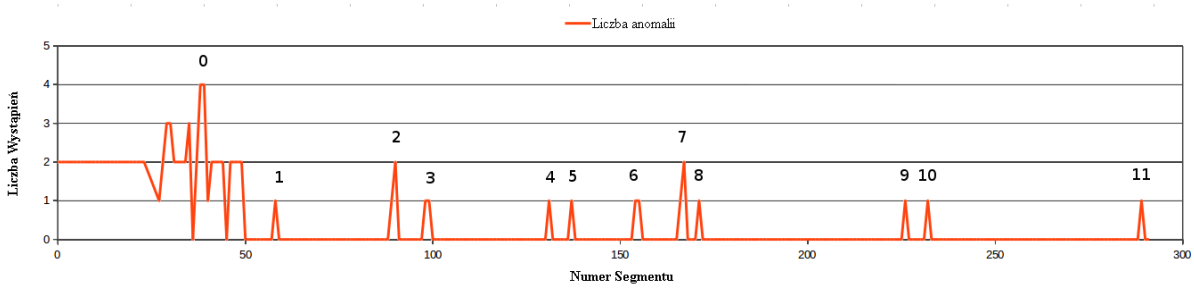
Nazwa funkcji	**Liczba wykonanych instrukcji**
SysTick_Handler	10
vApplicationSysTickHook	1
SysTick_timersService	68
xTaskIncrementTick	69
vListInsertEnd	22
xTaskIncrementTick	86
vPortClearInterruptMask	5
PendSV	13
vTaskSwitchContext	47

Tabela 2. Liczba instrukcji wykonanych przypadających na funkcję

W pierwszej kolumnie znajdują się nazwa funkcji. W drugiej kolumnie znajduje się liczba wykonanych instrukcji do momentu zmiany przepływu sterowania przez odpowiednią instrukcję np. instrukcje skoku. Systemowy proces reakcji na przerwanie zegarowe rozpoczyna się w funkcji *Systick*, której adres jest wpisany do wektora przerwań. W ramach tej funkcji wykonywane jest wstępne zachowanie kontekstu oraz wywołanie funkcji

vApplicationSysTickHook. Czynności te wymagają 10 instrukcji. *vApplicationSysTickHook* jest funkcją opakowującą dla *SysTick_timersService*- jej ciało nie zawiera ani preambuły, ani epilogu, tylko jedną instrukcję skoku. *SysTick_timersService* aktualizuje wartości liczników programowych przez 68 instrukcji, po czym przepływ sterowania trafia do funkcji *xTaskIncrementTick*, która inkrementuje licznik systemu operacyjnego oraz sprawdza czy system operacyjny powinien przełączyć kontekst na inny wątek. Jeśli istnieje wątek, który powinien zostać obudzony jest on dodawany do listy procesów do obudzenia za pomocą funkcji *vListInsertEnd*. Następnie, kontrola przepływu wraca do funkcji *xTaskIncrementTick*, która m.in przez 86 instrukcji przygotowuje i wywołuje przerwanie programowe. Na końcu wołana jest funkcja *vPortClearInterruptMask*, powodując odblokowanie przerwań i przepływ kontroli sterowania do funkcji obsługi przerwania programowego *PendSV*. *PendSV* woła funkcję *vTaskSwitchContext* realizującą przygotowanie wątku do wykonywania poprzez przywrócenie kontekstu wykonywania i skoku do następnika ostatniej wykonanej instrukcji. Przebieg wykonania instrukcji wskazuje, że w tym wypadku mikrokontroler potrzebował w sumie 242 instrukcji na przełączenie wątków.

W celu weryfikacji poprawności funkcjonowania algorytmu detekcji anomalii przeprowadzony został trzeci test. Podczas testu oprogramowanie terminala GPS/GPRS było uruchamiane w ramach jedenastu iteracji. Podczas każdej iteracji system agregował ślad wykonania instrukcji. Każdy ze śladów został podzielony na segmenty o rozmiarze 25 tys. adresów instrukcji. Scenariusz każdej iteracji rozpoczyna się od resetu oraz inicjalizacji oprogramowania terminala. Następnie terminal przez 44 sekundy funkcjonował podłączony do symulatora środowiska z interfejsem umożliwiającym zmianę stanu sygnału zapłonu w losowym momencie wraz z losową liczbą zmian. Podczas iteracji 1-9 stan linii zapłonu zachowywał stały poziom niski. Podczas dziesiątej iteracji stan zapłonu był modyfikowany cztery razy w krótkich odstępach czasu, w celu symulowania zakłóceń w sygnale. Iteracja jedenasta została uruchomiona na testowym terminalu GPS/GPRS z błędnie obsadzonym układem akcelerometru. Sprzęt ten na dokładnie tym samym oprogramowaniu powoduje przemijające błędy komunikacji I2C z akcelerometrem. Rysunek 3 prezentuje liczbę wykrytych anomalii w funkcji numeru segmentu. Tabela 3 zawiera dodatkowe informacje na temat wykrytych anomalii. Pierwsza kolumna zawiera numer anomalii. Dla każdej anomalii w drugiej kolumnie znajdują się numery segmentów, w których wykryte zostały anomalie. W trzeciej kolumnie wylistowane są numery iteracji, podczas których dana anomalia została wykryta. Informacje z rysunku 3 i tabeli 3 pozwalają określić dokładną różnicę, która spowodowała anomalie w przepływie sterowania.



Rysunek 3. Wykres liczby wykrytych anomalii w zależności od numeru segmentu

Numer anomalii	**Numer segmentu śladu**	**Numer iteracji testu, podczas której wykryto anomalię**
1	58	4
2	89-90	10;11
3	99-100	11
4	131	11
5	137	10
6	154-155	11
7	166-169	10;11
8	171	11
9	226	11
10	232	11
11	289	11

Tabela 3. Podsumowanie danych statystycznych wykrytych anomalii

Detekcja anomalii śladu została przeprowadzona z wartościami parametrów: promień dostępności równy 900 instrukcji i minimalna liczba punktów równa 3. Każde maksimum lokalne wykresu z rysunku 3 zostało oznaczone liczbą oznaczającą numer anomalii. Anomalia zerowa zawiera wiele maksimów w segmentach od 0 do 49. Podczas inicjalizacji sprzętu oraz oprogramowania, uruchamianie są wątki systemu operacyjnego, co skutkuje realizacją różnych etapów inicjalizacji w różnym czasie generując dużą liczbą anomalii w pierwszej fazie testu. Anomalia 1 została wykryta podczas iteracji 4. Dokładniejsza analiza śladu wykazała, że w wyniku wystąpienia błędu komunikacji I2C z akcelerometrem oraz uruchomienia mechanizmu detekcji przepełnienia stosu, segment zawierał znaczną liczbę instrukcji (ponad 2 tysiące) różną względem odpowiadających segmentów. Wszystkie pozostałe wykryte

anomalie powstały w wyniku zdarzeń zewnętrznych podczas iteracji 10 i 11 co potwierdza poprawność opisywanej metodologii i algorytmu. System był w stanie wykryć 3 z 4 wygenerowanych losowo anomalii podczas iteracji 10. Anomalie z iteracji 11 były spowodowane występowaniem błędów w komunikacji I2C.

2.4 Wnioski

Analiza śladu instrukcji pozwala na uzyskanie obszernych informacji o przepływie sterowania z dokładnością do pojedynczych instrukcji. Testy z użyciem śladu instrukcji umożliwiają detekcję błędów oprogramowania w rzeczywistym systemie wbudowanym. Systemy testowania oparte na danych ze śladu instrukcji ułatwiają testowanie implementacji mechanizmów tolerowania defektów, ze względu na możliwość obserwacji zmian kontroli przepływu. Opisywany system wykorzystuje ślad instrukcji w celu przeprowadzania czterech różnych testów, w tym testu opartego na autorskim algorytmie wykorzystującym metody grupowania do detekcji anomalii. Podczas korzystania z systemu bardzo ważnym aspektem jest odpowiedni dobór parametrów algorytmu. Na przykład, zwiększając promień dostępności możliwe byłoby zlikwidowanie wielu wykrytych fałszywych anomalii oznaczonych numerem 0 w przykładzie. Z drugiej strony w ten sposób większość anomalii podczas kolejnego etapu iteracji nie zostałoby wykryte. Manipulując wielkością promienia i minimalną liczbą tworzącą grupę możemy ustawiać poziom odróżniający faktyczną anomalię od np. przesuniętej w czasie obsługi okresowego zadania. Algorytm bazuje na obserwacji, że system wbudowany wykonuje okresowe zadania lub reaguje na zdarzenie zewnętrzne. Symulując i kontrolując środowisko zewnętrzne oraz wielokrotnie powtarzając testy, system umożliwia detekcję defektów przemijających, błędów typu heisenbug, defektów w mechanizmach synchronizacji. Ustalenie rozmiaru segmentu, na które podzielone są ślady instrukcji również ma wpływ na czułość algorytmu. W typowym scenariuszu niepożądane są takie zachowania systemu, że klasyfikują każde okresowe przerwanie od zegara systemu jako anomalię. Odpowiednio duży segment w ramach którego powinno znaleźć się kilka zgrupowań instrukcji odpowiedzialnych za obsługę danego przerwania, pozwala na zamaskowanie danego przerwania w taki sposób, aby stanowiło ono tło dla faktycznych anomalii.

3 Eksploracja szeregów czasowych

Systemy wbudowane, mimo iż często są dedykowane do realizacji jednej funkcji, podczas działania wykonują wiele różnych zadań. Diagramy stanów oprogramowania systemów wbudowanych mogą być złożone. Ponadto możliwość analizy zachowania takiego systemu bez ingerencji w oprogramowanie czy sprzęt systemu pozwala na lepsze zrozumienie systemu, a także wyższą jakość testów. Tworzenie profili behawioralnych systemu umożliwia wykrycie niepoprawnych i nieprzewidzianych zachowań. Jeśli system udostępnia szczegółowe logi z działania wielu podsystemów, powyższe cele są również osiągalne. Jednak systemy wbudowane często mają ograniczone zasoby sprzętowe i w szczególności w końcowych iteracjach rozwoju oprogramowania, logowanie jest ograniczane. Do pozyskiwania informacji o stanie wykonania można wykorzystać interfejs sprzętowy JTAG. Metoda ta zwykle wpływa na parametry dynamiczne wykonania oprogramowania, co może skutkować nierzeczywistymi wynikami analizy. Ponadto fizyczny dostęp do interfejsu w ostatecznych wersjach sprzętu jest często ograniczany. Rozdział 3.1 adresuje powyżej przedstawione wyzwania związane z analizą systemów wbudowanych. W tym celu zaprojektowane zostały algorytmy oraz system, który umożliwia analizę systemów bazując na szeregach czasowych sygnałów. Przykładem szeregu czasowego sygnału, który jest dostępny w każdym systemie wbudowanym oraz który charakteryzuje się właściwością korelacji z aktualnym stanem urządzenia jest szereg czasowy wartości próbek natężenia pobieranego prądu/pobieranej mocy przez badane urządzenie. Rozdział 3.2 poświęcony jest przykładowi systemu wbudowanego o skomplikowanej architekturze zarządzania energią oraz praktycznemu przypadkowi optymalizacji poboru energii. Opisywane algorytmy mają na celu przyśpieszenie wykrywania problemów opisanych w tym rozdziale, a także metodyczne podejście do zadań optymalizacji poboru energii. Przedstawiony przypadek optymalizacji był rzeczywistą motywacją dla stworzenia algorytmów opisywanych w tym rozdziale. W rozdziale 3.1 przedstawiony jest algorytm, który wykorzystuje szereg czasowy w celu wykrywania stanów urządzenia oraz ich grupowania i klasyfikacji.

3.1 Obiektowa analiza szeregu czasowego

Główną funkcją prezentowanego algorytmu jest rozpoznawanie obiektów na podstawie szeregu pomiarów oraz ich hierarchizacja. Wygenerowany hierarchiczny model obiektów pozwala na wnioskowanie o stanie urządzenie badanego systemu w zadanym momencie oraz o możliwych następnych stanach. W przypadku analizy sygnału, który jest szeregiem

czasowym próbek poboru energii, algorytm wykrywa stany poboru energii, w jakich przebywa urządzenie oraz stany przejściowe. Przy odpowiednio dużej liczbie danych wejściowych zebranych z urządzenia, które działało w środowisku symulującym każdy typ obciążenia, algorytm wygeneruje wszystkie możliwe stany energetyczne urządzenia. Zbiór stanów wraz z parametrami statystycznymi i dynamicznymi umożliwia analizę poboru energii urządzenia, a także detekcje anomalii i stanów generujących zawyżony pobór prądu. Na podstawie wykrytych stanów można także stworzyć graf przejść między stanami weryfikując poprawność algorytmów oszczędzania energii. Jeśli szereg wejściowy jest skorelowany z częścią logiki aplikacyjnej urządzenia, algorytm umożliwia weryfikację poprawności implementacji logiki aplikacyjnej systemu np. poprzez wykrycie anomalii w diagramie stanów. Ponadto informacja o czasie trwania stanu oraz znacznik czasowy przejścia do stanu pozwala dokładniej powiązać logi systemowe z wykresem czasowym poboru prądu.

3.1.1 Założenia oraz definicja modelu

Danymi wejściowymi algorytmu jest ciąg próbek $M_{cur}[1..N]$. Każda próbka zawiera pojedynczy pomiar analizowanego sygnału np. natężenia prądu pobieranego przez badane urządzenie wyrażonego w formacie liczby niecałkowitej. Przy stałym napięciu zasilania pobierany prąd jest wprost proporcjonalny do mocy. Kolejnymi parametrami wejściowym są: okres próbkowania oraz znacznik czasowy pierwszej próbki. Oprócz sekwencji próbek algorytm wymaga konfiguracji w postaci zestawu parametrów *params*: (1) *avg_eps*, (2) *deviation_eps*, (3) *min_eps*, (4) *max_eps*, (5) *rms_eps*. avg_eps jest to maksymalna wartość różnicy średnich arytmetycznych grupy segmentów i kandydata na członka grupy. Parametr *deviation_eps* jest to maksymalna wartość różnicy odchyleń standardowych grupy segmentów i kandydata na członka grupy, *min_eps* jest to maksymalna wartość różnicy wartości parametru minimum grupy segmentów i kandydata na członka grupy. Parametr *max_eps* jest to maksymalna różnica wartości maksimum grupy segmentów i kandydata na członka grupy. Parametr *rms_eps* jest to maksymalna wartość różnicy wartości średniokwadratowej sekwencji próbek grupy segmentów i kandydata na członka grupy. Parametry te mają bezpośredni wpływ na cechy jakościowe grupowania i klasyfikacji algorytmu. W zależności od typu sygnału prezentowany model przewiduje rozszerzenie parametrów z rodziny **_eps* o dodatkowe typy dla dodanych statystyk.

Po zakończeniu wykonywania algorytmu w S[1..X] znajdują się tablica obiektów reprezentujących wykryte stany. Każdy z obiektów zawiera listę grup, które są reprezentantami

danego stanu. Każda z grup posiada zbiór parametrów statystycznych ją określających oraz oznaczenie stanu, do którego przynależy. Drugim wynikiem jest obiekt grafu w postaci wektora węzłów/stanów. Każdy z elementów zawiera listę węzłów (stanów), które wraz z węzłem początkowym tworzą możliwe tranzycje. Trzecim wynikiem jest zbiór klas cykli. Każda klasa cykli zawiera informacje o wszystkich jej wystąpieniach (cyklach) w ramach sekwencji próbek analizowanego sygnału.

Analiza szeregu czasowego podzielona jest na trzy etapy: akwizycja danych, generowanie modelu obiektów, interpretacja obiektów modelu. Próbki sygnału (dane wejściowe) są agregowane i przetwarzane w celu wytworzenia struktury hierarchicznej obiektów. Wynikiem operacji jest wysokopoziomowy graf pozwalający na eksplorację nietrywialnej wiedzy na temat zachowania systemu. W szczególności algorytm odkrywa aktywność urządzenia powiązaną z konkretnymi stanami i sekwencjami operacji.

Szereg czasowy jest sekwencją N wartości próbek sygnału $TS = \{sample1, \dots, sampleN\}$. Próbki są zapisywane w kolejności przy stałym okresie próbkowania- czas między odczytem kolejnych wartości próbek jest stały. Z pojedynczą próbką skojarzone są: wartość próbki- określająca poziom sygnału w momencie odczytu wartości oraz znacznik czasowy- pozwalający na umiejscowienie odczytu w czasie.

Wartość próbki - wartość liczbowa $x_i \in X$ reprezentująca chwilową wartość sygnału np. poboru prądu przez badane urządzenie wyrażona w jednostce natężenia prądu.

Znacznik czasowy - wartość czasu (wyrażona w jednostce czasu np. milisekundy lub w jednostce czasu i daty) $t_i \in T$ pozwalająca jednoznacznie określić moment wystąpienia pomiaru.

Próbka- para: wartość próbki, znacznik czasowy próbki zdefiniowana jest jako $Sample_i = (x_i, t_i)$

Szereg czasowy jest dzielony na mniejsze szeregi zwane dalej segmentami. Każdy segment początkowo jest stałego rozmiaru wynoszącego n próbek. W wyniku operacji powstaje M=N/n segmentów.

Segment- Zbiór kolejnych próbek o początkowej liczbie próbek N. Segmenty nie współdzielą próbek.

$$seg_L = \{(t_L, x_L), \dots, (t_{L+N-1}, x_{L+N-1})\}$$

Z każdym segmentem związany jest zestaw parametrów wyznaczany na podstawie próbek segmentu. Przykładem takich parametrów są wartości statystyczne: minimum (*min*), maksimum (*max*), średnia arytmetyczna (*avg*), średnia kwadratowa (*rms*), odchylenie standardowe (*std*), całka oznaczona, itp. Sąsiadujące segmenty są porównywane ze sobą

na podstawie różnic wartości parametrów statystycznych segmentów. Jeśli segmenty są podobne (różnica wartości statystycznych jest mniejsza od parametru ε_g), przypisane są im takie same etykiety. Etykieta jest indeksem liczbowym ze zbioru liczb naturalnych (wliczając 0). Jeśli segmenty są różne kolejny segment oznaczany jest kolejną wartością etykiety. Ciągi segmentów, w których segmenty posiadają tą samą etykietę tworzą grupę.

Grupa- Ciąg kolejnych segmentów spełniających określony warunek.

$$g_k = \{seg_i : |metric(seg_i) - metric(seg_{i\pm1})| < \varepsilon_g\} \tag{3.1}$$

$i \in [1, |g_k|)$, k jest numerem grupy, a metric jest funkcją przyporządkowującą wektor liczbowy wartości cech danego segmentu, ε_g jest wektorem maksymalnych różnic między kolejnymi segmentami. Funkcja *metric* może być również określona inaczej pozwalając na bardziej złożone i dokładniejsze rozróżnianie podciągów podobnych segmentów. Segmenty, które nie spełniają warunku (3.1) tworzą grupy składające się z pojedynczych segmentów. Na tym etapie algorytm tworzy chronologiczną sekwencję grup, która reprezentuje sygnał wejściowy. W celu poprawienia dokładności opisu, grupy składające się z pojedynczych segmentów mogą być podzielone na trzy części w formie trzech segmentów o niestandardowych rozmiarach. Części zewnętrzne podzielonego segmentu/grupy są przyłączane do sąsiednich grup, które nie są grupami o pojedynczym segmencie. Środkowa część segmentu pozostaje w swojej początkowej grupie.

Każdej grupie g_k przypisywany jest stan s_x. Stan jest właściwym obiektem opisującym wykryty stan badanego systemu w ramach szeregu czasowego. Klasyfikacja grup do stanów bazuje na porównywaniu grup ze sobą bez uwzględnienia sąsiedztwa grup. Dokładny opis tworzenia stanów opisany jest w rozdziale 3.1.3. Stan jest zbiorem grup spełniających warunek z równania (*3.2*).

Stan- Jest to zbiór grup o zbliżonych właściwościach reprezentujący stan urządzenia, w którym możliwe jest zaobserwowanie profilu zachowania z przedziału czasowego określonego przez grupy danego stanu.

$$s_x = \{g_i : \forall g_i \in G \; \exists g_j \in G, i \neq j \; \wedge \; |metric(g_i) - metric(g_j)| < \varepsilon_s\} \tag{3.2}$$

gdzie ε_s jest wektorem maksymalnych różnic między grupami należącymi do danego segmentu. Podobnie jak w warunku (3.1) funkcja *metric* może być bardziej złożona pozwalając na dokładniejszą klasyfikację grup do stanów.

Korzystając z wykrytych stanów, dla danego szeregu czasowego TS można utworzyć graf skierowany stanów definiowany jako $SG = \{S_{g0}, S, E\}$. S_{g0} jest stanem początkowym- pierwszym stanem w ramach szeregu czasowego TS. S jest zbiorem wierzchołków-

w nomenklaturze opisywanego algorytmu jest to zbiór wykrytych stanów. E jest zbiorem tranzycji/krawędzi pomiędzy stanami/wierzchołkami.

Graf- Struktura określająca możliwe tranzycje między stanami w postaci grafu skierowanego, w którym węzłami są stany, a krawędziami możliwe tranzycje między tymi stanami zdefiniowane jako:

$$E = \{(u, v): u, v \in S, u \neq v, \exists g_i \in u \wedge g_j \in v \; \exists seg_k \in g_i \wedge seg_l \in g_j, k + 1 = l\} \quad (3.3)$$

gdzie, E jest zbiorem krawędzi grafu stanów, u i v są stanami oraz $i, j \in [0, |G|), k, l \in [0, M)$. Powyższy warunek oznacza, że krawędź jest tworzona na podstawie sąsiedztwa stanów zmapowanych na szereg czasowy TS. Krawędź między stanami u i v istnieje, jeśli istnieją segmenty należący do grup, które są oznaczone odpowiednio jako stany u i v, a także segmenty te są sąsiadami w ramach szeregu czasowego.

Algorytm definiuje zbiór krawędzi wychodzących z danego stanu/węzła:

$$E_{out(x)} = \{(s_x, v): (s_x, v) \in E\} \quad (3.4)$$

Zbiór krawędzi wchodzących do danego węzła to:

$$E_{in(x)} = \{(u, s_x): (u, s_x) \in E\} \quad (3.5)$$

W zależności od wyznaczonego celu analizy algorytm rozróżnia dwa specjalne rodzaje stanów: stan prosty oraz stan bazowy.

Stan prosty- to stan, do którego urządzenie może przejść tylko z pojedynczego innego stanu. W przypadku, gdy celem jest ukazanie zachowania systemu w sposób uproszczony, stany proste są sklejane z poprzednikiem. Stan taki zawiera tylko jedną grupę nazywaną grupą prostą. Stan prosty musi spełniać warunek:

$$|E_{out(x)}| = |E_{in(x)}| = 1 \quad (3.6)$$

Stan bazowy- to stan S_B, w którym urządzenie wykonuje standardowe zadania tła (np. pobiera najmniej energii). Algorytm definiuje stan bazowy jako stan, z którego najczęściej dochodzi do tranzycji do innego stanu oraz który jest najczęstszym stanem docelowym dla tranzycji.

$$\exists s_B, \forall s_i \in S, |E_{out(B)}| + |E_{in(B)}| \geq |E_{out(i)}| + |E_{in(i)}| \quad (3.7)$$

W przypadku analizy systemów wbudowanych stan bazowy zwykle opisuje czynność, którą system wykonuje periodycznie oraz częściej niż inne czynności. Tego typu wzorzec zachowania systemu jest powodowany przez procesy typowe dla działania systemu wykonujące się w tle np. regularne sprawdzanie stanu urządzeń peryferyjnych, agregacja danych z ADC, regularne przerwania od licznika RTC, itd. Wykrywając stan bazowy, algorytm może użyć jego występowanie jako punkt odniesienia do klasyfikacji różnych anomalii czy też tworzenia profili energetycznych dla zachowań reaktywnych na zmiany w danych wejściowych

lub zmiany w środowisku zewnętrznym. W niektórych aplikacjach znaczenie może mieć znalezienie więcej niż pojedynczego stanu bazowego (poprzez iteracyjne powtórzenie warunku (3.7)) jednak opisywany algorytm wykorzystuje pojedynczy stan bazowy.

Cykl- ciąg grup, którego kolejne elementy tworzą krawędź należącą do zbioru krawędzi E, pierwszy element należy do $E_{out(B)}$, a ostatni do $E_{int(B)}$. s_B jest stanem bazowym.

Cykl to ciąg $C = (g_i, \ldots, g_p)$ taki, że spełnia warunki:

1) $g_i, \ldots, g_p \in G$, i $i \ldots p$ są kolejnymi indeksami grup

2) $g_i \in s_i \; i \; (s_B, s_i) \in E_{out(B)}$

3) $g_p \in s_p \; i \; (s_p, s_B) \in E_{in(B)}$,

4) Niech S^{g_x} oznacza stan, do którego należy grupa g_x, wtedy:

$\forall x \in [i, p)$: $g_x, g_{x+1} \in C$, $(S^{g_x}, S^{g_{x+1}}) \in E$

5) $\forall x \in [i, p)$: $S^{g_x} \neq S_B$

Pojedynczy cykl jest sekwencją kolejnych grup, z których żadna nie jest stanem bazowym. Poprzednikiem pierwszej grupy cyklu jest stan bazowy oraz następnikiem ostatniej grupy z cyklu jest stan bazowy.

Przedstawiony hierarchiczny model obiektów szeregu czasowego TS jest możliwy do zastosowania w celu analizy systemów, które wykonują sporadyczne i niedeterministyczne czynności/reakcje oraz równocześnie wykonują periodyczne zadania w tle. Ponadto powyższe czynności muszą manifestować się w sygnale, który stanowi dane wejściowe algorytmu. Przykładem takiego próbkowanego sygnału może być szereg czasowy wartości poboru prądu lub mocy przez urządzenie. Opisywany model ujawnia hierarchiczną strukturę, której przedstawicielami są kolejno, segmenty, grupy, stany, graf, cykle i klasy cykli. Klasa cykli stanowi zbiór cykli, które spełniają dodatkowe warunki np. posiadają podobny profil poboru energii. Rozpoznane obiekty mogą być następnie analizowane czy też porównane z wiedzą ekspercką dotyczącą badanego systemu lub z danymi z logów (jeśli system takie udostępnia). Algorytm zastosowany do badania systemów, które wypełniają zakładany model behawioralny, ma na celu umożliwienie wykrywania anomalii w zachowaniu systemu lub też nieoptymalności w wykorzystaniu zasobów. Porównywanie wielu modeli uzyskanych z wyników algorytmu przeprowadzonego na kolejnych szeregach czasowych zgromadzonych w podobnych warunkach może pozwolić na weryfikacje powtarzalności działania systemu czy też wykrycia próbek urządzeń, które powtarzalnie generują anomalie.

3.1.2 Pseudokod opisu algorytmu

Algorytm zaprezentowany jest jako zbiór funkcji opisanych w pseudokodzie. Pseudokod ma strukturę bazującą na języku Python. Pseudokod wykorzystuje paradygmat obiektowy wraz z notacją używaną często przez języki obiektowe. Notacja *obiekt.parametr* oznacza odniesienie do wartości parametru *parametr,* który stanowi część stanu obiektu *obiekt.* Notacja *obiekt.metoda(argumenty)* oznacza wykonanie funkcji *metoda* z danymi wejściowymi w postaci parametrów *argumenty* oraz stanu obiektu *obiekt.* Nazwy obiektów, parametrów czy metod są znaczące- nazwy odpowiadają odpowiednio na pytania: „czyj stan reprezentuje *obiekt*?", „Co oznacza właściwość/atrybut wskazywany przez *parametr*?", „Jakie czynności wykonuje *metoda*?". Przykładem takiej notacji jest: *groupA.add(segmentB)* gdzie *add* oznacza funkcję dodającą do instancji kolekcji będącej grupą *groupA* segment *segmentB*. Funkcja *are_similar(segmentA, GroupB)* wylicza metrykę różnic wartości statystycznych między segmentem A, a połączonymi segmentami w grupę B i zwraca wartość logiczną: prawdę lub fałsz w zależności od wartości różnic. Funkcja zwraca prawdę, gdy wartości różnic mieszczą się w zakresach wartości konfiguracji algorytmu.

Większość obiektów używanych w pseudokodach algorytmu są to obiekty-kolekcje, co oznacza, że obiekty zawierają wiele innych obiektów, często zunifikowanego typu. Przykładami kolekcji mogą być zbiory (*set)* czy też listy *(list)*. Zbiór jest nieposortowaną kolekcją unikalnych obiektów tego samego typu. Lista jest kolekcją obiektów tego samego typu o dostępie sekwencyjnym. Do listy dodawany jest element na końcu lub początku. Z kolei z listy wybierany czy przeszukiwany jest element począwszy od początku lub końca listy.

Instancje obiektów tworzone są z wykorzystaniem notacji *instancja* = ***new*** *typ_obiektu* na przykład: *groupA* = ***new*** *Group,* tworzy obiekt grupy. Innym rodzajem złożonego typu obiektu jest graf *(Graph)*. Graf składa się z listy wierzchołków/węzłów (*nodes*) oraz z krawędzi (*edges*). Węzły i krawędzie są również kolekcjami, które przechowują informacje o dostępnych węzłach, a także o możliwych tranzycjach między nimi.

Pseudokod do kontroli przepływu wykonywanych kroków wykorzystuje instrukcje zbudowane ze słów kluczowych takich jak: **if, then, else, end, foreach, return**. Instrukcja warunkowa **if** *(warunek)* **then** *blok_prawda* **else** *blok_fałsz* **end if** pozwala na wykonanie kroków algorytmu z jednego z dwóch bloków instrukcji na podstawie wartości logicznej warunku. Warunek może składać się z wielu warunków połączonych operatorami logicznymi takimi jak: iloczyn (**and**) i suma (**or**). Do tworzenia warunków logicznych, w których parametrami są zmienne liczbowe, wykorzystuje się standardowe operatory relacji: $=, <, >, \leq, \geq$. Ponadto pseudokod wykorzystuje operatory arytmetyczne: mnożenie (*),

dzielenie (/), dodawanie (+), odejmowanie (-). Modelowanie pętli instrukcji realizowane jest poprzez notację **foreach** prezentowaną na listingu poniżej:

```
1:      foreach element in kolekcja do
2:           blok instrukcji
3:      end foreach
```

Konstrukcja **foreach** pozwala na zapis sekwencji instrukcji, który zakłada wielokrotne wykonanie kroków z *bloku instrukcji*. Liczba iteracji jest zależna od liczby elementów w obiekcie *kolekcja*. Po każdej iteracji obiekt *element* wskazuje na następny element w *kolekcji*.

Struktura pseudokodu zakłada możliwość definiowania procedur, które są blokami instrukcji przywoływanymi jako instrukcje algorytmu. Procedura jest sparametryzowana argumentami, które zdefiniowane są w notacji definicji procedury: **function** nazwa_procedury(argument1, argument2, …). Odwołanie do procedury w innym fragmencie opisu algorytmu w postaci: *nazwa_procedury(argument1, argument2, ...)* oznacza wykonanie kroków zdefiniowanych przez procedurę. Procedura może zwracać wynik działania. Realizowane jest to przez notację z użyciem słowa kluczowego **return.**

3.1.3 Specyfikacja algorytmu

Analiza szeregu czasowego podzielona jest na etap stworzenia modelu obiektów oraz na identyfikacje właściwości i interakcji między obiektami. Pseudokod algorytmu jest w podrozdziale opisany w postaci definicji (algorytmów) poszczególnych procedur. Dane wejściowe są przetwarzane przez kolejne procedury. Procedury zwracają przetworzone dane w postaci obiektów, które stają się danymi wejściowymi dla kolejnej procedury. W algorytmie 3-1 przedstawiony jest szkielet pełnego algorytmu, który za dane wejściowe przyjmuje szereg czasowy próbek, a w rezultacie zwraca obiekty: wykrytych stanów, graf oraz listę klas cykli. Kompletny algorytm jest opisany w formie czternastu procedur. Podsumowanie podziału algorytmu znajduje się w tabeli 4.

Nr algorytmu	Opis
1	Opis algorytmu najwyższego poziomu
2	Generowanie segmentów z szeregu próbek czasowych
3,4,5,6,7	Generowanie grup segmentów i rafinacja segmentów granicznych
8,9	Generowanie stanów na podstawie zbioru grup
10,11	Generowanie krawędzi na potrzeby stworzenia modelu grafu
12,13,14,15	Wykrywanie cykli oraz agregowanie ich do postaci klas cykli

Tabela 4. Zestawienie procedur algorytmu

Procedura *analyze_data* zawiera sekwencję wywołań kolejnych procedur algorytmu. Pierwszą wołaną procedurą jest *create_segments*, która dzieli szereg próbek na segmenty o stałej długości. Segmenty są danymi wejściowymi dla procedury *create_groups*. Procedura ta porównuje sąsiadujące segmenty i tworzy grupy sąsiadujących segmentów, które zawierają segmenty podobne. Na podstawie listy grup procedura *detect_states,* klasyfikuje grupy do stanów oznaczając je odpowiednimi etykietami. Analizując grupy z etykietami stanów procedura *detect_edges* generuje listę krawędzi (*edges*) między stanami. Obiekty stanów oraz krawędzi są agregowane w jeden obiekt grafu (*Graph*). Procedura *find_all_similar_cycles* analizuje model grafu w celu znalezienie cykli, a następnie klasyfikuje je tworząc obiekt klasy cykli *(CycleClasses).*

```
Wejście:
      Samples   - wartości próbek
      Period    - okres próbkowania
      Timestamp - znacznik czasowy pierwszej próbki
Wyjście:
      States       - wykryte stany
      Graph        - obiekt grafu stanów
      CycleClasses - klasy cykli.
1: function analyze_data(Samples, Period, Timestamp)
2:       Segments = create_segments(Samples, Period, Timestamp)
3:       Groups = create_groups(Segments)
4:       States = detect_states(Groups)
5:       Edges = detect_edges(Groups)
6:       Graph = new Graph(States, Edges)
7:       CycleClasses = find_all_similar_cycles(Graph)
8:       return States, Graph, CycleClasses
9:  end function
```

Algorytm 3-1. Procedura wykonująca analizę szeregu czasowego (*analyze_data*)

Tworzenie segmentów polega na podziale sekwencji próbek pomiarów chwilowego poboru prądu przez badane urządzenie na segmenty o stałym rozmiarze. Algorytm 3-2 prezentuje procedurę tworzenia listy segmentów. W pierwszych krokach algorytmu

początkowa wartość obecnego znacznika czasu (*timestamp)* jest ustawiana na wartość znacznika czasowego pierwszej próbki. Następnie w każdej iteracji pętli **foreach** dodawana jest próbka do listy próbek danego segmentu. Próbka w ramach segmentu składa się z wartości próbki oraz jej znacznika czasowego. Wartość znacznika czasowego próbki przypisywana jest na podstawie wartości zmiennej *timestamp*. Zmienna *timestamp* aktualizowana jest co iteracje poprzez dodawanie wartości okresu próbkowania. W ramach pętli istnieje zmienna *i*, która przechowuje rozmiar obecnie tworzonego segmentu- jest inkrementowana po dodaniu każdej próbki. Jeśli wartość rozmiaru segmentu osiągnie określoną wartość (w konfiguracji algorytmu oznaczaną jako *Seg_N*), to zmienna *i* jest ustalana jako 0. Jeśli rozmiar obecnego segmentu jest równy 0, tworzony jest nowy obecny segment *s* i odniesienie do niego dodawane jest do listy segmentów *Segments* za pomocą metody *add.* W ostatnim kroku procedura zwraca listę segmentów.

```
Wejście:
      Samples   - wartości próbek
      Period    - okres próbkowania
      Timestamp - znacznik czasowy pierwszej próbek
Wyjście:
      Segments - wygenerowane segmenty
1:  function create_segments(Samples, Period, Timestamp)
2:      Segments = new List
3:      timestamp = Timestamp
4:      Seg_N = params.seg_size
5:      s = None
6:      i = 0
7:      foreach sample in Samples do
8:          if (i >= Seg_N) then
9:              i = 0
10:         end if
11:
12:         if (i = 0) then
13:             s = new Segment
14:             Segments.add(s)
15:         end if
16:         i = i + 1
17:         s.add(sample, timestamp)
18:         timestamp = timestamp + Period
19:     end foreach
20:     return Segments
21: end function
```

Algorytm 3-2. Procedura tworząca segmenty szeregu czasowego (*create_segments*)

Algorytm 3-3 przedstawia procedurę tworzenia grup bazując na liście segmentów. Segmenty są przeglądane w kolejności ich utworzenia. Podczas każdej iteracji po liście segmentów (kroki 4-10) do obiektu grupy (wskazywanego przez zmienną *group*) dodawany jest obecnie przeglądany segment. Jeśli segment nie spełnia warunków podobieństwa do grupy,

obiekt wskazywany przez *group* dodawany jest do kolekcji *Groups* oraz tworzony jest nowy obiekt grupy (linia 7). O przyłączeniu do grupy decyduje funkcja *are_similar*. Na koniec procedury lista grup *Groups* jest zwracana jako wynik działania funkcji.

```
Wejście:
     Segments – lista segmentów
Wyjście:
     Groups - lista grup segmentów
1: function create_groups(Segments)
2:      Groups = new List
3:      group = new Group
4:      foreach segment in Segments do
5:           if not are_similar(segment, group) then
6:                Groups.add(group)
7:                group = new Group
8:           end if
9:           group.add( segment )
10:     end foreach
11:     return Groups
12: end function
```

Algorytm 3-3. Procedura tworząca grupy na podstawie listy segmentów (*create_groups*)

```
Wejście:
     x,y - 2 wartości liczbowe
Wyjście:
     wartość różnicy w relacji do wartości większej
1: function relative_diff(x, y)
2:     if (x < y) then
3:          return 1.0 - x/y
4:     else
5:          return 1.0 - y/x
6: end function
```

Algorytm 3-4. Procedura wyznaczająca różnicę względną (*relative_diff*)

Procedura *are_similar* służy do określenia czy dany segment podobny jest do innego segmentu lub grupy segmentów, którą stanowią próbki ze wszystkich segmentów należących do badanej grupy. W krokach 2-3 algorytmu 3-5 procedura może zakończyć się z wynikiem *Prawda*, jeśli obiekty grupy nie zawiera żadnych segmentów. Jest to zachowanie wybrane arbitralnie w celu uproszczenia innych procedur w ramach algorytmu. W następnych krokach procedura porównuje różnice zadanych wartości statystycznych wyznaczonych na podstawie próbek segmentu oraz próbek połączonych segmentów w ramach grupy. Jeśli dana różnica jest mniejsza od odpowiedniego parametru algorytmu (nazwa parametru zawiera przyrostek *_eps*), to dany warunek podobieństwa jest spełniony. Procedura pozwala na uznanie podobieństwa między segmentem a grupą, jeśli wszystkie warunki są spełnione- w przeciwnym przypadku procedura *are_simillar* zwraca *Fałsz*. Możliwych jest wiele wariantów funkcji

dyskryminacyjnej tego typu. Wybór odpowiednich funkcji metryk grupy i segmentu oraz parametrów dyskryminacyjnych tych metryk jest kluczowy dla poprawnego działania algorytmu. Lista użytych metryk oraz wartości parametrów jest silnie związana z aplikacją algorytmu. W przykładzie zaprezentowanych w algorytmie 3-5 obliczane są wartości: średniej arytmetycznej, odchylenia standardowego, minimum, maksimum oraz całki oznaczonej zarówno dla grupy jak i dla segmentu, który jest kandydatem do grupy. Dla wartości średniej arytmetycznej oraz całek oznaczonych obliczana jest różnica zależna (algorytm 3-4). Funkcja ta pozwala przedstawić wartość podobieństwa w postaci jednej liczby w zakresie (0.0, 1.0) bazując na stosunku obu wartości statystyk.

```
Wejście:
     segment – pojedynczy segment
     group   - grupa
Wyjście:
     Wartość logiczna - Prawda, jeśli segment jest podobny do grupy,
w innym przypadku Fałsz
1: function are_similar(segment, group)
2:     if (group.segment_count == 0) then
3:         return True
4:     end if
5:     if ((relative_diff(avg(group), avg(segment)) < params.avg_eps)
6:       and (abs(deviation(group) - deviation(segment)) <
           params.dev_eps)
7:       and (abs(min(group) - min(segment)) < params.min_eps)
8:       and (abs(max(group) - max(segment)) < params.max_eps)
9:       and (relative_diff(integral(group), integral(segment)) <
           params.int_eps))
10:      then
11:        return True
12:    else
13:        return False
14:    end if
15: end function
```

Algorytm 3-5. Przykład procedury określającej podobieństwo dwóch obiektów szeregu czasowego na podstawie cech statystycznych (*are_similar*)

Kolejnym etapem algorytmu jest poprawa dokładności wyznaczenia grup. Konsekwencją podziału na segmenty o identycznych rozmiarach jest wytworzenie pojedynczych segmentów, które nie należą do żadnej z sąsiadujących grup, generując grupy o liczbie segmentów równych jeden. Segment taki zwykle zawiera część próbek, które tworzą profile pasujące do grup poprzednika i następnika. Grupa zawierająca opisywany segment dalej nazywana jest grupą prostą. Procedura zaprezentowana w ramach algorytmu 3-6 dzieli segment na jeden do trzech części. Grupy proste składają się z segmentu podzielonego na fragmenty: (a) fragment należący do grupy poprzedzającej, (b) próbki reprezentujące zmianę stanu oraz (c) fragment należący do

grupy następnej. Algorytm 3-6 zawiera procedurę wykrywania oraz podziału grup prostych. Jeśli grupa spełnia definicję grupy prostej, jej segment dzielony jest na trzy części w szczególności na część: poprzedzającą (dopasowywaną do grupy poprzedzającej) oraz następującą (dopasowywaną do grupy następującej). Grupa jest klasyfikowana jako prosta, jeśli zawiera pojedynczy segment, a grupy poprzedzająca i następująca zawierają więcej niż jeden segment (linie 3-5). Podsegment poprzedzający jest porównywany z grupą poprzedzającą, a następujący z grupą następującą. Jeśli którykolwiek z podsegmentów zostanie zaklasyfikowany jako część innych grup, dodawany jest on do tej grupy. Ponadto z analizowanej grupy usuwany jest przepisany podsegment. Kroki te realizowane są w ramach funkcji *append_from_beggining* i *append_from_end* (linie 6 i 7). Funkcje te rekurencyjnie dzielą podsegmenty do momentu, w którym warunek podobieństwa jest spełniony (procedura *are_similar* zwróci wartość *Prawda*) lub głębokość podziału osiągnie wartość maksymalną. Długość odciętego segmentu może przyjąć wartość równą $\frac{N}{2^x}$, gdzie N to standardowa długość segmentu, a x to głębokość rekurencji w zakresie od jeden do *max_level*.

```
Wejście:
     Groups – lista grup
Wyjście:
     Groups – lista grup z podzielonymi segmentami brzegowymi
1:  function increase_accuracy(Groups)
2:      foreach group in Groups do
3:          if ((group.segment_count == 1)
4:            and (group.prev.segment_count > 1)
5:            and (group.next.segment_count > 1)) then
6:              first_segment = group.first_segment
7:              group.prev.append_from_beginning(first_segment)
8:              group.next.append_from_end(first_segment)
9:          end if
10:     end foreach
11: end function
```

Algorytm 3-6. Procedura dzieląca segmenty brzegowe na mniejsze części (*increase_accuracy*)

Na tym etapie algorytmu, hierarchiczna struktura segmentów oraz grup jest już zrafinowana. Ostatni etap algorytmu składa się z trzech kroków. Pierwszym z nich jest wykrycie stanów na podstawie sekwencji grup. Algorytm 3-7 prezentuje procedurę *detect_states*. Procedura iteracyjnie przegląda listę grup *Groups*. W każdej iteracji wywoływana jest procedura *add_to_similar_state* opisana w algorytmie 3-8. Procedura przyjmuje jako argumenty wejściowe listę wykrytych stanów oraz aktualnie analizowaną grupę. Procedura przeszukuje listę stanów by odnaleźć stan, do którego pasuje obecnie przetwarzana grupa (linie 2-8). Dopasowanie weryfikowane jest poprzez funkcje *are_similar*.

Grupa przypisywana do stanu oznaczana jest etykieta stanu. Funkcja *add_to_similar_state* zwraca obiekt stanu w przypadku znalezienia pasującego stanu lub symbol *None* w przeciwnym. Jeśli grupa *group* nie zostanie przypisana do żadnego z obecnych stanów, wykonywane są kroki 6-7 z algorytmu 3-7. Tworzony jest nowy obiekt stanu *state* za pośrednictwem funkcji *create_new_state*, która także oznacza grupą nową etykietą stanu. Obiekt stanu *state* zawiera wszystkie grupy oznaczone tą samą etykieta stanu oraz samą etykietę. Obiekt stanu dodawany jest do listy stanów *States* (linie 6-7). Po przejrzeniu wszystkich grup i ich klasyfikacji lista stanów zwracana jest w kroku 10.

```
Wejście:
     Groups – Lista grup segmentów
Wyjście:
     States - Lista wykrytych stanów
1:  function detect_states(Groups)
2:     States = new List
3:     foreach group in Groups do
4:          state = add_to_similar_state(States, group)
5:          if (state is None) then
6:                    state = create_new_state(group)
7:                    States.add(state)
8:          end if
9:     end foreach
10:    return States
11: end function
```

Algorytm 3-7. Procedura wstępnego rozpoznawania stanów oraz generowania grafu przejść między stanami (*detect_states*)

```
Wejście:
     States – lista obiektów stanów, group– grupa segmentów
Wyjście:
     state - obiekt stanu, do którego grupa jest podobna
1:  function add_to_similar_state(States, group)
2:     foreach state in States do
3:          if (are_similar(state, group)) then
4:               state.add(group)
5:               group.StateLabel = state.Label
6:               return state
7:          end if
8:     end foreach
9:     return None
10:  end function
```

Algorytm 3-8. Procedura agregująca zadaną grupę od odpowiedniego stanu (*add_to_similar_state*)

Po wykryciu wszystkich stanów i oznaczeniu każdej z grup numerem stanu procedura *detect_edges* w procesie iteracyjnym wykrywa wszystkie krawędzie tranzycji między stanami poprzez porównanie sąsiadujących pól *StateLabel* w każdej z grup. W kroku 3 algorytmu 3-9

tworzona jest pętla, która iteruje po wszystkich grupach z kolekcji *Groups*. W kroku 4 algorytm sprawdza czy obecna grupa jest ostatnią grupą w sekwencji szeregu czasowego. Jeśli grupa posiada następnik, to wartości etykiet stanów dla grup obecnie analizowanej oraz następnika są porównywane w kroku 5. Różne etykiety stanów oznaczają, że w danym miejscu szeregu czasowego sygnał odwzorowuje przejście między dwoma stanami. W ramach struktury hierarchicznej takie zdarzenie modelowane jest jako krawędź grafu stanów. W kroku 6 tworzony jest obiekt krawędzi i dodawany do zbioru krawędzi *Edges*. Wynikiem procedury *detect_edges* jest zbiór krawędzi (par: numer stanu wyjściowego, numer stanu wejściowego). Lista stanów *States* wraz ze zbiorem krawędzi *Edges* tworzy graf stanów reprezentowany przez obiekt *Graph*.

```
Wejście:
     Groups – lista grup
Wyjście:
     Edges – lista wykrytych krawędzi
1:  function detect_edges(Groups)
2:      Edges = new Set
3:      foreach group in Groups do
4:          if (group.next is not None) then
5:              if (group.StateLabel!=group.next.StateLabel) then
6:                  Edges.add(new Edge(group.StateLabel,
                        group.next.StateLabel))
7:              end if
8:          end if
9:      end foreach
10:     return Edges
11: end function
```

Algorytm 3-9. Procedura wykrywania krawędzi (*detect_edges*)

```
Wejście:
     graph – Obiekt grafu
Wyjście:
     graph – Obiekt grafu z zredukowanymi prostymi krawędziami
1: function merge_simple_sequences(graph)
2:     graph.Nodes.clear_visited_flags()
3:     sequences = new Set
4:     current_sequence = new List
5:     identify_simple_sequences(graph.Nodes.first, sequences,
          current_sequence)
6:     merge_states_and_nodes(sequences, graph)
7: end function
```

Algorytm 3-10. Procedura scalająca ze sobą sąsiadujące stany proste (*merge_simple_sequences*)

```
Wejście:
      node             - węzeł grafu
      sequences        - zbiór sekwencji
      current_sequence - wskazanie na obecnie analizowaną sekwencję
Wyjście:
      sequences - zaktualizowana lista sekwencji
1:function identify_simple_sequences(node, sequences,
    current_sequence)
2:     node.visited = True
3:     if (node.In_edges.count <= 1 and node.Out_edges.count <= 1)
         then
4:         current_sequence.add(node)
5:         if (node.Out_edges.count == 0) then
6:             sequences.add(current_sequence)
7:             current_sequence = new List
8:         end if
9:     else
10:        if (current_sequence.length > 1) then
11:            sequences.add(current_sequence)
12:            current_sequence = new List
13:        end if
14:    end if
15:    foreach neighbour in node.neighbours do
16:        if (neighbour.visited == False) then
17:          identify_simple_sequences(neighbour, sequences,
               current_sequence)
18:        end if
19:    end foreach
20: end function
```

Algorytm 3-11. Procedura wyszukująca sekwencje stanów prostych (*identify_simple_sequences*)

Wygenerowany graf stanów jest zwykle grafem złożonym. Graf może również lokalnie składać się z sekwencji przejść między stanami, które występują unikatowo (takie stany posiadają jedną krawędź wejściową i wyjściową). Stany tego typu określane są stanami prostymi. Przykładem zdarzenia, które może wygenerować serię stanów prostych grafu, jest proces uruchamiania się urządzenia. Proces ten wymaga zwykle inicjalizacji wielu komponentów zarówno sprzętowych jak i programowych. Inicjalizacje są wykonywane zwykle jednorazowo oraz w różnej kolejności, skutkując występowaniem pojedynczych stanów następujących po sobie. Tego typu zdarzenia mogą być nieinteresujące z perspektywy dalszej analizy. W tym celu algorytm umożliwia uproszczenie grafu poprzez sklejenie ze sobą stanów prostych do jednego stanu, który może reprezentować cały proces uruchamiania się urządzenia. Każdy stan, który wystąpił więcej niż raz w ramach próbek szeregu czasowego, skutkuje powstaniem przynajmniej trzech krawędzi oznaczających tranzycje z lub do stanu. Taki stan nie spełnia definicji stanu prostego. Algorytm wykrywania stanów prostych oraz upraszczania grafu jest przedstawiony w algorytmach 3-10 i 3-11. Procedura *merge_simple_sequences*

przyjmuje jako dane wejściowe obiekt grafu. Następnie algorytm w dwóch krokach (linie 5 i 6) wykrywa sekwencje stanów prostych za pomocą rekurencyjnej funkcji z algorytmu 3-11 *identify_simple_sequences.* Sekwencje są przekazywane do procedury *merge_states_and_nodes* w celu sklejenia stanów w pojedynczy bardziej ogólny stan. Procedura sklejająca agreguje wszystkie grupy z następnych stanów prostych względem pierwszego stanu prostego w sekwencji. Grupy te dodawane są do pierwszego stanu prostego oraz ich etykiety są aktualizowane. Kroki 2-4 algorytmu 3-10 zawierają instrukcje przygotowujące dane dla algorytmu identyfikacji stanów prostych poprzez wyzerowanie flag odwiedzenia grafu oraz utworzenie obiektu zbioru sekwencji *sequences* i obiektu pomocniczego *current_sequence.* Procedura identyfikacji sekwencji prostych wywoływana jest z argumentami: stan początkowy w ramach szeregu czasowego oraz obiekt grafu, do którego dostęp jest realizowany poprzez obiekt węzła stanu początkowego. Procedura identify*_simple_sequences* oparta jest na algorytmie grafowym przeszukiwania w głąb. W każdym wywołaniu procedury przeglądany jest pojedynczy węzeł/stan grafu. W pierwszym kroku procedury obecnie przetwarzany węzeł oznaczany jest jako odwiedzony. Następnie w kroku trzecim algorytm sprawdza czy węzeł jest stanem prostym poprzez weryfikację liczności krawędzi wyjściowych i wejściowych. Jeśli obecny stan jest stanem prostym, to w kroku 4 dodawany jest on do aktualnie tworzonej sekwencji stanów prostych. W przeciwnym przypadku algorytm dodaje obecną sekwencję do listy sekwencji będących wynikiem działania procedury, a także tworzy nową pustą sekwencję stanów prostych. Powyższe operacje opisane są w krokach 6-7 i 10-12 algorytmu 3-11. W pętli z kroków 15-19 algorytm rekurencyjnie przegląda wszystkich nieodwiedzonych sąsiadów (krawędzie wyjściowe) obecnie analizowanego stanu. W momencie, gdy procedura przeanalizuje wszystkie stany, zbiór *sequences* zawiera wszystkie sekwencje stanów prostych, które mogą zostać sklejone.

Na tym etapie algorytmu graf wraz ze stanami i grupami próbek szeregu czasowego jest gotowy do wykorzystania w analizie na wyższym poziomie abstrakcji. Możliwe jest wyszukiwanie anomalii w ramach sekwencji stanów, odnalezienie podobnych sekwencji stanów wskazujących na powtarzanie czynności, tworzenie profili zachowania urządzenia pod wpływem różnych zdarzeń zewnętrznych. Wykorzystanie struktury hierarchicznej obiektów szeregu czasowego na prezentowanym wyższym poziomie abstrakcji pozwala również korelować stany urządzenia jak i procesy z innymi danymi takimi jak logi systemowe czy aplikacyjne.

W systemach wbudowanych częstym wzorcem działania jest periodyczne wykonywanie powtarzalnych czynności w tle oraz dynamiczne reagowanie na te same zdarzenia w identyczny zaplanowany sposób. Zakładając taki wzorzec zachowań systemu można wydzielić dwa typy stanów: 1) stany, w których urządzenie wykonuje periodyczne i typowe czynności oraz 2) stany, których emanacją jest reakcja na czynnik zewnętrzny lub wewnętrzny. Powyższą obserwację wykorzystuje metoda wykrywania anomalii zastosowania w algorytmie. W ostatnim etapie algorytm oddziela zadania tła od zadań reaktywnych czy też anomalii systemu lub obsługi błędów. W typowym systemie wbudowanym zadania tła są stanem początkowym i końcowym dla wszystkich stanów reaktywnych. Na zadania tła składają się zarówno periodyczne przeglądy stanu systemu oraz zasobów systemu jak i sam stan spoczynku urządzenia, w którym urządzenie nie wykonuje żadnych czynności. Taki stan w grafie badanego systemu ma najwięcej krawędzi zarówno wejściowych jak i wyjściowych. W nomenklaturze algorytmu opisywany stan nazywany jest stanem bazowym. W zależności od aplikacji można przyjąć luźniejszą definicję stanu bazowego, co skutkować może powstaniem kilku stanów bazowych w ramach pojedynczego grafu. Stan bazowy stanowiący początek i koniec procesów w ramach systemu pozwala na identyfikacje cykli stanów, które opisują bardziej złożone procesy. We współczesnych systemach często reakcja na zdarzenie nie zachodzi w postaci pojedynczego sekwencyjnego procesu, ale w postaci serii równolegle wykonywanych czynności. Czynności te mogą być zrealizowane w różnej kolejności lub też różnie się przeplatać, jednak zagregowany profil wykonania reakcji dla zdarzeń tego samego typu powinien być podobny. Zaprezentowana obserwacja ukierunkowuje analizę na poszukiwanie podobieństwa między cyklami. Zbiór cykli, które są podobne, określany jest klasą cykli. Algorytm w ostatnim etapie wyszukuje cykle oraz klasyfikuje je w obiekty klasy cykli.

```
Wejście:
     graph – obiekt grafu
Wyjście:
     graph – graf
1: function find_all_similar_cycles(graph)
2:     detect_base_state(graph)
3:     find_cycles(graph)
4:     find_similar_cycles(graph)
5: end function
```

Algorytm 3-12. Procedura znajdująca wszystkie podobne cykle w grafie stanów (*find_all_similar_cycles*)

```
Wejście:
     graph – graf
Wyjście:
     graph – graf z oznaczonymi stanami bazowymi
1: function detect_base_state(graph)
2:      max = 0
3:      foreach node in graph.Nodes do
4:          if (node.Edges.count > max) then
5:              base_state = node.state
6:              max = node.Edges.count
7:          end if
8:      end foreach
9:      graph.base_state = base_state
10: end function
```

Algorytm 3-13. Procedura selekcji stanu bazowego (*detect_base_state*)

```
Wejście:
     graph – graf
Wyjście:
     graph – graf z oznaczonymi cyklami
1: function find_cycles(graph)
2:     cycle = None
3:     Cycles = new List
4:     foreach g in graph.Groups do
5:         if (g == graph.base_state) then
6:             if (cycle == None) then
7:                 cycle = new Cycle
8:             else
9:                 Cycles.add(cycle)
10:                cycle = None
11:            end if
12:        else
13:            if (cycle != None) then
14:                cycle.add(g)
15:            end if
16:        end if
17:    end foreach
18:    graph.Cycles = Cycles
19: end function
```

Algorytm 3-14. Procedura wyszukująca cykle w grafie stanów (*find_cycles*)

```
Wejście:
     graph – graf
Wyjście:
     graph – graz z zidentyfikowanymi podobnymi cyklami
1: function find_similar_cycles(graph)
2:     Cycle_Classes = new List
3:     Cycles.ClearVisitedFlag()
4:     foreach c1 in graph.Cycles do
5:         if (c1.visited == False) then
6:             c1.visited = True
7:             current_CC = new List
8:             current_CC.add(c1)
9:             foreach c2 in graph.Cycles do
10:                if ((c2.visited == False) and are_similar(c1,c2))
                     then
11:                    current_CC.add(c2)
12:                    c2.visited = True
13:                end if
14:            end foreach
15:            Cycle_Classes.add(current_CC)
16:        end if
17:    end foreach
18:    graph.Cycle_Classes = Cycle_Classes
19:  end function
```

Algorytm 3-15. Procedura wykrywające ciągi stanów tworzących klasy cykli (*find_similar_cycles*)

W algorytmie 3-12 przedstawione są poszczególne kroki ostatniego etapu. Najpierw za pomocą procedury *detect_base_state* odnajdywany jest stan bazowy (algorytm 3-13), następnie wykorzystując stan bazowy znajdywane są wszystkie cykle zapoczątkowane przez stan bazowy (w ramach procedury *find_cycles*- algorytm 3-14). W ostatnim kroku cykle są porównywane i grupowane w klasy cykli za pomocą funkcji *find_similar_cycles* z algorytmu 3-15.

Procedura znalezienia stanu bazowego z algorytmu 3-13 jest typowym przykładem iteracyjnego wyszukania węzła o największej liczbie krawędzi (zapis *node.Edges.count* zwraca liczbę krawędzi zadanego węzła). W linii 9, wykryty stan zapamiętywany jest w atrybucie grafu *base_state*. Procedura *find_cycle* z algorytmu 3-14 w kroku 4 tworzy pętlę iterującą po wszystkich grupach użytych podczas tworzenia stanów i grafu. Jeśli obecnie przetwarzana grupa jest oznaczona inną etykieta niż stan bazowy, grupa dodawana jest do tworzonego cyklu *cycle*. W przeciwnym przypadku, jeśli *cycle* jest utworzone, cykl jest zamykany i dodawany do listy cykli *Cycles* (linie 9 i 10). Jeśli zmienna *cycle* nie wskazuje na otwarty cykl, tworzony jest nowy pusty cykl.

Mając wygenerowane wszystkie cykle, algorytm grupuje je w klasy cykli. Grupowanie realizowane jest poprzez porównanie każdego cyklu z wszystkimi innymi. Algorytm realizuje

powyższe założenie w dwóch pętlach (linie 4 i 9 w algorytmie 3-15) oraz korzystając z atrybutów *visited* obiektów klas w celu uniknięcia podwójnego grupowania tych samych cykli. Porównanie wykonywane jest poprzez funkcję *are_similar*. Użycie funkcji *are_similar* jest możliwe, ponieważ cykle są ciągłe (próbki tworzące cykl są swoimi następnikami lub przednikami w ramach szeregu czasowego). Funkcja porównująca wylicza statystyki z próbek od początku do końca porównywanych cykli. Jeśli cykle są podobne, cykl dodawany jest do utworzonej listy podobnych cykli *current_cc* (krok 8 i 11). Po opuszczeniu pętli wewnętrznej nowa klasa cykli dodawana jest do listy klas cykli *Cycle_classes*. Na koniec procedury lista klas cykli włączana jest do obiektu grafu. Każdy cykl oznaczany jest indexem C1, C2, itd. Podobne cykle tworzą klasę cykli oznaczaną CCx.

3.1.4 Właściwości algorytmu

Złożoność algorytmu została wyznaczona przy użyciu notacji dużego O(n), gdzie n określa klasę złożoności np. $O(n^x)$ opisuje złożoność wielomianową, w szczególności n^2 oznacza złożoność kwadratową. Notacja O(n) określa górny limit złożoności/złożoność pesymistyczną występującą dla zestawu danych wejściowych skutkujących największym wykorzystaniem zasobów. Zmienna n określa liczność danych wejściowych, więc przy czasowej złożoności kwadratowej algorytm wykona $n^2 * C$ instrukcji/kroków (C- jest stałą wartością). Podczas analizy algorytmu rozpatrywane były dwa typy złożoności: a) czasowa oraz pamięciowa. Złożoność czasowa określa liczność wykonywanych kroków algorytmu w zależności od rozmiaru danych wejściowych. Złożoność pamięciowa określa wymiar rozmiaru pamięci niezbędnego do działania algorytmu dla zadanej liczności danych wejściowych. W tabeli 5 przedstawiono złożoności czasowe i pamięciowe dla poszczególnych etapów działania algorytmu.

Procedura algorytmu	**Parametr danych wejściowych**	**Złożoność czasowa**	**Złożoność pamięciowa**
create_segments (2)	N- Liczba próbek	O(N)	O(N)
create_groups (3,4,5)	M- Liczba segmentów	O(NM)	O(M)
increace_accuracy (6)	P- liczba grup	O(P)	O(1)
detect_states (7,8)	P- liczba grup	$O(P^2)$	O(P)
detect_edges (9)	P- liczba grup	O(P)	$O(P^2)$
merge_simple_sequences (10) identify_simple_sequences (11)	S- liczba stanów E- liczba krawędzi	O(S+E)	O(S)
detect_base_state (13)	E- liczba krawędzi	O(E)	O(1)
find_cycles (14)	P- liczba grup	O(P)	O(P)
find_similar_cycles (15)	C- liczba cykli	$O(C^2)$	0

Tabela 5. Złożoność algorytmów

Pesymistyczna złożoność pamięciowa dla procedury tworzenia grup wynosi O(M), ponieważ w najgorszym wypadku liczba grup jest równa liczbie stworzonych segmentów. Najmniejszym rozmiarem segmentu jest segment o liczności próbek równej jeden. W takim wypadku liczba segmentów jest równa liczbie próbek. Liczba segmentów jest wyliczalna ze wzoru: liczba segmentów = liczba próbek/rozmiar segmentu (M=N/n). W trakcie tworzenia segmentów przeglądana jest każda próbka. Pesymistyczny przebieg tworzenia grup zakłada dołączanie do jednej grupy kolejnych segmentów. Podczas takiego procesu po każdym segmencie należy przeliczyć wartości statystyk dla grupy skutkując złożonością czasową O(NM). Upraszając złożoność tylko do parametru liczby segmentów N=nM złożoność tworzenia grup wynosi $O(M^2)$. Liczba wygenerowanych grup zależy liniowo od liczby segmentów (w notacji O(n)). Podczas poprawy dokładności wykrywania grup przeglądane są wszystkie grupy generując złożoność czasową O(P)~O(N)~O(M). Wykrywanie stanów w pesymistycznym wariancie wymaga porównania każdej grupy z wszystkimi dostępnymi skutkując kwadratową złożonością czasową $O(P^2)$~$O(M^2)$. Wykrywanie krawędzi jest prostym liniowym przejrzeniem wszystkich grup (złożoność O(P)~O(M)). Kwadratowa złożoność pamięciowa $O(P^2)$ tworzenia krawędzi wynika z możliwości powstania grafu gęstego. Para procedur sklejania prostych sekwencji oraz ich identyfikacji bazuje na algorytmie przeglądania grafu w głąb (DFS) i charakteryzuje się sumaryczną liniową złożonością czasową O(S+E) względem liczby stanów i krawędzi. Po odpowiednich podstawieniach wartości na podstawie

złożoności pamięciowych tworzenia stanów i krawędzi, złożoność czasową można uprościć do $O(M+M^2) \sim O(M^2)$. Wykrywanie stanu bazowego zakłada przejrzenie wszystkich krawędzi powodując złożoność czasową $O(E) \sim O(M^2)$. Wykrywanie cykli również charakteryzuje się liniową złożonością czasową $O(P) \sim O(M)$- polega na przeglądaniu wykrytych i oznaczonych grup. Maksymalna liczba cykli jest równa połowie liczby grup, przy założeniu, że stan bazowy etykietuje co drugą grupę, a więc złożoność pamięciowa jest również liniowa O(P). Procedura grupowania cykli w klasy podobnych cykli generuje złożoność czasową pesymistyczną $O(C^2)$. Upraszczając wyrażenie analogicznie jak dla poprzednich przykładów, złożoność czasową grupowania cykli można zapisać jako $O(M^2)$. Sumując złożoności czasowe wszystkich etapów algorytmu możliwe jest wyznaczenie złożoności czasowej całego algorytmu. Sumaryczna złożoność czasowa algorytmu względem parametru M (liczby segmentów) wynosi: $O(M) + O(M^2) + O(M) + O(M^2) + O(M) + O(M^2) + O(M^2) + O(M) + O(M^2)$. Zgodnie z notacją O(n) wynikiem takiej sumy jest złożoność czasowa o największym wymiarze, w tym wypadku jest to złożoność $O(M^2)$. Analogicznie złożoność pamięciowa algorytmu wynosi: $O(M^2)$.

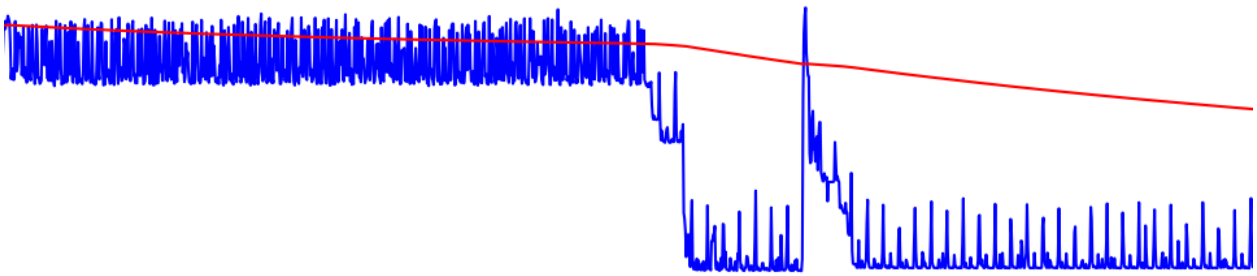
Dobór parametrów algorytmu może być wykonany na 3 sposoby: 1) bazując na poprzednich analizach (np. kopiując wartości parametrów z poprzedniej iteracji analizy tego samego lub podobnego systemu), 2) bazując na praktycznym doświadczeniu eksperta, który spodziewa się wykrycia konkretnych klas cykli na podstawie wiedzy o poziomach oszczędzania energii czy też periodyczności zadań badanego systemu, 3) iteracyjnie poszukując wartości parametrów, bazując na reprezentatywnym fragmencie szeregu czasowego. Metody 2) i 3) wykorzystują podejście heurystyczne. Na podstawie znanych cech algorytmu można zdefiniować wskazania czy reguły, które ułatwiają ustalenie początkowych wartości parametrów. Wartość n rozmiaru segmentu powinna przyjąć taką wartość minimalną, aby segment pokrywał fragment sygnału/profil energetyczny reprezentujący stan bazowy, który opisuje periodyczny proces działający w tle. Początkową wartość n można na przykład oszacować korzystając ze wzoru $\frac{k * T_{max}}{sample\_period}$. Wartość T_{max} jest największym okresem aktywności procesu należącego do zbioru periodycznych procesów tła. wartość *sample_period* jest okresem próbkowania, a wartość k jest liczbą naturalną, która powinna być tak dobrana, aby segment zawierał przynajmniej kilka wystąpień periodycznego procesu. Górne ograniczenie wartości n może być określone na podstawie najkrócej trwającego zdarzenia celowego. Jako zdarzenie celowe możemy zdefiniować przeciwieństwo stanu bazowego- jest to zdarzenie, które algorytm sklasyfikuje jako wystąpienie klasy cyklu. Algorytm wykorzystuje parametry progowe (*avg_eps, min_eps, max_eps, ...*) do klasyfikacji podobnych segmentów,

stanów i cykli oraz rozróżnienia różnych stanów i aktywności. Przyjęcie zbyt dużych wartości progowych zmniejsza wykrywalność cykli, skutkując mniejszą liczbą cykli i prawdopodobnie wykrytych zdarzeń. Zdarzenia, które trwają relatywnie krótko, mają charakterystykę piku sygnału i mogą być rozróżnione za pomocą parametru *max_eps*. Natomiast w przypadku stanów energetycznych, których czas trwania po wystąpieniu jest długi oraz podnosi pobór prądu, parametr *min_eps* pozwala na wykrycie anomalii (klasy cyklu).

Poszukiwanie parametrów algorytmu może być wspierane przez oprogramowanie. Przykładowo ekspert może oznaczać na wykresie badanego szeregu czasowego poszczególne fragmenty, które jego zdaniem reprezentują procesy tła lub przykłady interesujących zdarzeń. System wspierający na podstawie tych fragmentów obliczy wartości statystyczne i dobierze przykładowe parametry algorytmu tak, aby rozróżnić oznaczone grupy od siebie w ramach procesu wykrywania klasy cykli i stanu bazowego.

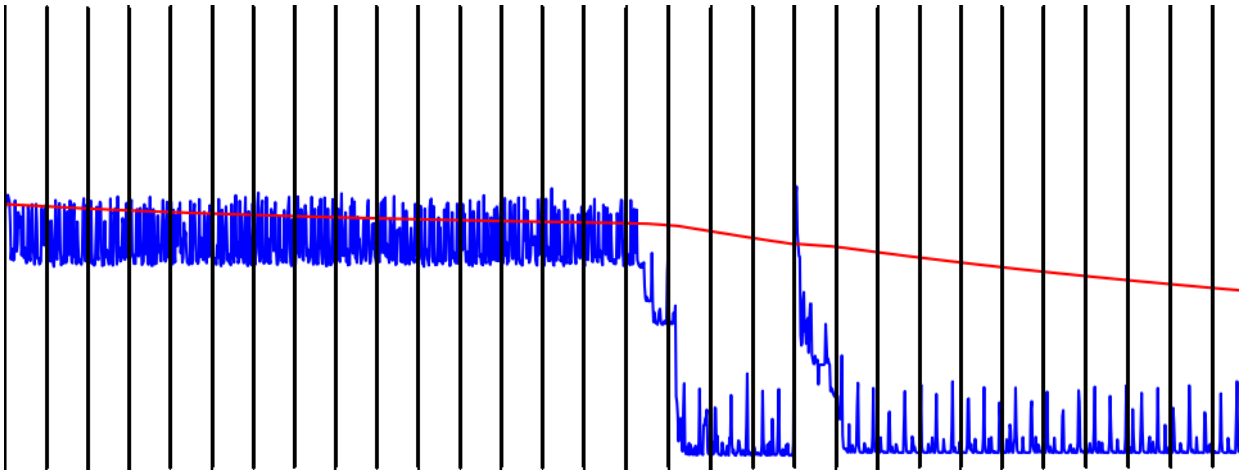
3.1.5 Ilustracja wykonania algorytmu

W tym podrozdziale zaprezentowany został przykładowy przebieg algorytmu dla danych wejściowych w postaci ciągu próbek pomiaru prądu pobieranego przez badane urządzenie. Wykresy poboru prądu są uproszczone w celu zwrócenia uwagi na najważniejsze zmiany wizualizujące kolejne etapy algorytmu. Rysunek 4 przedstawia wykres fragmentu poboru prądu przez badane urządzenie. Pomiar realizowany jest na zaciskach bateryjnych urządzenia. Czerwoną linią zaznaczono wykres średniej skumulowanej poboru prądu. Na osi pionowej jest wartość poboru prądu, na poziomej numer próbki. Każda z próbek była mierzona z okresem próbkowania równym 200us.



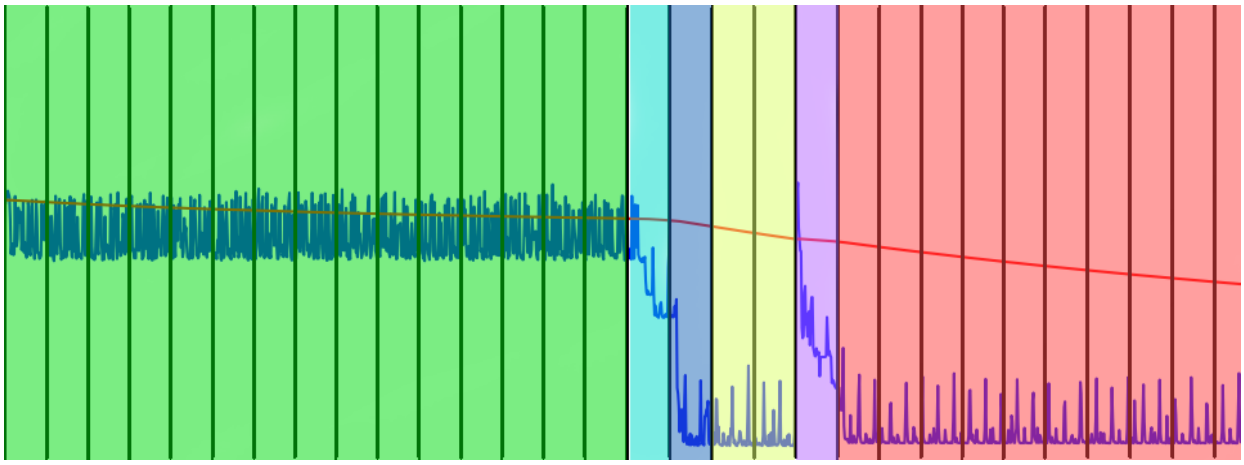
Rysunek 4. Wykres poboru prądu teoretycznego systemu wbudowanego

W ramach działania algorytmu próbki dzielone są na segmenty o stałej długości 250 próbek. Proces ten został zobrazowany na rysunku 5 za pomocą pionowych czarnych linii.



Rysunek 5. Wykres poboru prądu wraz z oznaczonym podziałem próbek na segmenty (pionowe linie)

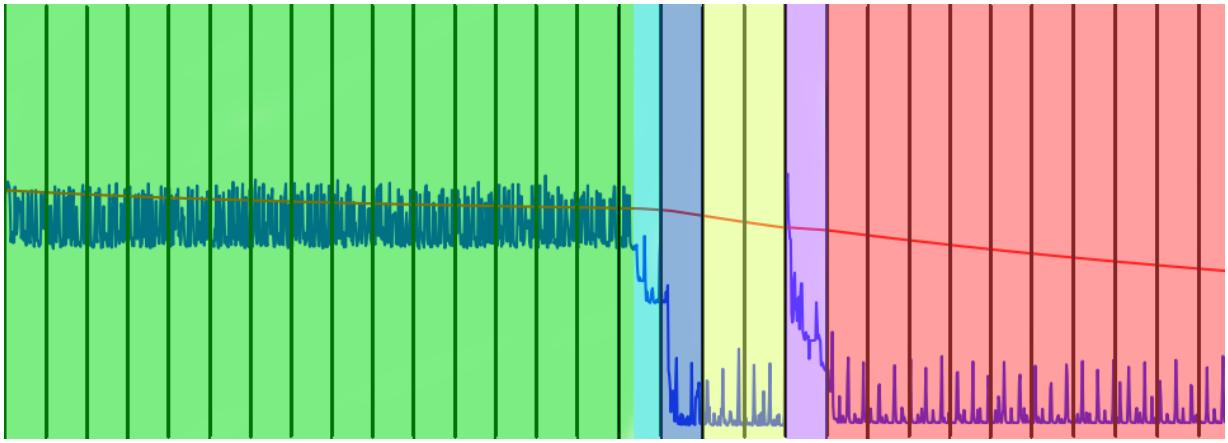
Następnie algorytm dokonał wstępnego przydziału segmentów do grup (grupowanie). Dla każdego z segmentów wyliczono cechy: średnia arytmetyczna, minimum, maksimum, odchylenie standardowe. Parametry (1) *avg_eps*, (2) *deviation_eps*, (3) *min_eps*, (4) *max_eps* równe odpowiednio: 3mA, 5mA, 1mA, 10mA pozwoliły przypisać segmenty do grup, co zostało zobrazowane na rysunku 6 w postaci pokolorowanych fragmentów wykresu. Kolorem zielonym oznaczono segmenty należące do grupy 1, błękitnym do 2, niebieskim do 3, żółtym do 4, fioletowym do 5 i czerwonym do ostatniej grupy 6.



Rysunek 6. Wykres poboru prądu wraz z podziałem na grupy (kolor tła określa przynależność do grupy)

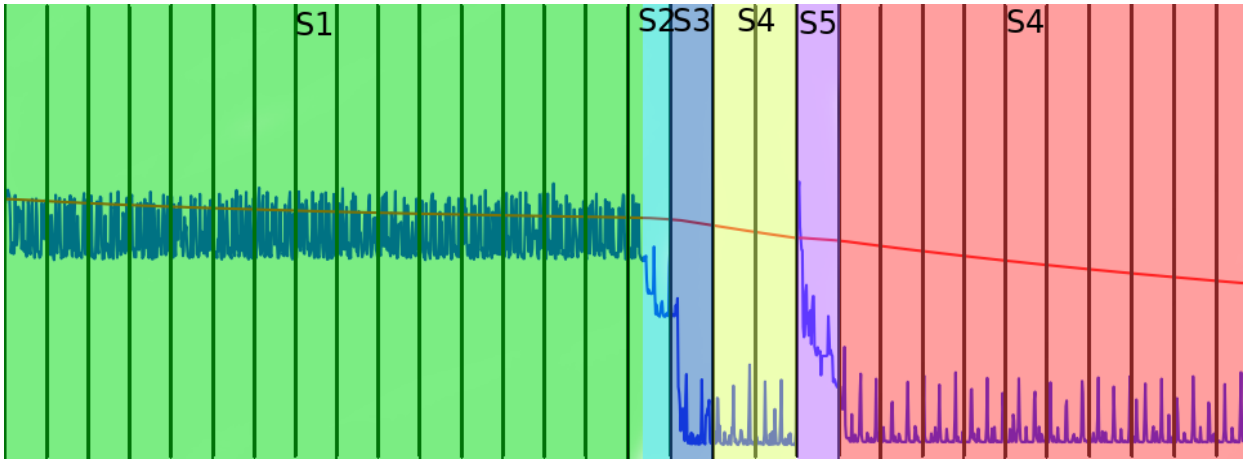
Zwiększenie dokładności dopasowania segmentów do grup następuje poprzez podział na mniejsze podsegmenty z segmentów granicznych oraz dołączeniu ich do grup sąsiadujących. Przykład tego kroku algorytmu zobrazowano na rysunku 7. Segment z grupy 2 połowicznie spełniał wymagania na segment graniczny tzn. sąsiadował z jedną grupą o rozmiarze większym

niż 1 segment. W celu ukazania opisywanego kroku segment z grupy 2 został w ramach procedury rekurencyjnej podzielony na dwa podsegmenty. Najbardziej skrajny lewy podsegment został przydzielony do grupy 1 w wyniku porównania obliczonych wcześniej cech. Pozostałe podsegmenty utworzyły pojedynczy segment, który stanowi grupę 2.



Rysunek 7. Wykres poboru prądu wraz z podziałem na grupy po procesie poprawy dokładności

Ostatnim etapem było przypisanie etykiet grupom w celu detekcji stanów. W tym celu algorytm korzystał z wcześniej wyliczonych cech grup oraz wartości parametrów granicznych. Nazwy stanów są przypisywane do każdej grupy w kolejności rosnącej. Każda grupa jest weryfikowana w poszukiwaniu podobnych grup, które oznaczane są tą samą nazwą stanu i nierozpatrywane w kolejnych iteracjach opisywanego kroku algorytmu. Na rysunku 8 oznaczono nazwy stanów przypisane do grup. Grupa 4 i grupa 6 zostały zaklasyfikowane jako instancje tego samego stanu.



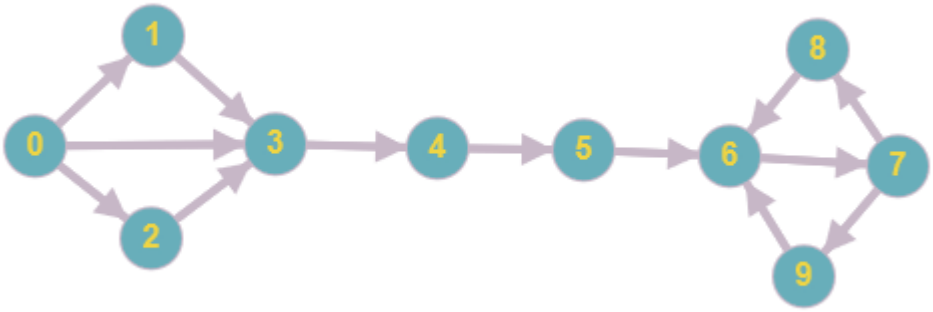
Rysunek 8. Wykres poboru prądu wraz z podziałem na segmenty, grupy oraz stany

Ostatni krok algorytmu generuje ponadto graf przejścia między stanami. Na rysunku 9 przedstawiono graf tranzycji między stanami powstały podczas detekcji stanów. Z grafu można

odczytać, że z każdego stanu można dokonać tranzycji do stanu o indeksie o jeden większym. Wyjątkiem jest stan 5, z którego można osiągnąć stan 4.

Rysunek 9. Graf tranzycji między wykrytymi stanami

Przykład szeregu czasowego opisywanego powyżej można zapisać w postaci sekwencji symboli Sx_g^k, gdzie x jest numerem segmentu, k etykietą stanu, a g numerem grupy. Przykład z rysunku 5 przyjmuje postać: $S1_1^1 S2_1^1 S3_1^1 S4_1^1 S5_1^1 S6_1^1 S7_1^1 S8_1^1 S9_1^1 S10_1^1 S11_1^1 S12_1^1 S13_1^1 S14_1^1$ $S15_1^1 S16_1^1 S17_2^2 S18_3^3 S20_4^4 S21_4^4 S22_5^5 S23_6^4 S24_6^4 S25_6^4 S26_6^4 S27_6^4 S28_6^4 S30_6^4 S31_6^4 S32_6^4$



Rysunek 10. Przykład innego możliwego grafu tranzycji między stanami

Gdyby graf stanów był reprezentowany przez graf na rysunku 10, proces scalania i usuwania stanów połączyłby stany 4 i 5 w nowy stan 4 zachowując przy tym tranzycję stanu 5.

3.2 Przykład analizy szeregu czasowego poboru prądu

W tym podrozdziale jest zaprezentowane studium przypadku procesu analizy szeregu czasowego poboru prądu komercyjnego systemu telemedycznego w postaci mobilnego holtera EKG (*Holter #1*). Zaprezentowany proces został zrealizowany manualnie podczas praktyki zawodowej autora bez użycia opisywanego algorytmu. Problemy oraz obserwacje zauważone podczas analizy i optymalizacji poboru prądu stały się inspiracją do przygotowania algorytmów opisanych w pracy. W podrozdziale 3.2.1 opisana jest charakterystyka badanego urządzenia

wraz ze specyfikacją celu analizy zaprezentowanej w podrozdziale 3.2.2. Opisane urządzenie wykorzystywane było także podczas doświadczeń i ewaluacji algorytmu. Wyniki doświadczeń zawarte zostały w rozdziałach 5.1 i 5.2.

3.2.1 Założenia

Jednym z wymagań nowej generacji mobilnego holtera EKG (*Holter #1*) było działanie na w pełni naładowanej baterii przez 36h. Osiągnięcie tego celu jest możliwe wtedy, gdy pobór energii przez urządzenie jest na odpowiednio niskim poziomie, tzn. średnia wartość pobieranego natężenia prądu za okres 24h przy stałym napięciu zasilania 4V wynosi ok. 32 mA. Na pewnym etapie rozwijania nowego urządzenia wymaganie to uzyskało priorytet najwyższy, gdyż podczas kompleksowych testów użytkowych czas działania urządzenia na zasilaniu bateryjnym okazał się niezgodny ze specyfikacją.

Oprogramowanie uruchamiane na *Holterze #1* działało na finalnej wersji sprzętu realizując w różnym stopniu wszystkie wymienione w specyfikacji funkcje takie jak: próbkowanie, agregacja, wstępna analiza sygnału EKG, uruchamianie aplikacji pozwalającej na kontrolowanie przebiegu badania, informowanie o stanie samopoczucia czy zauważonych objawach chorób przez pacjenta, a także notyfikacje o stanie urządzenia i wymaganych działaniach np. wymianie baterii. Oprogramowanie bazuje na systemie Android 4.4 oraz pojedynczej aplikacji realizującej funkcje systemu. Uruchomienie systemu Android na sprzęcie *Holter #1* wymagało przeprowadzenia procesu portowania, czyli przystosowania kodu jądra i warstwy pośredniej systemu Android do procesora z rodziny Omap3xx (rdzeń ARM Cortex-A8). Ponadto oprogramowanie obsługuje modem LTE Telit i dodatkową baterię zapasową na stałe wlutowaną w płytę urządzenia. W ramach przystosowania systemu do procesora modyfikacji uległa także polityka zarządzania energią, na która składają się: 1) zarządzanie częstotliwością taktowania zegara rdzenia procesora (MPU) oraz jego napięciem zasilania, 2) zarządzanie przejściem w stan pełnego uśpienia procesora i jego wybudzeniu, 3) zarządzanie zmianami stanów domen mocy procesora, 4) zarządzanie włączaniem i wyłączaniem ekranu urządzenia oraz innymi urządzeniami zewnętrznymi względem CPU, 5) zarządzanie usypianiem i wybudzaniem aplikacji 6) dodanie obsługi płynnego przejścia na baterię zapasową oraz politykę funkcjonowania bez baterii głównej.

Pierwszy obszar polityki oparty jest na przełączaniu się pomiędzy punktami wydajnościowymi (ang. Operation Performance Point- OPP) przez procesor. Pojedynczy punkt jest zdefiniowany jako para: częstotliwość taktowania zegara rdzenia oraz napięcie zasilania rdzenia przy zadanej częstotliwości. Lista punktów OPP jest zdefiniowana w pliku DTS (ang.

Device Tree Source). Kod jądra systemu Linux dla procesorów ARM zawiera jedynie logikę sterowników sprzętu jak i również logikę konfiguracji procesora. W dużym uproszczeniu logika działania sterownika systemowego można zamodelować jako: a) konfigurację sprzętu poprzez zapisy do odpowiednich rejestrów czy obszarów pamięci, b) reagowanie na przerwania poprzez odczyt i zapis do odpowiednich obszarów przestrzeni adresowej oraz c) świadczenie usług na rzecz przestrzeni użytkownika. Do poprawnego funkcjonowania kodu jądra w funkcjach a) i b) wymagane są m.in. konkretne adresy rejestrów i innych parametrów specyficznych dla konkretnego sprzętu. Adresy te są zdefiniowane są w pliku DTS. Plik DTS jest kompilowany za pomocą kompilatora DTC do binarnej struktury danych. Następnie w procesie uruchamiania urządzenia program ładujący jądro systemu do pamięci ładuje także skompilowany plik DTS, a jako argument wejściowy jądra podaje adres do obszaru pamięci z plikiem DTS. W rezultacie każdy sterownik czy fragment kodu jądra ma dostęp do konfiguracji sprzętowej systemu. Jednym z takich modułów jest *Linux CpuFreq Scalling Governor*. Jest to podsystem, który decyduje o przejściu procesora na inny punkt OPP. W systemie w zależności od architektury dostępnych jest kilka polityk zarządzania punktami OPP np. *Performance*- utrzymuje stale maksymalną częstotliwość, *Powersave*- stara się utrzymać minimalną częstotliwość i napięcie, *OnDemand*- stara się utrzymać minimalną częstotliwość i napięcie oraz monitoruje wykorzystanie procesora tak, by w przypadku skokowej zmiany wartości obciążenia przejść na wyższy OPP, *Userspace*- w tej konfiguracji oprogramowanie przestrzeni użytkownika może samodzielnie wybierać odpowiedni punkt OPP. W przypadku systemów, w których profil obciążenia CPU jest z góry znany albo łatwo przewidywalny najefektywniejszym trybem jest *Userspace*, gdyż przełączanie pomiędzy punktami OPP przed zmianą obciążenia pozwala uniknąć strat energii/wydajności związanych z latencją występującą w przypadku polityki *OnDemand*. Czas ten wynika z potrzeby wykrycia przez system wzrostu obciążenia jak i też obliczenia wartości tego wzrostu. Typowym systemem, w którym można przewidzieć obciążenie jest klasa systemów wbudowanych, do której zalicza się mobilny holter EKG. Z przyczyn opisanych powyżej dyskutowany holter działa wg polityki *Userspace*. W ramach tej polityki oprogramowanie przestrzeni użytkownika poprzez pliki w systemie plików sysfs dokonuje zmian częstotliwości taktowania rdzenia, a jądro dostosowuje napięcie zasilania wg danych z pliku DTS. O wyborze punktów decyduje serwis warstwy pośredniej *DisplayManager*, który ma dostęp do informacji m.in. o stanie ekranu. Jeśli ekran jest włączony, serwis zarządza wejście procesora na najwydajniejszy OPP, w celu bezproblemowego rysowania interfejsu użytkownika przez biblioteki graficzne aplikacji. Po wyłączeniu ekranu, a więc przejściu aplikacji w tryb uśpienia, następuje zmiana

na drugi w kolejności punkt OPP. Ze względu na konfigurację funkcjonowanie urządzenia sprowadza się do przełączania pomiędzy częstotliwościami i napięciami: OPP1 (1Ghz; 1,45V) oraz OPP2 (600Mhz, 1.25V).

Drugi obszar polityki zarządza pełnym uśpieniem procesora (ang. suspend). W momencie usypiania wszystkie sterowniki przechodzą w stan niskiego poboru mocy, który umożliwia także wznowienie pracy bez potrzeby pełnej reinicjalizacji sprzętu. Następnie procesor wyłącza się zostawiając uruchomiony jedynie sprzęt umożliwiający wybudzenie systemu np. zegar generujący przerwanie do wybudzenia systemu. Suspend jest to stan systemu, w którym procesor jest wyłączony, a pamięć RAM jest w stanie podtrzymania zasilania. W momencie wybudzenia następuje minimalna reinicjalizacja sprzętu/ aktualizacja stanu oprogramowania w celu zachowania spójności stanów między oprogramowaniem i sprzętem. W typowym jądrze systemu Linux suspend jest to stan S3, czyli stan, do którego system przechodzi, gdy wszystkie wątki i procesy są zawieszone w oczekiwaniu na zdarzenie (np. synchronizacja między wątkowa) lub upływ czasu. W systemie Android do jądra dodano specjalny mechanizm blokad (*wake locks*). Wpis dowolnego ciągu znaków (np. nazwy blokady) do pliku pod ścieżką */sys/power/wake_lock*, spowoduje uniemożliwienie uśpienia systemu. Mechanizm jest wykorzystywany przede wszystkim przez serwisy i aplikacje przestrzeni użytkownika, aby umożliwić np. odbieranie danych w tle przez sieć WiFi, czy też obliczenia w ramach aplikacji uruchomionych na wirtualnych maszynach *JAVA-y*. Mechanizm usypiania wymaga odpowiedniego obsłużenia zdarzeń przejścia do uśpienia i wybudzenia przez wszystkie sterowniki systemu, a także warstwę HAL (ang. Hardware Abstraction Layer) przestrzeni użytkownika systemu Android. Ze względu na niedoskonałość tychże elementów w opisywanym systemie oraz nakłady pracy, których wymagałoby dopracowanie modułów oprogramowania, *Holter #1* blokuje możliwość przejścia w stan *suspend* poprzez wpis do pliku *wake_lock* przy starcie systemu.

Zarządzanie przejściami pomiędzy stanami domen mocy realizowane jest poprzez jądro systemu Linux. Sterowniki systemowe wykorzystują API (ang. Application Programming Interface) włączając lub wyłączając sygnał zegarowy dla poszczególnych fragmentów procesora. System monitorując stan zegarów jest w stanie określić które domeny mocy mogą zmienić stan. W momencie, gdy zegary wszystkich modułów sprzętowych wchodzących w skład domeny mocy są wyłączone lub są w stanie odciętym, jądro dokonuje wpisu do rejestru konfiguracyjnego danej domeny z żądaniem przejścia w stan obniżonego poboru mocy np. *inactive*, *retention* lub *off*. Następnie sprzęt sprawdza czy żądanie jest możliwe do wykonania i jeśli wszystkie warunki są spełnione przełącza domenę w stan obniżonego poboru prądu.

W sytuacji, gdy sterownik zostanie wybudzony i włączy on zegar, dana domena zostaje ponownie włączona w stan aktywny i w zależności od poziomu obniżonego poboru mocy może wymagać: rekonfiguracji sprzętu (ze stanu OFF) lub załadowania stanu z pamięci RAM (ze stanu OFF, ale z pamięcią w stanie *retention*). W przypadku stanu *RETENTION* uruchomienie domeny jest bez kosztowe. Jednym z parametrów warunkujących przejścia między stanami domen mocy są zależności usypiania i wybudzania domen mocy. W *Holterze #1* konfiguracja zależności domen mocy początkowo wskazywała na silną zależność domen CORE oraz PER, a także innych pomniejszych domen.

Warstwą decyzyjną czwartego obszaru zarządzania energią są serwisy warstwy pośredniczącej oraz HAL. Elementami zarządzanymi są: podświetlenie ekranu, sterownik kontrolera ekranu, sterownik kontrolera panelu dotykowego, podsystem USB oraz modem LTE. Ekran kontrolowany jest poprzez serwis *DisplayManager*, który poprzez wpisy do odpowiednich plików w systemie SYSFS ma możliwość: włączenia/wyłączenia podświetlenia ekranu, wstrzymania wysyłania ramek obrazu do ekranu, zablokowania generowania przerwań związanych z dotykiem ekranu i jego obsługą. Polityka działania *DisplayManager*-a sprowadza się do prostych zależności np. żądanie wygaszenia ekranu skutkuje: wyłączeniem podświetlenia, wyłączeniem wysyłania ramek danych, zablokowaniem pomiaru dotyku dla kontrolera panelu dotykowego. Analogicznie, gdy wykryte zostanie wciśnięcie przycisku głównego urządzenia, zostanie wybudzona aplikacja, a *DisplayManager* uruchomi ekran. Szczególnie ważna jest kontrola panelu dotykowego, gdyż w sytuacji wyłączenia ekranu i uśpienia aplikacji, pomiar dotyku i generowanie przerwań jest bezcelowe. Taka sytuacja występuje np. gdy *Holter #1* trzymany jest w kieszeni. W opisywanym stanie procesor jest wybudzany co 20ms, obsługuje przerwanie, odczytuje dane z kontrolera dotyku po interfejsie I2C tylko po to by stwierdzić, że dane te są niepotrzebne, gdyż aplikacja jest uśpiona. Wyłączenie dotyku w momencie uśpienia aplikacji pozwala zaoszczędzić cenne mA natężenia prądu pobieranego z baterii przez urządzenie. *Holter #1* wykorzystuje modem LTE, który podłączony jest do procesora za pośrednictwem interfejsu USB. Moduł USB posiada tryb automatycznego usypiania prowadzący do wyłączenia domeny USBHOST po 2 sekundach bezczynności na magistrali. Za pomocą plików w systemie SYSFS można manualnie sterować przechodzeniem w tryby uśpienia lub całkowicie je zablokować. Domyślnie system operacyjny steruje modułem USB. Oprogramowanie Vendor RIL (Radio interface Layer), które zarządza modemem LTE ma możliwość także sterować modułem USB oraz piniami GPIO (ang. General Purpose Input Output) podłączonymi do modemu LTE. Vendor RIL dla modułu Telit-866 został w pełni przygotowany przez autora wraz z zespołem deweloperskim. Gdy zostanie

włączony tryb samolotowy (ang. airplane mode) vendor RIL wyłącza modem za pośrednictwem tranzystora kluczującego zasilanie modemu. Tryb samolotowy jest włączany i wyłączany przez aplikację, w zależności od potrzeby łączności ze światem zewnętrznym.

Aplikacja jest wybudzana w momencie wciśnięcia przycisku głównego urządzenia. Usypianie aplikacji następuje na żądanie jej samej w sytuacji, gdy np. pacjent skończy raportować objaw lub wciśnie ponownie przycisk główny. Uśpienie aplikacji polega na wstrzymaniu jej wykonywania w ramach wirtualnej maszyny uruchomionej w systemie Android.

Ostatnim obszarem, który rozszerza politykę zarządzania energią jest system zarządzania bateriami. System zrealizowany jest w postaci serwisu *HealthDaemon*. Typowy system Android obsługuje jedynie pojedynczą baterię. W *Holterze #1* zastosowano dodatkową baterię zapasową na stałe wlutowaną w płytę urządzenia. W momencie, gdy zostanie wyjęta bateria główna (lub jej napięcie spadnie do poziomu mniejszego od 3.5V) system przełącza się na baterię zapasową, na której powinien funkcjonować przez czas ok. 10 minut, pozwalając wymienić baterię główną bez potrzeby przerywania badania. Obsługę dodano poprzez rozwinięcie oryginalnego *HealthDaemona* o dodatkowe struktury danych i logikę opisującą nową baterię. Projekt płyty PCB został wzbogacony o układ scalony automatycznie przełączający źródło zasilania przy wymianie baterii. Układ informuje o wykrytych zmianach stanu przerwaniem. Po stronie jądra dodano sterownik drugiej baterii. Informacja o zmianie baterii jest propagowana do aplikacji, która decyduje o zmianie profilu działania urządzenia np. poprzez natychmiastowe wyłączenie modemu LTE.

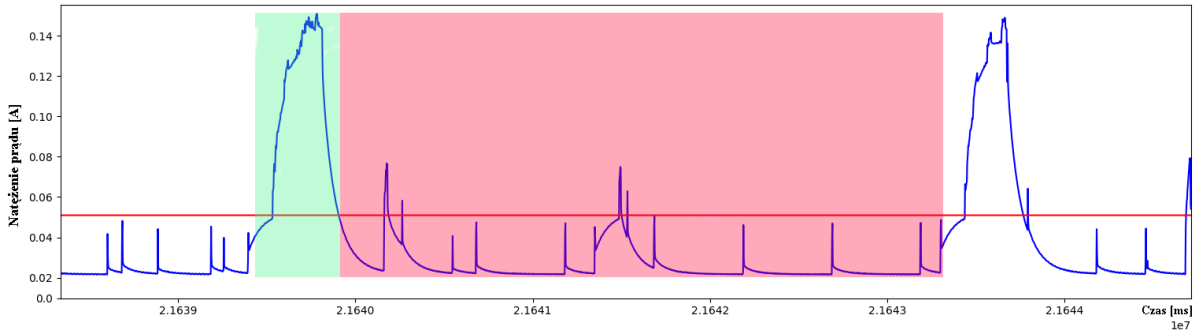
Innymi czynnikami wpływającymi na profil poboru mocy przez *Holter #1* są: częstotliwość wysyłanych danych na serwer (modem LTE podczas badania przy idealnych parametrach sieci LTE włączany jest co 1h 10m na czas wysyłania danych o rozmiarach od kilku

do kilkudziesięciu MB), częstotliwość zapisywanych logów na karcie SD (ok. 1/15s.), częstotliwość przesyłania próbek przez dodatkowy mikrokontroler MSP430 do procesora głównego i ich analizy (1/4s.).

Holter #1 w konfiguracji opisanej powyżej przy napięciu zasilania 4V początkowo pobierał prąd o średnim natężeniu 57mA, przy minimum na poziomie 21mA. Jest to punkt wyjścia dla opisanego poniżej procesu optymalizacji energetycznej systemu.

3.2.2 Optymalizacja zarządzania energią w urządzeniu Holter #1

Pierwszym obszarem, który podlegał analizie efektywności wykorzystywania energii z baterii była konfiguracja punktów OPP. W przypadku stałego obciążenia procesora w trybie aktywnym wraz ze spadkiem częstotliwości sygnału zegarowego taktującego rdzeń proporcjonalnie spada wartość pobieranej mocy. Ponadto mniejsza częstotliwość sygnału zegarowego pozwala obniżyć wartość napięcia zasilania rdzenia, co skutkuje obniżeniem poboru mocy proporcjonalnie do kwadratu różnicy napięcia. Na rysunku 11 przedstawiono wykres natężenia prądu pobieranego przez *Holter #1* podczas agregacji i wstępnej analizy sygnału EKG.

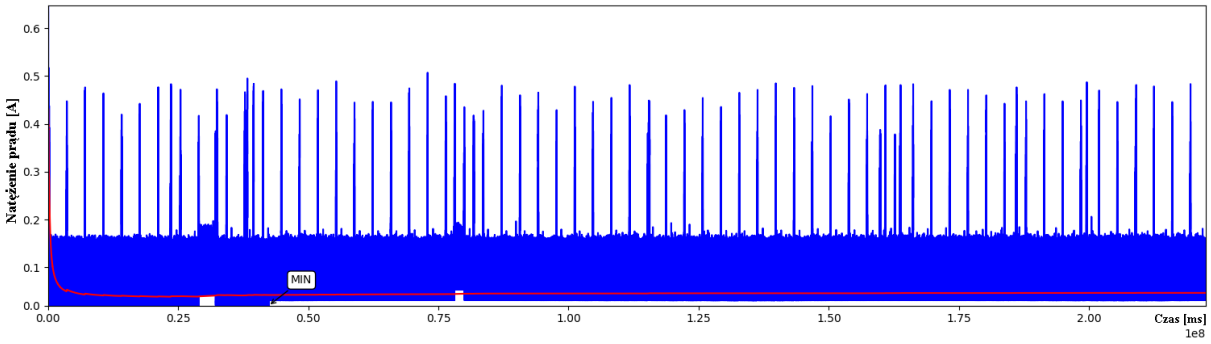


Rysunek 11. Wykres natężenia prądu podczas agregacji i analizy sygnału EKG przez *Holtera #1*

Czerwoną linią zaznaczono średnią wartość natężenia prądu podczas 18 godzinnego badania. Podczas badania procesor główny co 4s sekundy odbiera zbuforowane próbki sygnału EKG przez mikrokontroler MSP430 za pośrednictwem interfejsu SPI, następnie procesor główny wykonuje operacje przetwarzania wstępnego oraz klasyfikacji morfologii sygnału EKG. Sygnał jest kompresowany i zapisywany do pliku na karcie SD. Proces ten oznaczony jest na wykresie kolorem zielonym. Na wykresie czerwonym kolorem oznaczono fragment wartości natężenia prądu, gdy urządzenie oczekuje na kolejny odczyt sygnału EKG. W tym okresie procesor główny jest w stanie niskiego poboru energii i nie wykonuje instrukcji. Pojedyncze krótkie skoki natężenia spowodowane są przez wybudzenia związane z obsługą systemu operacyjnego Android oraz agregacją próbek przez MSP430. Z wykresu wynika, że podczas przetwarzania przy częstotliwości sygnału zegarowego procesora 800Mhz natężenie osiąga wartość maksymalną 154 mA. Zmniejszając częstotliwość procesora natężenie prądu w obszarze zielonym spadnie, jednak równocześnie czas przetwarzania ulegnie zwiększeniu, a czas przebywania procesora w stanie niskiego poboru prądu spadnie. Producent procesora w zmianach zaaplikowanych do jądra systemu Linux zaleca listę punktów OPP, między innymi punkty 1) 300MHz, 2) 600MHz oraz 3) 800MHz. Bazowo w prezentowanym przykładzie

procesor działał w stanie punktu 3). Podczas testowania punktu 1) okazało się, że urządzenie nie było w stanie dokończyć przetwarzania danych przed czasem 4s, skutkując stratami co drugiej paczki próbek sygnału EKG. Z drugiej strony okresu odczytywania próbek nie można wydłużyć, ze względu na ograniczoną pamięć mikrokontrolera MSP430. Wyniki pomiarów działania urządzenia w punkcie 2) pokazały, że wartość maksymalna spadła do poziomu 112mA, a czas przetwarzania uległ wydłużeniu. Ostatecznie średnia wartość natężenia prądu spadła z 57 mA do 44mA.

Z pomiarów poboru prądu można zauważyć w ciągu ok. 5-11 godzin od uruchomienia badania wzrasta wartość lokalnie minimalna natężenia prądu. Tą sytuację można zauważyć na wykresie z rysunku 12. Strzałka z etykietą MIN wskazuje na miejsce na osi czasu, od którego wartość minimum wzrasta do poziomu 46mA.



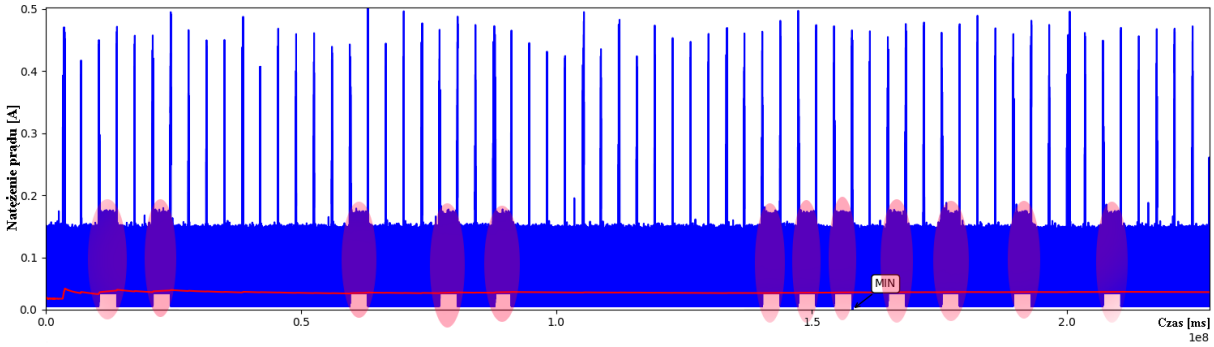
Rysunek 12. Wykres natężenia prądu *Holtera #1* podczas badania EKG

Za wartość lokalnie minimalną uznajemy minimalne natężenie prądu podczas pojedynczego okresu odczytu i przetwarzania sygnału EKG wraz z etapem oczekiwania na kolejną sekwencję. Wykonano 10 pomiarów na trzech różnych próbkach systemu trwających 18h i w każdej sytuacji wystąpiło podobne zdarzenie w postaci wzrostu wartości lokalnie minimalnego natężenia prądu z poziomu 21mA do 40mA. Urządzenie przez znaczną część czasu badania przebywa w stanie niskiego poboru energii co oznacza, że wzrost minimum ma znaczący wpływ na średnią wartość natężenia prądu za okres całego badania. Niestety analiza z wykorzystaniem systemu *FTrace*[4] oraz innych narzędzi nie pomogła wskazać dokładnej przyczyny takich zdarzeń. Z tego powodu do systemu operacyjnego dodano funkcję zapisującą zawartość wszystkich rejestrów konfigurujących domeny mocy oraz wszystkie mechanizmy kontrolowania poboru energii w procesorze Omap3xxx. Zrzut danych był wykonywany do plików tworzonych na karcie SD. Każdy z plików w nazwie zawierał znacznik

[4] FTrace- narzędzie systemu linux pozwalające na agregację wydajnych logów binarnych opisujących zdarzenia wewnątrz jądra i sterowników systemu na osi czasu.

czasowy. Pliki były tworzone co 20 minut. Po zakończeniu jednej próby trwającej 18 godzin, wykonano analizę porównawczą utworzonych plików. Wyniki pokazały, że do pewnego momentu konfiguracja nie ulegała zmianie. Następnie wystąpiło nieokreślone zdarzenie, które spowodowało ustawienie bitu w rejestrze konfigurującym mechanizm *SmartReflex*. Nowa wartość tego rejestru utrzymała się do końca trwania pomiaru. Z tych danych można także uzyskać dwa znaczniki czasu (znacznik z pliku z wartością rejestru *SmartReflex* niezmienioną od początku oraz czas w pierwszym pliku z nową zawartością rejestru), które definiują obszar poszukiwań innych skutków tego zdarzenia. Wykres poboru prądu potwierdza związek skoku minimum lokalnego natężenia prądu, z opisywaną zmianą zawartości rejestru. *SmartReflex* jest mechanizmem sprzętowym wymagającym podstawowej konfiguracji z poziomu oprogramowania. *SmartReflex* monitoruje obciążenie CPU na zadanej częstotliwości taktowania sygnału zegarowego i w zależności od obciążenia modyfikuje napięcie zasilające rdzeń procesora. Napięcie to można zmodyfikować poprzez wydanie rozkazu układowi PMIC (ang. power managment integrated circuit) za pośrednictwem interfejsu I2C. Układ PMIC odpowiada między innymi za zasilanie odpowiednimi i konfigurowalnymi wartościami napięć różnych części urządzenia takich jak: rdzeń CPU, karta SD, pady wejścia-wyjścia itp. Sterownik *SmartReflex* był przenoszony z wersji jądra systemu 2.6 do nowej 4.4. Pomiędzy tymi wersjami występują duże zmiany w obszarze architektury zarządzania energią, definicji sprzętu oraz architektury przechodzenia systemu w stan niskiego poboru prądu. W wyniku testów okazało się, że najprostszym rozwiązaniem problemu jest wyłączenie komponentu *SmartReflex* i usunięcie sterownika z systemu. Ten prosty zabieg spowodował zniwelowanie problemu oraz spadek średniej wartości natężenia prądu do poziomu 37-39mA. Rozrzut wartości wynika z innego problemu opisanego w paragrafie poniżej.

Kolejnym problemem generującym podwyższony pobór energii były okresowe skoki wartości lokalnego minimum natężenia prądu. Opisywana sytuacja zaprezentowana jest na rysunku 13, który przedstawia wykres natężenia prądu w funkcji czasu (ms) podczas przeprowadzonego badania. Krótko trwające, ale wysokie skoki natężenia prądu powstały podczas wysyłania danych na serwer. Urządzenie włącza modem, otwiera połączenie FTP, wgrywa pliki z sygnałem EKG i adnotacjami na serwer, aby finalnie zamknąć połączenie. Dane wysyłane są co 70 minut. Na wykresie można zauważyć, że po zakończeniu części transmisji wartość lokalnego minimum natężenia prądu nie wracała do poziomu sprzed rozpoczęcia transmisji. Stan ten utrzymywał się do kolejnej transmisji danych. Opisywane fragmenty wykresu zostały oznaczone czerwoną elipsą.



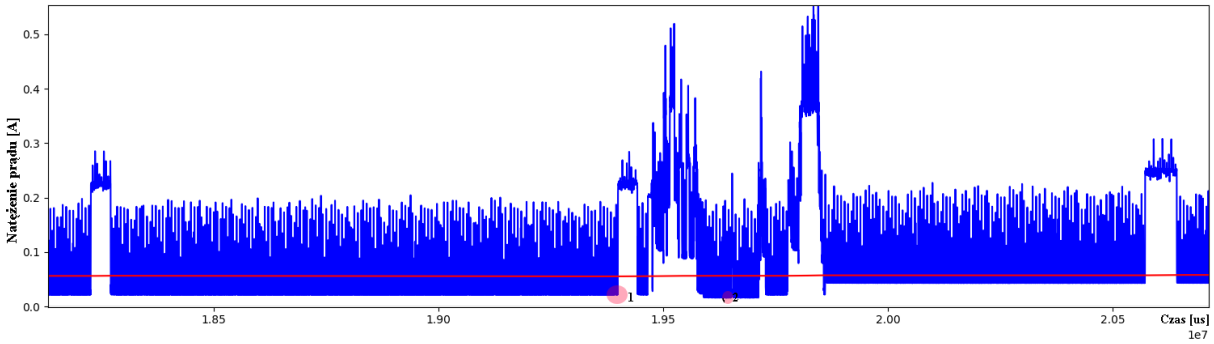
Rysunek 13. Wykres natężenia prądu pobieranego przez *Holtera #1* podczas badania EKG

Z przesłanek takich jak: występowanie wzrostu lokalnego minimum w momencie uruchomienia modemu, powrót do wcześniejszego minimum również po transmisji danych przez modem LTE można wywnioskować związek ze zmianami stanów oprogramowania i sprzętu, które odbywają się podczas uruchamiania modemu, wysyłania danych i wyłączania modemu. Bazując na doświadczeniu, które uzyskano podczas pracy nad stabilnością funkcjonowania magistrali USB, przez którą podłączony jest modem do procesora, pierwszą hipotezą wyjaśniającą zjawisko były problemy z niezawodnością kontrolera USBHOST oraz układów wspierających. Podczas implementacji obsługi modemu Telit wykryto niestabilność w utrzymywaniu urządzenia widocznego na szynie danych w momencie, gdy szyna USB jest wybudzana ze stanu uśpienia. Objawami problemów był częsty brak odpowiedzi na ramkę kontrolną generowaną przez kontroler. Bezpośrednim skutkiem było znikanie modemu z listy urządzeń podłączonych do szyny. Ostatecznie oprogramowanie traciło możliwość komunikacji z modemem LTE i ze zdalnym serwerem. Automatyczna sprzętowa kontrola usypiania modułu USBHOST została wyłączona z powodów błędów sprzętowych udokumentowanych w erracie do procesora Omap3xxx. Producent zaleca niekorzystanie z tego mechanizmu z powodu błędów w projekcie procesora. Usypianie jest więc sterowane przez oprogramowanie. Domyślnie po 2s braku transmisji USB innej niż odpowiedź na ramkę kontrolną, sterownik usypia kontroler jak i całą szynę. W tym momencie prawdopodobnie dochodzi do utraty spójności stanów między transreceiverem, a sterownikiem USB i kontrolerem USBHOST po stronie procesora. Transreceiver USB sterowany i konfigurowany jest poprzez interfejs równoległy ULPI przez procesor główny. Komunikacja w tej warstwie jest realizowana w pełni sprzętowo tzn. nie ma możliwości, aby oprogramowanie uzyskało informacje o komunikatach wysyłanych przez ULPI.

W momencie wystąpienia utraty spójności stanów transreceiver może być w dowolnym trybie poboru mocy. Rozwiązaniem opisywanych skoków wartości prądu okazał się

następujący mechanizm: po transmisji modem jest wyłączany oraz usuwany jest sterownik OMAP-EHCI (sterownik specyficzny dla kontrolera USB ECHCI dla procesorów OMAP). Po transmisji transreceiver utrzymywany jest w stanie uśpienia. Podczas rozpoczynania transmisji procedura polegała na: podniesieniu stanu linii ULPI_RESET do stanu nieaktywnego (wyłączenie resetu), załadowaniu sterownika OMAP_EHCI, uruchomieniu modemu i jego konfiguracji. Rozwiązanie okazało się skuteczne i spowodowało spadek średniej wartości natężenia prądu do poziomu 34mA. Wadą tego rozwiązania jest wyłączanie modemu, podczas gdy nie są wysyłane dane. Ponadto wydłużono czas uruchamiania modemu (ładowanie sterownika OMAP_ECHCI, ponowna enumeracja modemu na szynie USB).

Holter #1 najwięcej czasu podczas badania spędza w stanie niskiego poboru energii, dlatego wartość minimum i jego stabilność ma duży wpływ na średnią wartość natężenia prądu. Większość wartości lokalnego minimum natężenia prądu dla *Holtera #1* wynosiła ok. 21 mA. Jednak jak pokazuje rysunek 15, *Holter #1* ma możliwość osiągnąć stan, w którym układ pobiera minimum 14mA. Na rysunku 14 zaznaczono dwa punkty. Punkt pierwszy zaznaczony na rysunku oznacza wartość natężenia prądu 21mA. Następnie na wykresie widać skoki natężenia związane z włączeniem i inicjalizacją modemu. Punkt drugi oznaczono w miejscu, w którym pobór prądu spada do poziomu 14mA. Poziom natężenia utrzymuje się przez ok 2 min.

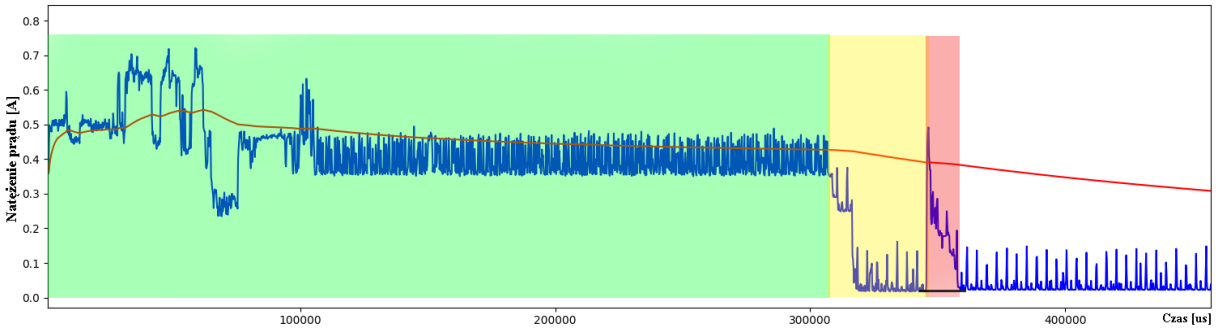


Rysunek 14. Wykres natężenia prądu pobieranego przez urządzenie, zawierający moment osiągnięcia minimum wartości natężenia (14mA)

Stan oznaczony cyfrą 2 był bardzo trudny do uchwycenia- w celu jego wykrycia przez ponad tydzień dwie próbki *Holtera #1* wykonywały ciągłe badanie. Trudność w uchwyceniu tego stanu wynikała z rzadkiej kombinacji stanów po uruchomieniu modemu. Prawdopodobna kombinacja wyglądała następująco: 1) modem jest w stanie włączonego zasilania, 2) inicjalizacja z powodów niedoskonałości oprogramowania modemu lub błędów w warstwie RIL (ang. Radio Interface Layer) zatrzymuje się i pozwala przejść procesorowi w stan niskiego

poboru prądu, 3) kontroler USBHOST wraz z transreceiverem poprawnie usypiają magistralę USB, 4) Modem przechodzi do trybu głębokiego uśpienia.

Innym przykładem sytuacji, w której na krótki okres *Holter #1* osiąga lepsze minimum lokalne pobieranej mocy niż zwykle jest start systemu operacyjnego zaprezentowany na wykresie w rysunku 15. Na zielono zaznaczony jest fragment wykresu reprezentujący natężenie prądu podczas uruchamiania systemu operacyjnego począwszy od programu rozruchowego poprzez jądro systemu, aż do podstawowych serwisów systemowych. Kolorem żółtym oznaczono fragment wykresu zmierzony, gdy urządzenie było w stanie między zakończeniem uruchamiania systemu, a startem warstwy radiowej systemu. Kolor czerwony reprezentuje start warstwy radiowej, m.in uruchomienie modemu i wstępna weryfikacja poprawności działania, a także wyłączenie modemu. Ostatnia sekcja biała reprezentuje pomiar natężenia prądu w stanie, gdy modem został już wyłączony. Na czarno zaznaczono różnicę w minimalnych wartościach natężenia pomiędzy obszarem żółtym (16mA) i białym(21mA).



Rysunek 15. Wykres natężenia prądu pobieranego przez *Holter #1*, zawierający moment osiągnięcia minimum wartości natężenia (16mA)

Modem pobiera mniej energii w stanie poprawnego uśpienia niż w momencie, gdy za pomocą tranzystora kluczującego odłączono zasilanie modułu. Ponadto stan w jakim znajduje się modem przed uruchomieniem warstwy RIL jest bardziej energooszczędny niż stan, w którym RIL kluczuje zasilanie modułu. Korzystając z autorskiego narzędzia zlokalizowano właściwy plik *FTrace*, który pokrywa zdarzenia powstałe podczas uruchamiania warstwy RIL. Analiza porównawcza stanów linii GPIO pomiędzy stanem "żółtym", a "białym" wykazała, że przy wyłączonym modemie w obu stanach linia RESET modemu ma inne wartości logiczne. W przypadku podwyższonych wartości poboru prądu linia ta była zwarta do masy zasilania. Zwarcie linii RESET do masy podczas, gdy modem był odłączony od zasilania skutkował zasilaniem się modemu ze źródła VIO (poprzez rezystor podciągający modułu). Zaaplikowanie rozwiązania ustawiającego sygnał linii RESET w stan wysokiej impedancji po wyłączeniu modemu spowodowało, że wartość średniego poboru prądu spadła o 5mA. Pomiary wykazały,

że powyższa poprawka niweluje opisane niedoskonałości w poborze energii. Średnia wartość natężenia prądu spadła do poziomu 30,9 mA- wartość mniejsza od wymaganej przez specyfikacje.

Wykrycie powyższego błędu jak i jego diagnoza były najbardziej pracochłonnymi czynnościami podczas procesu optymalizacji energetycznej systemu *Holter #1*. Posiadanie narzędzia opisanego w rozdziałach 3.1 i 8.2 pozwoliłoby na znaczące skrócenie opisywanych czynności oraz zautomatyzowanie części pracy. *StateEventAnalyzer* wskazałby stany zawyżonego poboru energii oraz oznaczyłby tranzycje, które prowadzą do tych stanów. Po uwzględnieniu logów systemowych wykryte stany zostałyby opisane, ułatwiając analizę bezpośredniej przyczyny. Największym praktycznym zyskiem z zastosowania *StateEventAnalyzer* byłaby automatyzacja wykrywania przebiegów, w których przedstawiany wyciek prądu występował. Reprodukowalność błędu była niedeterministyczna- odtworzenie sytuacji wymagało kilku iteracji testów. Warto zauważyć, że rozwiązanie większości problemów opisanych w tym rozdziale wymagało fuzji wyników równoległej analizy obserwacji pochodzących z wielu źródeł np. szeregu czasowego poboru prądu, logów aplikacyjnych czy też danych z systemu FTrace. Problem częściowej automatyzacji procesu analizy tego typu opisany jest w rozdziale 4, który zawiera propozycję korelacji i filtracji wyników obserwacji z logami tekstowymi.

4 Algorytm korelacji logów tekstowych z szeregiem czasowym sygnału

W rozdziale 4 przedstawiony jest algorytm korelacji logów tekstowych z zaobserwowanymi lub wykrytymi zdarzeniami/stanami badanego systemu. Algorytm rozwiązuje problem braku synchronizacji między zegarami urządzeń próbkujących sygnał zewnętrzny urządzenia, a znacznikami czasowymi rekordów logów. Część systemów wbudowanych loguje zachowanie oraz stan poszczególnych podsystemów w formie logów tekstowych, które są przechowywane w pamięci urządzenia. Każdy rekord może przechowywać informacje o zdarzeniach, o wykonywanych czynnościach oraz zaistniałych błędach i wyjątkach. Kolejność wygenerowania rekordów jest znacząca podczas odtwarzania przebiegu wydarzeń oraz analizy przyczyny wystąpienia błędów. Logi oznaczane są znacznikiem czasu, który umożliwia ulokowanie danej informacji na osi czasu analizowanej sytuacji oraz określenie kolejności wystąpienia logów. Zegar używany do znakowania logów jest zegarem lokalnym, który jest synchronizowany w określonych momentach. W systemach, które nie posiadają modułów komunikacyjnych, zegar może być niezsynchronizowany i udostępniać niepoprawny czas dla systemu logowania. W takim przypadku rekordy logów zachowują właściwość umożliwiającą odzyskanie poprawnej kolejności występowania danych zdarzeń, jednak niemożliwe jest powiązanie danego rekordu ze zdarzeniem lub stanem uzyskanym na podstawie zewnętrznej metody obserwacji badanego urządzenia. Rozwiązaniem problemu braku powiązania może być metoda korelacji między sekwencją logów a zdarzeniami z obserwacji zewnętrznej. Zewnętrzną obserwacją może być szereg czasowy próbek sygnału generowanego przez urządzenie jak i wynik analizy takiego sygnału.

Jedną z metod korelacji pomiędzy takimi sekwencjami jest metoda wstawiania znacznika. Metoda polega na zmodyfikowaniu logiki funkcjonowania urządzenia w taki sposób, by wstawiać do sygnału charakterystyczne, unikalne i identyfikowalne zaburzenie. System loguje wygenerowanie takiego zaburzenia wraz z jego parametrami. Następnie podczas korelacji algorytm dopasowuje czas wystąpienia zaburzenia do czasu odpowiadającego rekordowi logu, znajdując w ten sposób przesunięcie w czasie i dokonując korekcji znaczników czasu rekordów. Niektóre systemy wbudowane mogą być obserwowany jedynie poprzez sygnały elektryczne, które sterują innym podsystem wykonawczym. Wprowadzanie zaburzeń do sygnału może skutkować utratą funkcji lub generować niezgodności ze specyfikacją urządzenia. Ponadto każda modyfikacja oprogramowania może wpływać na ostateczny sposób funkcjonowania urządzenia. Wyniki analizy uzyskanej dzięki korelacji przy pomocy

zmodyfikowanego oprogramowania urządzenia nie opisują zachowania systemu w rzeczywistym środowisku. Jednym z wymagań proponowanego w tym rozdziale algorytmu jest nieinwazyjność na poziomie oprogramowania i sprzętu badanego systemu.

Przykładami opisywanych zewnętrznych obserwacji są zarówno próbkowany sygnał generowany przez badany system np. szereg czasowy próbek natężenia pobieranego prądu jak i sekwencja obiektów wyekstrahowanych na podstawie sygnału. Metoda pozyskiwania sekwencji obiektów jest przedstawiona w rozdziale 3.1. Wynikiem działania algorytmu na szeregu czasowym (TS) jest sekwencja instancji klas cykli. Każda z instancji opisuje złożony stan badanego urządzenia. Jednym z atrybutów instancji jest znacznik czasu początku i końca wystąpienia klasy cyklu. Znacznik czasowy klasy cyklu jest zsynchronizowany z urządzeniem dokonującym akwizycji próbek. Natomiast logi tekstowe urządzenia są oznaczone znacznikami czasowymi wygenerowanymi przez zegar o nieznanym statusie synchronizacji. Celem opisywanego algorytmu jest znalezienie takiego przesunięcia w czasie, aby logi tekstowe odpowiadały stanom urządzenia. Podczas operacji dopasowania algorytm powinien dokonać filtracji rekordów logów odrzucając te, które nie opisują klas cykli. Algorytm przetwarza rekordy logów tekstowych w ustandaryzowane obiekty worków słów, które umożliwiają operację wyszukania podobnych do siebie rekordów wskazujących na podobne zdarzenia. Algorytm dokonując analizy podobieństwa rekordów, grupuje je w rekordy opisujące podobne zdarzenia, a następnie określa charakterystyki czasowe (np. okres/częstotliwość) występowania takich grup rekordów. Wybierając odpowiednie grupy do analizy znajduje takie, których charakterystyki odpowiadają właściwościom czasowym instancji klas cykli. Różnica wartości między znacznikami czasu odpowiadających sobie zdarzeniom (obserwacjom i rekordom logów tekstowych) pozwala określić przesunięcie w czasie. Ostatecznie algorytm dokonuje korekcji znaczników czasowych logów, co jest równoważne z korelacją między logami tekstowymi a szeregiem czasowym.

4.1 Założenia oraz definicja modelu

Danymi wejściowymi dla algorytmu są dwa rodzaje informacji. Pierwszą jest lista obiektów zdarzeń pochodzących z obserwacji zewnętrznej. Każdy obiekt powinien dotyczyć zdarzenia tego samego rodzaju np. uruchomienia modemu oraz prezentować dwa znaczniki czasu: rozpoczęcia zdarzenia i jego zakończenia. Opisywane dane wejściowe można przedstawić jako listę par wartości-interwałów *intervals*: znacznik czasu początku i znacznik czasu końca trwania zdarzenia.

Drugim argumentem wejściowym jest ciąg znakowy tworzący zestaw rekordów logów-*log_text*. Każdy rekord logu tekstowego został wygenerowany przez badane urządzenie wg formatu: „*%znacznik czasowy %poziom_logu*/*%źródło rekordu* (*%numer PID*): *%komunikat logu"* zakończony znakiem nowej linii. Znakiem *%* oznaczono w formacie pola rekordu. Pierwszym polem jest znacznik czasowy opisujący datę i czas powstania logu. Następnym znaczącym polem jest źródło rekordu, zawierające nazwę serwisu/procesu lub wątku, który wygenerował rekord logu. Numer PID (ang. proccess identification number) jest numerem procesu, z którego został wygenerowany rekord logów. Ostatnim polem jest ciąg alfanumeryczny wraz ze znakami białymi, który opisuje zdarzenie lub stan urządzenia. Przykładem takiego rekordu jest: „R11-08 07:05:38.657 D/AT (134): AT> AT+CCWA=1". Rekord ten ma znacznik czasowy określający datogodzinę rekordu logu na 8 listopada godzinę 7, 5 minut i 38 sekund oraz 657ms. Litera R przed znacznikiem czasowym określa typ zegara użytego do oznaczenia znacznika: R-zegar czasu rzeczywistego. Typ zegara jest stały dla wszystkich logów i nie ma wpływu na przedstawianą analizę. Źródłem jest komponent AT-biblioteka odpowiedzialna za przetwarzanie komend AT przy komunikacji z modemem LTE. Funkcja biblioteczna została wywołana przez proces o numerze PID=134, a treść komunikatu logu to: AT> AT+CCWA=1. Komunikat oznacza, że proces wysłał do modemu zadaną komendę AT+CCWA=1.

Podstawowym wynikiem działania algorytmu jest wartość przesunięcia podana w jednostce czasu. Dodanie przesunięcia do każdego znacznika czasu w logach zdarzeniowych tworzy logi, które są zsynchronizowane z sygnałem zewnętrznym/interwałami. Drugim wynikiem jest lista pogrupowanych sekwencji rekordów, które odpowiadają wydarzeniom opisanym przez interwały czasowe *intervals.* Jest to lista rekordów logów odfiltrowana z logów nieznaczących lub takich, których charakterystyki czasowe wskazują na niedopasowanie do interwałów.

Model danych prezentowanego algorytmu opisuje obiekty, które są tworzone na różnych etapach korelacji między szeregiem czasowym TS oraz sekwencją rekordów logów tekstowych EL. Szereg czasowy $TS = (s_1, s_2, \dots, s_r)$ składa się z wartości próbek sygnału $(s_i, 1 \leq i \leq r)$ próbkowanych z okresem T. Każda z próbek jest oznaczona znacznikiem czasu pochodzącym z zegara lokalnego urządzenia próbkującego. Sygnał TS jest następnie poddawany analizie i przetwarzaniu zgodnie z algorytmem z rozdziału 3.1. W wyniku działania algorytmu otrzymujemy hierarchię różnych obiektów opisujących zdarzenia i stany na różnym poziomie abstrakcji. Kolejnym etapem jest wybór jednego typu zdarzeń np. wystąpień pojedynczej klasy cykli. Każde z tych wystąpień zdarzeń ma swój znacznik początku i końca. Wybierając

wszystkie instancje danej klasy cykli tworzona jest sekwencja par znaczników czasu określana dalej jako sekwencja interwałów czasowych $OI = \{(t_1, t_2), (t_3, t_4), \dots, (t_k, t_{k+1})\}$. W dalszym opisie pojedyncza para znaczników czasowych jest określana jako $z_i = (t_{i*2-1}, t_{i*2})$ i $z_i(1) = t_{i*2-1}$, $z_i(2) = t_{i*2}$. Taka sekwencja *OI* może również powstać w wyniku innego typu obserwacji zewnętrznych i stanowić dane wejściowe dla przedstawianego w tym rozdziale algorytmu.

Logi zdarzeniowe są generowane przez badane urządzenie w postaci tekstowych rekordów logów oddzielonych znakami nowej linii. Sekwencja logów zdarzeniowych jest oznaczana $EL = (E_1.E_2 \dots., E_v)$. Każdy rekord jest tworzony zgodnie z formatem i opisem z paragrafu wyżej. Pole komunikatu tekstowego jest dzielone na poszczególne słowa, które następnie są klasyfikowane.

Rozważany problem korelacji jest definiowany jako optymalne dopasowanie interwałów OI wygenerowanych na podstawie TS do logów zdarzeniowych w postaci sekwencji EL. Metoda korelacji bazuje na znalezieniu relacji odpowiedniości między obiektami EL i IO poprzez przesuwanie skali czasu jednego z typów obiektów. W rozdziale przedstawiony jest opis struktury modelu danych wykorzystywany przez algorytm. W tym celu rozdział wykorzystuje następującą notację: a) $\bigcup_x Y(x)$- suma zbiorów wygenerowanych przez wyrażenie Y(x) indeksowane x należącymi do zdefiniowanej przestrzeni, b) $\sum_x Y(x)$- suma algebraiczna wyrażeń skalarnych indeksowanych zmienną x należącą do zdefiniowanej przestrzeni, c) $\forall_x : Y(x)$- kwantyfikator ogólny („Dla każdego x (…), Y(x)") oznacza, że wyrażenie Y(x) jest spełnione dla każdej wartości zmiennej x w ramach zdefiniowanej przestrzeni wartości, d) $\exists_x : Y(x)$- kwantyfikator egzystencjalny („Istnieje taki x (…), że Y(x)"), oznacza, że istnieje taka wartość, którą zmienna x może przyjąć w ramach zdefiniowanej przestrzeni wartości, że wyrażenie Y(x) jest prawdziwe. Operator $A \cap B$ oznacza iloczyn zbiorów A i B. Relacja $a_i \in A$ oznacza, że element a_i należy do zbioru A. Symbol – nad zbiorem oznacza przestrzeń wszystkich elementów tego samego typu np. $\overline{A}$, oznacza przestrzeń wszystkich elementów a_i.

Opis najniższego poziomu algorytmu wykorzystuje pojęcie słowa $w_i \in \overline{W}$. Słowo jest zdefiniowane jako sekwencja znaków alfanumerycznych (np. w kodowaniu ASCII). Słowo jest oddzielone od sąsiednich słów przez wybrane znaki takie jak: _, -, [,], (,), + oraz znaki białe. Każde słowo jest klasyfikowane do jednej z wartości: 1) *l-word*- dowolna sekwencja zaczynająca się od znaków alfabetu łacińskiego, które nie tworzy słowa typu *source,* 2) *d-word*- sekwencje znaków 0-9, które nie są *PID*-em, 3) *PID*- sekwencja cyfr tworząca numer procesu

generującego log tekstowy, 4) *source*- nazwa programu, usługi, procesu, wątku lub biblioteki generującego log tekstowy, 5) *timestamp*- znacznik czasowy rekordu. Rekord logu rozkładany jest na pojedyncze słowa, które są klasyfikowane. Para $cw = (w_i, m_j)$, gdzie $w_i \in \overline{W}$ i $m_j \in \overline{M}$, $\overline{M} = \{l-word, d-word, pid, source, timestamp\}$ tworzy *słowo sklasyfikowane.* Opis algorytmu korzysta z notacji *cw(1)* i *cw(2)* do odnoszenia się odpowiednio do: słowa i jego typu (klasyfikacji).

Każdy rekord logu EL jest reprezentowany poprzez worek słów B_L zdefiniowany jako zbiór słów sklasyfikowanych. Funkcja $TS(B_L)$ zwraca znacznik czasowy rekordu L. Funkcja $BW(cw_i)$ zwraca wartość numeryczną wagi zdefiniowaną jako parametr algorytmu dla każdej kategorii słowa $cw_i(2)$. Jednym z etapów algorytmu jest porównywanie worków słów. W tym celu algorytm korzysta z metryki *similarity*, która określa poziom podobieństwa między workami słów $B_a i\ B_b$:

$$SIMILARITY(B_a, B_b) = \frac{\sum_{cw_i \in B_a \cap B_b} BW(cw_i)}{max\left(\sum_{cw_j \in B_a} BW(cw_j), \sum_{cw_k \in B_b} BW(cw_k)\right)} \quad (4.1)$$

Podczas wykonywania algorytmu powstają zbiory podobnych worków słów (SBS_x). W ramach każdego SBS_x istnieją worki słów $B_a\ i\ B_b$ spełniające warunek SIMILARITY$(B_a, B_b) \geq eps_s$, gdzie *eps_s* jest parametrem algorytmu ustalającym próg podobieństwa dla zbiorów podobnych worków słów. Każdy worek słów wygenerowany z logów EL jest przypisywany do unikalnego *zbioru podobnych worków słów*.

Sekwencja zdarzeń S jest zdefiniowana jako zbiór uporządkowany (relacją większości względem znacznika czasowego *timestamp*) elementów SBS_x, które spełniają warunek:

$$S \subseteq SBS_x: \forall B_j \in SBS_x/S, \forall B_k \in S\ , \left|TS(B_j) - TS(B_k)\right| > eps\_tolerance \quad (4.2)$$

gdzie *eps_tolerance* jest parametrem dzielącym SBS_x. Elementy *sekwencji zdarzeń* są ze sobą w relacji: $TS(S(1)) \leq TS(S(2)) \leq TS(S(3)) \leq \cdots \leq TS(S(N))$, gdzie N jest licznością *sekwencji zdarzeń* S, a $TS(S(i))$ jest znacznikiem czasowym i-tego sklasyfikowanego worka słów (w indeksowaniu od początku sekwencji S). W ten sposób zbiór SBS_x jest dzielony na podzbiory uporządkowane S_i, tworzące szereg $D_x = (S_1, S_2, \ldots, S_M)$. Zbiór wszystkich sekwencji zdarzeń oznaczany jest $\overline{S(SBS_x)}$. Celem takiego podziału SBS_x jest wyszczególnienie rekordów logów, które opisują to samo wystąpienie zdarzenia. Algorytm zakłada, że rekordy logów, które wcześniej zostały sklasyfikowane jako podobne, opisują to samo wystąpienie zdarzenia pod warunkiem, że zostały wygenerowane w niewielkich odstępach czasowych od siebie. Wnioskiem z założenia jest potrzeba grupowania worków słów

w grupy oddzielone odstępami czasowymi. Algorytm determinuje podział na takie grupy poprzez sprawdzanie czy odstęp czasowy (różnica znaczników czasowych) między dwoma kolejnymi workami słów jest większa niż parametr *eps_tolerance.* Jeśli warunek jest spełniony oznacza to, że dany odstęp czasowy rozdziela dwie sekwencje zdarzeń opisujących prawdopodobnie dwa kolejne wystąpienia tego samego zdarzenia zarówno w ramach EL jak i obiektów TS.

Algorytm wykorzystuje szereg $IS(SBS_x)$, który zdefiniowany jest jako szereg wartości odstępów czasowych między kolejnymi *sekwencjami zdarzeń* szeregu D_x. Znacznik czasowy początku i końca *sekwencji zdarzeń* S_i oznaczany jest odpowiednio jako: $TS_{DB}(S_i) = TS(S_i(0))$ oraz $TS_{DE}(S_i) = TS(S_i(N))$, gdzie N jest licznością zbioru S_i. Powyższe wyrażenia można również zapisać używając szeregu D_x jako: $TS_{DB}(D_x(i))$ oraz $TS_{DE}(D_x(i))$. Używając przedstawionej notacji model danych algorytmu definiuje $IS(SBS_x)$ jako:

$$IS(SBS_x) = \Big(TS_{DB}(D_x(2)) - TS_{DE}\big(D_x(1)\big); TS_{DB}\big(D_x(3)\big) - TS_{DE}\big(D_x(2)\big); \dots; TS_{DB}(D_x(N)) - TS_{DE}\big(D_x(N-1)\big)\Big) \tag{4.3}$$

W wielu systemach wbudowanych można zaobserwować występowanie periodycznych procesów tła. Algorytm odfiltrowuje rekordy logów pochodzące z takich procesów. Opisywane rekordy są określane w nomenklaturze algorytmu jako szum. Warunek klasyfikujący dany zbiór SBS_x jako szum przedstawiony jest w równaniu (4.4).

$$\frac{\left|avg\big(IS(SBS_x)\big) - med(IS(SBS_x))\right|}{avg(IS(SBS_x))} < noise\_0 \; AND \; \frac{std(IS(SBS_x))}{avg(IS(SBS_x))} < noise\_1 \tag{4.4}$$

gdzie *noise_0* i *noise_1* są predefiniowanymi numerycznymi parametrami algorytmu, *avg* oznacza średnią arytmetyczną, *med*- medianę, *std*- odchylenie standardowe wyliczone na podstawie odstępów czasowych szeregu D_x. Jeśli zanegowany warunek z równania (4.4) jest spełniony (przynajmniej jedna z wartości wyrażeń jest większa bądź równa od wartości progowych), to testowany szereg sekwencji zdarzeń ma szansę zawierać zdarzenia wygenerowane w wyniku sytuacji wyjątkowej lub będącej reakcją na zdarzenie nieregularne.

Celem algorytmu jest znalezienie dopasowania między obiektami interwałów OI, a *sekwencjami zdarzeń.* Dopasowanie jest realizowane zarówno poprzez wyżej zdefiniowaną filtrację jak i znalezienie wartości przesunięcia na osi czasu podczas analizy korelacji. Wynikiem działania algorytmu jest zbiór dopasowanych rekordów logów (w formie sekwencji zdarzeń) do interwałów OI oznaczany jako RES (opis tworzenia zbioru zawarty jest w rozdziale 4.2). Zbiór RES jest definiowany w równaniach (4.5) i (4.6):

$$RES(SBS_x, k) = \Big\{ S \subseteq \overline{S(SBS_x)} : \forall z \in OI, \exists B_j \in S, TS\left(B_j(1)\right) + k \geq z(1)\ AND\ TS\left(B_j\left(N_{B_j}\right)\right) + k \leq z(2) \Big\} \tag{4.5}$$

$$RES(k) = \bigcup_{SBS_x \in \overline{SBS}} RES(SBS_x, k) \tag{4.6}$$

gdzie k jest wartością przesunięcia w czasie, która jest wynikiem działania algorytmu korelacji. Zbiór RES jest sumą wszystkich podzbiorów zbiorów podobnych worków słów sklasyfikowanych SBS_x, dla których znaleziono dopasowanie, poprzez przesunięcie osi czasu o wartość k. Zbiór SBS_x jest dopasowany, gdy posiada taki szereg sekwencji zdarzeń, że znaczniki czasowe początków i końców kolejnych zdarzeń mieszczą się w zakresach zdefiniowanych przez kolejne pary interwałów czasowych $z \in OI$. Sumując wszystkie takie zbiory otrzymywany jest zbiór wynikowy RES(k). Wynikiem działania algorytmu jest wartość przesunięcia k oraz zbiór RES(k) takie, że liczność zbioru RES(k) (liczona jako liczba sekwencji zdarzeń) jest największa w przestrzeni poszukiwań dopasowania.

Przedstawiony model danych jest bezpośrednio wykorzystywany przez algorytm opisany w rozdziale 4.2. W celu przedstawienia intuicji działania algorytmu w rozdziale 4.4 został zaprezentowany sposób działania algorytmu na małym zbiorze danych wejściowych.

4.2 Specyfikacja algorytmu

Specyfikacja algorytmu jest oparta o notację pseudokodu opisaną w rozdziale 3.1.2. Specyfikacja wykorzystuje metody i funkcje, których nazwy opisują czynności realizowane przez daną procedurę. Wyjaśnienia szczegółowe każdej procedury znajdują się przy opisie każdego z algorytmów.

Algorytm korelacyjny jest podzielony na dwie fazy: 1) konwersji logów tekstowych na obiekty oraz ich grupowaniu i klasyfikacji do formy pozwalającej na dopasowanie do par interwałów czasowych zdarzeń, 2) określenia przesunięcia czasowego pozwalającego na najlepsze dopasowanie zdarzeń do struktury logów tekstowych. Algorytm 4-1 przedstawia wysokopoziomową perspektywę wykonania algorytmu korelacji logów. Procedura zawiera odniesienia do procedur reprezentujących kolejne kroki algorytmu. Argumentami wejściowymi algorytmu są: 1) ciąg tekstowy rekordów logów- *log_text*, 2) zbiór par interwałów czasowych obserwacji zewnętrznych- *intervals.* Argumenty odpowiadają odpowiednio obiektom EL i OI z opisu modelu danych z rozdziału 4.1. Znaczniki czasowe obiektu OI są znormalizowane względem początku pierwszego znacznika czasowego, co oznacza, że kolejne znaczniki czasowe wskazują na upływ czasu od początku pierwszego zdarzenia. Funkcja

create_bags_of_logs zwraca listę worków słów wygenerowanych na podstawie rekordów logów tekstowych. Procedura dzieli logi tekstowe na rekordy oddzielone znakiem nowej linii, następnie dokonuje ekstrakcji pojedynczych słów. Słowa są klasyfikowane i przypisywane do odpowiednich kategorii. Takie pary w ramach pojedynczego rekordu są agregowane w jeden worek słów. Na przykład dla rekordu logu: „R11-08 07:05:38.657 D/AT (134): AT> AT+CCWA=1", procedura wygeneruje worek słów: < R11-08 07:05:38.657, timestamp>, <AT, source>, <134, pid>, <AT, l-word>, <CCWA, l-word>, <1, d-word>. Podczas tworzenia worków słów znaczniki czasowe kolejnych worków są normalizowane względem pierwszego rekordu logów tekstowych. Utworzona lista worków słów jest argumentem wejściowym dla procedury *cluster_into_ssb*. Procedura grupuje worki słów w zbiory podobnych worków słów, korzystając z metryki *similarity*. Każdy zbiór podobnych worków słów jest dzielony na sekwencje. Każda sekwencja składa się z jednego lub więcej występujących po sobie worków słów. Obiekty *sequences* i *intervals* są argumentami procedury *candidates_from_sequence*, która zwraca kandydatów na dopasowanie wraz wartością przesunięcia (*offset*) generującego dopasowanie. Kandydaci agregowani są do listy *candidates*. Lista *candidates* wykorzystywana jest przez procedurę *find_best_matching*. Funkcja zwraca listę wybranych kandydatów ze wspólną wartością przesunięcia (wartości przesunięcia poszczególnych wybranych kandydatów mieszczą się w zakresie tolerancji definiowanej przez parametr *eps_tolerance*), która zapewnia największą liczbę dopasowanych kandydatów. Wynik procedury wraz z wartością przesunięcia jest rezultatem działania algorytmu. Zbiór dopasowanych kandydatów odpowiada definicji zbioru $RES(k)$.

```
Wejście:
     log_text  - rekordy logów w formacie tekstowym
     intervals - lista par interwałów opisujących zdarzenie
Wyjście:
     matched_logs - zbiór logów dopasowanych do obiektu interwałów
1: function match_events_with_logs (log_text, intervals)
2:      logs_as_bags = create_bags_of_logs(log_text)
3:      ssb = cluster_into_ssb(logs_as_bags)
4:      candidates = new List
5:      foreach (similar_bags in ssb) do
6:          sequences = create_sequences(similar_bags)
7:          candidates.add(candidates_from_sequence(sequences,
              intervals))
8:      end foreach
9:      matched_logs = find_best_matching(candidates)
10:     return matched_logs
11:end function
```

Algorytm 4-1. Procedura korelacji logów z perspektywy wysokopoziomowej (*match_events_with_logs*).

Funkcja *create_bags_of_logs* jest procedurą analizy i rozbioru tekstu (ang. parsing) i jest zależna od formatu logów tekstowych. Z tego powodu jej definicja nie znajduje się w opisie algorytmu. Procedura dokonuje podziału logów na rekordy szukając znaków nowej linii. Następnie rekordy dzielone są na poszczególne słowa. Podział realizowany jest poprzez znalezienie wystąpień wszystkich znaków białych takich jak spacja, tabulacja, itp. Bazując na pozycji danego słowa w rekordzie oraz jego zawartości pod względem rodzajów znaków, słowo klasyfikowane jest do jednej z kategorii: *l-word, d-word, PID, timestamp, source*. W wyniku takiego podziału zwracana jest lista worków słów. Każdy ze znaczników czasowych w danym worku słów jest normalizowany. Normalizacja znacznika czasowego polega na odjęciu wartości znacznika czasowego pierwszego analizowanego rekordu logu.

Algorytm *cluster_into_ssb* grupuje worki słów w zbiory podobnych worków słów (SBS z rozdziału 4.1). W pierwszym kroku lista worków słów kopiowana jest do zmiennej *unmatched_bags.* Dopóki *unmatched_bags* nie jest pustą listą, algorytm w pętli while (linia 5) wykonuje następujące kroki (linie 6-7): 1) pobranie pierwszego elementu z listy worków słów, 2) stworzenie zbioru/listy podobnych worków słów bazując na pobranym worku słów oraz pozostałej liście worków słów, 3) dodanie zbioru do obiektu agregującego zbiory podobnych worków słów *ssb*. Operacja 2) realizowana jest poprzez procedurę *create_similar_bags_list* przedstawioną w algorytmie 4-3. Procedura tworzy nową listę jednoelementową nowego zbioru worków podobnych słów. Pojedynczy element stanowi pobrany w operacji 1) worek słów. Następnie w podwójnej pętli foreach (linie 5 i 6) przeglądane są worki słów. Pierwsza pętla iteruje się po obecnie tworzonym zbiorze podobnych worków słów (similar_bags), a pętla wewnętrzna po zbiorze niedopasowanych worków słów (unmatched_bags). Za pomocą metryki *SIMILARITY* (linia 7) weryfikowane jest podobieństwo między elementami obu kolekcji. Jeśli worki zostaną sklasyfikowane jako podobne, worek słów z *unmatched_bags* usuwany jest ze swojej kolekcji i dodawany jest do listy *similar_bags*. Po przejrzeniu kombinacji dopasowania worków słów, nowa lista worków słów jest zwracana.

```
Wejście:
      bags - lista worków słów
Wyjście:
      ssb - lista zbiorów podobnych worków słów
1: function cluster_into_ssb(bags)
2:       ssb = new Set
3:       unmatched_bags = bags.copy()
4:       candidates = new List
5:       while (not unmatched_bags.empty()) do
6:            seed = unmatched_bags.take_first()
7:            ssb.add(create_similar_bags_list(seed, unmatched_bags))
8:       end while
9:       return ssb
10:end function
```

Algorytm 4-2. Procedura grupowania worków słów w zbiory podobnych worków słów (*cluster_into_ssb*)

```
Wejście:
      seed           - pierwszy worek słów tworzący obiekt zwrotny
      unmatched_bags - zbiór niezgrupowanych worków słów
Wyjście:
      similar_bags - lista worków słów należących do tego samego zbioru
podobnych worków słów
1: function create_similar_bags_list(seed, unmatched_bags)
2:       similar_bags = new List
3:       similar_bags.add(seed)
4:       unmatched_bags.remove(seed)
5:       foreach (b in similar_bags) do
6:            foreach (a in unmatched_bags) do
7:                 if (similarity(b,a) > eps_s)
8:                      similar_bags.add(a)
9:                      unmatched_bags.remove(a)
10:                end if
11:           end foreach
12:      end foreach
13:      return similar_bags
14:end function
```

Algorytm 4-3. Procedura generowania pojedynczego zbioru podobnych worków słów (*create_similar_bags_list*).

Procedura *create_sequences* przyjmuje jako argument zbiór podobnych worków słów. Zbiór jest sortowany względem znaczników czasowych worków słów reprezentujących rekordy logów. Zbiór jest przeglądany w kolejności rosnącej znaczników czasowych za pomocą pętli foreach (linie 6-15). Pętla operuje na wskazaniu na obecnie tworzoną sekwencję *current_sequence*. W linii 7 obliczana jest różnica między znacznikiem czasu obecnego worka słów a poprzednikiem. Jeśli różnica jest większa od parametru *eps_time* to lista *current_sequence* jest zamykana i dodawana do obiektu szeregu sekwencji *sequences* za pośrednictwem metody *add*. Ponadto dodawana do obiektu jest para znaczników czasu

początku i końca sekwencji (metoda *addTimeRange* z linii 9). Metoda *addInterval* uzupełnia obiekt *sequences* o wartość różnicy znaczników czasowych między początkami sekwencji: obecnej i kolejnej. Jest to parametr potrzebny do wyznaczania interwałów między sekwencyjnych w szeregu D_x ($IS(SBS_x)$). W linii 11 algorytm tworzy nową pustą listę worków słów stanowiących obecnie przetwarzaną sekwencję. W linii 13 obecnie przetwarzana sekwencja jest uzupełniania o obecnie przeglądany worek słów. Po przejrzeniu wszystkich worków słów ze zbioru w linii 16 wyznaczane są wszystkie statystki szeregu sekwencji, niezbędne dla kolejnych procedur wykorzystując metodę *calculateStatisticsFromIntervals*. Utworzony obiekt szeregu sekwencji jest zwracany w linii 17.

```
Wejście:
     sbags - lista podobnych worków słów
Wyjście:
     sequences - podzielone na sekwencje worki słów z listy podobnych
worków słów sbags
1: function create_sequences(sbags)
2:      sequences = new sequences
3:      sbags.sort()
4:      current_sequence = new List
5:      last = sbags.first_bag()
6:      foreach (b in sbags) do
7:          if (b.timestamp - last.timestamp > eps_time)
8:              sequences.add(current_sequence)
9:              sequences.addTimeRange(current_sequence[0].timestam,
                  last.timestamp)
10:             sequences.addInterval(b.timestamp -
                  current_sequnce[0].timestamp)
11:             current_sequence = new List
12:         end if
13:         current_sequence.add(b)
14:         last = b
15:     end foreach
16:     sequences.calculateStatisticsFromIntervals()
17:     return sequences
18:end function
```

Algorytm 4-4. Procedura tworzenia sekwencji zdarzeń (*create_sequences*)

Algorytmy 4-5 i 4-6 przedstawiają procedury generujące kandydatów na dopasowanie na podstawie szeregu sekwencji. Procedura *candidates_from_sequence*s przyjmuje dwa argumenty: szereg sekwencji *sequences* oraz czasy zdarzeń *events* obiektów OI. W liniach 4 oraz 5 następuje odróżnienie szeregów, które klasyfikowane są jako szum od szeregów nieokresowych/nieregularnych. Warunek z linii 4 bazuje na metrykach opisanych w równaniu (4.3). Jeśli warunek jest spełniony, to szereg jest szumem, a procedura zwraca pustą listę kandydatów. W następnych krokach algorytm generuje kolejnych kandydatów na dopasowanie. Kandydat jest obiektem składającym się z: 1) wartości przesunięcia czasowego względem

pierwszego rekordu logu z pliku oraz 2) wybranych sekwencji z szeregu sekwencji, których czas trwania oraz znaczniki wystąpień zmodyfikowanych o przesuniecie czasowe mieszczą się w zakresach par znaczników czasowych obiektu *events.* Wstępna wartość przesunięcia jest ustawiana na wartość znacznika czasowego pierwszej sekwencji. Następnie w każdej iteracji pętli while (linia 8) weryfikowane jest dopasowanie do zdarzeń z obiektu *events.* Procedura *create_candidate_seq* zwraca listę sekwencji z szeregu sekwencji, której znaczniki czasowe mieszczą się wewnątrz znaczników czasowych zdarzeń *events* uwzględniając obecne przesunięcie (linia 9). Jeśli zwrócony obiekt dopasowanych sekwencji jest pusty, oznacza to, że dla danego przesunięcia czasowego dopasowanie jest niemożliwe (linia 10). W innym wypadku (linia 11) tworzony jest obiekt kandydata. Kandydat uzupełniany jest o wartość przesunięcia oraz dopasowane sekwencje (*matched_sequences*). Kandydat dodawany jest do listy kandydatów *candidates.* W linii 13 pod koniec iteracji wartość przesunięcia inkrementowana jest o parametr *time_increment.* Warunkiem zakończenia pętli jest przejrzenie wszystkich możliwych dopasowani co jest równoznaczne z osiągnięciem przez zmienną *offset* wartości równej znacznikowi czasowemu ostatniej sekwencji (linia 8). Procedura zwraca listę kandydatów w linii 15.

Weryfikacja dopasowania jest realizowana poprzez procedurę create*_candidate_seq,* która przyjmuje argumenty: szereg sekwencji *sequences,* zdarzenia *events* oraz przesunięcie czasowe *offset.* Wyznaczając dopasowanie należy zmodyfikować znaczniki czasu wybranego typu obiektu (sekwencji lub zdarzeń). Ze względów wydajnościowych (znaczników czasu zdarzeń najczęściej jest mniej niż analizowanych worków słów w szeregach sekwencji) algorytm w linii 2 kopiuje obiekt zdarzeń modyfikując znaczniki czasowe zdarzeń o wartość przesunięcia czasowego. Operacja kopiowania i modyfikacji przesunięcia realizowana jest przez metodę *copyWithTimeOffset*. Algorytm iteruje po wszystkich sekwencjach szeregu za pomocą pętli foreach (linia 5). W każdej iteracji algorytm przechowuje dwa wskazania: *current_ev*- wskazanie na obecnie analizowaną parę znaczników pojedynczego zdarzenia z obiektu *ts_objects,* oraz *filtered_sequences*, który stanowi podszereg sekwencji możliwych do dopasowania do zdarzeń *ts_objects*. Zmienna *current_ev* początkowo wskazuje na pierwsze zdarzenie. Jeśli w ramach pętli z linii 5-15 znaczniki czasowe początku i końca obecnej sekwencji zawierają się w wartościach znaczników czasowych początku i końca obecnego zdarzenia (*current_ev*), to taka sekwencja dodawana jest do podszeregu sekwencji *filtered_sequences* za pomocą metody *addSequence*. Weryfikacja relacji zawierania realizowana jest przez metodę *include*. W przypadku potwierdzenia zawierania obecnej sekwencji sprawdzany jest również warunek zawierania dla sekwencji następnej (linia 8). Jeśli

kolejna sekwencja nie spełnia warunku, algorytm przechodzi do analizy dopasowań dla kolejnego zdarzenia (linia 9). Jeśli obecna zmienna *current_ev* wskazuje na ostatnie zdarzenie, to po kroku 9 będzie wskazywała na koniec listy poprzez literał *None*. Spełnienie warunku wskazywania na koniec listy jest sprawdzane w linii 12 i jest równoznaczne ze znalezieniem wszystkich pasujących sekwencji. Obiekt *filtered_sequences* jest zwracany w linii 13. Dotarcie procedury *create_candidate_seq* do kroku 16 oznacza, że dany szereg sekwencji nie może być w pełni dopasowany do wszystkich zdarzeń z zadanych przesunięciem. W takim przypadku zwracany jest literał *None*, aby poinformować o braku dopasowania.

```
Wejście:
     sequences - lista sekwencji zdarzeń, events- pary znaczników
czasu obiektu OI.
Wyjście:
     candidates - lista kandydatów na dopasowanie
1: function candidates_from_sequences(sequences, events)
2:      candidates = new List
3:      s = sequences
4:      if (abs(s.avg-s.med)/s.avg < noise0 and s.std/s.avg < noise1)
5:          return candidates
6:      end if
7:      offset = sequences.firstSequence().timestamp
8:      while (offset < sequences.lastSequence().timestamp) do
9:          matched_sequences = create_candidate_seq(sequences,
              events, offset)
10:         if (matched_sequences != None) then
11:             candidates.add(new Candidate(offset,
                  matched_sequences))
12:         end if
13:         offset = offset + time_increment
14:     end while
15:     return candidates
16:end function
```

Algorytm 4-5. Procedura tworzenia listy kandydatów na podstawie sekwencji (*candidates_from_sequences*)

```
Wejście:
      i_sequences - lista sekwencji zdarzeń
      ts_objects  - pary znaczników czasu obiektu OI, offset-
przesunięcie w czasie
Wyjście:
      filtered_sequences - lista sekwencji dopasowanych do interwałów
z zadanym przesunięciem
1: function create_candidate_seq(i_sequences, ts_objects, offset)
2:      evs = ts_objects.copyWithTimeOffset(offset)
3:      current_ev = evs.first()
4:      filtered_sequences = new Sequences
5:      foreach (s in i_sequences) do
6:          if (current_ev.timeRanges.includes(s.timeRanges)) then
7:              filtered_sequences.addSequence(s)
8:              if (not current_ev.timeRanges.includes(
                  s.next().timeRanges))
9:                  current_ev = current_ev.next()
10:             end if
11:         end if
12:         if (current_ev is None) then
13:             return filtered_sequences
14:         end if
15:     end foreach
16:     return None
17:end function
```

Algorytm 4-6. Procedura generowanie listy sekwencji, które są dopasowane do interwałów przy zadanym przesunięciu czasu (*create_candidate*)

Algorytm 4-7 przedstawia procedurę *find_best_matching*. Procedura przyjmuje argument- listę kandydatów na dopasowanie. Procedura w dwóch zagnieżdżonych pętlach (linia 3 i 4) przeszukuje listy kandydatów grupując je w zbiory kandydatów (*matched_candidates*) o tej samej wartości przesunięcia czasowego ± wartość *eps_tolerance* (<offset, offset+eps_tolerance>). Następnie poprzez proste wybieranie zbioru o największej liczności kandydatów (linie 12-14) generowana jest lista dopasowanych kandydatów. Za pośrednictwem listy dopasowanych kandydatów algorytm umożliwia odczyt wartości przesunięcia, która koreluje logi tekstowe EL wraz ze zdarzeniami OI.

```
Wejście:
     candidates - lista kandydatów zdefiniowanych przez przesuniecie
i dopasowane sekwencje zdarzeń
Wyjście:
     max_matched_candidates - lista kandydatów wraz z przesunięciem,
które generuje największe dopasowanie z obiektami OI
1: function find_best_matching(candidates)
2:      max_matched_candidates = new List
3:      foreach (c in candidates) do
4:          matched_candidates = new List
5:          matched_candidates.add(c)
6:          foreach (d in candidates) do
7:              if (d.offset >= c.offset and d.offset < c.offset +
                  eps_tolerance and d is not in matched_candidates)
                  then
8:                  matched_candidates.add(c)
9:              end if
10:         end foreach
11:         if (matched_candidates.count() >
              max_matched_candidates.count()) then
12:             max_matched_candidates = matched_candidates
13:         end if
14:     end foreach
15:     return max_matched_candidates
16:end function
```

Algorytm 4-7. Procedura wyszukująca najlepsze dopasowanie dla kandydatów (*find_best_matching*)

4.3 Dobór parametrów algorytmu

Algorytm wymaga konfiguracji parametrami: 1) *word_weights*- wagi typów słów, 2) *eps_s*- próg podobieństwa, 3) *eps_time*- próg dyskryminacyjny tworzenia sekwencji podany w jednostce czasu, 4) parametry progowe szumu: *noise_0* i *noise_1*, 5) *time_increment*- krok algorytmu w jednostce czasu, 6) *eps_tolerance*. Wartości parametrów 1) i 2) są krytyczne dla klasyfikacji rekordów logów w formacie worków słów do zbiorów podobnych worków słów. Parametr *word_weights* jest mapą wag (wartości liczbowych) dla każdego z typów słów (*d-word*, *l-word*, *timestamp*, itp). Wagi są używane podczas wyliczania metryki *similarity* używanej do porównywania dwóch worków słów. Wartości wag powinny być dopasowane do używanego stylu logowania. Na przykład, jeśli przeciętny rekord logów tekstowych zawiera dużo liczb, które są tylko parametrami zdarzenia i nie definiują typu zdarzenia, to wartość wagi dla słów typu *d-word* powinna być obniżona. Podobnie jest w przypadku, w którym wiele serwisów generuje logi w ramach pojedynczego procesu, co obniża wartość rozróżniającą słów typu PID.

Parametr *eps_s* jest wartością progową, od której dwa worki słów uznawane są za podobne. Przyjmuje on w wartości w zakresie <0.0, 1.0>. Dobierając wartość parametru *eps_s* można wziąć pod uwagę właściwości logów, między innymi: różnorodność form i liczby rekordów logów wygenerowanych dla pojedynczego zdarzenia, stopień identyczności rekordów logów wygenerowanych dla tych samych typów zdarzeń czy też zróżnicowanie formatu komunikatu rekordów wygenerowanych dla różnych zdarzeń. Zwiększenie wartości *eps_s* powoduje wzrost liczby zbiorów podobnych worków słów (SBS) o mocach zbiorów pomniejszonych. Zbyt wysoka wartość parametru może spowodować, że część szeregów sekwencji będzie rozdzielona na drobniejsze uniemożliwiając ich zakwalifikowanie jako kandydatów na dopasowanie i tym samym negatywnie wpływać na wynik zarówno etapu filtracji dopasowania jak i wyznaczenia wartości przesunięcia. Z kolei spadek wartości parametru może doprowadzić do sytuacji, w której algorytm nie rozróżni dwóch różnych zdarzeń, generując fałszywe szeregi sekwencji. Ostatecznie taka konfiguracja uniemożliwi osiągnięcie dopasowania do obiektów OI.

Parametr *eps_time* wpływa na liczbę i właściwości statystyczne wygenerowanych sekwencji. Większe wartości powodują tworzenie się dłuższych sekwencji (zarówno w przestrzeni czasu jak i liczności podobnych worków słów). Jeśli wartość jest zbyt duża, algorytm wygeneruje pojedynczą sekwencję zawierającą wszystkie elementy z zbioru podobnych worków słów, a więc może pominąć opisy niektórych zdarzeń. Odpowiednia wartość parametru *eps_time* zależy od charakterystyk czasowych par interwałów zdarzeń. Wartości parametrów noise_0 i noise_1 wpływają na liczbę odrzucanych rekordów logów-rekordów sklasyfikowanych jako szum. Im większa jest wartość obu tych parametrów tym więcej szeregów sekwencji jest klasyfikowana jako szum i odrzucana. W praktyce zauważono, że dla większości badanych systemów wartość równa 0.2 generuje zadowalające wyniki.

Parametr 5) *time_increment* jest odpowiedzialny za liczbę wygenerowanych kandydatów na dopasowanie. Zmniejszanie wartości parametru zwiększa dokładność wyznaczenia wartości przesunięcia kosztem czasu obliczeń. Zwiększanie lub zmniejszanie wartości parametru 6) *eps_tolerance* zmienia liczbę finalnie wybranych kandydatów na dopasowanie. Wartość parametru 6) nie powinna być większa od parametru 5). Na przykład jeśli wiadomo, że najkrótszy interwał między zdarzeniami wynosi 5s, to konfiguracja *time_increment = 5s.* i *eps_tolerance = 2.5s.*, powinna wygenerować poprawne wyniki. Parametry mogą zostać dobrane eksperymentalnie lub bazując na wartościach poprzednich dopasowań dla podobnych systemów.

4.4 Przykład procesu analizy

Podrozdział przedstawia działanie algorytmu na przykładzie niewielkich i prostych danych wejściowych. Algorytm jest skonfigurowany parametrami: *eps_time*=2s, eps_tolerance=1s, *word_wages*={*source*=2.0, *l-word*=1.0, *d-word*=1.0}, *eps_s*=0.7. Obiekt zdarzeń OI składa się z par znaczników czasu opisujących początek i koniec 3 kolejnych zdarzeń: {<7,13>, <25,28>, <53,58>} (wartości podane są w sekundach). Log tekstowy z rekordami EL zawarty jest w tabeli 6.

Log tekstowy	**Zbiory podobnych worków słów**
1: <2> [PER] checking it 2: <5> [DHCP] activity one 3: <6> [DHCP] activity two 4: <9> [MANAGER] task one 5: <10> [PER] checking it 6: <18> [PER] checking it 7: <23> [MANAGER] task two 8: <24> [MANAGER] task three 9: <25> [DHCP] activity three 10: <26> [PER] checking it 11: <27> [MANAGER] task four 12: <34> [PER] checking it 13: <42> [PER] checking it 14: <50> [PER] checking it 15: <51> [DHCP] activity four 16: <52> [MANAGER] task five 17: <53> [DHCP] activity five 18: <54> [MANAGER] task six 19: <58> [PER] checking it	Zbiór 1 <2> [PER] checking it <10> [PER] checking it <18> [PER] checking it <26> [PER] checking it <34> [PER] checking it <42> [PER] checking it <50> [PER] checking it <58> [PER] checking it Zbiór 2 <5> [DHCP] activity one <6> [DHCP] activity two <25> [DHCP] activity three <51> [DHCP] activity four <53> [DHCP] activity five Zbiór 3 <9> [MANAGER] task one <23> [MANAGER] task two <24> [MANAGER] task three <27> [MANAGER] task four <52> [MANAGER] task five <54> [MANAGER] task six

Tabela 6. Przykłady rekordów logów tekstowych i zbiorów podobnych worków słów

Lewa kolumna tabeli 6 zawiera 19 rekordów logów w formacie uproszczonym (bez numeru PID procesu). Pierwsza liczba w każdej linii jest to numer rekordu, następnie w nawiasach <> podany jest znacznik czasowy w sekundach. W nawiasach [ ] jest nazwa źródła rekordu logu. Po nawiasie kwadratowym znajdują się kolejne wyrazy rekordów logu. Z każdego rekordu logu tekstowego tworzony jest worek słów sklasyfikowanych np. dla rekordu 1 powstaje worek słów: [2, *timestamp*], [PER, *source*], [checking, *l-word*], [it, *l-word*]. W celu ułatwienia śledzenia stanu danych rekordów logów, w opisach zamiast worków słów znajdują się numery rekordów logów lub rekordy w postaci tekstowej.

Pierwszym etapem algorytmu jest normalizacja znaczników czasowych zarówno zdarzeń jak i worków słów. Normalizacja powoduje, że rekord 1 otrzymuje znacznik czasowy <0>, rekord 2 znacznik czasowy <3>, itd. Analogicznie znaczniki zdarzeń przyjmą wartości: <0,7>, <18,21>, <46, 52>. Zabieg normalizacji nie ma wpływu na wynik działania algorytmu i jego analizę, natomiast upraszcza formę algorytmu. W celu ułatwienia śledzenie zmian i stanu rekordów i zdarzeń obiekty te występują w swojej oryginalnej postaci. Algorytm grupuje podobne worki słów w zbiory podobnych worków słów. Wynik takiego grupowania zaprezentowany jest w prawej kolumnie tabeli 6. Grupowanie bazuje na metryce *SIMILARITY*. Na przykład porównując logi „<9> [MANAGER] task one" oraz „<23> [MANAGER] task two" otrzymujemy metrykę *SIMILARITY*= 0.75 > *eps_s*=0.7, a więc oba te rekordy są podobne. Obliczenia metryki prezentują się następująco: rekordy 4 i 7 mają wspólne źródło MANAGER oraz słowo „task", a więc licznik metryki wynosi 2.0+1.0+0.0=3.0. Oba rekordy mają trzy wyrazy o identycznej sumie wag wynoszącej 2.0+1.0+1.0=4.0. Ostatecznie wartość *SIMILARITY* dla tych rekordów wynosi 3.0/4.0=0.75. Analogicznie wartość metryki *SIMILARITY* dla rekordów 5 i 4 wynosi 0.25 < eps_s. W wyniku grupowania powstały trzy zbiory podobnych worków słów: zbiór 1, zbiór 2 i zbiór 3.

Następnie dla każdego zbioru algorytm dzieli go na sekwencje, tworzące szereg sekwencji zdarzeń. Zakładając, że *eps_time*=7s, algorytm wygenerował dla każdego zbioru indeksowanego parametrem X szereg sekwencji X, gdzie X przyjmuje wartości od 1 do 3. Kolejne sekwencje w szeregu oznaczane są sequence_Y, gdzie Y jest indeksem sekwencji w szeregu. Szeregi sekwencji wszystkich trzech zbiorów są przedstawione na listingu poniżej:

Dla zbioru 1-> szereg sekwencji 1:
sequence_0 = {<2> [PER] checking it}
interval_0 = 8
sequence_1 = {<10> [PER] checking it}
interval_1 = 8
sequence_2 = {<18> [PER] checking it}
interval_2 = 8
sequence_3 = {<26> [PER] checking it}
interval_3 = 8
sequence_4 = {<34> [PER] checking it}
interval_4 = 8
sequence_5 = {<42> [PER] checking it}
interval_5 = 8
sequence_6 = {<50> [PER] checking it}
interval_6 = 8
sequence_7 = {<58> [PER] checking it}

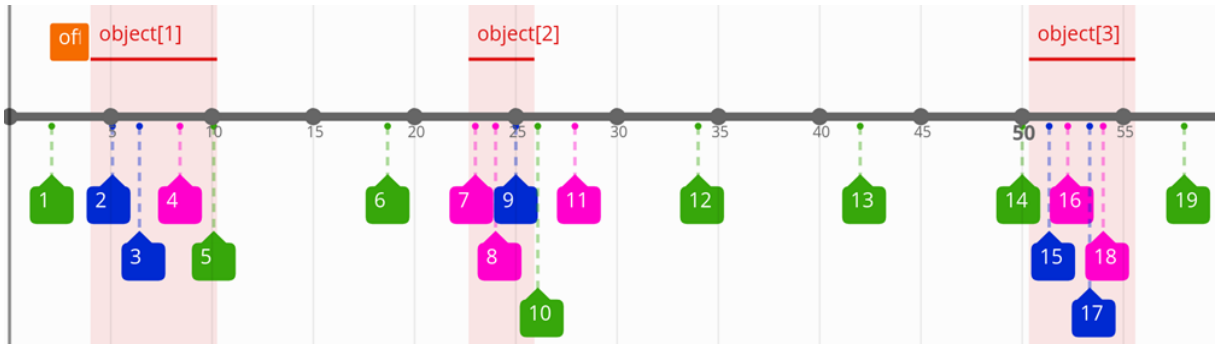
Dla zbioru 2-> szereg sekwencji 2:
sequence_0 = {<5> [DHCP] activity one, <6> [DHCP] activity two}
interval_0 = 19
sequence_1 = {<25> [DHCP] activity three}
interval_1 = 26
sequence_2 = {<51> [DHCP] activity four,<53> [DHCP] activity five}

Dla zbioru 3-> szereg sekwencji 3:
sequence_0 = {<9> [MANAGER] task one}
interval_0 = 14
sequence_1 = {<23> [MANAGER] task two, <24> [MANAGER] task three,
<27> [MANAGER] task four}
interval_1 = 29
sequence_2 = {<52> [MANAGER] task five, <54> [MANAGER] task six}

Każdy szereg sekwencji oprócz sekwencji zawiera wartości interwałów czasowych między sekwencjami obliczanych zgodnie z algorytmem *create_sequences*. Sekwencje są rozróżniane wtedy, gdy różnica między znacznikami czasowymi kolejnych rekordów jest większa od *eps_time*=7s. Na przykład, różnice czasu między rekordami 7 i 8 oraz 16 i 18 to odpowiednio 1s, 2s ≤ *eps_time*. Natomiast wartości różnic między znacznikami czasowymi dla rekordów 8 i 11 oraz 11 i 16 wynoszą 3s i 25s > *eps_time*. Z tego powodu w ramach zbioru trzeciego powstały sekwencje: sequence_1 składająca się z rekordów 7 i 8, sequence_2: rekord 11 oraz sequence_3: rekordy 16 i 18.

Kolejnym etapem działania algorytmu jest wytworzenie kandydatów na dopasowanie. Szereg sekwencji 1 został sklasyfikowany jako szum z perspektywy algorytmu, ponieważ wszystkie wartości interwałów wynoszą 8 co skutkuje, że obie wartości koniunkcji warunków (procedura *create_candidates_seq*- linia 4) wynoszą 0 i z pewnością są poniżej wartości noise_0 i noise_1. Pozostałe szeregi sekwencji są dalej procedowane. W wyniku działania algorytmu wygenerowane są dwa obiekty kandydatów: <offset=2, szereg sekwencji 2: sekwencje 0, 1, 2> oraz <offset =2, szereg sekwencji 3: sekwencje 0, 1, 3>. Jest to para kandydatów, która jako jedyna może być dopasowana. Ponadto para ta może być zgrupowana, ponieważ przesunięcie czasowe każdego kandydata mieści się w wartości *offset*±*eps_tolerance*. Ostatecznym wynikiem działania algorytmu dla powyższych danych jest przesunięcie czasowe = 2s. Zakładając, że jako bazę przyjmujemy oś czasu logów tekstowych, wartości znaczników czasowych zdarzeń obiektu OI wyniosą: <4,10>, <22,25>, <50,55>. Analogicznie używając oryginalnej osi czasowej obiektu OI, znaczniki czasowe można przeliczyć jako: dla rekordu 1: <5>, dla rekordu 2: <8>, dla rekordu 3: <9>, itd. Na rysunku 16 przedstawiono wyniki działania algorytmu przyjmując za oś odniesienia oryginalny czas logowani rekordu logów. Oś czasu narasta w kierunku prawej strony. Zielonymi, niebieskimi i różowymi polami oznaczono

rekordy logów, składających się odpowiednio na szeregi sekwencji 1, 2 i 3. W każdym polu jest wpisany indeks rekordu logów. Każde pole zaznacza na osi czasu wartość swojego znacznika czasowego. Pomarańczowe pole oznacza wartość przesunięcia czasowego równego 2s względem pierwszego rekordu logów (wynik działania normalizacji znaczników czasowych). Półprzeźroczyste czerwone pola zarysowują początek i koniec oraz czas trwania zdarzeń opisanych przez obiekt OI po dopasowaniu ich przez algorytm przy wartości offset=2.



Rysunek 16. Graficzna reprezentacja wyników działania algorytmu korelacji logów tekstowych i zdarzeń

5 Badanie eksperymentalne holterów

W rozdziale przedstawiono wyniki trzech eksperymentów z wykorzystaniem autorskiego systemu *StateEventAnalyzer* implementującego algorytmy zaprezentowane w rozdziałach 3 i 4. Obiektami podlegającymi monitorowaniu są dwa typy holterów EKG oznaczone jako: 1) Holter#1 oraz 2) Holter#2. Holer#1 jest urządzeniem realizującym kolejno zadania: 1) agregacji sygnału EKG z trzech elektrod tworzących dwa kanały EKG, 2) buforowania sygnału, 3) komunikacji z centrum monitorowania z wykorzystaniem technologii LTE oraz 4) wysyłania danych na serwery centrum monitorowania. Holter#1 był wykorzystywany podczas eksperymentów polegających na wykryciu stanów energetycznych badanego systemu (rozdział 5.1) oraz wygenerowaniu zautomatyzowanego opisu wykrytych stanów na podstawie dopasowanych logów tekstowych (rozdział 5.2).

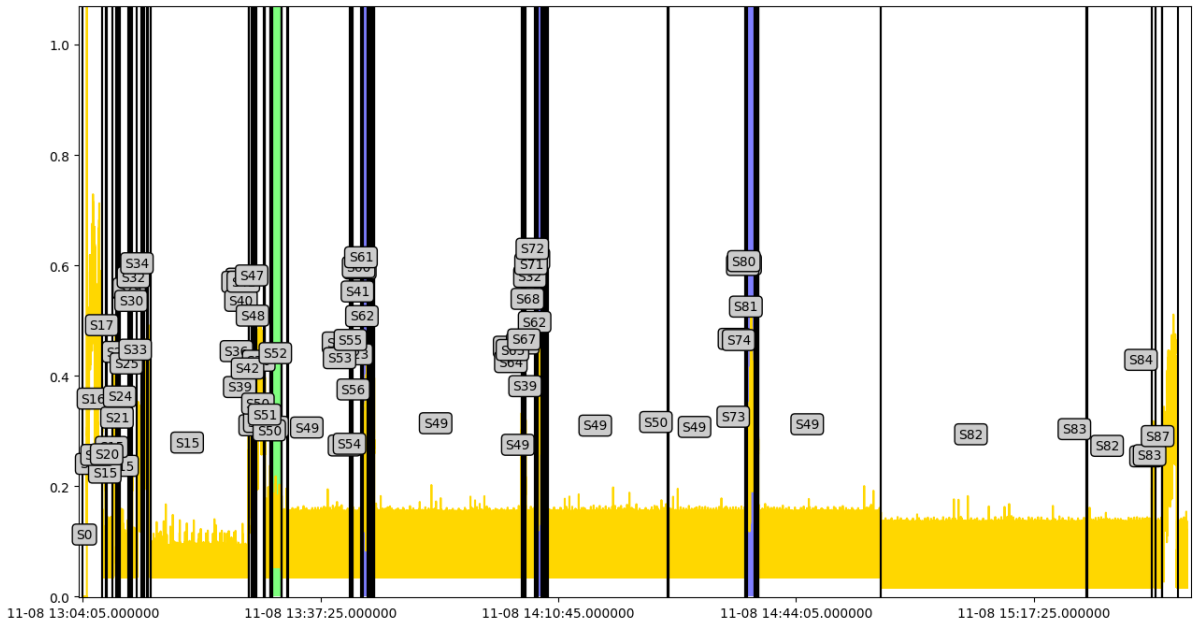
Holter#2 jest systemem wbudowanym, który nie realizuje funkcji bezprzewodowej łączności. Zapisuje on zagregowane próbki jednokanałowego sygnału EKG w pamięci nieulotnej. Po zakończeniu badania dane są odczytywane poprzez port USB. W rozdziale 5.3 przedstawiona jest kolejna analiza z wykorzystaniem algorytmów z rozdziału 3 (Holter#2 nie generuje logów tekstowych). Ponadto rozdział zawiera praktyczną propozycję metod doboru parametrów algorytmów.

Rysunki zaprezentowane w tym rozdziale pochodzą z systemu *StateEventAnalyzer*. Na osi pionowej znajdują się wartości natężenia pobieranego prądu elektrycznego przez urządzenie przy stałym napięciu. Oś pozioma reprezentuje wartość znacznika czasu próbki szeregu czasowego. Czarne linie poziome oddzielają kolejno wykryte stany na wykresie. Stany oznaczone są etykietami *S{numer stanu}*. Kolejne klasy cykli oznaczone są etykietami *CC{numer klasy}* oraz odpowiadający im fragment wykresu jest zamalowany różnym kolorem.

5.1 Obiektowy model stanów energetycznych w Holterze #1

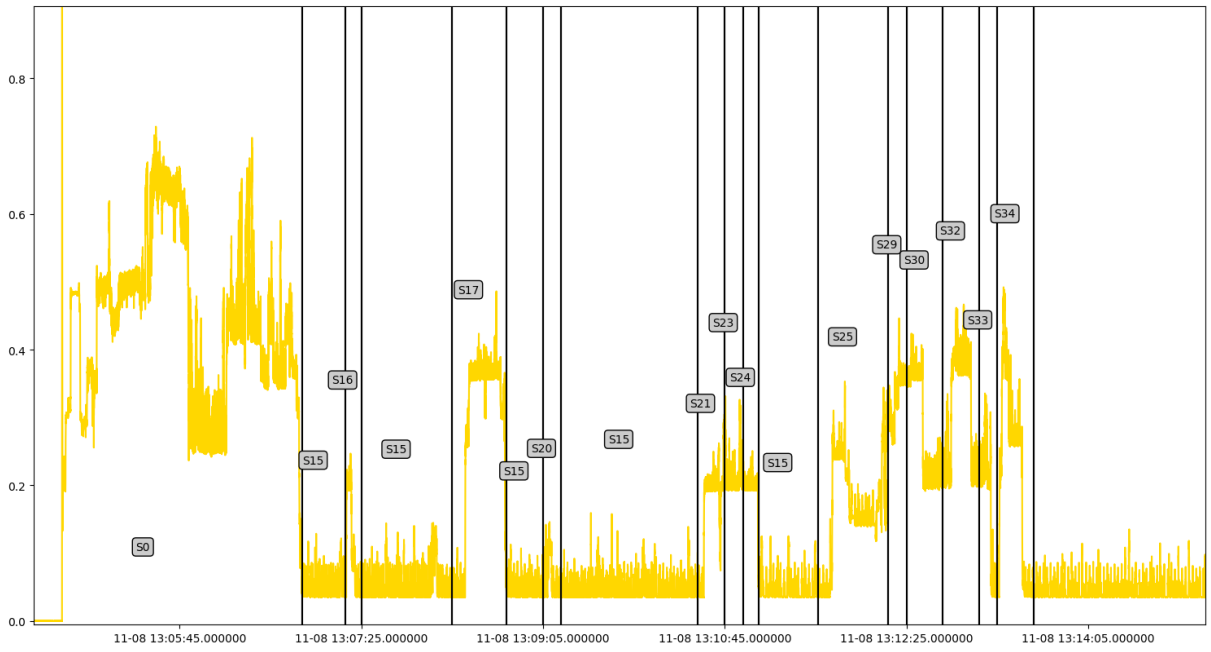
W celu przetestowania skuteczności opisywanego algorytmu i systemu, autor wykonał eksperyment analizy pobieranej energii przez urządzenie typu holter. Dnia 08.11.2019 o godz. 13:04 *Holter #1* został podłączony do sztucznej baterii. Pomiar prądu trwał do godziny 15:40. Napięcie zasilania zostało ustalone na poziomie 4.1V. Agregacja próbek pomiarowych wykonana została z okresem próbkowania równym 2 ms. Zakres wartości próbki wynosił 3A oraz 262144 poziomów próbkowania. Podczas trwania eksperymentu operator wykonywał czynności zmieniające stan logiczny urządzenia lub generujące zdarzenia takie jak: 1) włączenie badanego urządzenia, 2) włączenia i wyłączenia ekranu urządzenia, 3)

generowanie zgłoszenia sytuacji medycznej do centrum monitoringu, 4) uruchomienie badania EKG, 5) odłączenie przystawki HOLTER <-> UART służącej do konfiguracji urządzenia. Po zakończeniu agregacji plik z surowymi danymi osiągnął rozmiar 484 MB. Następnie system *StateEventAnalyzer* został uruchomiony z parametrami algorytmu: (0) liczba próbek na segment N=500, (1) *avg_eps=0.01*, (2) *deviation_eps=0.009*, (3) *min_eps*=0.007, (4) *max_eps=0.050*. Program wykonał algorytm detekcji stanów energetycznych oraz wygenerował wykres poboru prądu zaprezentowany na rysunku 17.



Rysunek 17. Wykres poboru prądu za cały badany okres z naniesionymi etykietami wykrytych stanów

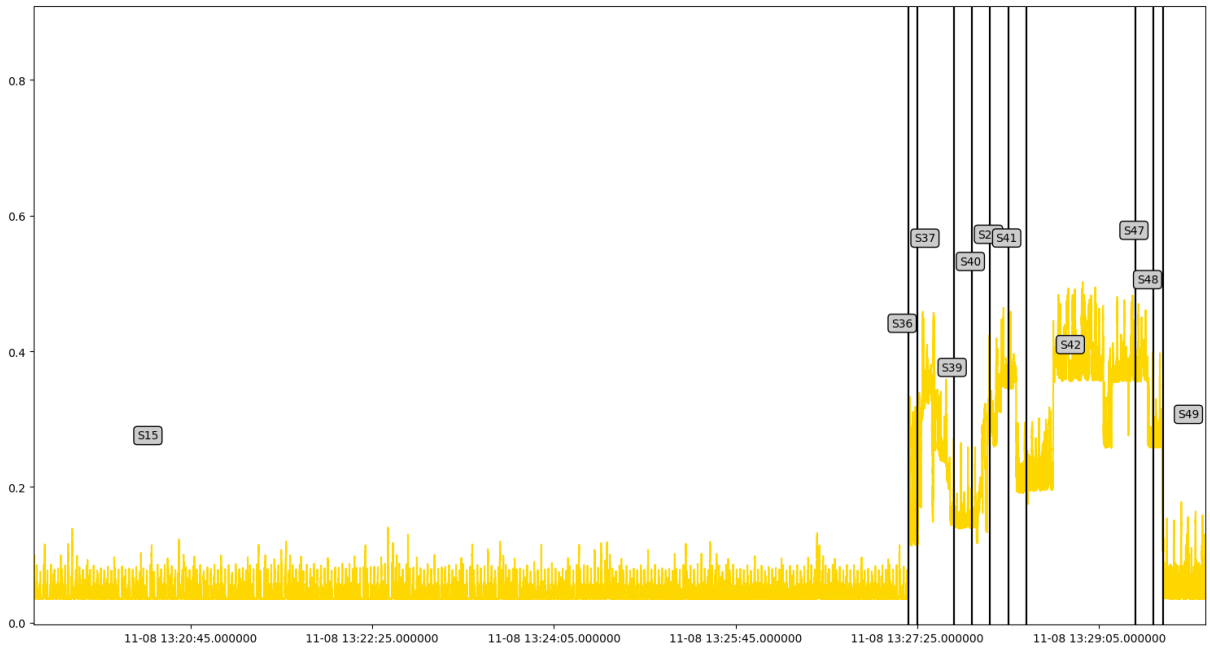
Z powodu dużego rozmiaru danych oraz zakresu czasu analizowanego wykresu wiele etykiet oraz linii dzielących stany nakłada się na siebie zmniejszając czytelność danych. Ponadto w prezentowanym widoku nie można zauważyć wszystkich szczegółów, które wpłynęły na podział na stany energetyczne. W kolejnych paragrafach podrozdziału opisane zostały kolejne fragmenty wykresu wraz z wykrytymi stanami.



Rysunek 18. Wykres poboru prądu w pierwszych chwilach po uruchomieniu urządzenia.

Rysunek 18 przedstawia pierwszy fragment wykresu pobieranego prądu przez *Holter #1*. Jako stan S0 został sklasyfikowany profil energetyczny urządzenia od chwili włączenia zasilania do momentu zakończenia uruchamiania ostatniego serwisu oprogramowania systemowego. Na rysunku można również zauważyć, że stan S15 jest stanem, w którym zaobserwowano najmniejszą wartość pobieranego prądu na poziomie 36mA. Skok wartości prądu oznaczony jako S16 był spowodowany komunikacją z serwerem DHCP oraz uzyskaniem adresów IP i DNS. O godz. 13:08:36 system wykrył brak odpowiedzi na ramki USB modemu. W wyniku czego system dokonał ponownej enumeracji urządzeń USB oraz ponownie uzyskał połączenie z siecią Internet przez APN operatora LTE. Jednym z elementów procesu oznaczonego jako S17 była także ponowna komunikacja z serwerem DHCP objawiająca się impulsem 460 mA. W stanie S20 system wybudził się i zwalniał pamięć operacyjną (zadziałał Garbage Collector maszyny wirtualnej Dalvik). Stany S21, S23 oraz S24 opisują moment ponownej utraty komunikacji między procesorem a modemem LTE. Próba odzyskania komunikacji polegała na ponownej enumeracji urządzeń USB oraz rozpoczęciu wymiany komend AT. Modem nie odpowiedział na pierwszą komendę AT. Procedura odzyskania komunikacji z modemem konfiguruje magistralę USB oraz modem LTE w tryb uniemożliwiający usypianie modemu jak i magistrali USB. Taka konfiguracja skutkuje średnim poborem prądu na poziomie 223mA przez wszystkie trzy stany. Gdy upłynął czas oczekiwania na odpowiedź (30s) modem został odłączony od zasilania, przez co urządzenie przeszło

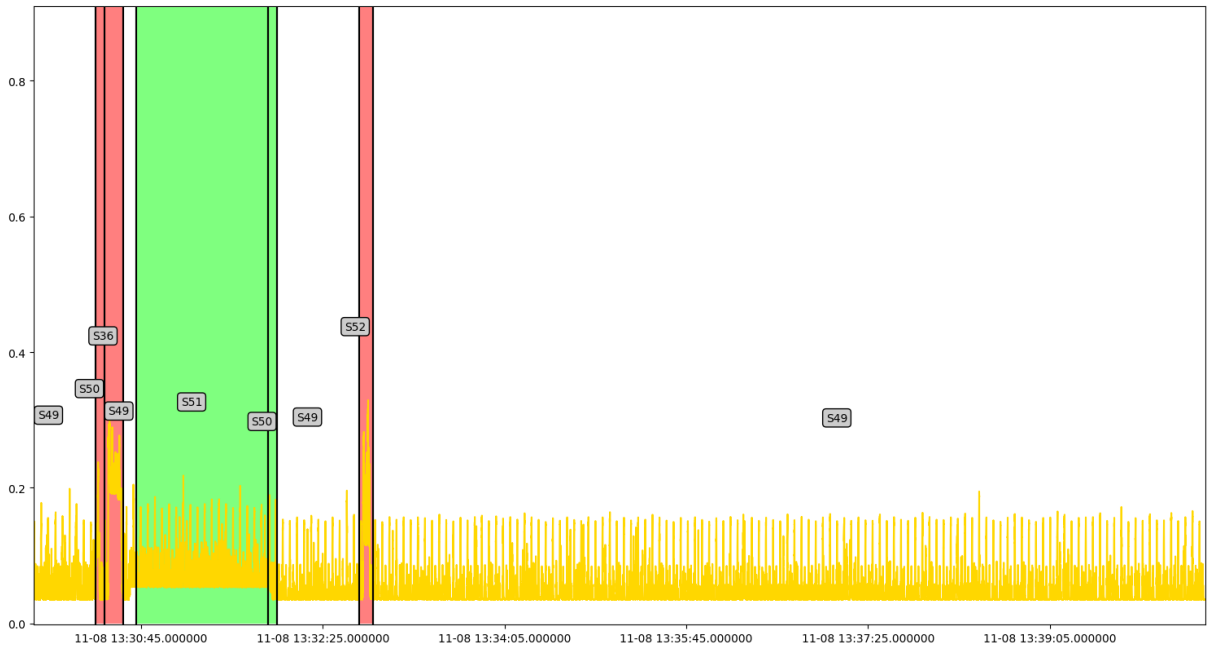
do stanu S15. Po upływie 40s zasilanie modemu zostało ponownie włączone oraz *Holter #1* rozpoczął procedurę konfiguracji modemu komendami AT, a także nawiązania uzyskania adresów IP i DNS z usługi DHCP. Proces ten skutkuje poborem prądu oznaczonym S25, S29, S30, S32, S33 oraz S34. Po skonfigurowaniu modemu *Holter #1* przeszedł do stanu S15 (USB uśpione, modem wyłączony, procesor uśpiony i wybudzany periodycznie).



Rysunek 19. Wykres poboru prądu podczas rozpoczynania badania EKG

Stan S36 przedstawia pobór prądu przez urządzenie w sytuacji, gdy został włączony ekran oraz wywołano 2 zdarzenia dotykowe za pomocą dłoni. Każde z tych zdarzeń jest nieobserwowalne na wykresie poboru prądu. W wyniku zdarzeń dotykowych rozpoczęto nowe badanie EKG. Podczas startowania nowego badania *Holter #1* próbuje skomunikować się z serwerem zdalnym w celu uzyskania konfiguracji badania. Stany od S37-S41 ukazują pobór energii podczas włączania modemu i jego konfiguracji. Stany S42, S47, S48 reprezentuje proces uzyskania adresów IP i DNS, a także komunikację z serwerem zdalnym w celu pobrania konfiguracji badania oraz komunikację w poszukiwaniu aktualizacji oprogramowania. Podczas włączania modemu system nie zarejestrował żadnych zdarzeń z interfejsu użytkownika i po 5s wyłączył ekran. Wyłączenie ekranu można zauważyć porównując stan S37 oraz S25. Pierwszy skok wartości poboru prądu jest znacznie wyższy (osiąga 424mA) w S37 oraz szerokość czasowo impulsu w stanie S37 jest szersza dwukrotnie. Różnica jest spowodowana włączonym ekranem, który pobiera ok 80mA. Wartość prądu po pierwszym impulsie spada jednak do tego samego poziomu w przypadku obu stanów (ok. 180mA) wskazując na moment wyłączenia

ekranu. Obserwację potwierdzają logi systemowe. Po zakończeniu komunikacji modem zostaje uśpiony wraz z magistralą USB, urządzenie przechodzi w stan niskiego poboru energii S49. Jednak w stanie S49 procesor komunikuje się co 2s poprzez interfejs SPI z mikrokontrolerem MSP430, aby odebrać pakiet zawierający próbki EKG. Komunikacja powoduje wzrost średniej wartości pobieranego prądu z poziomu 38mA do 41mA.

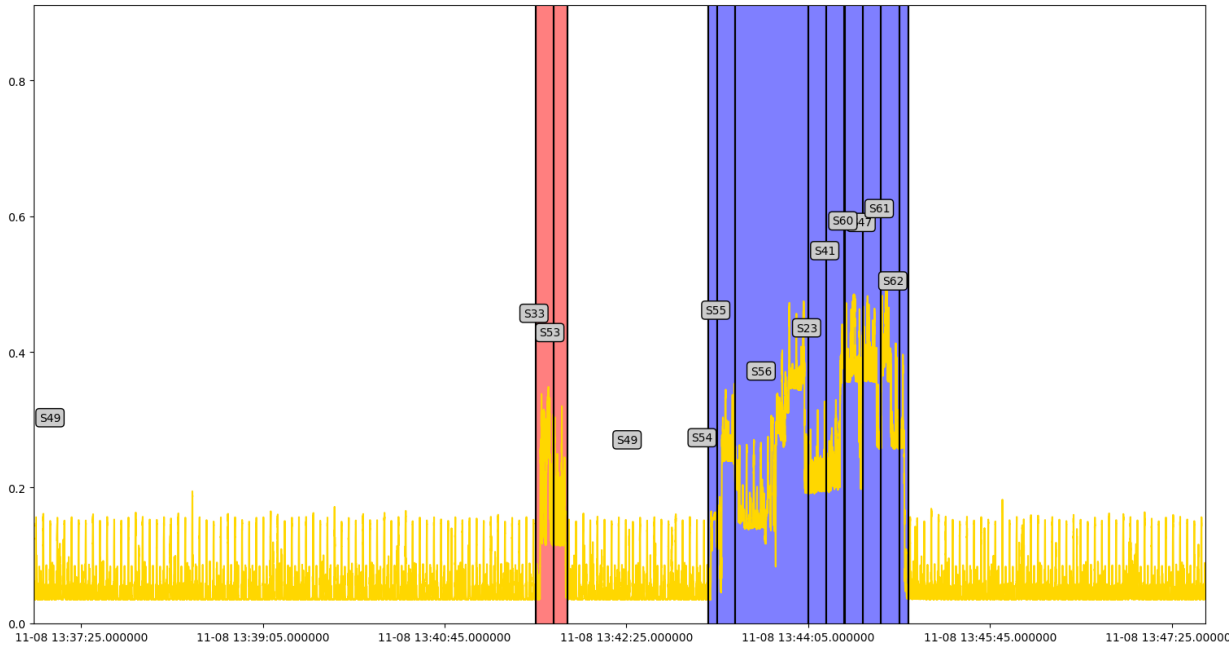


Rysunek 20. Wykres poboru prądu z oznaczeniami stanów oraz oznaczeniami klas cykli stanów

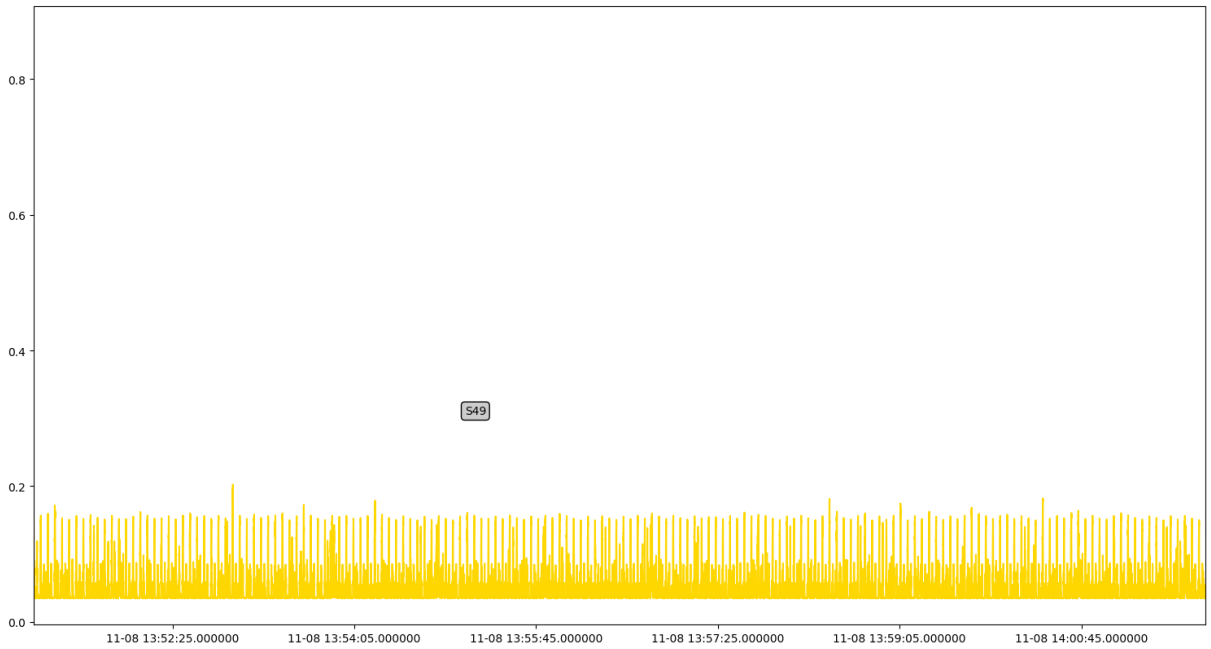
Pierwszą wykrytą instancją klasy cykli (CC1) jest klasa oznaczona czerwonym kolorem na rysunku 20. W pierwszym wystąpieniu CC1 w jej skład wchodzą stany S50 i S36, co oznacza, że impulsy prądowe i skok wartości został spowodowany chwilowym 15s. włączeniem ekranu, aby wyświetlić komunikat. Następna klasa CC2 oznaczona kolorem zielonym składa się ze stanów S50 i S51. W tym wystąpieniu klasy CC2 modem wyszedł z trybu uśpienia do trybu pracy w oczekiwaniu na odłączenie zasilania, co poskutkowało wzrostem minimum poboru prądu do poziomu 52 mA. Kolejnym wystąpieniem klasy CC1 było zdarzenie oznaczone stanem S82. W tym stanie urządzenie miało włączony ekran na żądanie użytkownika. Na żądanie użytkownika również ekran został wyłączony.

Rysunek 21 przedstawia typowe zachowanie w wyniku wystąpienia zdarzenia zgłoszenia symptomu przez użytkownika przy użyciu interfejsu dotykowego. Najpierw włączany jest ekran za pomocą głównego przycisku, następnie użytkownik wybiera odpowiedni symptom i zaznacza odpowiedzi na pytania związane z danym symptomem. Po potwierdzeniu poprawności wprowadzonych danych ekran zostaje wyłączony. Proces ten zobrazowany jest

poprzez profil energetyczny na fragmencie wykresu oznaczonym kolorem czerwonym. Jest to również kolejne wystąpienie klasy CC1 (stany S33 i S53). Jak można zauważyć instancje klasy CC1 opisują stany urządzenia, w których ekran jest uruchomiony i wyświetla dane. Po przetworzeniu informacji *Holter #1* uruchamia modem, konfiguruje go, uzyskuje parametry połączenia (adresy IP oraz DNS), zapisuje zgromadzony sygnał EKG w postaci plików na karcie SD, łączy się do serwera zdalnego i przesyła pliki. Cały proces z perspektywy poboru prądu- od uruchomienia modemu po transmisję- można zaobserwować we fragmencie wykresu oznaczonym na niebiesko na rysunku 21. Fragment oznaczony na niebiesko reprezentuje pojedyncze wystąpienie klasy CC3 składającej się w tym przypadku ze stanów S54, S55, S56, S23, S41, S60, S47, S61 oraz S62. Bardzo podobną sekwencję wystąpień klas cykli można zaobserwować także na rysunku 23. Wystąpienia CC3 na obu rysunkach w większości składają się z różnych stanów, mimo to współdzielą obiekty takie jak S61 czy S62. Różnica ta spowodowana jest tym, że przy każdorazowym uruchomieniu modemu, w zależności od wielu czynników takich jak: stan sieci LTE, błędy w oprogramowaniu modułu LTE, przeplot wątków w oprogramowaniu procesora, czasy odpowiedzi zdalnego serwera, a nawet obciążenie CPU, urządzenie może w różny sposób w czasie rozłożyć energię pobraną w celu komunikacji. W większości przypadków jednak całka pobranej energii powinna być zbliżona.

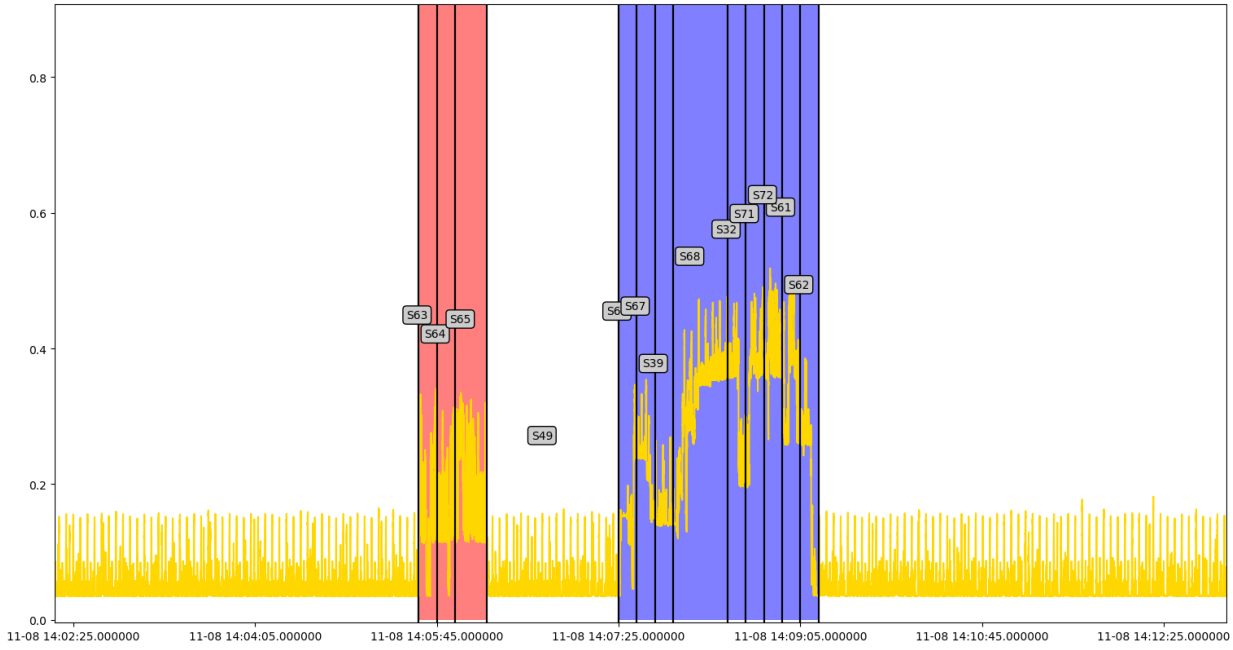


Rysunek 21. Profil energetyczny *Holtera #1* przy zgłaszaniu symptomu podczas badania (1)



Rysunek 22. Profil energetyczny *Holtera #1* podczas uruchomionego badania w stanie spoczynku

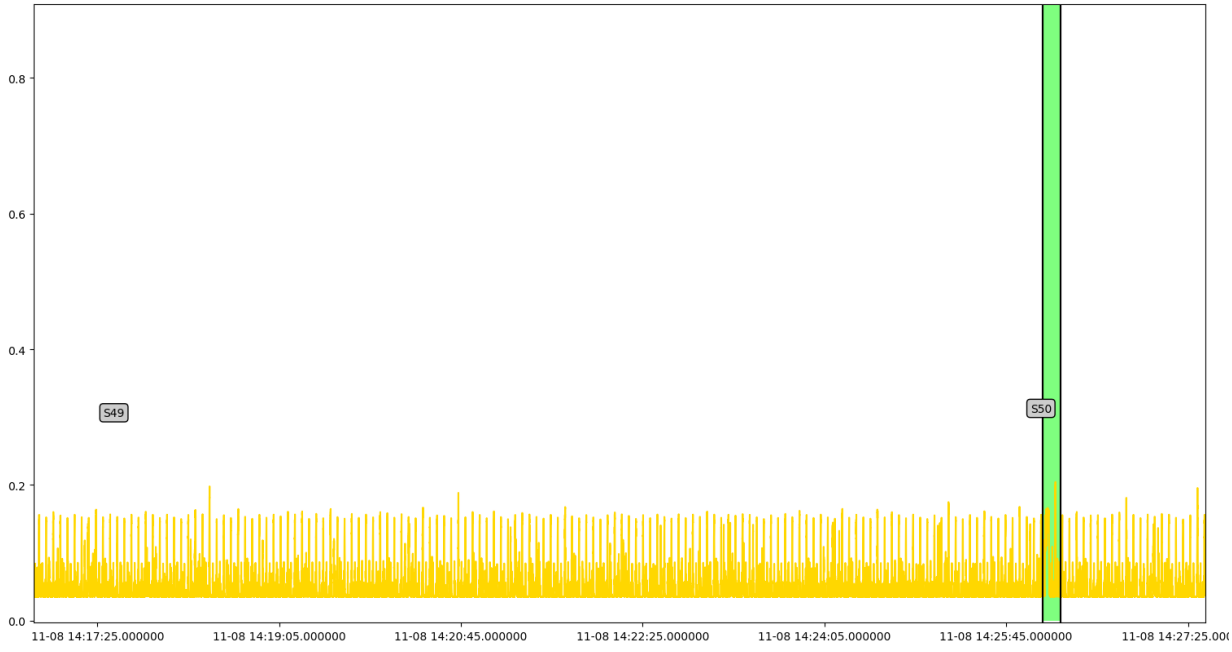
Rysunek 22 przedstawia wykres poboru prądu podczas stanu bazowego poboru prądu w warunkach regularnej komunikacji pomiędzy procesorem a mikrokontrolerem MSP430. Stan ten jest oznaczony S49.



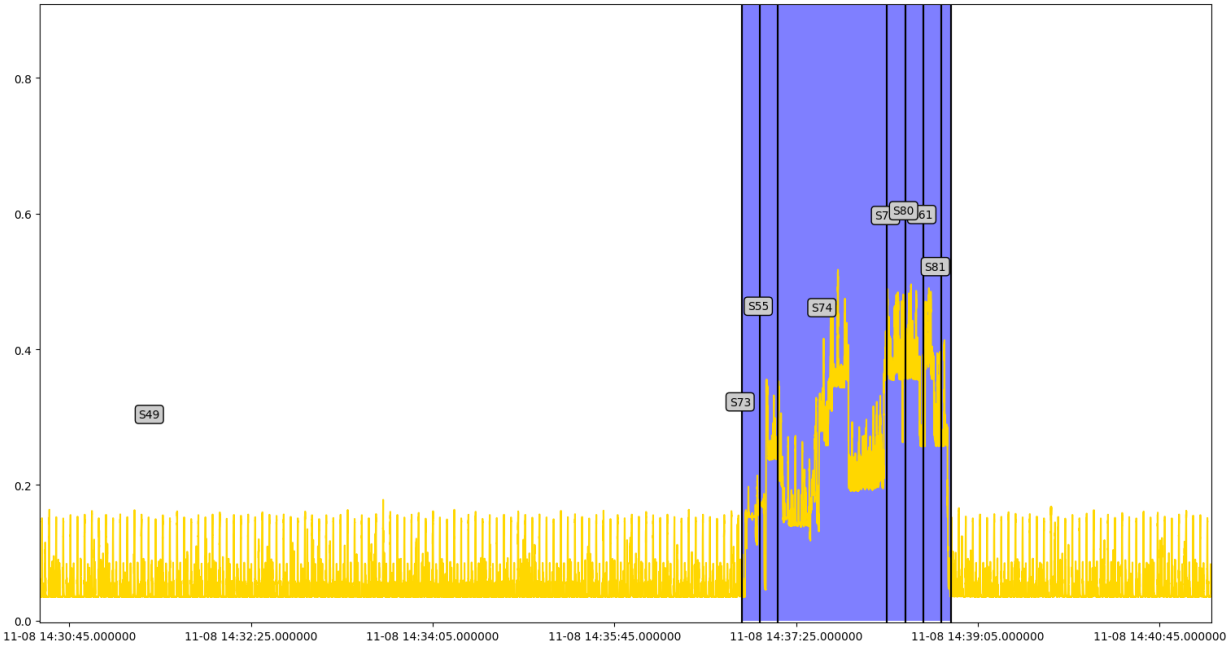
Rysunek 23. Profil energetyczny *Holtera #1* przy zgłaszaniu symptomu podczas badania (2)

Na rysunku 24 można zauważyć wykrytą anomalię S50 pomiędzy typowym spoczynkowym profilem energetycznym S49. Anomalia została zaklasyfikowana do klasy

cykli CC2 oznaczonej zielonym kolorem. Na rysunku 25 znajduje się wykres poboru prądu w stanie S50 (CC2) wraz z otoczeniem przy pięciokrotnie większej rozdzielczości osi czasu. W stanie S49 urządzenie pobiera regularnie impulsowo prąd o wartości ok 165 mA z częstotliwością 2s. Pomiędzy impulsami osiągającymi maksymalną wartość dla tego stanu można zauważyć dwa dodatkowe o mniejszej wartości szczytowej. Pierwszy impuls jest odbiciem zapotrzebowania energetycznego podczas komunikacji po interfejsie SPI pomiędzy procesorem głównym a mikrokontrolerem urządzenia. Kolejne dwa powstają w wyniku działania procesora głównego, który najpierw przetwarza dane w celu klasyfikacji załamków sygnału EKG oraz detekcji niebezpiecznych dla zdrowia zdarzeń, a następnie buforuje przetworzone dane w pamięci operacyjnej urządzenia. O godzi 14:26 urządzenie weszło w stan S50 (CC2). W tym stanie *Holter #1* zapisywał zgromadzone dane w postaci sekwencji plików na karcie SD. Na rysunku 25 można zauważyć, że pierwszy impuls o maksymalnej wartości 164 mA trwał ok 5s, co wyróżnia S50 od S49. Pozostałe impulsy są tłem który jest związany z profilem energetycznym stanu S49. Tło spowodowane jest tym, że urządzenie mimo potrzeby zapisywania danych na karcie SD, równolegle wykonuje zadania typowe dla stanu spoczynkowego: komunikacja z mikrokontrolerem i agregacja próbek EKG.

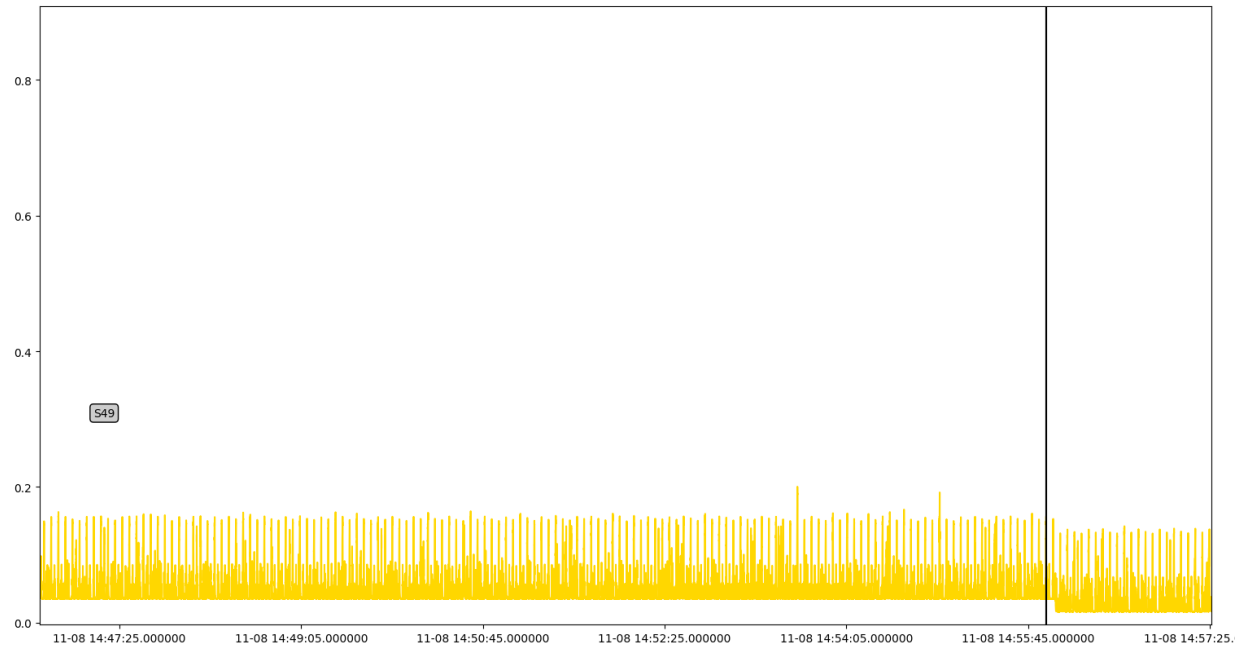


Rysunek 24. Wykres poboru prądu z oznaczoną wykrytą anomalią

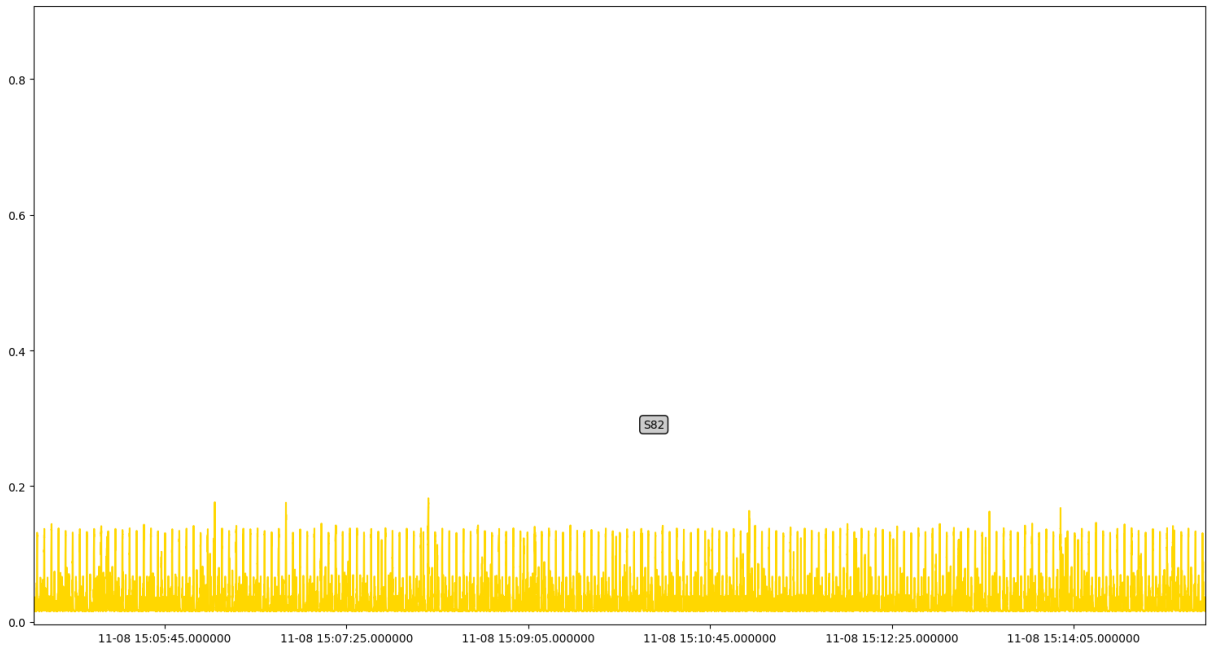


Rysunek 25. Profil energetyczny podczas uruchamiania i wysyłania danych na zdalny serwer

Na rysunku 25 naniesiony jest wykres poboru prądu podczas kolejnej transmisji danych EKG na zdalny serwer. Stany, które reprezentują ten proces, zostały zaklasyfikowane do klasy cykli CC3 i oznaczona kolorem niebieskim na wykresie. Logi systemowe skorelowane z wartościami czasowymi stanów S73-S81 pokazują, że powód transmisji nie był anomalią. Urządzenie regularnie co 1 godzinę i 10 minut wysyła zagregowane i przetworzone dane.

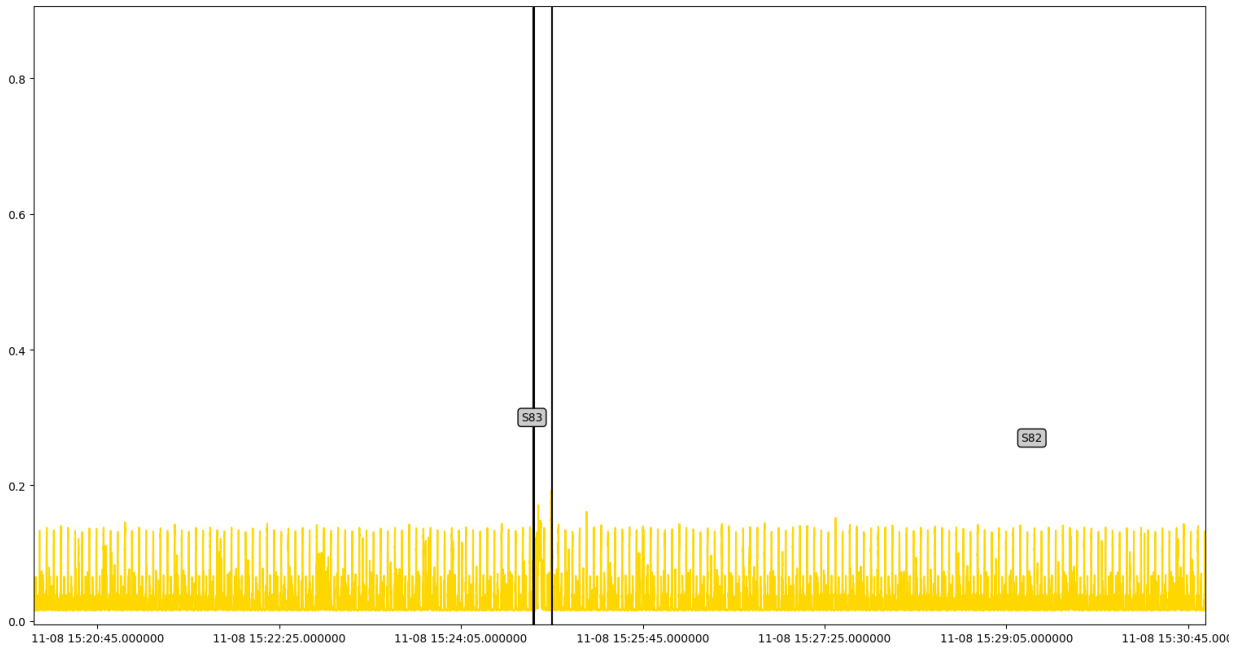


Rysunek 26. Wykres zmiany poboru prądu po odłączeniu przystawki serwisowej

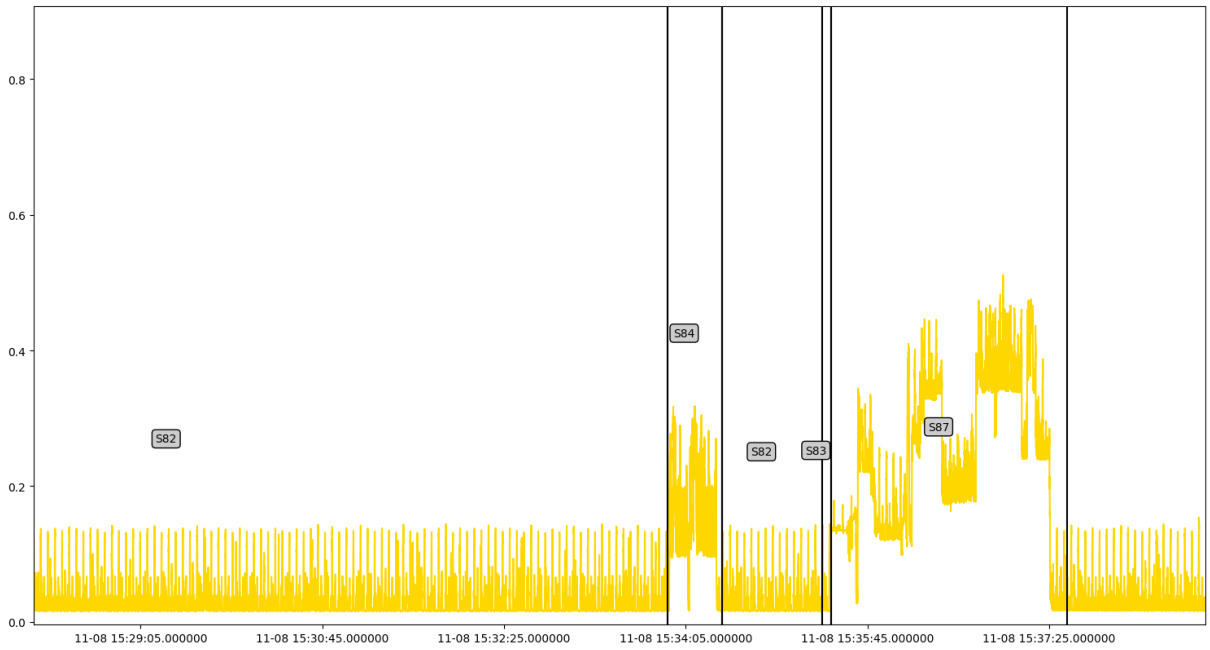


Rysunek 27. Wykres poboru prądu w stanie minimalnego poboru energii po odłączeniu przystawki serwisowej.

Podczas pierwszych 2 godzin do badanego systemu podłączona była przystawka serwisowa umożliwiająca komunikację z interfejsem konsolowym urządzenia. Przystawka pobierała stały prąd o wartości 19mA. O godzinie 14:56 (rysunek 26) przystawka została odłączona, co spowodowało spadek poboru prądu o 19 mA, zmieniając wartości parametrów statystycznych wszystkich kolejnych segmentów. Z tego powodu stan bazowego poboru energii został oznaczony nową etykietą S82 (rysunek 27). Z kolei stan S83 oznaczony na rysunku 28 jest tożsamy ze stanem S50 w ramach CC2, co pokazuje różnica między rysunkiem 30 i 31. Na obu rysunkach w ramach analizowanego stanu występuje charakterystyczny impuls trwający ok 4s. Podobieństwo również potwierdzają rekordy logów systemowych wskazujące na zapis danych na karcie SD w postaci sekwencji plików.



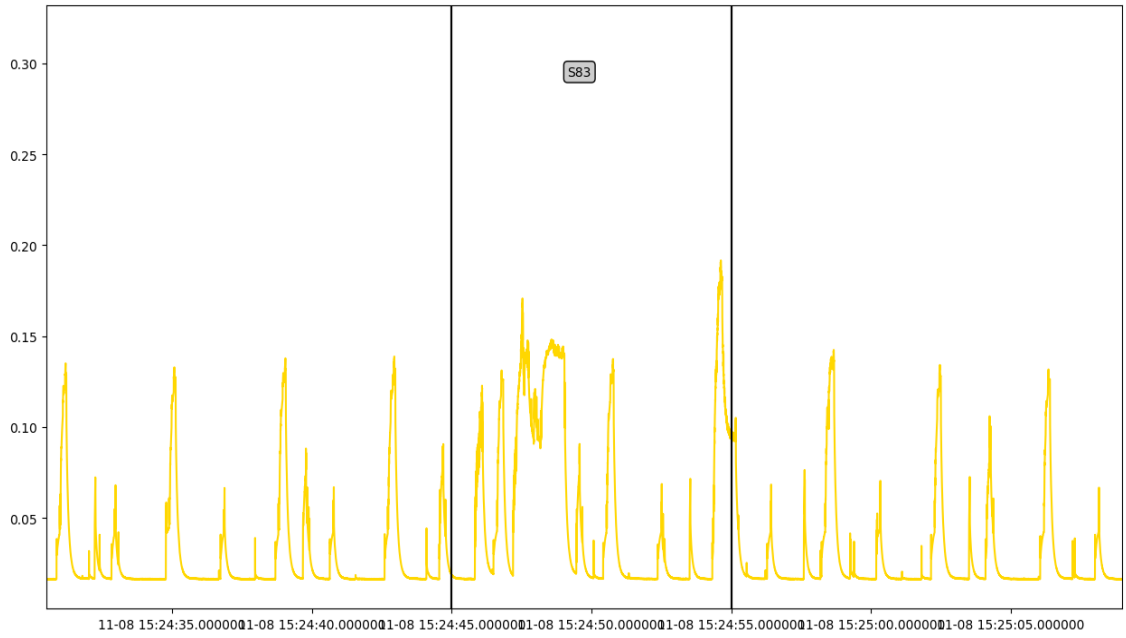
Rysunek 28. Wykres poboru energii podczas anomalii oznaczonej jako stan S83



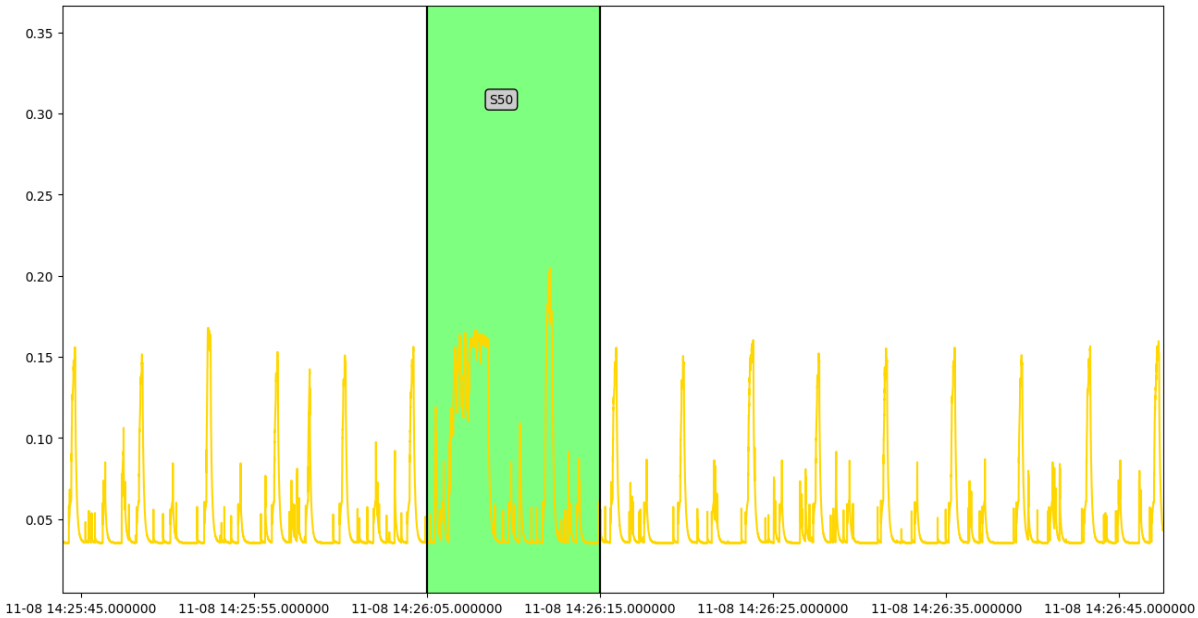
Rysunek 29. Wykres poboru prądu podczas ostatniego wygenerowanego zdarzenia zgłoszenia symptomu

Analogicznie czytając rysunek 29 można zauważyć, że stan S84 jest odpowiednikiem stanów z CC1, a S87 jest równoważne z SG2. Rysunek 29 przedstawia wykres poboru prądu podczas kolejnej transmisji danych spowodowanej zgłoszeniem symptomu przez użytkownika. Stan S84 wskazuje na włączenie ekranu (skok poboru prądu do poziomu ok 300 mA), wybranie

przez użytkownika symptomu oraz jego potwierdzeniem. Następnie ekran jest wyłączany (S82). W stanie S87 modem zostaje włączony i następuje transmisja danych.



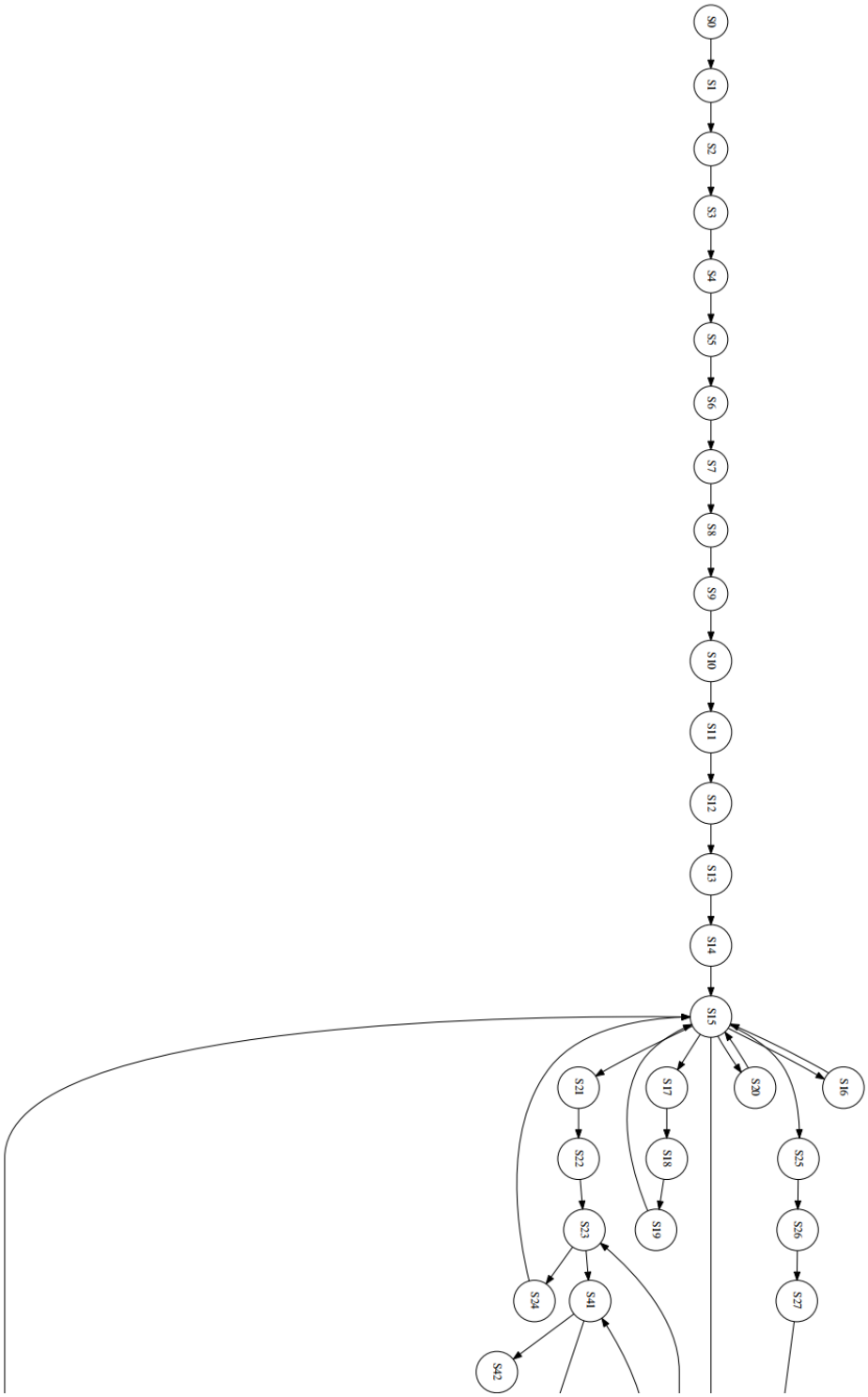
Rysunek 30. Wykres poboru prądu podczas zapisu na karcie SD przy odłączonej przystawce serwisowej

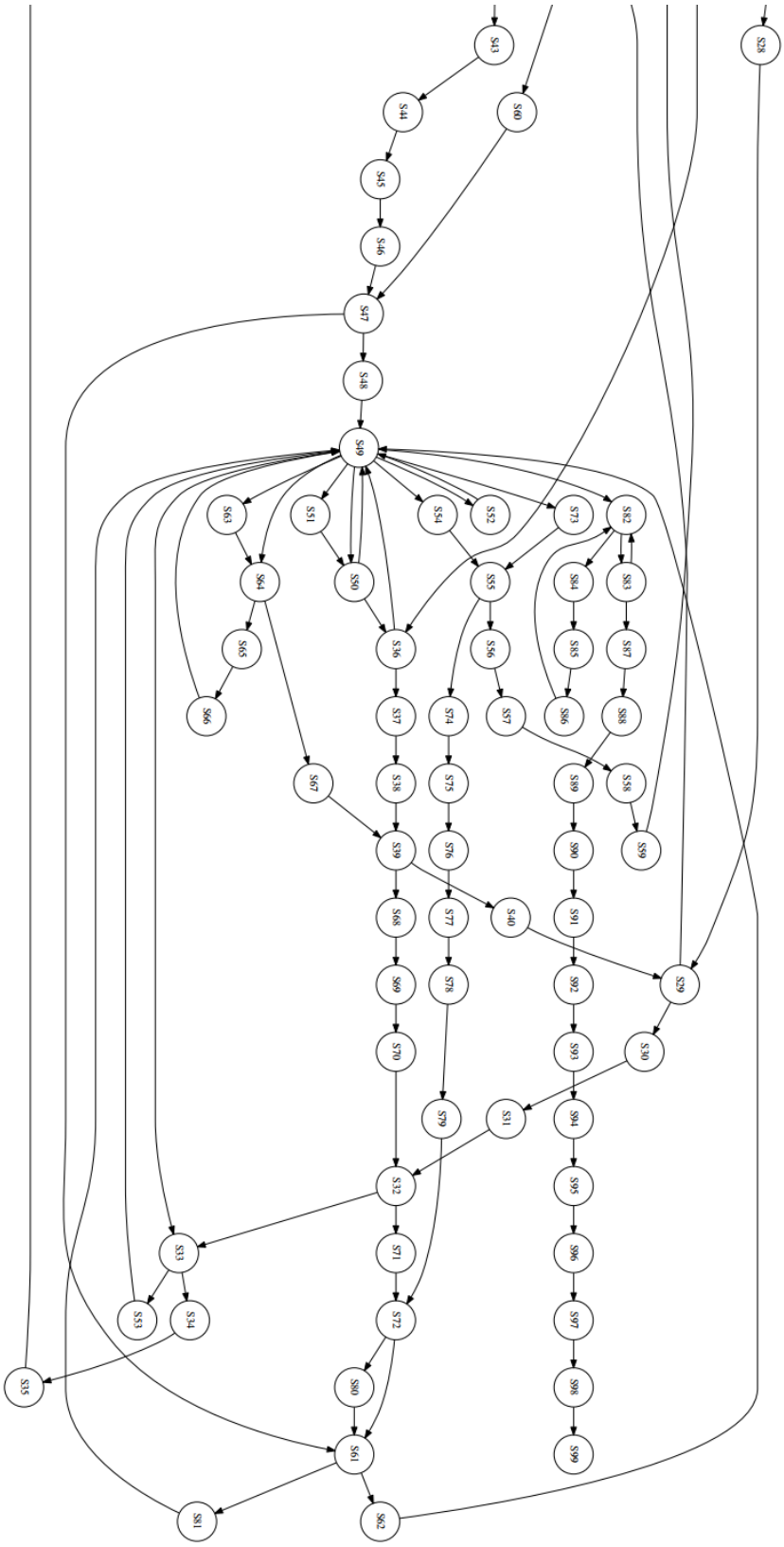


Rysunek 31. Wykres poboru prądu podczas zapisu na karcie SD

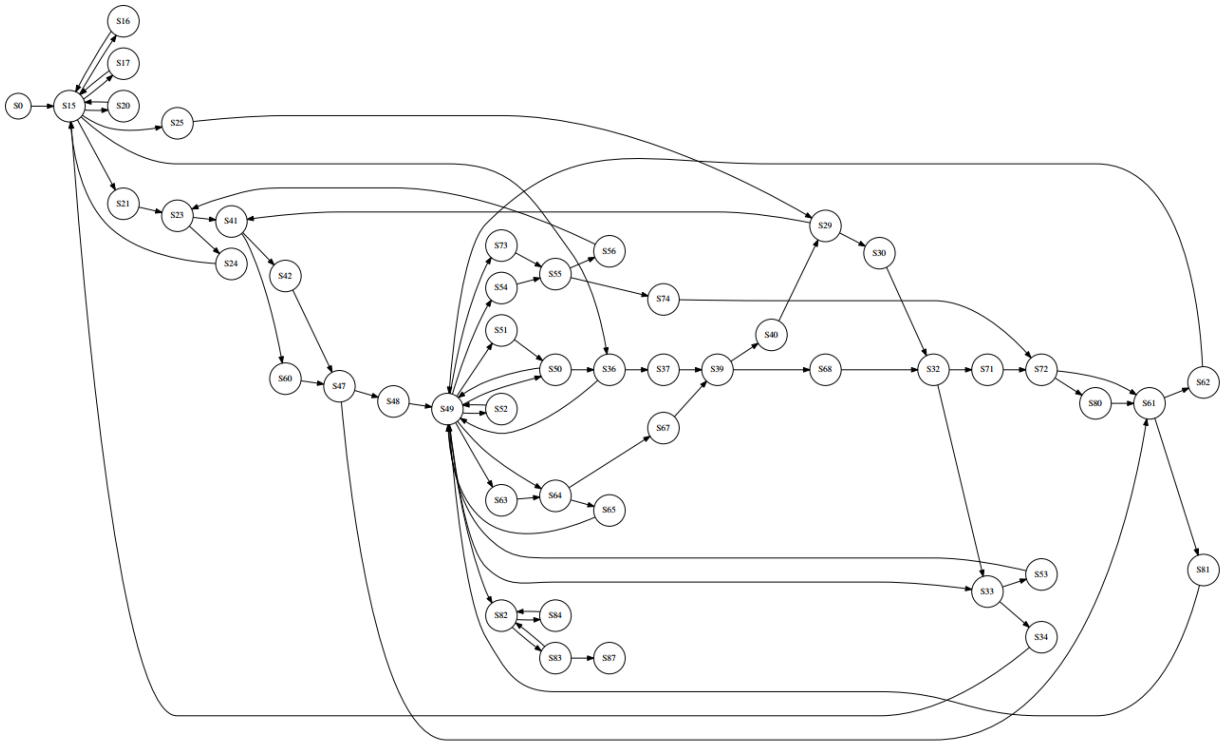
Stany S84 oraz S87 nie zostały podzielone na krócej trwające/mniejsze, z tego powodu, że proces zgłaszania symptomów i transmisji danych po odłączeniu przystawki wystąpił tylko raz. Pojedyncze wystąpienie sekwencji mniejszych stanów powoduje, że procedura redukcji

stanów algorytmu scala te stany w jeden większy (większość takich stanów klasyfikowana jako stany nadmiarowe). Ponadto żaden z tych stanów nie został sklasyfikowany do klasy cyklów, ponieważ nie sąsiadował ze stanem bazowym. Na stan bazowy został wybrany zgodnie z algorytmem stan S49.





Rysunek 32. Graf przejść między stanami przed redukcją stanów prostych



Rysunek 33. Zredukowany graf przejść między stanami

Rysunek 33 przedstawia graf przejść między stanami wygenerowany przez program implementujący opisywany algorytm podczas testów z wykorzystaniem *Holtera #1*. Na rysunku 32 znajduje się graf przejść między stanami wraz z usuniętymi stanami prostymi. Graf z rysunku 33 jest czytelniejszy od grafu z rysunku 32. Ponadto uproszczony graf służy jako podstawa do klasyfikacji sekwencji stanów do klas cykli w ostatnim kroku algorytmu.

Warto zauważyć, że w prezentowanym przypadku zmiana wykrywania stanu bazowego z globalnego wyszukiwania na lokalny pozwoliłaby na wykrycie dwóch stanów bazowych (przed i po odłączeniu przystawki UART, która spowodowała zmianę profili energetycznych klas cykli i stanów). Ponadto obserwacje wyników działania algorytmu potwierdzają, że hierarchiczny model danych wykorzystywany przez algorytm umożliwia detekcję anomalii oraz oznaczanie podobnych zachowań badanego systemu jako obiekty tego samego typu np. wystąpienia klasy cykli CC3 były tożsame z procesem udanego wysyłania danych EKG na serwer.

5.2 Korelacja logów systemowych z modelem energetycznym Holtera #1

W ramach analizy algorytmu przedstawionego w rozdziale 4 przeprowadzona została korelacja logów tekstowych zebranych przez urządzenie *Holter #1* w eksperymencie z rozdziału 5.1. Jako obserwację zewnętrzną wybrano wystąpienia klasy cykli CC3. Znaczniki czasowe wystąpień klasy cykli są następujące {(2020-11-08 13:43:10, 2020-11-08 13:45:00), (2020-11-08 14:07:25, 2020-11-08 14:09:15), (2020-11-08 14:36:40, 2020-11-08 14:40:00)}. Pozostałe parametry konfiguracyjne wynoszą: 1) *eps_s* = 0.3, 2) *eps_time* = 2m. 8s., 3) *eps_tolerance*= 1s., 4) wagi słów *WT(x)* ={*word* = 1.0; *pid* = 2.0; *source* = 3.0; *non_word* = 0.0}. Korelowane logi były w pojedynczym pliku tekstowym o 19716 rekordach logów (2.01MB) obejmujących czas od 11-08 07:05:04.093 do 11-08 09:43:38.651. Wyniki działania algorytmu przez system *StateEventAnalyzer* są dostępne przez komendy *print_correlated_logs* i *print_correlated_offsets*.

W wyniku działania algorytmu dopasowanych zostało 1602 rekordy logów. Większość rekordów jest powtarzana w każdej z co najmniej 3 sekwencji dopasowanych do zdarzeń OI. Sekwencje pochodzą z odfiltrowanych 11 zbiorów worków podobnych słów. Biorąc pod uwagę liczbę zbiorów oraz liczbę dopasowanych sekwencji można stwierdzić, że na pojedyncze zdarzenie przypada średnio 1600/3=550 rekordów logów, które są przeglądane i analizowane w 11 klasach (w każdej z nich średnio przypada 550/11=50 rekordów logów). Taka struktura znacząco ułatwia dalszą analizę i ustalenie faktycznego znaczenia korelowanych zdarzeń. Dopasowane szeregi sekwencji powstały z rekordów logów wygenerowanych przez moduły: 1) AT, 2) RILJ, 3) DHCP, 4) *MobileDataStateTracker*, 5) *SocketSenderThread* oraz 6) DCT. Komponent AT jest to fragment oprogramowania odpowiedzialny bezpośrednio za formatowanie i analizę tekstu komunikatów wysyłanych i odbieranych od modemu GSM. Cała komunikacja związana z logiką działania modemu jest logowana z poziomu komponentu AT. RILJ jest komponentem serwisu systemu Android zarządzającym wszelkimi systemami bezprzewodowymi. Serwis loguje jako RILJ zmiany związane ze stanem komponentów sprzętowych i oprogramowania podsystemów bezprzewodowych- w szczególności modemu LTE. Loguje komunikaty takie jak rozkaz włączenia czy wyłączenia zasilania modemu, podniesienia poboru prądu przez modem poprzez wydanie rozkazu podłączenia się do sieci Internet itp. Serwis DHCP implementuje obsługę protokołu DHCP potrzebnego do uzyskania konfiguracji warstwy 3 w modelu OSI (warstwy IP) takiej jak adres IP, maska sieci, adres głównej bramy. Zwykle serwis DHCP jest aktywowany po podłączeniu do sieci Internet

oraz gdy dzierżawa adresu IP zostanie cofnięta lub jej termin ważności minie. *MobileDataStateTracker* jest komponentem systemu Android, który jest częścią *NetworkManagera*- serwisu zarządzającego pakietowymi interfejsami sieciowymi. Moduł logujący jako *SocketSenderThread* jest fragmentem bibliotek przestrzeni użytkownika obsługujących interfejs gniazd sieciowych. Ten komponent loguje stworzenia, otwarcia czy zamknięcia gniazd sieciowych przez daną aplikację użytkownika lub serwis. W szczególności *SocketSenderThread* informuje o nieobsłużonych czy niespodziewanych wyjątkach działania, które mogą sygnalizować błędy czy nagłe zamknięcia połączeń TCP. Powodem takiego zamknięcia często może być reset samego modemu LTE czy też niepoprawna obsługa wyłączenia modułu. Komponent DCT jest fragmentem oprogramowania serwisu odpowiedzialnego za śledzenie przepustowości połączenia. Na etapie analizy samych źródeł oprogramowania, które logowały rekordy dopasowane przez algorytm do zdarzeń, można wywnioskować, że zmiana w stanie poboru prądu jest związana ze zmianą w stanie łączności *Holtera #1* z siecią LTE i siecią Internet. Jednak, aby dokładnie wskazać przyczynę należy przejrzeć rekordy logów, w szczególności ze źródeł AT i DCT, gdyż one pozwolą na analizę zmiany stanu urządzenia od dwóch skrajnych warstw oprogramowania- warstwy komunikacji ze sprzętem i warstwy aplikacyjnej.

Przeglądając worki słów, z które posiadają słowo AT o typie równym źródło (*source*) można zauważyć rekord: *Time normalized:* 0:41:40.469000 *log:*<[source], AT> <[pid], 134> <[word], at> <[word], cgact>. Pochodzi on z rekordu: R11-08 07:45:19.034 D/AT (134): AT> AT+CGACT=1,3. Rekord ten powtarza się w czasie każdego z zakresów opisujących zdarzenia wejściowe. Jest to komenda, która wysyłana jest do włączonego modemu, aby połączyć modem z konkretnym kontekstem APN (ang. Access Point Name). Jest to operacja wymagająca dużej wymiany danych na poziomie sprzętowym z stacją bazową. Jest to też moment prawdopodobnego silnego wzrostu poboru prądu przez urządzenie. Innym rekordem wskazującym na możliwy duży pobór prądu jest: *Time normalized:* 0:41:42.881000 *Log:* <[*source*], DCT> <[*pid*], 542> <[*word*], setupdataonconnectableapns> <[*word*], apncontext> <[*word*], {mapntype> <[*word*], default> <[*word*], mstate> <[*word*], idle> <[*word*], mwaitingapns> <[*word*], {> <[*word*], apnsettingv> <[*word*], verizon> <[*word*], internet> <[*word*], null> <[*word*], null> <[*word*], null> <[*word*], null> <[*word*], null> <[*word*], default> <[*word*], supl> <[*word*], ip> <[*word*], ip> <[*word*], true> <[*word*], }> <[*word*], mwaitingapnspermanentfailurecountdown> <[*word*], mapnsetting> <[*word*], {null}> <[*word*], mreason> <[*word*], radioturnedoff> <[*word*], mdataenabled> <[*word*], true> <[*word*], mdependencymet> <[*word*], true}>. Wskazuje on na to, że rozkaz podłączenia się do sieci

wyszedł z najwyższych warstw oprogramowania, a więc jest intencjonalnym zachowaniem spowodowanym przez algorytm sterujący włączaniem i wyłączaniem sieci na żądanie w celu oszczędzania energii przez *Holter #1*. Pozostałe rekordy również wskazują na operacje związane z: włączaniem modemu, jego konfiguracją, nawiązaniem połączenia z siecią, połączeniem do serwera, wysłaniem danych, a następnie zamknięciem wszystkich procesów związanych z obsługą sieci.

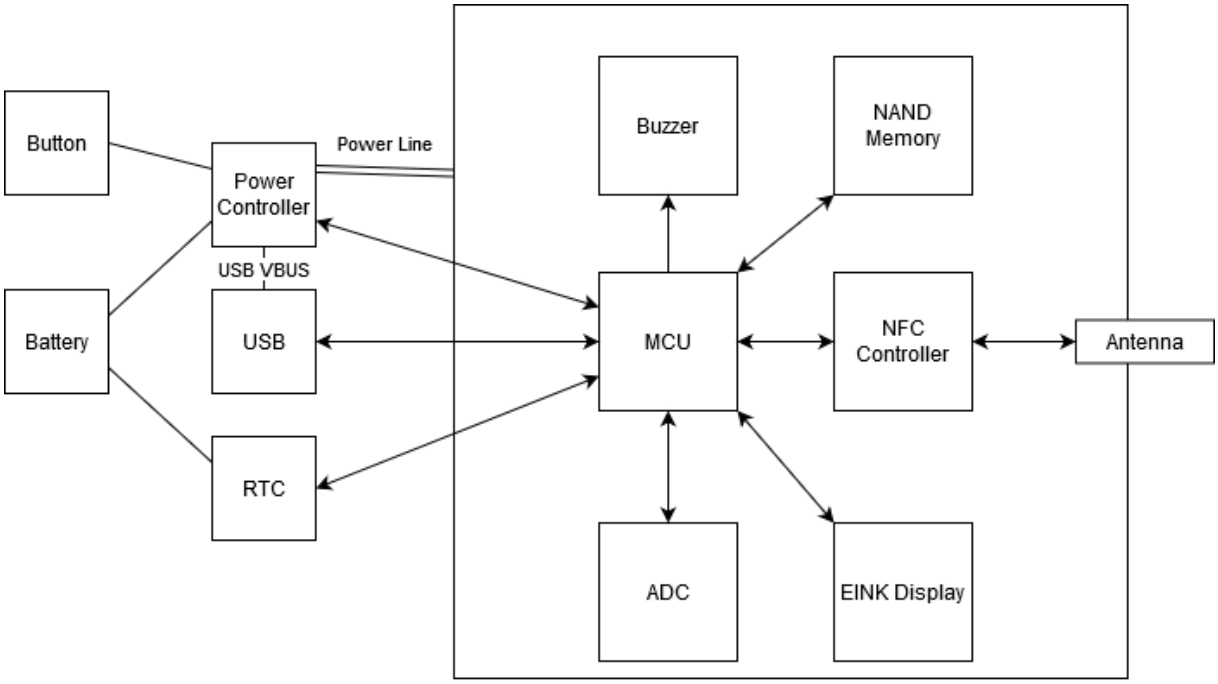
W wyniku działania algorytmu możliwe było osiągnięcie w sposób zautomatyzowany podobnych efektów, jakie zostały wykonane ręcznie poprzez przeglądanie logów systemowych w rozdziale 5.1. Opis zdarzeń, które spowodowały podwyższony pobór energii jest tożsamy z opisem z poprzedniego rozdziału. Warto odnotować, że prezentowany algorytm ograniczył liczbę rekordów logów, które należy przeanalizować, aby określić przyczyny zmian w poborze prądu z 19716 do 1602. Ponadto już sama selekcja źródeł oprogramowania logujących opisywane 1602 rekordy logów pozwala w przybliżeniu określić przyczynę anomalii w poborze prądu.

5.3 Analiza stanów energetycznych oraz anomalii w Holterze #2

Urządzenie typu holter patch (nazywane dalej *Holter #2*) jest agregatorem pojedynczego kanału sygnału EKG, który jest jednokrotnie przytwierdzany do klatki piersiowej pacjenta za pośrednictwem samoprzylepnych elektrod. Urządzenie dedykowane jest czternastodniowym ciągłym badaniom. Podczas agregacji danych urządzenie nie łączy się zdalnie z centrum monitoringu, tylko zapisuje próbki EKG, przetwarzając je w minimalnym stopniu w pamięci wewnętrznej. W przeciwieństwie do urządzenia z rozdziału 5.1. *Holter #2* wykonuje całe badanie na pojedynczej baterii CR2032. Z tego powodu zarządzanie energią w urządzeniu jest wyjątkowo ważne, aby umożliwić funkcjonowanie przez 14 dni ciągłego próbkowania i zapisu danych. Z powodu ograniczonych zasobów urządzenie nie generuje logów tekstowych. Jedyną możliwością analizy zmian stanów urządzenia przez zewnętrzne systemy, jest analiza sygnałów zewnętrznych urządzenia w szczególności szeregu próbek natężenia prądu pobieranego przez urządzenie. Opisana właściwość ogranicza zastosowanie algorytmu do procedur z rozdziału 3.1.

Holter #2 jest systemem wbudowanym o niewielkich rozmiarach, służącym do akwizycji jednokanałowego EKG. Urządzenie z dwoma odprowadzeniami na elektrody EKG jest przyklejane do ciała osoby badanej. Po sparowaniu urządzenia z badaniem za pośrednictwem interfejsu NFC, *Holter #2* rozpoczyna akwizycję sygnału, który jest zapisywany w pamięci NAND. Stan *Holtera #2* jest wyświetlany na ekranie typu EINK.

Zmiany stanu sygnalizowane są sygnałem dźwiękowym. Pacjent ma możliwość zgłaszania wystąpienia symptomów chorobowych za pośrednictwem przycisku oraz poprzez aplikację NFC.



Rysunek 34. Schemat logiczny wysokiego poziomu architektury sprzętowej *Holtera #2*

Holter #2 zasilany jest z baterii CR2032. Bateria pozwala na przeprowadzenie czternastodniowego badania. Podczas odczytywania sygnału EKG system zasilany jest interfejsem USB.

Tryb poboru mocy	Częstotliwość zegara	12 MHz	96MHz
Active		System wykonuje instrukcje z zadaną częstotliwością. Stan energetyczny peryferiów zależy od konkretnego stanu	
Sleep		System wstrzymuje wykonywanie instrukcji. Stan energetyczny peryferiów zależy od konkretnego stanu	
Powerdown		Rdzeń i pamięć operacyjna są w trybie podtrzymania. Wszystkie urządzenia peryferyjne są wyłączone z wyjątkiem RTC i GPIO	

Shutdown	Procesor i wszystkie urządzenia z wyjątkiem zewnętrznego RTC są odcięte od zasilania przez układ kontrolera zasilania.

Tabela 7. Dostępne konfiguracje oszczędzania energii w *Holterze #2*

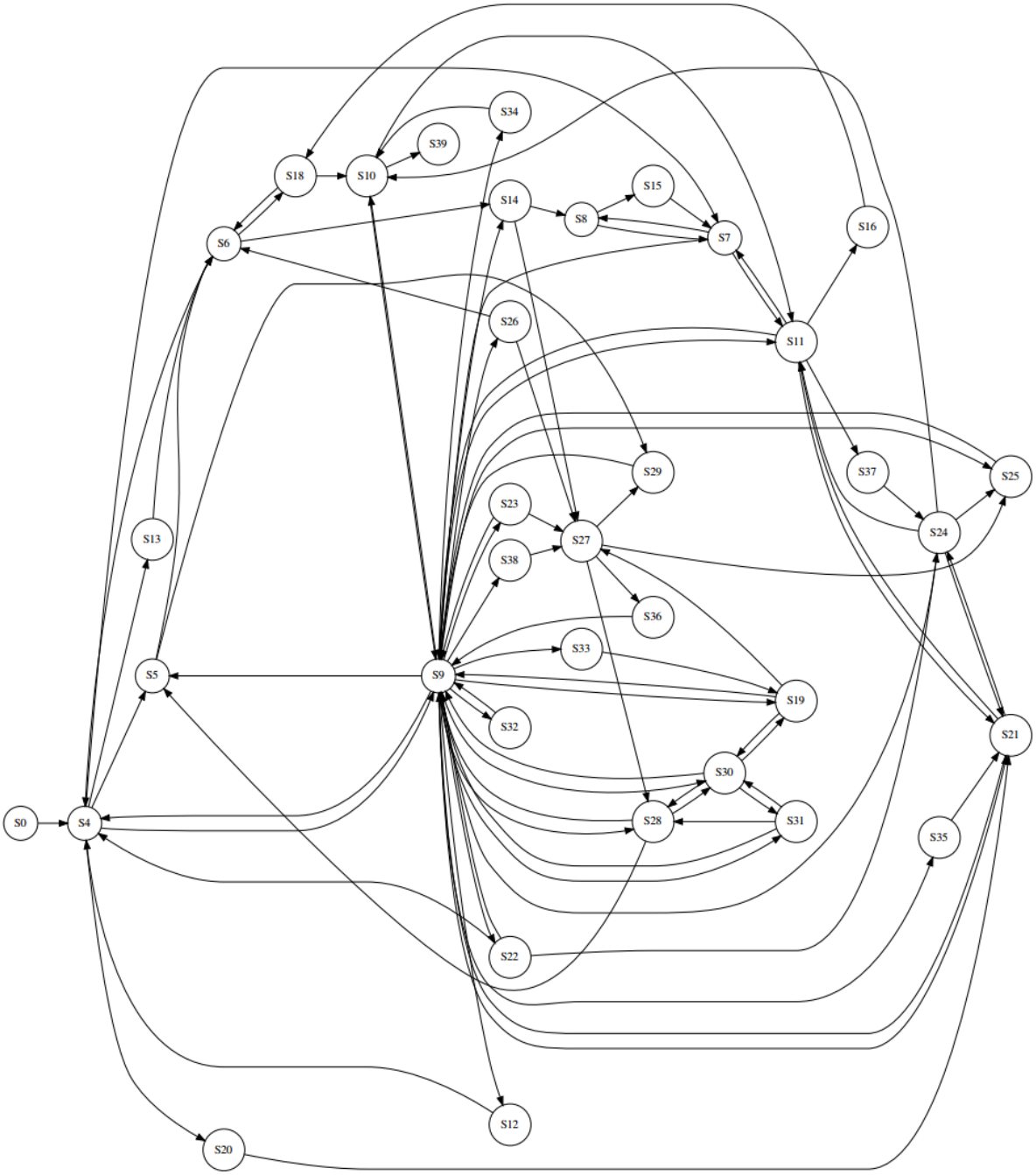
Tabela 7 przedstawia dostępne tryby poboru mocy, które są skojarzone z obecnym stanem logicznym urządzenia. Wartość częstotliwości zegara jest zależna od obecnie obsługiwanego urządzenia peryferyjnego np. obsługa pamięci NAND realizowana jest z częstotliwością zegara głównego równą 96MHz, natomiast podczas komunikacji z przetwornikiem ADC równa się 12MHz. Skojarzenie stanu urządzenia (np. akwizycja EKG, oczekiwanie na skojarzenie badania, odłączenie elektrod itd.) z konkretną konfiguracją energetyczną pozwala na ułatwione zarządzanie poborem prądu przez urządzenie. Efektem takiego podejścia jest również możliwość obserwacji stanu urządzenia poprzez obecny profil poboru prądu. *Holter #2* podczas przeprowadzania badania cały czas wykonuje okresowe czynności tzn. pod wpływem przerwania od ADC oprogramowanie budzi system z trybu *Powerdown,* odczytuje próbki z ADC, buforuje je w pamięci RAM, aby na koniec wejść w tryb *Powerdown*. Gdy bufor w pamięci RAM zostanie zapełniony, sygnał EKG zapisywany jest w pamięci NAND, co wymaga wejścia w tryb aktywny z włączonym zegarem 96MHz i pamięcią NAND. Wykonywanie okresowych czynności przez większą cześć czasu badania wpasowuje się w model behawioralny wymagany przez algorytm opisany w rozdziale 3.

5.3.1 Detekcja stanów urządzenia na podstawie poboru prądu

Holter #2 został uruchomiony z podłączonym do źródła baterii zasilaczem podającym stałe napięcie 3.0V oraz multimetrem cyfrowym pozwalającym na akwizycję danych z częstotliwością próbkowania 500Hz. Wartość natężenia pobieranego prądu była próbkowana przez ponad 19h (35809091 nieprzetworzonych próbek, okres próbkowania 5ms). System wygenerował graf zredukowany zawierający 35 stanów, w tym stan bazowy, który charakteryzowany jest przez 17 krawędzi wejściowych i 14 wyjściowych. Stan bazowy zawiera profil obrazowany wieloma pikami prądu do poziomu 1.8mA z okresem występowania 150ms i czasem piku równym 25ms. Przez pozostałe 125ms urządzenie pobiera 40uA. Algorytm zidentyfikował 14 klasy cykli (CC0-CC13). Wystąpienia klasy cykli CC2 są dominujące na tle innych- jest ich najwięcej (2104 wystąpień oraz są periodyczne z największą częstotliwością).

W trakcie testu urządzenie zostało uruchomione poprzez włączenie zasilania. System wykonał test wewnętrzny komponentów po czym przeszedł do stanu gotowości do uruchomienia. Wciśnięcie przycisku zmieniło stan na „gotowość do sparowania

z konfiguracją badania”. Po sparowaniu i potwierdzeniu rozpoczęcia badania, system konfiguruje przetwornik analogowo cyfrowy i rozpoczyna akwizycję próbek sygnału EKG. Dane są zapisywane do pamięci NAND z okresem równym 42 sekundy. Podczas zapisu do pamięci NAND konfiguracja energetyczna systemu zakłada ustawienie częstotliwości zegara głównego na wartość 96MHz oraz zablokowanie wejścia w stan *Powerdown* oraz *Shutdown*. Podczas eksperymentu kilkukrotnie odczytano status urządzenia za pomocą aplikacji mobilnej NFC, a także zaraportowano symptomy medyczne przy użyciu przycisku oraz aplikacji NFC. Po zakończeniu agregacji próbek wartości poboru prądu, dane zostały podane na wejście algorytmu detekcji stanów urządzenia. Parametry algorytmu zostały dobrane eksperymentalnie i wyniosły: (0) N=225, (1) *avg_eps=0.4,* (2) *deviation_eps=0.035,* (3) *min_eps*=0.5, (4) *max_eps=4.5, (5) integral_eps=0.25*. W celu poprawy klasyfikacji dodano dodatkowy parametr *integral_eps* rozróżniający stany na podstawie stosunku wartości energii sygnału.



Rysunek 35. Graf stanów *Holtera #2* za testowany okres poboru prądu

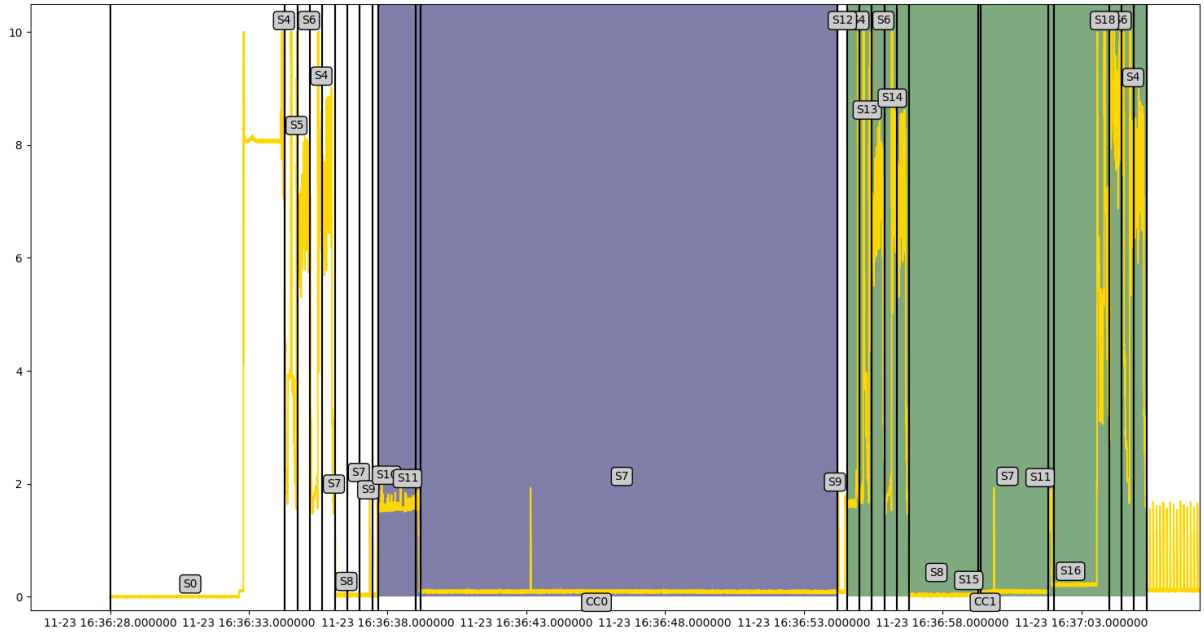
Rysunek 35 przedstawia graf stanów wygenerowany przez algorytm. Porównując prezentowany graf z grafem *Holtera #1* można zauważyć, że stan bazowy S9 posiada znacznie więcej tranzycie wejściowych i wyjściowych a także cykle S9<->S9 są znacznie krótsze pod względem liczby węzłów. Wskazuje to na większe zróżnicowanie profilu poboru prądu w różnych stanach *Holtera #2*, a także na mniejszy poziom skomplikowania zadań realizowanych w każdym z tych stanów. Na podstawie wykrytego stanu bazowego S9 wykryto 13 klas cykli opisanych w tabeli 8.

Klasa cyklu	% wystąpień wykrytych klas	Suma czasów [s] cyklu klas (% udział w czasie agregacji próbek)	Opis
CC0	0,05%	16,538 (0,0231%)	CC0 jest klasą opisującą stan urządzenia, w którym *Holter #2* czeka na sparowanie z badaniem
CC1	0,05%	10,798 (0,0151%)	Klasa cyklu reprezentuje: sparowanie telefonu przez NFC, przejście w stan oczekiwania na rozpoczęcie badania oraz samo rozpoczęcie badania.
CC2	97,72%	392,638 (0,5482%)	Wybudzenie systemu oraz zapis zbuforowanych próbek do pamięci NAND
CC3	0,05%	1,542 (0,0022%)	CC3 oznacza wystąpienie wykrycia odpięcia elektrod oraz reakcji systemu w postaci sygnału dźwiękowego emitowanego przez buzzer, a także odświeżenia ekranu z komunikatem o zdarzeniu.
CC4	0,05%	1,092 (0,0015%)	Klasa CC4 zawiera stan, w którym nastąpiło podłączenie elektrod oraz zniknięcie komunikatu z ekranu o odłączeniu elektrod.
CC5	0,43%	15,046 (0,021%)	CC5 opisuje operacje *Holter #2* podczas zapisu do pamięci SRAM raportu wystąpienia symptomu medycznego zgłoszonego przez pacjenta za pomocą przycisku lub aplikacji NFC
CC6	0,38%	8,534 (0,0119%)	Wybudzenie systemu oraz zapis zbuforowanych próbek do pamięci NAND, a także aktywne oczekiwanie na koniec transmisji NFC (co jest

			wynikiem wykrytego błędu oprogramowania)
CC7	0,05%	0,192 (0,0003%)	Wybudzenie systemu z powodu aktywacji licznika upływu czasu na koniec transmisji NFC
CC8	0,05%	1,684 (0,0024%)	Odczyt stanu *Holter #2* przez aplikację NFC zakończony sukcesem
CC9	0,95%	3,852 (0,0054%)	Wybudzenie systemu w celu inkrementacji licznika godzin spędzonych na realizacji badania
CC10	0,05%	2,528 (0,0035%)	Odczyt stanu *Holter #2* przez aplikację NFC zakończony sukcesem z pojedynczym ponowieniem transakcji
CC11	0,10%	3,316 (0,0046%)	Nieudana transmisja raportu z symptomem medycznym zrealizowana przez aplikację NFC
CC12	0,05%	0,896 (0,0013%)	Zapis do pamięci NAND analogiczny jak w klasie CC2 z nałożoną poprawną obsługą wykrycia zaniku pola NFC po komunikacji.
CC13	0,05%	4,72 (0,0066%)	Finalizacja badania za pomocą aplikacji NFC.

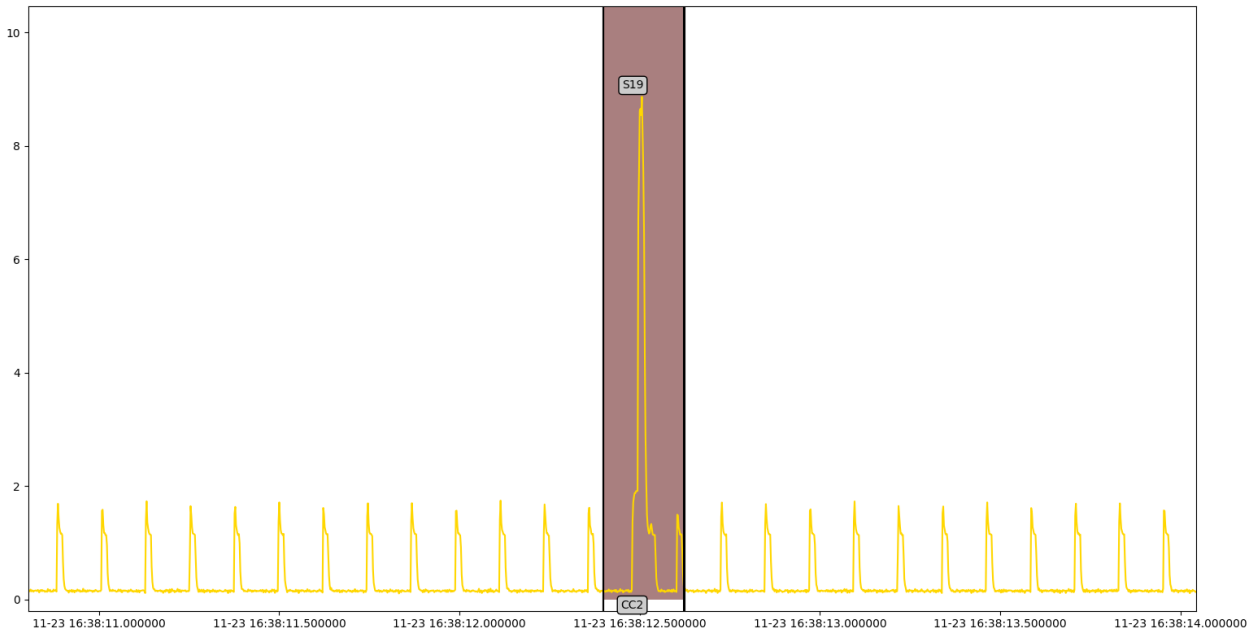
Tabela 8. Lista klas cykli wraz z opisem

Stan S9 klasyfikuje fragmenty sygnału poboru prądu, podczas których urządzenie okresowo odczytuje próbki sygnału EKG, zapisuje je w wewnętrznym buforze pamięci SRAM, a następnie przechodzi w stan niskiego poboru energii oczekując przerwania z przetwornika ADC.



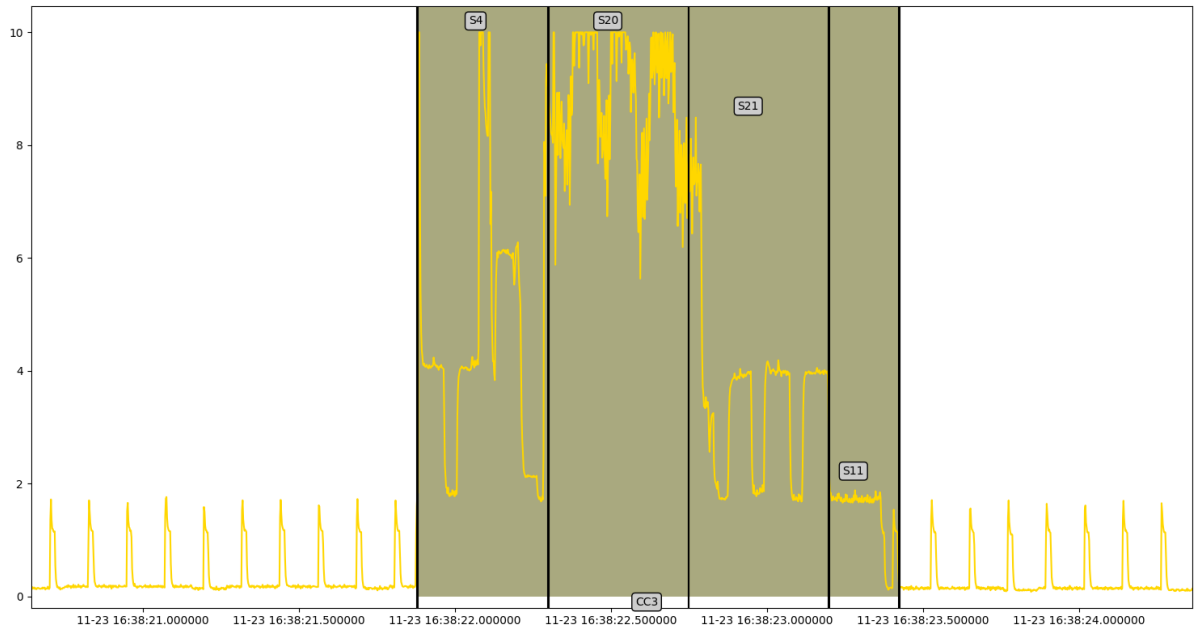
Rysunek 36. Wykres poboru prądu *Holtera #2* od momentu zasilenia do chwili rozpoczęcia badania z naniesionymi oznaczeniami klas cykli 0 i 1

Rysunek 36 przedstawia wykres poboru prądu z oznaczeniami klas cykli 0 i 1. W stanie S0 urządzenie zaczyna być zasilane. Następnie system przechodzi kolejno przez stany S4, S5, S6, S4 i S7. W tych stanach system wyświetla ekran powitalny oraz wykonuje auto test. W stanie S8 system czeka na zakończenie się testu, by w stanie S9 rozpocząć inicjalizację przetwornika ADC oraz konfigurację archiwum pamięci NAND. Po procesie inicjalizacji system przechodzi do stanu niskiego poboru prądu, który trwa przez większość instancji CC0. Klasa CC1 składa się z zdarzeń: (1) sparowania z konfiguracją badania przez aplikację NFC (pierwsze sekwencje skoków wartości poboru prądu), (2) odświeżenia ekranu z imieniem i nazwiskiem pacjenta, (3) oczekiwanie w stanie niskiego poboru prądu na polecenie rozpoczęcia badania, (4) detekcja wciśnięcia przycisku i utrzymania tego stanu przez ponad 2s, (5) rozpoczęcie badania. Zdarzenie (4) można rozpoznać po wzroście wartości minimum poboru prądu. Rozpoczęcie badania wiąże się z równoległą aktualizacją stanu ekranu oraz emisją sygnału dźwiękowego.



Rysunek 37. Wykres poboru prądu zawierający reprezentanta klasy cykli CC2

Na rysunku 37 zawarty jest fragment wykresu opisany klasą cykli 2 (CC2) oraz widoczny jest profil energetyczny stanu S9 przed i po wystąpieniu CC2. Profil energetyczny CC2 osiąga maksymalną wartość poboru prądu z przedziale 6-10mA. Rozrzut wartości wynika z częstotliwości próbkowania, maksimum zwykle jest reprezentowane przez pojedynczą próbkę i w przypadku nietrafienia w szczyt wzrostu prądu, obserwowane wartości maksimum mogą być mniejsze. W pierwszej fazie CC2 pobór prądu wzrasta do poziomu 2 mA (wybudzenie i konfiguracja zegara na 96MHz), następnie następuje zapis do pamięci NAND co skutkuje wzrostem prądu do nawet 10mA. Po zapisie system przełącza główny zegar na częstotliwość 12MHz. Zmiana ta odwzorowana jest poprzez spadek poboru prądu do ok. 1mA. W tym czasie system przygotowuje bufor pamięci na kolejne próbki EKG oraz wejście w tryb *Powerdown*.



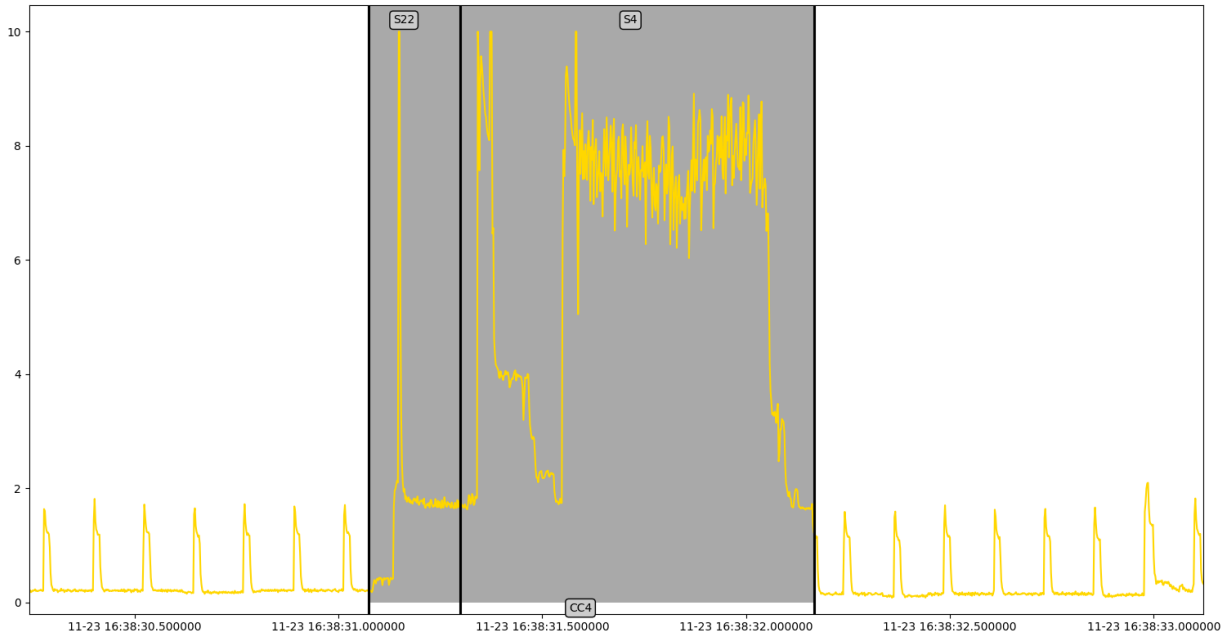
Rysunek 38. Wykres poboru prądu przedstawiający moment wystąpienia zdarzenia odłączenia elektrod

Klasa cykli CC3 jest przypisywana do zdarzeń odłączenia elektrod. Cykl CC3 w swoim profilu energetycznym zawiera zdarzenia odświeżenia ekranu (trzy szczyty wzrostu poboru prądu, które osiągają maksymalny zakres wartości prądu (S20)). W stanie S21 można obserwować pobór energii podczas generowania sygnału dźwiękowego. Wartość poboru prądu jest wprost proporcjonalna do głośności generowanego dźwięku, co z kolei oznacza, że na podstawie poboru prądu można odtworzyć charakterystykę komunikatu dźwiękowego.

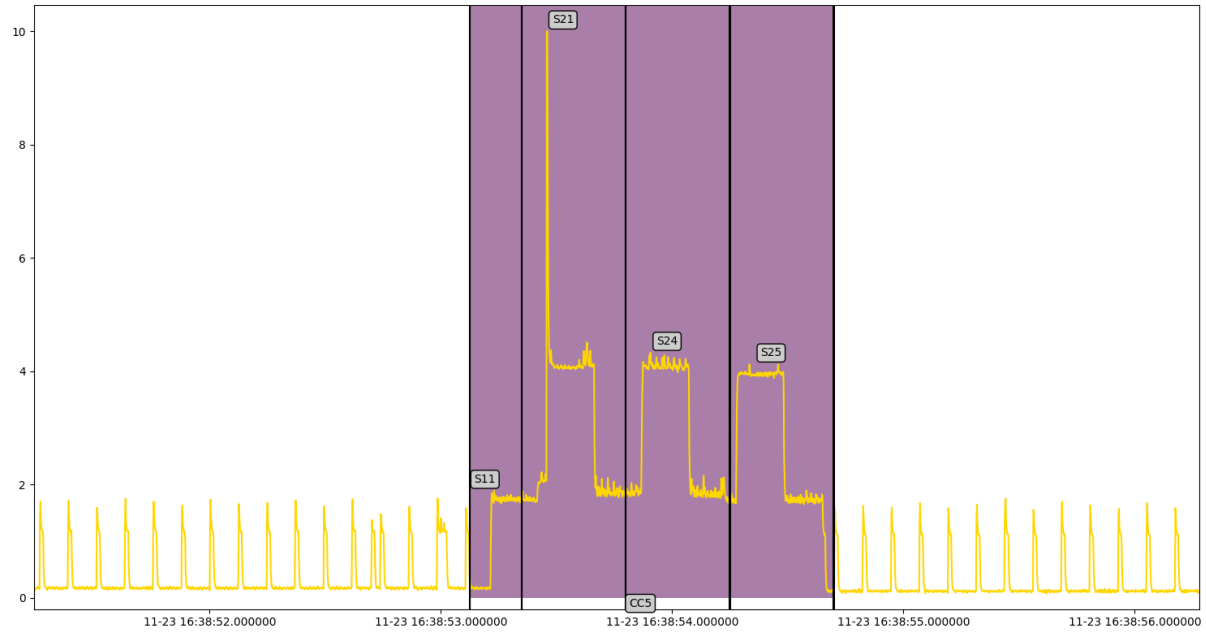
Klasa CC4 opisuje moment wykrycia ponownego podłączenia elektrod do urządzenia. W wyniku wykrycia podłączenia elektrod, moduł przetwornika analogowo cyfrowego jest rekonfigurowany, a ekran EINK jest odświeżany tak, by komunikat o braku elektrod nie był już widoczny. Podłączenie elektrod nie jest komunikowane dźwiękiem co skutkuje tym, że CC4 trwa 0.5 sekundy krócej w stosunku do CC3.

Rysunki od 40 do 43 przedstawiają pobór prądu podczas różnych instancji klasy cykli CC5. Etykieta CC5 jest przypisywana do profili energetycznych urządzenia, które obsługuje stan zgłoszenia raportu z symptomem medycznym przez pacjenta. Zgłoszenie tego typu może być zrealizowane przez przycisk urządzenia lub interfejs NFC. Po wykryciu zgłoszenia symptomu włączana jest przetwornica typu BOOST, co obrazuje stan S21- skok wartości poboru prądu do 10mA. Następnie po buforowaniu symptomu generowany jest sygnał dźwiękowy, którego charakterystykę amplitudy podobnie jak w CC3 można odczytać obserwując wartość poboru prądu w stanach S24 i S25. Jeśli użytkownik wyłączy aplikację

NFC zanim zakończy się komunikacja, to w czasie 3s. nastąpi wybudzenie urządzenia w celu zerowania stanu modułu oprogramowania odpowiedzialnego za komunikację, co można za obserwować jako CC7 na rysunku 43.



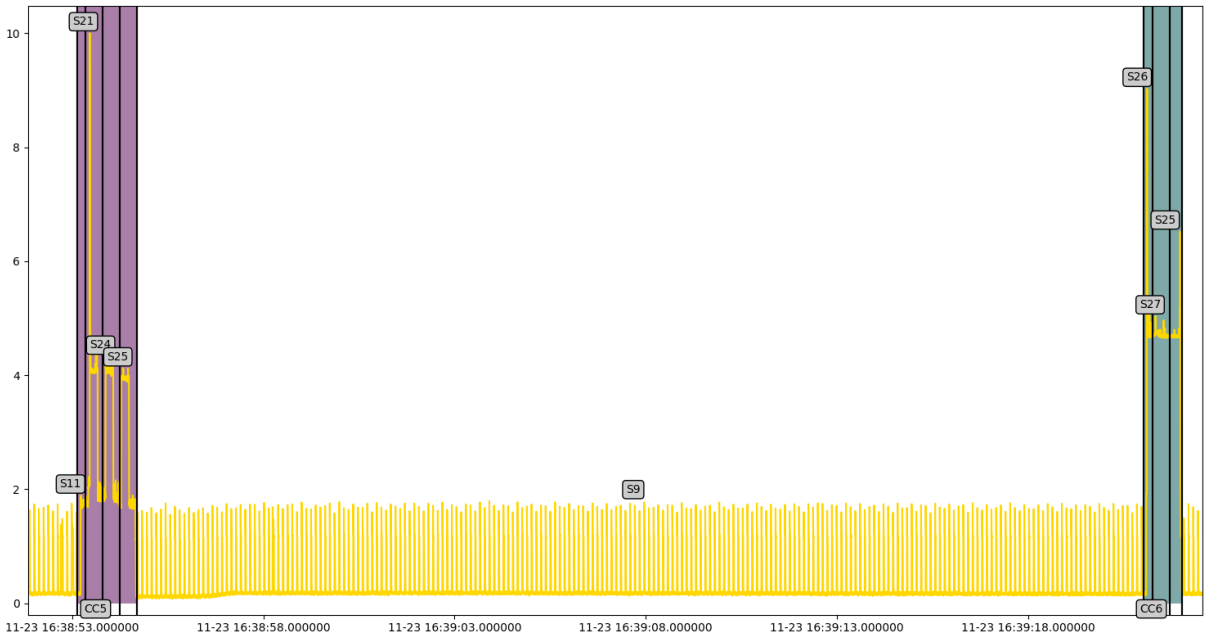
Rysunek 39. Wykres poboru prądu podczas podłączania elektrod do pacjenta



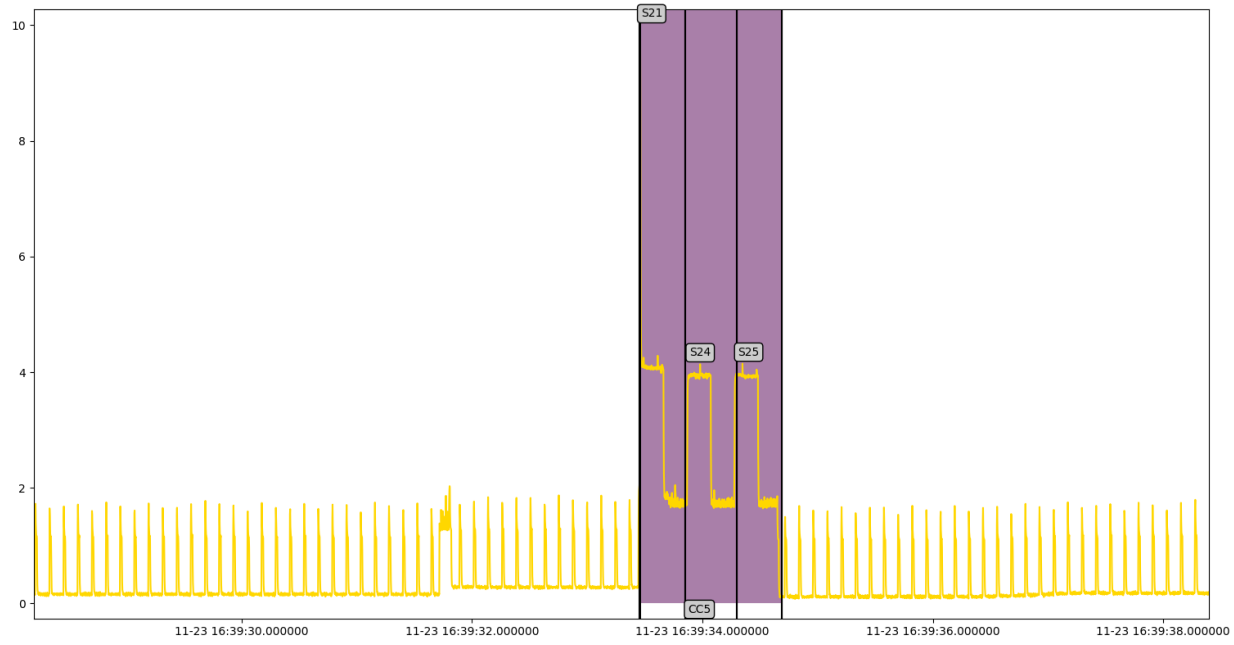
Rysunek 40. Wykres poboru prądu podczas zapisywania raportu symptomu medycznego zgłoszonego za pomocą NFC

Obserwując wykresy zawierające CC5 można zauważyć, że zwykle po CC5 następuje instancja klasy cykli CC6. Kolejne instancje klasy cykli można rozpisać w postaci sekwencji

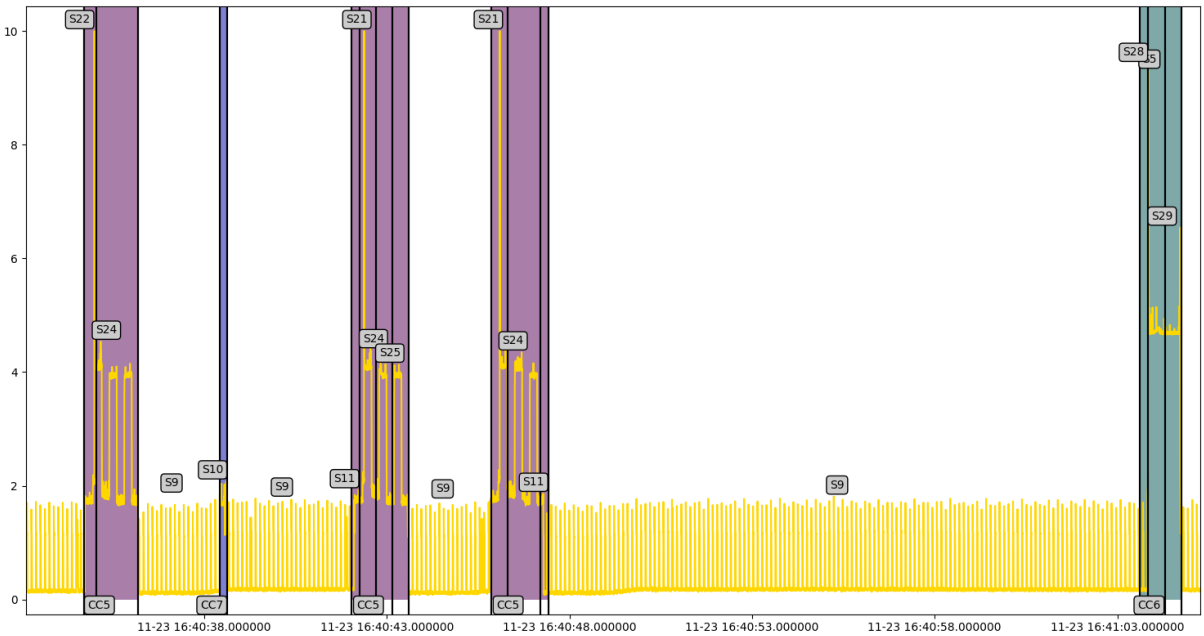
klasy cykli np. (…) CC2 CC2 CC5 CC6 CC2 CC2 CC5 CC2 (..) itd. Analizując taką sekwencje można wyznaczyć zależności: (1) grupy CC6 występują zawsze w czasie, w którym powinna się znajdować grupa zaklasyfikowana jako CC2 oraz (2) przed wystąpieniem CC6 występuje grupa jednej z klasy cykli opisującej komunikacje z wykorzystaniem interfejsu NFC (CC5: rysunki 40, 41, 43; CC8: rysunek 44; CC10: rysunek 45 ;CC11: rysunek 46). Wyjątkiem jest rysunek 42 na którym po jednej grupie CC5 nie występuje CC6, tylko CC2. Przyglądając się CC5 z rysunku 42 można zauważyć, że ta instancja CC5 nie posiada stanu przed stanem S21, a ponadto na ok 2s. przed grupą CC5 następuje wzrost minimum poboru prądu, który jest spowodowany zwarciem linii przycisku i przepływem prądu przez rezystor podciągnięty do zasilania. Oznacza to, że opisywana instancja CC5 została wygenerowana w wyniku zgłoszenia symptomu przyciskiem. Z powyższych przesłanek można wnioskować, że pojawienie się instancji CC6 jest uwarunkowane przynajmniej próbą komunikacji aplikacji NFC z urządzeniem. Dzięki takiemu wnioskowi programista mógł: (1) zauważyć sam problem oraz (2) ukierunkować poszukiwania defektu na moduł komunikacji NFC. Zauważone zjawisko stanowi realne zagrożenie dla wykonywania funkcji urządzenia, gdyż wydłuża o 2s. (z ok 120 ms.) przebywanie przez system w stanie podwyższonego poboru prądu. W przypadku pacjenta, który często zgłasza symptomy mogłoby to oznaczać niemożność wykonania badania w czasie czternastu dni na pojedynczej baterii. Dzięki ukierunkowaniu poszukiwań, defekt został odnaleziony w bardzo krótkim czasie jednego dnia roboczego oraz wyeliminowany. Błąd był wynikiem defektu oprogramowania, który powodował aktywację układu NFC podczas najbliższej zmiany stanu energetycznego systemu (np. z trybu akwizycji przejście do stanu zapisu do pamięci NAND). Układ NFC był wyłączany dopiero po 2s. bezczynności, gdy licznik systemowy zgłaszał brak komunikacji przez 2s.



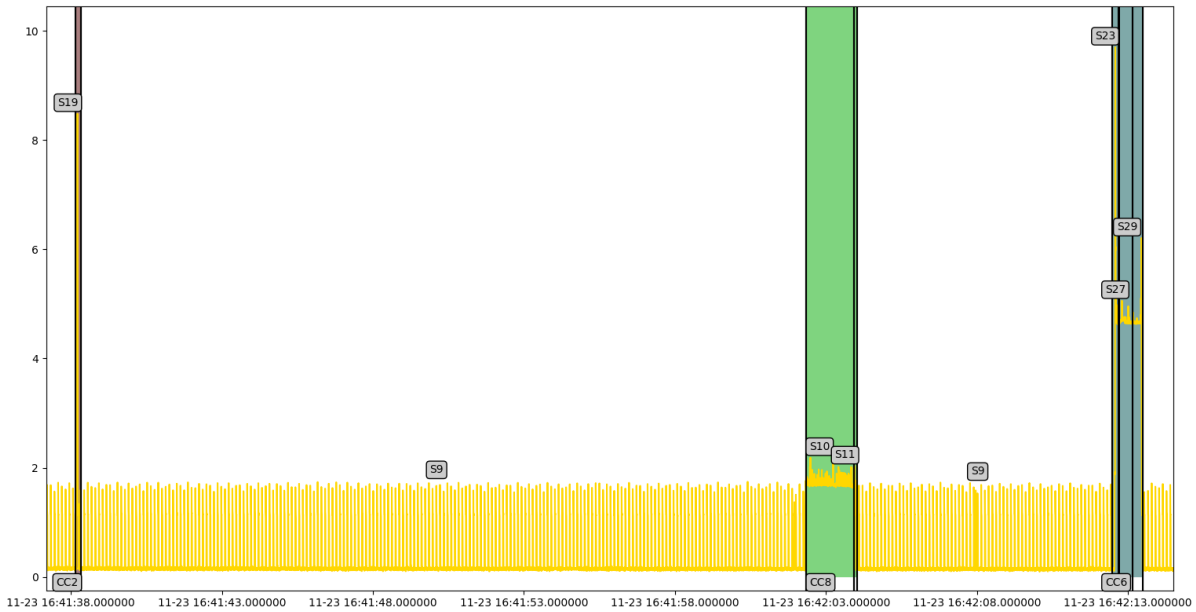
Rysunek 41. Wykres poboru prądu podczas zapisywania raportu symptomu medycznego zgłoszonego za pomocą aplikacji mobilnej, a także podczas zapisu do pamięci NAND zbuforowanych próbek EKG i symptomów



Rysunek 42. Wykres poboru prądu podczas zapisywania raportu symptomu medycznego zgłoszonego za pomocą przycisku



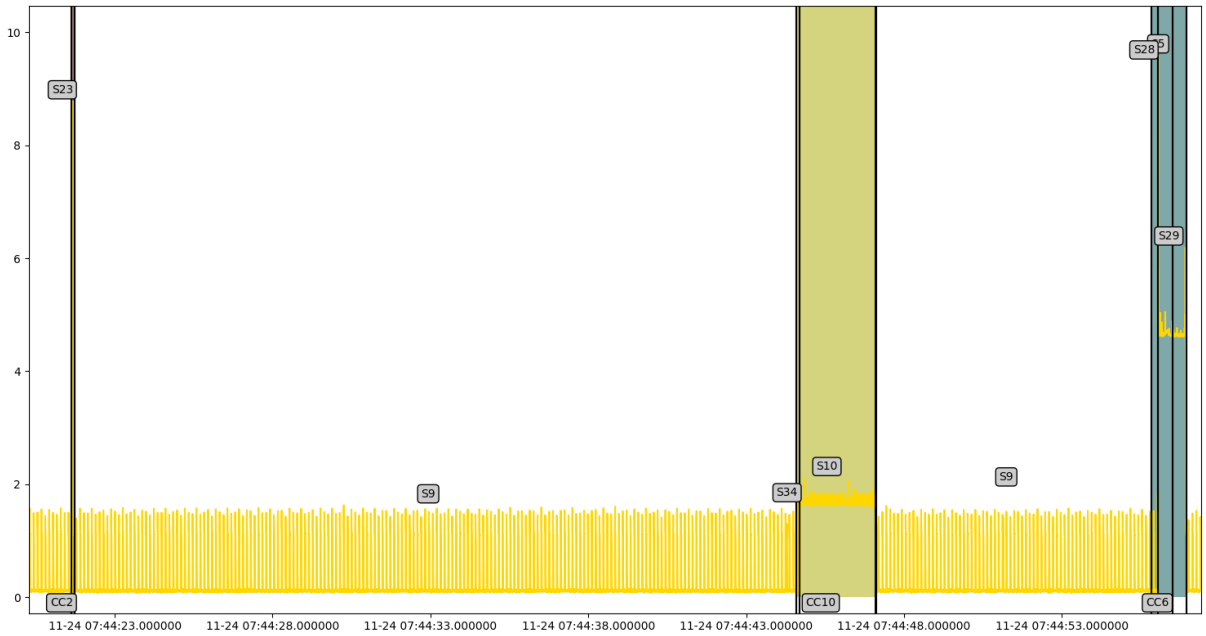
Rysunek 43. Wykres poboru prądu urządzenia podczas zgłaszania, w krótkim odstępie czasu dwóch symptomów przez aplikację NFC oraz dodatkowego symptomu za pomocą przycisku.



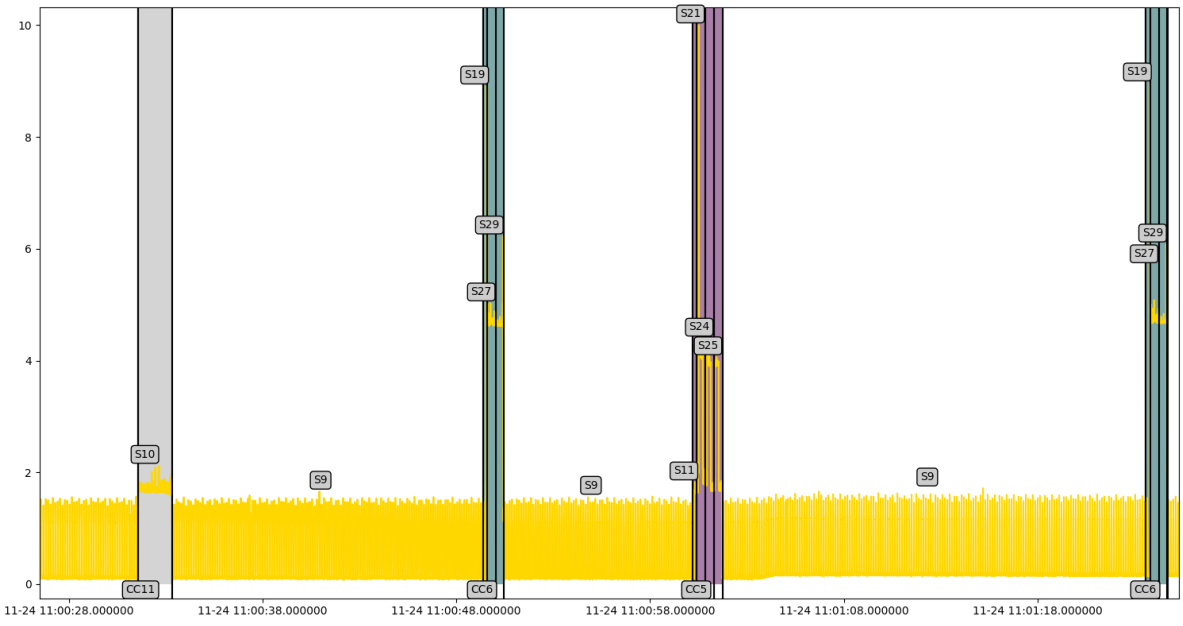
Rysunek 44. Wykres poboru prądu podczas operacji odczytu statusu przez aplikację NFC zakończonej sukcesem

Rysunki 44, 45 oraz 46 zawierają wykresy z oznaczonymi instancjami klas cyklów: CC8, CC10, CC11. Wszystkie trzy klasy cyklów opisują to samo zdarzenie- próbę komunikacji z wykorzystaniem interfejsu NFC. CC8 opisuje odczyt statusu zakończony sukcesem. CC10 oznacza również odczyt status ostatecznie zakończony sukcesem, który trwał 2 razy dłużej

z powodu błędu transmisji skutkującym ponownym wykonaniem procedury. CC11 prezentuje profil energetyczny próby komunikacji NFC, która skończyła się wynikiem negatywnym.



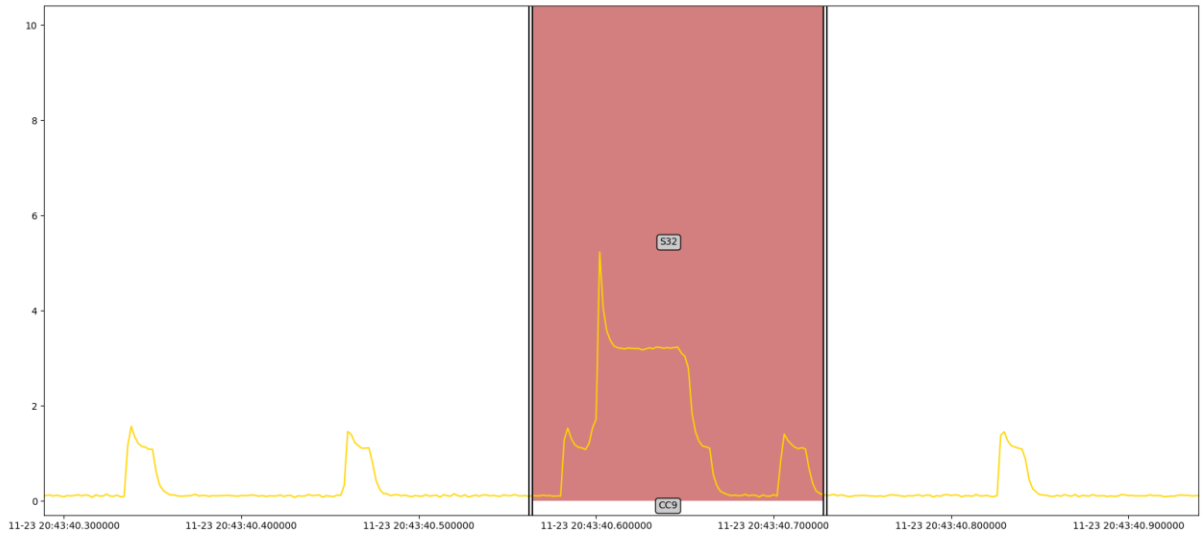
Rysunek 45. Wykres poboru prądu podczas odczytu statusu (wraz z pojedynczą retransmisją) przez aplikację NFC.



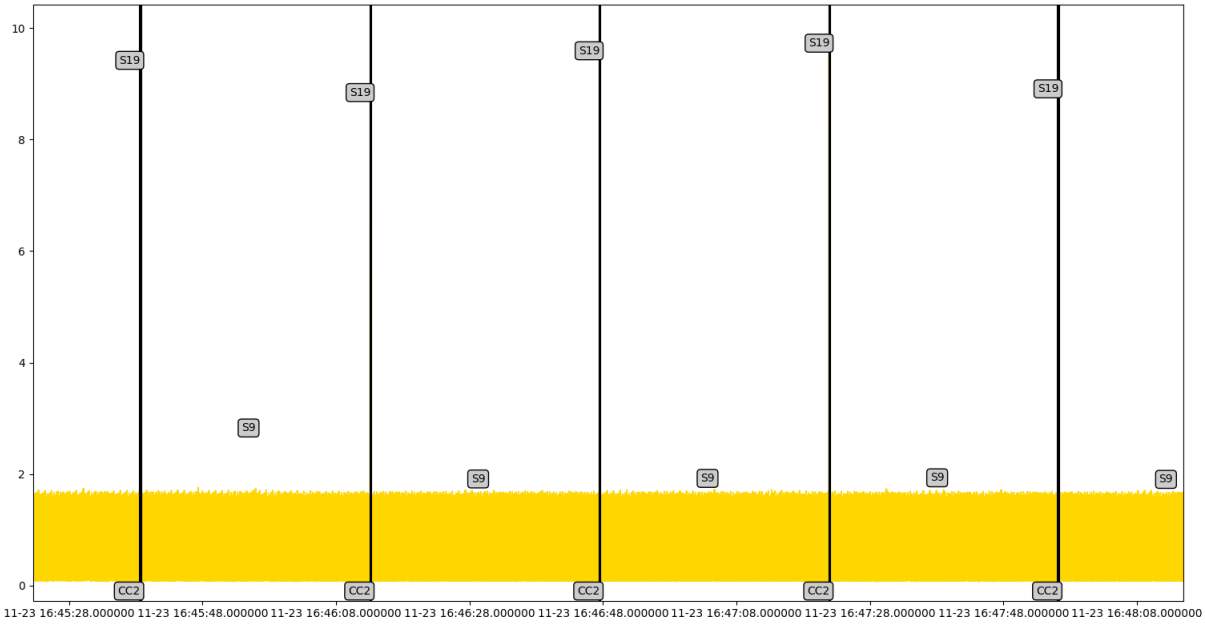
Rysunek 46. Wykres poboru prądu podczas nieudanej próby raportowania symptomu medycznego z wykorzystaniem NFC

Narzędzie klasyfikujące profile energetyczne stanów urządzenia można także wykorzystać do zgrubnej analizy dokładności sygnałów zegarowych wykorzystywanych

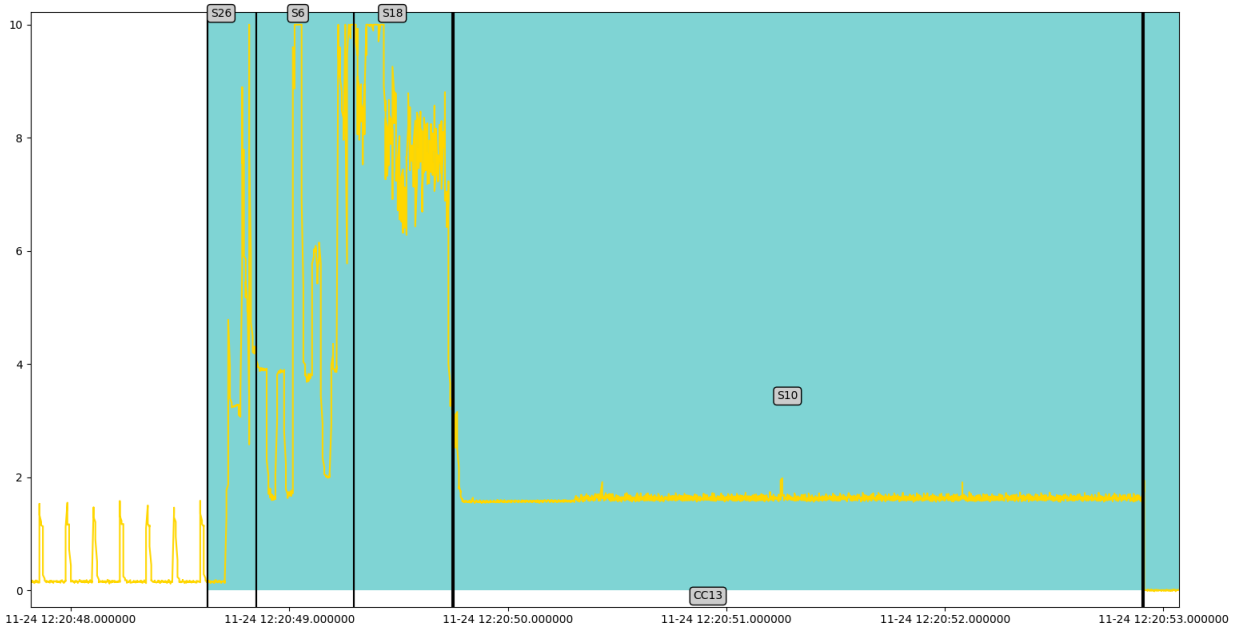
do synchronizacji czasu czy też jako źródło sygnału próbkującego. Rysunek 47 prezentuje profil energetyczny CC9. Klasa ta opisuje wybudzenie systemu podczas akwizycji sygnału EKG do stanu aktywnego w celu weryfikacji czasu (odczyt czasu z układu RTC) oraz inkrementacji licznika godzinowego badania. Tworząc listę wszystkich instancji CC9 wraz ze znacznikami czasu charakterystycznych punktów np. próbek wartości maksymalnej poboru prądu instancji, można ocenić jakość sygnału zegara czy też stabilność opóźnienia wybudzania. Z tak przeprowadzonej analizy wynika, że rozrzut sygnału zegara RTC jest poniżej < 1s. w ciągu 20 godzinnego badania. Analogicznie pomiary można oprzeć o czas występowania instancji CC2 (rysunek 48). Zapis do pamięci NAND jest realizowany w momencie zapełnienia się obszaru bufora w pamięci SRAM o stałym rozmiarze. Przy stałej częstotliwości oraz rozdzielczości próbkowania, zapis ten realizowany jest dokładnie co 42 sekundy.



Rysunek 47. Wykres poboru prądu urządzenia podczas wybudzenia w celu weryfikacji czasu badania



Rysunek 48. Wykres poboru prądu urządzenia podczas przeprowadzania badania EKG



Rysunek 49. Wykres poboru prądu urządzenia w trakcie kończenia badania

Instancja klasy CC13 podczas badania występuje tylko raz. Wynika to z faktu, że klasa CC13 opisuje pobór prądu podczas operacji kończenia badania. W przykładzie z rysunku 49 badanie zostało zakończone poprzez rozkaz przesłany przez interfejs NFC. Pierwszym krokiem jest zapis do pamięci NAND bufora próbek EKG, co widoczne jest jako stan S26, następnie równolegle rozpoczyna się odświeżanie ekranu w celu wyświetlenia komunikatu z informacją o możliwości pobrania danych oraz nadawana jest sekwencja sygnałów dźwiękowych

odwzorowanych poprzez stan S6. Po wykonaniu tych kroków system czeka na podłączenie do komputera PC przez 30s., a następnie przechodzi w stan *Shutdown*.

5.3.2 Wpływ wartości parametrów na wyniki

W ramach doświadczenia sprawdzony został wpływ wartości parametrów na działanie i wyniki testowanego algorytmu. Opisane wyżej wyniki zostały wygenerowane algorytmem z parametrami: *n*=225, *avg_eps*=0.4, *dev_eps*=0.35, *min_eps*=4.5, *max_eps*=4.5. Parametry te zostały dobrane na podstawie wiedzy eksperckiej oraz na podstawie 15 minutowego fragmentu pomiarów poboru prądu. Czas generowania modelu za pomocą aplikacji *StateEventAnalyzer* wyniósł 27 minut na komputerze klasy PC z procesorem Intel core i5-8600K. Rozmiar danych wejściowych wynosił 2.64GB. Wywołania algorytmu były powtarzane dla tych samych danych wejściowych, ale z innymi parametrami algorytmu. Zmiany wartości parametru n przy utrzymaniu wartości pozostałych parametrów pokazały, że wzrost wartości n powoduje spadek wykrytej liczby klas cykli CC. Liczba wykrytych anomalii zmienia się wolniej niż liczba stanów wraz ze zmianą wartości n. Zbyt duża wartość parametru n, powoduje błędną detekcję stanu bazowego i konsekwentnie błędne detekcje anomalii/zdarzeń. Na przykład dla n=10000 liczba wystąpień klas cykli spada z 2082 do 1473, które składają się na jedynie 5 stanów oraz 5 klas cykli. Przy wartości n=675 algorytm wygenerował 20 stanów i 9 klas cykli w formie 2082 instancji, dla n=32: 44 stany 17 klas cykli i 2090 instancji. Czas działania algorytmów wyniósł odpowiednio 11 min i 1 godz. 24 min. Czas wykonania jest odwrotnie proporcjonalny do rozmiarów segmentu (parametr n). Kolejne etapy obniżania wartości parametru n skutkowały znaczącym wzrostem liczby klasy cykli do postaci w całości niespójnych z fizycznym modelem działania badanego urządzenia.

Kolejnym etapem testowania wpływu wartości parametrów na wyniki były testy porównawcze przy stałych n=225 i *dev_eps*=0.35. Dla wartości *avg_eps*=0.20, *min_eps*=0.25 i *max_eps*=2.25 system wygenerował 51 stanów, 29 klas cykli i 134 instancje CC. Z kolei dla *avg_eps*=0.80, *min_eps*=1.50, *max_eps*=9 algorytm wygenerował 14 stanów, 15 klas cykli i 2106 instancji. Wzrost wartości parametrów *avg_eps*, *min_eps* i *max_eps* na odpowiednio niskim poziomie ma niewielki wpływ na wyniki algorytmu. Spadek wartości dyskutowanych parametrów powoduje wzrost liczby stanów. Niedopasowane wartości parametrów mogą skutkować detekcją fałszywego stanu bazowego. Błędnie określony stan bazowy, może skutkować wykryciem błędnych klas cykli. Taka sytuacja może być zaobserwowana dla parametru *avg_eps*=0.2- liczba klas cykli spadła z 2104 do 134.

6 Podsumowanie

Przedstawiona problematyka jest adresowana poprzez propozycje trzech zestawów algorytmów: a) analizy śladu instrukcji (rozdział 2, algorytm 2-1), b) obiektowej analizy szeregu czasowego (rozdział 3, algorytmy od 3-1 do 3-15), c) korelacji logów tekstowych z szeregiem czasowym (rozdział 4, algorytmy od 4-1 do 4-7). Każdy z proponowanych algorytmów powstał w wyniku doświadczeń zawodowych autora. W szczególności algorytm b) miał swój początek w formie potrzeby narzędzia automatyzującego analizę i optymalizację poboru energii systemu wbudowanego. Algorytm c) jest naturalnym rozszerzeniem możliwości analizy o dodatkowe źródło danych, umożliwiając jeszcze większą automatyzację procesu. Algorytmy proponowane w pracy uwzględniają specyfikę systemów wbudowanych przedstawioną w rozdziale 1. Algorytm a) dzieli ślad instrukcji na segmenty, które następnie są grupowane w celu detekcji anomalii. Na potrzeby algorytmu grupowania zaproponowana została metryka. Algorytmy b) i c) również opierają się na porównywaniu utworzonych obiektów. Porównywanie i dyskryminacja są realizowane poprzez autorskie metody zaprezentowane w rozdziałach 3.1.1, 3.1.3, 4.1 i 4.2. Algorytmy rozwiązują problemy związane z monitorowaniem programowym systemów wbudowanych. Skuteczność algorytmów została potwierdzona na danych pochodzących z rzeczywistych komercyjnych systemów wbudowanych z branży medycznej. Wyniki działania algorytmów na rzeczywistych danych przedstawione zostały w rozdziale 5. Do najważniejszych osiągnięć pracy należą:

1. Opracowanie algorytmu analizy śladu instrukcji
    a. Opracowanie metody tworzenia struktury danych z sekwencji śladu instrukcji oraz wykorzystania struktury do wykrywania anomalii
    b. Propozycja metryki wykorzystywanej podczas grupowania
    c. Implementacja algorytmu w systemie *TraceAnalyzer*.
    d. Weryfikacja algorytmu na rzeczywistym systemie wbudowanym
2. Opracowanie hierarchiczno-obiektowego algorytmu analizy szeregu czasowego
    a. Opracowanie metod ekstrakcji obiektów za pomocą podziału i grupowania szeregu czasowego
    b. Opracowanie metod porównywania wykrytych obiektów na różnych poziomach abstrakcji
    c. Opracowanie modelu danych umożliwiającego hierarchizację obiektów oraz przedstawienie wyekstrahowanych danych w formie grafu stanowego
    d. Przygotowanie systemu *StateEventAnalyzer* implementującego algorytm.

e. Weryfikacja automatyzacji analizy szeregu czasowego (wykorzystując jako obiekty badań komercyjne systemy medyczne)

3. Opracowanie algorytmu korelacji logów tekstowych z szeregiem czasowym
    a. Opracowanie modelu danych dla obiektów reprezentujących rekordy logów tekstowych
    b. Opracowanie metody ekstrakcji i filtrowania zdarzeń z sekwencji rekordów logów tekstowych
    c. Opracowanie metody dopasowania/korelacji rozpoznanych obiektów do obiektów reprezentujących zdarzenia z innego źródła
    d. Rozwinięcie systemu *StateEventAnalyzer* o implementację algorytmu
    e. Weryfikacja funkcji algorytmu wykorzystując jako obiekty badań komercyjne systemy medyczne

Opisy poszczególnych osiągnięć zostały rozwinięte w kolejnych paragrafach.

Algorytmy przedstawione w rozdziałach 3 i 4 na podstawie szeregu czasowego poboru energii umożliwiają wykrycie obiektów, utworzenie hierarchii z tych obiektów, klasyfikacje obiektów oraz korelację ich z rekordami logów tekstowych. Prezentowany model danych wraz z logami tekstowymi pozwala na zautomatyzowanie analizy zachowania badanego urządzenia/systemu. Wykryte obiekty odpowiadają konkretnym stanom, w których przebywał badany system. Poprzez hierarchię obiektów oddane są różne poziomy abstrakcji modelu stanowego lub interakcji analizowanego systemu. Znaczenie obiektom jest nadawane poprzez analizę eksperta w oparciu o wykryte obiekty lub poprzez rekordy logów dopasowane to wystąpień obiektów. Oba algorytmy zostały zaimplementowane w ramach systemu *StateEventAnalyzer* przedstawionego w załączniku 8.2. Kontrola nad algorytmami i ich parametrami realizowana jest poprzez interfejs konsolowy. System udostępnia zestaw komend. Komendy można łączyć w sekwencje, tworząc skrypty interpretowane przez system. W ten sposób można przygotować sekwencję np. wczytania i przetworzenia danych, skonfigurowania algorytmu dekompozycji szeregów czasowych, wybrania z nich interesujących klas cykli, które następnie staną się danymi wejściowymi dla algorytmu korelacji logów tekstowych. Ostatecznie skrypt może dane te zapisać w postaci tekstowej do pliku wyjściowego. System umożliwia również wizualizacje różnych poziomów hierarchii obiektów poprzez wygenerowanie wykresu przebiegu szeregu czasowego z nałożonymi etykietami oraz oznaczeniami kolorami stanów i klas cykli. Model stanów jest wyświetlany w postaci grafu stanów z możliwością zapisu do plików grafiki wektorowej. Zaprezentowany system pozwala na powtarzalne odtwarzanie wyników różnych etapów działania algorytmów w celu analizy ich

poprawności jak i obserwacji cech. Ponadto wykorzystana architektura pozwala w nieskomplikowany sposób dodawać kolejne rozwinięcia algorytmów oraz umożliwia pracę z wieloma zestawami parametrów konfiguracyjnych.

Jednym z głównych osiągnięć pracy jest algorytm obiektowej dekompozycji szeregów czasowych z rozdziału 3. Algorytm w pierwszym etapie dzieli szereg czasowy na segmenty o stałej długości. Następnie sąsiadujące segmenty są porównywane ze sobą za pomocą autorskiego szeregu metryk bazujących na różnicach w parametrach statystycznych segmentów. Segmenty sklasyfikowane jako podobne grupowane są w obiekty- grupy. Grupy porównywane są ze sobą w celu oznaczenia podobnych grup etykietami- stanami. Znajdywany jest stan bazowy- jest to obiekt, który reprezentuje stan i operacje, które badany system wykonuje periodycznie i które nie stanowią reakcji systemu na zdarzenia- są to czynności tła. Stan bazowy stanowi punkt odniesienia niezbędny do wyszczególnienia procesów/sekwencji stanów urządzenia, które docelowo mają reprezentować atypowe zachowanie. Sekwencje stanów określane są poprzez znalezienie cykli zaczynających się i kończących w stanie bazowym. Podobne cykle grupowane są w klasy cykli. Klasa cykli wykryta przez algorytm reprezentuje proces pierwszego planu zapoczątkowany m.in. przez awarię lub zmianę w środowisku sygnałów wejściowych urządzenia.

Algorytm powstał w wyniku doświadczenia autora z pracy przy projektowaniu i implementacji systemów wbudowanych (w szczególności różnych typów urządzeń medycznych typu holter EKG). Rozwój opisywanych systemów wiąże się także z optymalizacją poboru energii, której przykład opisany został w rozdziale 3.2.2. W przypadku złożonych systemów wbudowanych, które wykonują wiele procesów na różnych poziomach abstrakcji realizując różne grafy stanów, proces optymalizacji energetycznej jest skomplikowany oraz wymaga analizy na wielu poziomach logicznych oprogramowania. Prezentowany algorytm dekompozycji szeregów czasowych pozwala na zrealizowanie trzech celów: 1) weryfikacje poprawności założeń projektowych badanego systemu i identyfikowanie naruszeń specyfikacji, umożliwiając określenie komponentu oprogramowania, który je powoduje, 2) porównywanie profili operacyjnych ze wzorcem referencyjnym, oraz 3) detekcje błędów w formie anomalii odstających od typowego profilu zachowania systemu. Przykłady wykorzystania algorytmu w opisywanych celach zostały zaprezentowane w rozdziale 5, wykazując skuteczność systemu. Dzięki metodom agregacji oraz dekompozycji danych pochodzących z monitorowania poboru prądu, algorytm umożliwił przeprowadzenie efektywnej analizy, której wyniki nie tylko wykryły awarię systemu, ale również wskazały prawdopodobne źródło defektu. Tego typu analiza oparta o monitorowanie była możliwa

poprzez dostęp do wygenerowanego hierarchicznego modelu obiektów opisujących stany systemu. Możliwość powiązania stanów badanego systemu z konkretnymi profilami poboru prądu znacząco upraszcza sam proces analizy, a także pozwala na zautomatyzowaną ocenę wpływu wprowadzanych zmian na zachowanie systemu. Po stwierdzeniu zawyżonego poboru energii przez badany system, można za pomocą modyfikacji parametru *min_eps* i *max_eps*, zlokalizować stany systemu, w których urządzenie przekracza oczekiwanie zakresy poboru energii. Innym wykorzystaniem algorytmu może być analiza pod kątem wykrycia przypadku wejścia urządzenia w stany logiczne niezgodnie ze specyfikacją diagramu maszyny stanowej.

W rozdziale 3.1.4 opisano właściwości algorytmu. Algorytm bazuje na założeniu, że analizowany system spełnia wymagania dotyczące generowanego sygnału w formie szeregu czasowego oraz periodyczności przynajmniej części czynności systemu. System podlegający analizie powinien wykonywać część czynności periodycznie np. budzić się, weryfikować wybrany parametr oraz podejmować zadaną czynność. W wyniku zmian środowiska system podejmuje akcje związane ze swoją funkcją. Jest to charakterystyka typowa dla systemów wbudowanych. Algorytm wymaga takiej charakterystyki w celu określenia stanu bazowego. Profil szeregu czasowego analizowanego sygnału powinien być skorelowany z tymczasowym stanem systemu lub operacjami jakie obecnie realizuje. Przykładem takiego szeregu jest szereg próbek wartości natężenia prądu pobieranego przez urządzenie. Możliwe jest wykorzystanie innych typów sygnału w celu analizy z wykorzystaniem algorytmu. Innym sygnałem może być wybrany sygnał analogowy sterujący urządzeniem wykonawczym.

Kolejną cechą algorytmu jest jego parametryzacja opisana w rozdziale 3.1.4. Algorytm posiada wiele parametrów, które mają wpływ na wynikowy poziom abstrakcji modelu obiektów, na liczbę wykrytych stanów i klas cykli oraz sposób określania podobieństwa każdego z typów obiektów. Dobranie poprawnych wartości parametrów jest niezbędne do poprawnego działania algorytmu. Wartości parametrów są związane z typem badanego systemu. Analiza podobnych systemów pozwala na wykorzystanie parametrów już raz ustalonych. Rozdział opisuje także inne metody dobierania parametrów oraz proponuje metodykę ustalania wartości parametrów dla systemu badanego po raz pierwszy. Wartości parametrów oraz rozmiary danych wejściowych wpływają również na czas wykonania oraz wykorzystywaną pamięć przez algorytm. Algorytm cechuje się pesymistyczną złożonością czasową kwadratową $O(n^2)$ – n to liczba próbek szeregu czasowego, oraz złożonością pamięciową kwadratową $O(n^2)$.

Wykryte instancje klas cykli jako rezultat działania algorytmu z rozdziału 3 mogą nie wystarczyć do określania rodzaju i natury stanów urządzenia, które reprezentują. Jeśli

urządzenie generuje logi systemowe/aplikacyjne, wskazane jest ich wykorzystanie do opisania wykrytych obiektów zdarzeń. Algorytm zaprezentowany w rozdziale 4 umożliwia dopasowanie sekwencji rekordów logów tekstowych do listy obiektów zdarzeń. Badane urządzenie generujące logi tekstowe korzysta z własnego zegara, który w systemach wbudowanych często jest niezsynchronizowany z zegarami zewnętrznymi np. używanymi przez urządzenie pomiarowe podczas agregowania sygnału dla algorytmu dekompozycji szeregu czasowego. Skutkuje to utrudnieniem wykorzystania logów tekstowych do opisu wykrytych zdarzeń. Algorytm na podstawie charakterystyk czasowych, grupowania i klasyfikacji logów tekstowych dopasowuje oraz filtruje rekordy logów tekstowych w celu korelacji ich z obiektami zdarzeń. Obiekty zdarzeń posiadają znaczniki czasowe o wartościach z innej osi czasowej. Zaprezentowane podejście pozwala rozwiązać problem synchronizacji a posteriori dwóch różnych typów sekwencji obiektów bez użycia sztucznie wstrzykiwanych markerów. Realizowane jest to poprzez ekstrakcję rekordów logów do formatu worków słów, których postać umożliwia zastosowanie autorskiej metryki podobieństwa w celu zgrupowania logów podobnych oraz odróżnienia ich od logów określanych jako szum. W wyniku działania algorytmu generowana jest wartość przesunięcia między osiami czasów zdarzeń i rekordów logów, a także sekwencje rekordów logów dopasowane do obiektów zdarzeń. Algorytm został przetestowany podczas analizy urządzenia *Holter #1* (rozdział 5.1). Skuteczność zarówno w filtracji jak i w dopasowaniu znaczących rekordów do zdarzeń została potwierdzona. Algorytm jest konfigurowalny poprzez zestaw parametrów. Determinują one poziom graniczny podobieństwa między dwoma rekordami, określają wartość podziału sekwencji rekordów na sekwencje opisujące zdarzenia, a także wyznaczają poziom oddzielenia szumu od rekordów logów znaczących dla zdarzeń. Wartości parametrów w znacznym stopniu zależą od liczby rekordów logów generowanych w jednostce czasu oraz formatu logów tekstowych. W mniejszym stopniu wartości te zależą od typu systemu. Cecha ta pozwala na wykonywanie algorytmu dla wielu różnych kategorii systemów. Ponadto obiektowe podejście do analizy tekstu jak i ich klasyfikacji pozwala na wykonywanie algorytmu na danych logowanych w dowolnym formacie, który zawiera znaczniki czasowe (wymagane są jedynie niewielkie modyfikacje na poziomie implementacji analizatora strukturalnego tekstu).

Następnym etapem w pracach nad monitorowaniem programowym systemów mogą być próby rozwinięcia zaprezentowanych algorytmów poprzez modyfikacje metryk czy też rozszerzenie analizy o wielowymiarowe dane np. w postaci skorelowanych sygnałów. Rozwinięcia te rozszerzyłyby liczbę typów systemów, które wpasowują się charakterystyką w przedmiot analizy algorytmów. Istnieje również możliwość tworzenia kolejnych algorytmów

dokonujących ekstrakcji wiedzy z innych źródeł pochodzących z systemów wbudowanych w celu ich ewaluacji i analizy. Przykładami takich źródeł mogą być: emisje elektromagnetyczne urządzenia lub też wewnętrzna komunikacja między interfejsami urządzenia. Ciekawym podejściem do przedstawianych problemów jest analiza holistyczna zagadnienia z wykorzystaniem fuzji wyników wszystkich trzech zaprezentowanych algorytmów z rozdziałów 2, 3 oraz 4. Fuzja taka stwarza problem powiązania niskopoziomowych wyników analizy śladu instrukcji z wynikami wysokiego poziomu abstrakcji hierarchii obiektów szeregu czasowego. Wiele systemów bazuje na oprogramowaniu, do których źródeł dostęp mają deweloperzy. Fakt ten pozwala na podejmowanie prób powiązania konkretnych bloków kodów źródłowych z obiektami zdarzeń czy anomalii śladów instrukcji. W badaniach korelacji logów zdarzeniowych z obiektami szeregu czasowego wykorzystano logi z rzeczywistych systemów, o dość prostej strukturze. W przypadku bardziej złożonych urządzeń zestaw logów zdarzeniowych może być bardzo bogaty i różnorodny. W tym przypadku korzystnym może być ich analiza strukturalna w celu wyróżnienia określonych klas oraz wyodrębnienia najbardziej interesujących do badań [12] [92].

Proces monitorowania może być również ukierunkowany na aspekty wskazywane w analizach repozytoriów dotyczących zgłaszanych błędów (np. Jira, Bugzillla) lub kontroli wersji kodu (np. Git) [93]. W pracy zakładano dostępność zestawu scenariuszy testowych do przeprowadzenia procesu monitorowania. Generacja takich scenariuszy jest odrębnym problemem silnie zależnym od aplikacji (np. [9] [56]). Scenariusze te można by rozszerzyć o badanie zachowania się systemu przy zasymulowaniu zakłóceń np. błędów przemijających [58] wykorzystując odpowiednie symulatory błędów [7] [8]. W tym przypadku pojawia się również problem korelacji czasowej dyskutowany w rozdziale 4.

7 Bibliografia

[1] H. Kopetz, „The Complexity Challenge in Embedded System Design,” w *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2008.

[2] K. Krosman and J. Sosnowski, "Correlating Time Series Signals and Event Logs in Embedded Systems," *Sensors*, vol. 21, p. 7128, 1 2021.

[3] K. Krosman, J. Sosnowski and P. Gawkowski, "Object oriented time series exploration: Applied to power consumption analysis of embedded systems," *Expert Systems with Applications*, vol. 184, p. 115531, 12 2021.

[4] K. Krosman, „Instruction trace analysis and enhanced debugging in embedded systems,” w *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2018*, 2018.

[5] K. Krosman i J. Sosnowski, „ESD problems in embedded systems,” w *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2019*, 2019.

[6] K. Krosman i J. Sosnowski, „Fault tolerance techniques for embedded telemetry system: case study,” w *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2016*, 2016.

[7] D. Trawczyński, J. Sosnowski and P. Gawkowski, "Testing Distributed ABS System with Fault Injection," in *Innovations in Computing Sciences and Software Engineering*, Dordrecht, 2010.

[8] M. Mosdorf and J. Sosnowski, "Fault injection in embedded systems using GNU Debugger," *Pomiary Automatyka Kontrola*, Vols. R. 57, nr 8, p. 825–827, 2011.

[9] K. Krosman i J. Sosnowski, „Exploring disk performance benchmarks,” w *Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2017*, 2017.

[10] J. Sosnowski, P. Gawkowski and K. Cabaj, "Exploring the Space of System Monitoring," R. Bembenik, L. Skonieczny, H. Rybinski, M. Kryszkiewicz and M. Niezgodka, Eds., Berlin, Heidelberg: Springer, 2013, p. 501–517.

[11] M. Kubacki i J. Sosnowski, „Exploring Operational Profiles and Anomalies in Computer Performance Logs,” *Microprocessors and Microsystems,* tom 69, 5 2019.

[12] M. Kubacki i J. Sosnowski, „Holistic Processing and Exploring Event Logs,” w *Software Engineering for Resilient Systems*, Cham, 2017.

[13] J. Sosnowski i M. Poleszak, „On-line monitoring of computer systems,” w *Third IEEE International Workshop on Electronic Design, Test and Applications (DELTA'06)*, 2006.

[14] P. Gawkowski, "Extending Icinga Monitoring Capabilities," *Information Systems in Management*, vol. 5, p. 481–491, 2016.

[15] S. Mensah, J. Keung, K. E. Bennin i M. F. Bosu, *Multi-Objective Optimization for Software Testing Effort Estimation,* SEKE, 2016.

[16] C.-G. Guo, X.-L. Li i J. Zhu, „A Generic Model for Software Monitoring Techniques and Tools,” w *2010 Second International Conference on Networks Security, Wireless Communications and Trusted Computing*, 2010.

[17] S. Kong, M. Lu, L. Li i L. Gao, „Runtime Monitoring of Software Execution Trace: Method and Tools,” *IEEE Access*, tom 8, p. 114020–114036, 2020.

[18] D. Wang i J. Pan, „An approach of automatically performing Fault Tree Analysis and failure mode and effect techniques to software processes,” w *The 2nd International Conference on Software Engineering and Data Mining*, 2010.

[19] H. Kawashima, „KRAFT: A Real-Time Active DBMS for Signal Streams,” w *2007 Fourth International Conference on Networked Sensing Systems*, 2007.

[20] D. F. Koranek, S. R. Graham, B. J. Borghetti i W. C. Henry, „Identification of Return-Oriented Programming Attacks Using RISC-V Instruction Trace Data,” *IEEE Access*, tom 10, p. 45347–45364, 2022.

[21] C. Acarturk, M. Sirlanci, P. G. Balikcioglu, D. Demirci, N. Sahin i O. A. Kucuk, „Malicious Code Detection: Run Trace Output Analysis by LSTM,” *IEEE Access*, tom 9, p. 9625–9635, 2021.

[22] S. Kong, J. Ai, M. Lu, S. Wang i W. E. Wong, „DistAD: Software Anomaly Detection Based on Execution Trace Distribution,” *arXiv:2202.13898 [cs]*, 4 2022.

[23] G. Pagano, D. Dosimont, G. Huard, V. Marangozova-Martin i J. Vincent, „Trace Management and Analysis for Embedded Systems,” w *2013 IEEE 7th International Symposium on Embedded Multicore Socs*, 2013.

[24] J. Kraft, A. Wall and H. Kienle, "Trace Recording for Embedded Systems: Lessons Learned from Five Industrial Projects," in *Runtime Verification*, Berlin, 2010.

[25] D. Zheng, F. Li and T. Zhao, "Self-adaptive statistical process control for anomaly detection in time series," *Expert Systems with Applications*, vol. 57, p. 324–336, 9 2016.

[26] H. Chen i H. Liu, „A remote electrocardiogram monitoring system with good swiftness and high reliablility," *Computers & Electrical Engineering*, tom 53, 2 2016.

[27] S. Raj i K. Ray, „Sparse representation of ECG signals for automated recognition of cardiac arrhythmias," *Expert Systems with Applications*, tom 105, 3 2018.

[28] R. San-Segundo, J. Montero, R. Barra-Chicote, F. Fernández-Martínez i J. Pardo, „Feature extraction from smartphone inertial signals for human activity segmentation," *Signal Processing*, tom 120, p. 359–372, 3 2016.

[29] A. Carroll i G. Heiser, „An Analysis of Power Consumption in a Smartphone," w *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USA, 2010.

[30] B. Martinez, M. Montón, I. Vilajosana i J. D. Prades, „The Power of Models: Modeling Power Consumption for IoT Devices," *IEEE Sensors Journal*, tom 15, p. 5777–5789, 10 2015.

[31] A. Kozłowski and J. Sosnowski, "Energy Efficiency Trade-Off Between Duty-Cycling and Wake-Up Radio Techniques in IoT Networks," *Wireless Personal Communications*, vol. 107, p. 1951–1971, 8 2019.

[32] H. Huang, J. Chen, R. Sun and S. Wang, "Short-term traffic prediction based on time series decomposition," *Physica A: Statistical Mechanics and its Applications*, vol. 585, p. 126441, 1 2022.

[33] M. Kubacki i J. Sosnowski, „Applying time series analysis to performance logs," w *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2015*, 2015.

[34] B. Pilastre, L. Boussouf, S. D'Escrivan i J.-Y. Tourneret, „Anomaly detection in mixed telemetry data using a sparse representation and dictionary learning," *Signal Processing*, tom 168, p. 107320, 3 2020.

[35] R. E. Sperl i S. M. Chung, „Two-Step Anomaly Detection for Time Series Data," w *2019 International Conference on Data and Software Engineering (ICoDSE)*, 2019.

[36] J. Zhao i L. Itti, „Decomposing time series with application to temporal segmentation,” *2016 IEEE Winter Conference on Applications of Computer Vision (WACV),* 2016.

[37] H. Kamalzadeh, A. Ahmadi i S. Mansour, „A Shape-Based Adaptive Segmentation of Time-Series Using Particle Swarm Optimization,” *Inf. Syst.,* tom 67, p. 1–18, 7 2017.

[38] V. Rajagopalan and A. Ray, "Symbolic time series analysis via wavelet-based partitioning," *Signal Processing,* vol. 86, p. 3309–3320, 11 2006.

[39] H. Yahyaoui i R. Al-Daihani, „A Novel Trend based SAX Reduction Technique for Time Series,” *Expert Systems with Applications,* tom 130, 4 2019.

[40] D. Katircioglu-Öztürk, H. A. Güvenir, U. Ravens and N. Baykal, "A window-based time series feature extraction method," *Computers in Biology and Medicine,* vol. 89, p. 466–486, 10 2017.

[41] S. Liu and K. Yu, "Successive multivariate variational mode decomposition based on instantaneous linear mixing model," *Signal Processing,* vol. 190, p. 108311, 1 2022.

[42] V. Mazaheri and H. Khodadadi, "Heart arrhythmia diagnosis based on the combination of morphological, frequency and nonlinear features of ECG signals and metaheuristic feature selection algorithm," *Expert Systems with Applications,* vol. 161, p. 113697, 12 2020.

[43] C. Johnpaul, M. V. N. K. Prasad, S. Nickolas and G. R. Gangadharan, "Trendlets: A novel probabilistic representational structures for clustering the time series data," *Expert Systems with Applications,* vol. 145, p. 113119, 5 2020.

[44] Y. Qin, B. Tang and Y. Mao, "Adaptive signal decomposition based on wavelet ridge and its application," *Signal Processing,* vol. 120, p. 480–494, 3 2016.

[45] M. Bendre and R. Manthalkar, "Time series decomposition and predictive analytics using MapReduce framework," *Expert Systems with Applications,* vol. 116, p. 108–120, 2 2019.

[46] R. Al-Hmouz, W. Pedrycz i A. Balamash, „Description and prediction of time series: A general framework of Granular Computing,” *Expert Systems with Applications,* tom 42, p. 4830–4839, 6 2015.

[47] L. G. B. Ruiz, M. C. Pegalajar, R. Arcucci and M. Molina-Solana, "A time-series clustering methodology for knowledge extraction in energy consumption data," *Expert Systems with Applications,* vol. 160, p. 113731, 12 2020.

[48] S. Liu, B. Zhou, Q. Ding, B. Hooi, Z. b. Zhang, H. Shen i X. Cheng, „Time Series Anomaly Detection with Adversarial Reconstruction Networks,” *IEEE Transactions on Knowledge and Data Engineering,* p. 1–1, 2022.

[49] J. F. Torres, D. Hadjout, A. Sebaa, F. Martínez-Álvarez i A. Troncoso, „Deep Learning for Time Series Forecasting: A Survey,” *Big Data,* tom 9, p. 3–21, 2 2021.

[50] K. Choi, J. Yi, C. Park i S. Yoon, „Deep Learning for Anomaly Detection in Time-Series Data: Review, Analysis, and Guidelines,” *IEEE Access,* tom 9, p. 120043–120065, 2021.

[51] J. Hills, J. Lines, E. Baranauskas, J. Mapp and A. Bagnall, "Classification of time series by shapelet transformation," *Data Mining and Knowledge Discovery,* vol. 28, p. 851–881, 7 2014.

[52] G. Li, W. Yan and Z. Wu, "Discovering shapelets with key points in time series classification," *Expert Systems with Applications,* vol. 132, p. 76–86, 10 2019.

[53] P. Schäfer i U. Leser, „Fast and Accurate Time Series Classification with WEASEL,” *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management,* p. 637–646, 11 2017.

[54] C. H. Lubba, S. S. Sethi, P. Knaute, S. R. Schultz, B. D. Fulcher and N. S. Jones, "catch22: CAnonical time-series CHaracteristics," *1852,* 11 2019.

[55] A. Bondu, D. Gay, V. Lemaire, M. Boullé i E. Cervenka, „FEARS: a Feature and Representation Selection approach for Time Series Classification,” w *Proceedings of The Eleventh Asian Conference on Machine Learning*, Nagoya, 2019.

[56] M. J. Jeleński and J. Sosnowski, "Mobile Application Testing and Assessment," in *Theory and Applications of Dependable Computer Systems*, Cham, 2020.

[57] J. Sosnowski, „Transient fault tolerance in digital systems,” *IEEE Micro,* tom 14, p. 24–35, 2 1994.

[58] A. Kozłowski i J. Sosnowski, „Evaluating energy consumption in wireless sensor networks,” tom 0808, p. 108085W, 10 2018.

[59] F. Xu, Y. Liu, Q. Li i Y. Zhang, „V-edge: Fast Self-constructive Power Modeling of Smartphones Based on Battery Voltage Dynamics,” w *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, Lombard, 2013.

[60] M. Al-Kofahi, M. Al-Shorman i O. Al-Kofahi, Toward Energy Efficient Microcontrollers and IoT Systems, 2019.

[61] P. He, J. Zhu, S. He, J. Li i M. R. Lyu, „Towards Automated Log Parsing for Large-Scale Log Data Analysis," *IEEE Transactions on Dependable and Secure Computing,* tom 15, p. 931–944, 11 2018.

[62] B. Zhang, H. Zhang, P. Moscato i A. Zhang, „Anomaly Detection via Mining Numerical Workflow Relations from Logs," 9 2020.

[63] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng i M. R. Lyu, „Tools and Benchmarks for Automated Log Parsing," *arXiv:1811.03509 [cs],* 1 2019.

[64] S. Locke, H. Li, T.-H. P. Chen, W. Shang i W. Liu, „LogAssist: Assisting Log Analysis Through Log Summarization," *IEEE Transactions on Software Engineering,* p. 1–1, 2021.

[65] S. Mishra, Z. Shafi and S. Pathak, "Time series event correlation with DTW and Hierarchical Clustering methods," 2019.

[66] R. Harper i P. Tee, „A Method for Temporal Event Correlation," w *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019.

[67] A. Prokoph i H. El Bilali, „Cross-Wavelet Analysis: A Tool for Detection of Relationships Between Paleoclimate Proxy Records," *Mathematical geosciences,* tom 40, p. 575–586, 7 2008.

[68] E. Ghaderpour and T. Vujadinovic, "The Potential of the Least-Squares Spectral and Cross-Wavelet Analyses for Near-Real-Time Disturbance Detection within Unequally Spaced Satellite Image Time Series," *Remote Sensing,* vol. 12, p. 2446, 1 2020.

[69] F. S. L. G. Duarte, R. A. Rios, E. R. Hruschka and R. F. de Mello, "Decomposing time series into deterministic and stochastic influences: A survey," *Digital Signal Processing,* vol. 95, p. 102582, 12 2019.

[70] B. Bai, G. Li, S. Wang, Z. Wu i Y. Wen, „Time Series Classification based on Multi-feature Dictionary Representation and Ensemble Learning," *Expert Systems with Applications*, tom 169, p. 114162, 12 2020.

[71] J. Wang, Y. Tang, S. He, C. Zhao, P. K. Sharma, O. Alfarraj and A. Tolba, "LogEvent2vec: LogEvent-to-Vector Based Anomaly Detection for Large-Scale Logs in Internet of Things," *Sensors,* vol. 20, p. 2451, 1 2020.

[72] X. Li, Y. Kang and F. Li, "Forecasting with time series imaging," *Expert Systems with Applications*, vol. 160, p. 113680, 12 2020.

[73] W. Fan, Y. Li, K. L. Tsui i Q. Zhou, „A Noise Resistant Correlation Method for Period Detection of Noisy Signals,” *IRE Transactions on Audio,* tom 66, p. 2700–2710, 5 2018.

[74] Y. Li, H. Zhao, W. Fan i C. Shen, „Extended Noise Resistant Correlation Method for Period Estimation of Pseudoperiodic Signals,” *IEEE Transactions on Instrumentation and Measurement,* tom 70, p. 1–11, 2021.

[75] Z. Faget, P. Rigaux, D. Gross-Amblard i V. Thion-Goasdoué, „Modeling synchronized time series,” w *Proceedings of the Fourteenth International Database Engineering & Applications Symposium*, New York, NY, USA, 2010.

[76] N. Nava, T. Di Matteo and T. Aste, "Dynamic correlations at different time-scales with empirical mode decomposition," *Physica A: Statistical Mechanics and Its Applications,* vol. 502, p. 534–544, 7 2018.

[77] J. Cabrieto, F. Tuerlinckx, P. Kuppens, B. Hunyadi i E. Ceulemans, „Testing for the Presence of Correlation Changes in a Multivariate Time Series: A Permutation Based Approach,” *Scientific Reports,* tom 8, p. 769, 1 2018.

[78] C. Zhang, X. Wang, H. Zhang, H. Zhang i P. Han, „Log Sequence Anomaly Detection Based on Local Information Extraction and Globally Sparse Transformer Model,” *IEEE Transactions on Network and Service Management,* tom 18, p. 4119–4133, 12 2021.

[79] C. Luo, J.-G. Lou, Q. Lin, Q. Fu, R. Ding, D. Zhang i Z. Wang, „Correlating events with time series for incident diagnosis,” w *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining,* New York, NY, USA, 2014.

[80] P. Xun, P.-D. Zhu, C.-L. Li i H.-Y. Zhu, „Discovering Multi-type Correlated Events with Time Series for Exception Detection of Complex Systems,” w *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, Barcelona, 2016.

[81] M. van Dortmont, S. van den Elzen and J. van Wijk, "ChronoCorrelator: Enriching Events with Time Series," *Computer Graphics Forum,* vol. 38, p. 387–399, 2019.

[82] B. Minaei-Bidgoli i S. B. Lajevardi, „Correlation Mining between Time Series Stream and Event Stream,” *2008 Fourth International Conference on Networked Computing and Advanced Information Management,* 2008.

[83] H. Yigitler, B. Badihi and R. Jäntti, "Overview of Time Synchronization for IoT Deployments: Clock Discipline Algorithms and Protocols," *Sensors,* vol. 20, p. 5928, 1 2020.

[84] K. Skiadopoulos, A. Tsipis, K. Giannakis, G. Koufoudakis, E. Christopoulou, K. Oikonomou, G. Kormentzas and I. Stavrakakis, "Synchronization of data measurements in wireless sensor networks for IoT applications," *Ad Hoc Networks,* vol. 89, p. 47–57, 6 2019.

[85] P. Arafa, D. Solomon, S. Navabpour i S. Fischmeister, „Debugging Behaviour of Embedded-Software Developers: An Exploratory Study," w *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC),* Raleigh, 2017.

[86] A. Avizienis, J.-C. Laprie, B. Randell i C. Landwehr, „Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing,* tom 1, p. 11–33, 1 2004.

[87] A. Cuzzocrea, E. Mumolo i R. Cecolin, „Runtime Anomaly Detection in Embedded Systems by Binary Tracing and Hidden Markov Models," w *2015 IEEE 39th Annual Computer Software and Applications Conference,* 2015.

[88] W. Zhang, F. Bastani, I. Yen, K. Hulin, F. Bastani i L. Khan, „Real-Time Anomaly Detection in Streams of Execution Traces," w *Ninth IEEE International Symposium on High-Assurance Systems Engineering (HASE'05),* Los Alamitos, CA, USA, 2012.

[89] M. Ester, H. P. Kriegel, J. Sander i X. Xiaowei, „A density-based algorithm for discovering clusters in large spatial databases with noise," 12 1996.

[90] D. Birant and A. Kut, "ST-DBSCAN: An algorithm for clustering spatial–temporal data," *Data & Knowledge Engineering,* vol. 60, p. 208–221, 1 2007.

[91] K. Khan, S. U. Rehman, K. Aziz, S. Fong i S. Sarasvady, „DBSCAN: Past, present and future," w *The Fifth International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2014),* 2014.

[92] Y. Liu, X. Zhang, S. He, H. Zhang, L. Li, Y. Kang, Y. Xu, M. Ma, Q. Lin, Y. Dang, S. Rajmohan and D. Zhang, "UniParser: A Unified Log Parser for Heterogeneous Log Data," 2022.

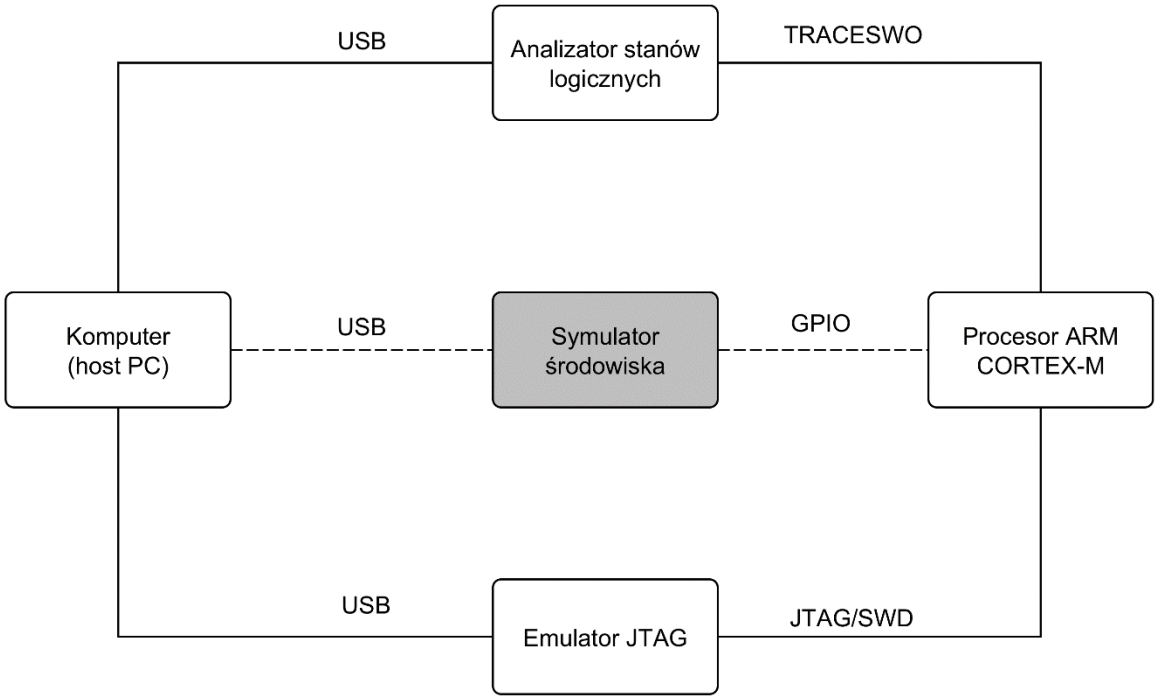
[93] J. Sosnowski, B. Dobrzyński and P. Janczarek, "Analysing problem handling schemes in software projects," *Information and Software Technology,* vol. 91, p. 56–71, 11 2017.

[94] Saleae, „Saleae," 2018. [Online]. Available: http://www.saleae.com/. [Data uzyskania dostępu: 10 04 2018].

[95] G. Pagano, D. Dosimont, G. Huard, V. Marangozova-Martin i J.-M. Vincent, „Trace Management and Analysis for Embedded Systems,” w *2013 IEEE 7th International Symposium on Embedded Multicore Socs*, 2013.

8 Załączniki

8.1 System agregacji i analizy śladu instrukcji [TraceAnalyzer]



Rysunek 50. Architektura systemu detekcji anomalii śladu instrukcji

System składa się z komponentów takich jak: (a) autorska aplikacja sterująca uruchamiana na komputerze gospodarzu (HOST PC), (b) dostępny komercyjnie analizator stanów logicznych [94], (c) dostępny komercyjnie emulator/debugger JTAG, (d) autorski symulator środowiska (ENVIROMENT SIMULATOR), (e) badany system wbudowany. Wszystkie elementy architektury zaprezentowane są na rysunku 50. Aplikacja sterująca jest nieskomplikowanym programem terminalowym uruchamianym na komputerze PC. Aplikacja jest odpowiedzialna za kontrolę kolejnych iteracji testów, sterowanie symulatorem środowiska, konfigurację agregacji śladu instrukcji i jego zapis. Autorska aplikacja została napisana w języku c++ wykorzystując zewnętrzne biblioteki i oprogramowanie wykonywane na żądanie. Aplikacja implementuje przedstawiany autorski algorytm wykorzystujący ślad instrukcji. Ślad jest odczytywany przy użyciu analizatora stanów logicznych kontrolowanego przy użyciu biblioteki libsigrok. SIGROK jest otwartym oprogramowaniem obsługującym ponad 100 różnych platform sprzętowych, które można wykorzystać jako analizatory stanów logicznych. Ponadto możliwe jest dodawanie wtyczek, które dekodują sygnał cyfrowy zgodnie z zadanym protokołem. Aplikacja wykorzystuje oprogramowanie SIGROK do odczytywania

i dekodowania śladu ETM. Konfiguracja SIGROK z wtyczką dekodującą strumienie TPIU oraz ETM zawiera kolejno: (1) nazwę wykorzystywanego sterownika analizatora logicznego, (2) numer linii analizatora logicznego na którym realizowana będzie transmisja śladu, (3) częstotliwość transmisji, (4) numery strumieni TPIU, (5) wartość flagi włączającą dekodowanie strumienia ITM, (6) wartość flagi włączającą dekodowanie ETM, (7) ścieżkę do pliku wykonywalnego badanego systemu zawierającego symbole, (8) czas agregacji śladu, (9) ścieżkę do pliku wynikowego zawierającego przetworzony ślad instrukcji. Plik wynikowy jest zapisany w formacie tekstowym np. ślad 5 kolejnych instrukcji może mieć postać: "PC: 0x8000ab1\nPC: 0x8000ab11\nPC: 0x8000ab12\nPC: 0x8000ab13\nPC: 0x8000ab14". Po ciągu znakowym "PC:" wypisywany jest adres instrukcji wykonanej przez procesor w kodzie szesnastkowym. Pojedynczy rekord śladu z adresem instrukcji kończy znak nowej linii. W pliku wynikowym śladu można także odnaleźć informację o zaraportowanej przyczynie skoku np. przerwanie wraz z numerem przerwania. Plik wynikowy stanowi podstawowe dane wejściowe dla algorytmu detekcji anomalii. W prezentowanym systemie wykorzystywany jest analizator Saleae 8, który agreguje sygnał z linii TRACESWO standardowego złącza microARM.

W celu konfiguracji testowanego systemu aplikacja sterująca za pośrednictwem oprogramowania OpenOCD wstrzymuje pracę badanego systemu, a następnie poprzez interfejs JTAG/SWD konfiguruje makrocele ETM, ITM i TPIU. Konfiguracja jest realizowana jako zapis do rejestrów poprzez magistralę ATB, do której dostęp możliwy jest z poziomu interfejsu JTAG. OpenOCD jest oprogramowaniem umożliwiającym zunifikowaną kontrolę nad różnymi emulatorami. OpenOCD udostępnia język skryptowy, który umożliwia konfigurację oprogramowania pod kątem wykorzystania konkretnego sprzętu-emulatora, a także wykonywanie poleceń debugujących badanego systemu, takich jak: wstrzymanie wykonywania kodu na rdzeniu mikrokontrolera, zmiana zawartości pamięci, odczyt rejestrów i pamięci czy też załadowanie obrazu oprogramowania i rozpoczęcia jego wykonywania. Wszystkie te operacje oraz ich sekwencje są przygotowane w postaci plików skryptu OpenOCD, które są wywoływane przez aplikację sterującą. Przykładem takiego skryptu jest listing 1, który przedstawia skrypt konfigurujący agregację śladu instrukcji. Każdy z trzech modułów sprzętowych (analizator stanów logicznych, emulator JTAG, symulator środowiska) podłączony jest do komputera PC za pośrednictwem USB. Zaprezentowana architektura sprzętowa pozwala na realizację, kontrolę i monitoring testów/eksperymentów nad dowolnym systemem opartym o mikrokontroler z rdzeniem Cortex-M, który zawiera złącze JTAG/SWD wraz z pinem TRACESWO.

```
#                IO Pin Configuration
setbits          $RCC_APB2ENR 1 ;                        # alternative mode for TRACESWO pin
setbits          $AFIO_MAPR 0x2000000 ;                  # disable JTAG
setbits          $DBGMCU_CR 0x20 ;                       # Enable ETM trace during sleep mode
#                TPIU
setbits          $COREDEBUG_DEMCR 0x1000000 ; # Unlock key
mww              $TPI_ACPR 0 ;                           # TPIU clock divider HCLK/(x+1)-> 8Mhz
mww              $TPI_SPPR 2 ;                           # Set protocol to UART 1 pin
mww              $TPI_FFCR 0x102 ;                       # Enable TPIU (2 channels)
#                DWT
mww              $DWT_CTRL 0x40011a01 ;        # PC sampling- 1 sample for each 512 instruction
#                ITM
mww              $ITM_LAR 0xC5ACCE55
mww              $ITM_TCR 0x0001000d ;          # ITM will be transmitted via 1st TPIU channel
mww              $ITM_TER 0xffffffff ;          # Enable all 32 ITM channels
#                ETM
mww              $ETM_LAR 0xC5ACCE55
setbits          $ETM_CR 0x400 ;                # ETM programming mode
mww              $ETM_CR 0xd80 ;                # Report all branches
mww              $ETM_TRACEIDR 2 ;              # ETM will be transmitted via 2nd TPIU channel
mww              $ETM_TECR1 0x1000000 ;         # Enable ETM
mww              $ETM_FFRR 0x1000000 ;          # Always block on FIFO full
mww              $ETM_FFLR 24 ;                 # Block when only 24 bytes lefts in FIFO
```

Listing 1. Skrypt OpenOCD konfigurujący generowanie śladu ETM.

Skrypt z listingu 1 przedstawia kolejne instrukcje, które wykonuje OpenOCD w badanym systemie. W pierwszym kroku konfigurowane są linie IO, służące do transferu śladu. Włączany jest tryb alternatywny SWO jako TRACESWO. Wyłączane są piny dostępne dla interfejsu JTAG- komunikacji z modułami sprzętowymi debugowymi będzie realizowana tylko poprzez linie sygnałowe SWD. W rejestrze DBGMCU_CR włączane jest generowanie śladu także w sytuacji, gdy mikrokontroler wchodzi w tryb uśpienia. Kolejnym krokiem jest konfiguracja makroceli TPIUI. Konfiguracja składa się z: (1) odblokowania rejestrów ETM/TPIU wpisem wartości klucza odblokowującego do odpowiedniego rejestru, (2) ustawienia dzielnika zegara dla układu wyjściowego TPIU- w tym wypadku częstotliwość sygnału zegarowego TPIU równa jest częstotliwości zegara rdzenia mikrokontrolera np. 8 MHz, (3) wybrania trybu 1-pin UART, (4) włączenia TPIU w trybie dwukanałowym- jeden dla danych z makroceli ITM, a drugi dla śladu ETM. Do rejestru DWT_CTRL wpisywana jest wartość częstotliwości próbkowania wartości program licznika m.in w celu synchronizacji ze śladem ETM. W listingu 1 DWT jest konfigurowane do wysyłania wartości rejestru PC co 512 wykonanych instrukcji. Konfiguracja makroceli ITM rozpoczyna się od wpisania stałej wartości do rejestru ITM_LAR w celu odblokowania możliwości zapisu do rejestrów ITM. Następnie makrocela ITM jest podłączana do kanału 1 TPIU. Analogicznie konfiguracja ETM inicjowana jest przez odblokowanie zapisu i wejście w tryb programowania ETM. W rejestrze ETM_CR ustawiana jest flaga odpowiedzialna za raportowanie każdej wykonanej instrukcji skoku. Pakiet

z informacją o skoku, jego adresie i wyniku pozwala wyznaczyć dokładnie kolejne adresy wykonanych instrukcji, które składają się na ślad. Ślad generowany przez ETM przypisywany jest do kanału 2 makroceli TPIU. Końcowa część konfiguracji wybiera tryb wstrzymywania wykonania instrukcji w momencie, gdy w kolejce FIFO TPIU zostało mniej niż 24 bajty. Ostatni wpis w skrypcie uruchamia generowanie śladu ETM. Po wykonaniu skryptu badany system rozpoczyna generowanie i wysyłanie śladu wykorzystując transmisję szeregową UART na linii TRACESWO.

Program sterujący jest programem terminalowym, który jest konfigurowany tekstowym plikiem konfiguracyjnym wczytywanym przy starcie systemu. Dane konfiguracyjne zapisane są w formacie [klucz]=[wartość]- jedna para przypada na jedną linię pliku. Plik definiuje wartości parametrów takich jak: (a) liczba iteracji testu, (b) warunek zakończenia pojedynczej iteracji testu, (c) typ testu wraz z parametrami konfiguracyjnymi analizy, (d) parametry konfiguracyjne dla oprogramowania sterującego emulatorem JTAG, (e) parametry konfiguracyjne dla oprogramowania sterującego agregacja śladu za pośrednictwem analizatora stanów logicznych, (f) ścieżki do plików zawierających ślad instrukcji oraz obraz oprogramowania badanego urządzenia w formacie ELF, (g) ścieżka do pliku do którego będą zapisywane wyniki testu. Parametry (a), (b) i (c) określają metodę testu. Warunek zakończenia iteracji testu może być zdefiniowany jako czas trwania testu lub rozmiar śladu. Skrypty OpenOCD umożliwiają konfigurację generowania śladu oraz przygotowanie badanego systemu do danej iteracji np. poprzez reset oprogramowania. Parametry konfiguracyjne oprogramowania kontrolującego analizator stanów logicznych pozwalają uruchomić i zapisywać ślad instrukcji do wybranego pliku. Tworzony jest osobny plik śladu dla każdej pojedynczej iteracji testu.

System pozwala wykonać cztery typy analizy badanego systemu: (1) profilowanie wydajnościowe oprogramowania o granulacji funkcji, (2) test pokrycia kodu, (3) analiza przebiegu wykonania instrukcji w czasie, (4) detekcja anomalii. Profil wydajnościowy oprogramowania generowany przez system zawiera częstości wykonywania kodu danej funkcji znormalizowane do procentowej wartości względem wszystkich wystąpień. Profilowanie realizowane jest poprzez próbkowanie wartości rejestru licznika rozkazów przy użyciu makroceli DWT. Odczytując wartość adresu instrukcji w stałych odstępach czasu system otrzymuje listę adresów. Następnie na podstawie pliku obrazu oprogramowania grupuje adresy instrukcji względem przynależności do danej funkcji na poziomie języka programowania. Liczność wystąpień funkcji w analizowanym zbiorze wyznacza częstość przepływu sterowania oprogramowania przez daną funkcję, pozwalając oszacować ranking sumarycznego czasu

procesora poświęconego na wykonywanie operacji w ramach procedury. Próbkowanie realizowane okresowo przy użyciu DWT gwarantuje, że przy odpowiedniej konfiguracji makrocel ETM/DWT/ITM (m.in wyłączone generowanie śladu ETM) mikrokontroler nie wstrzyma pracy, a cały test zostanie zrealizowany bez wpływu na parametry czasowe oprogramowania i sprzętu. Test pokrycia kodu bazuje na danych śladu instrukcji. Analizując kod wykonywalny oprogramowania każdą funkcję można podzielić na bloki. Każdy blok kodu rozpoczyna się i kończy instrukcją powodującą zmianę w przepływie sterowania. Przykładem takich instrukcji są: instrukcje skoku, skoków warunkowych, skoków z zapamiętam adresu powrotu. System dysponując śladem wykonania instrukcji jest w stanie wskazać które bloki kodu zostały wykonane. Informacje o blokach kodu, do których trafiło sterowanie zapisywane są do pliku wynikowego w postaci: <nazwa funkcji, adres pierwszej instrukcji bloku>. Ponadto wyliczana jest metryka pokrycia kodu testami: liczba bloków odwiedzonych/liczba wszystkich bloków. Na podstawie śladu instrukcji aplikacja sterująca może wygenerować przebieg wykonania kolejnych instrukcji. Aplikacja przegląda ślad od pierwszej instrukcji do ostatniej. Każda instrukcja skoku z adresem powrotu oraz instrukcja skoku do adresu powrotu generuje rekord przebiegu. Rekord przebiegu jest parą <znacznik czasowy, nazwa funkcji>. Nazwa funkcji jest odczytywana z pliku wykonywalnego na podstawie adresu skoku/powrotu. Znacznik czasowy wyznaczany jest na podstawie pakietów synchronizacyjnych ETM zawierających wartość licznika CYCCNT. Rekordy zapisywane są do pliku wynikowego. Danymi wejściowymi dla algorytmu detekcji anomalii jest ciąg plików zawierających ślady instrukcji z każdej iteracji testu. W wyniku działania algorytmu generowana jest lista wykrytych anomalii w formie segmentów instrukcji, które zostały sklasyfikowane jako anomalie. Każda taka sekwencja zawiera numer segmentu oraz numer grupy anomalii, do której został sklasyfikowany. Ponadto generowane są dane w formacie CSV pozwalające na wydruk wykresu ilości anomalii w zależności od numeru sekwencji.

8.2 System detekcji sekwencji stanów/klas cykli oraz korelacji logów tekstowych [StateEventAnalyzer]

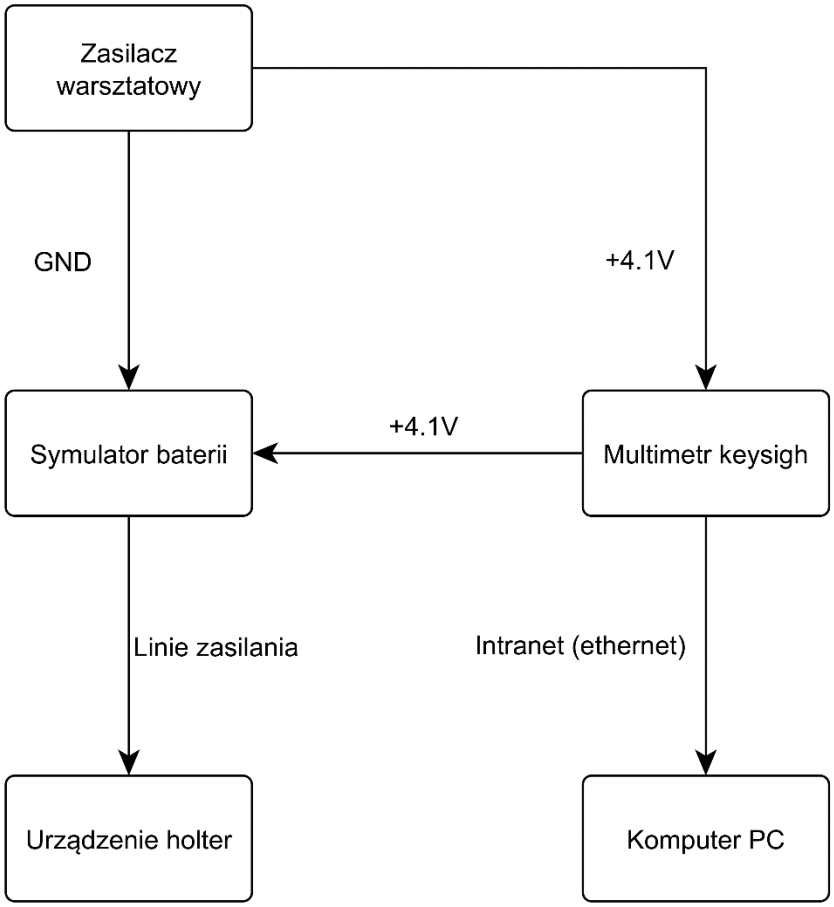
W rozdziale 8.2 opisany został system *StateEventAnalyzer*, który implementuje oba zaprezentowane algorytmy (z rozdziałów 3 i 4) oraz pozwala na eksport i dalszą analizę badanych systemów na podstawie wiedzy już wywnioskowanej. Algorytmy te zostały zaimplementowane w postaci rzeczywistego systemu analizy poboru prądu przez systemy wbudowane. System został zaprojektowany jako program, który przetwarza wcześniej zapisane

pliki zawierające próbki poboru prądu oraz rekordy logów tekstowych. Rysunek 51 przedstawia schemat zestawu testowego, dzięki któremu uzyskano ciąg próbek natężenia prądu pobieranego przez *Holter #1*. W pierwszym kroku autorski skrypt łączy się za pośrednictwem sieci wewnętrznej z urządzeniem *Keysight*, które jest wpięte w układ testowy w roli amperomierza. Skrypt za pośrednictwem protokołu SCPI konfiguruje parametry akwizycji danych takie jak: zakres wartości natężenia prądu oraz częstotliwość próbkowania. Następnie skrypt zapisuje pakiety z próbkami zebranymi przez *Keysight* do zadanego pliku. Format danych jest tekstowy. Przykład fragmentu danych przedstawiony jest na listingu 2. Czas agregacji próbek jest ustalony, bądź też eksperyment może zostać przerwany na żądanie.

```
# Connected to 192.168.1.129
# Running script scripts/current.scpi
#
# *rst
# *idn?
# Keysight Technologies,34461A,MY57218846,A.02.17-02.40-02.17-00.52-04-02
# disp:view tch
# syst:date?; time?; lab?
# +2020,+11,+23;16,3,23.184;"telit1"
# sens:curr:dc:term 3; rang 10mA; aper MIN
# sens:func:on "curr"
# trig:coun INF
# *cls
# init:imm
# [2020-11-23 16:36:28.041150843]
+1.35057532E-05
-5.46834143E-06
-8.67726908E-06
+5.07822930E-06
-9.77047828E-06
+4.23981532E-06
-7.09581717E-06
-7.47800999E-06
+1.07497183E-05
-5.23489392E-06
+1.21831135E-06
+3.33402981E-06
-6.52620066E-06
-2.57090667E-06
+1.86930165E-06
-2.38681138E-06
+6.21413391E-06
(…)
-1.21503344E-04
-1.12202056E-04
-1.30072268E-04
# [2020-11-23 21:35:18.468382386]
-9.62034420E-05
-1.14176375E-04
-1.30383761E-04
-1.11922585E-04
(…)
```

Listing 2. Przykład fragmentu danych otrzymywanych poprzez protokół SCPI.

Na listingu 2 można zaobserwować, że wartości próbek pomiaru natężenia prądu wysyłane są w postaci periodycznie otrzymywanych pakietów liczb zmiennoprzecinkowych w formacie tekstowym oraz notacji wykładniczej. Każdy pakiet jest oznaczony znacznikiem czasowym oraz zawiera stałą i konfigurowalną liczbę próbek. W pierwszym pakiecie za znakiem '#' wysyłana jest użyta konfiguracja zawierająca informacje o zakresie oraz jednostce wartości próbek.



Rysunek 51. Schemat zestawu testowego do agregacji danych o poborze prądu przez badane urządzenie

Uzyskany plik jest plikiem wejściowym dla autorskiego programu *StateEventAnalyzer*.

8.2.1 Opis systemu StateEventAnalyzer

System udostępnia interfejs konsolowy ze zdefiniowanymi komendami. Za pomocą komend użytkownik steruje potokiem wykonania algorytmu, ustawieniem parametrów, konwersją danych, wczytaniem i zapisem danych oraz wyświetleniem interaktywnego graficznego interfejsu użytkownika. System przyjmuje skrypty tekstowe zawierające

sekwencje komend wykonywanych po kolei. Listing 3 przedstawia przykładowy skrypt pozwalający skonfigurować i wykonać algorytm detekcji stanów i klas cykli.

```
1: load_params
2: convert
3: print_all_constants
4: detection_execute
5: save_dump
6: plot_view
```

Listing 3. Skrypt wykonujący algorytm detekcji stanów energetycznych.

W pierwszej instrukcji system wczytuje parametry algorytmu z pliku konfiguracyjnego pod domyślną ścieżką. Następnie polecenie *convert* wczytuje próbki z pliku tekstowego i przetwarza je na postać binarną oraz do każdej próbki dodaje znacznik czasowy. Polecenie *print_all_constants* jest dodane w celu weryfikacji poprawności parametrów algorytmu. W czwartym kroku wywoływany jest właściwy algorytm. *Save_dump* zapisuje wyniki działania algorytmu do pliku, a *plot_view* otwiera interfejs graficzny wyświetlający wykres sygnału wraz z adnotacjami stanów i klas cykli nałożonych na odpowiednie fragmenty wykresu. Tabela 9 przedstawia kompletną listę komend systemu wraz z ich opisem.

Format komendy	Opis
configuration_files *argFileType argFilePath*	Polecenie zapamiętuje listy plików konfiguracyjnych. Pierwszym argumentem jest typ pliku konfiguracyjnego. Dostępne wartości są następujące: „params_file” (plik z parametrami algorytmów), „input_file” (plik z wartościami próbek sygnału w formacie tekstowym), „input_binary_file” (plik z wartościami próbek w formacie binarnym), „dump_file” (plik, do którego zapisywany lub z którego ładowany jest obraz pamięci systemu). Drugim argumentem jest ścieżka do pliku.
load_params	Polecenie ładuje do pamięci parametry algorytmów z pliku konfiguracyjnego algorytmu „params_file”.
save_params	Polecenie zapisuje do pliku typu „params_file” parametry algorytmów.
set_param *argParameterName argValue*	Polecenie ustawia wartość parametru konfiguracyjnego algorytmu *argParameterName* na wartość *argValue.*
convert	Polecenie wykonuje konwersję wartości próbek z pliku tekstowego „input_file” do postaci szeregu czasowego próbek.

	Każdy rekord zawiera znacznik czasowy oraz binarną postać wartości próbek. Szereg czasowy zapisywany jest do dwóch plików binarnych „input_binary_file", które można automatycznie mapować do pamięci programu.
load_binary	Polecenie mapuje pliki binarne reprezentujące szereg czasowy do pamięci programu.
print_all_constants	Wypisuje na standardowe wyjście wszystkie ustawienia i konfiguracje zarówno programu jak i algorytmów w formacie „nazwa parametru = wartość parametru".
detection_execute	Polecenie wykonuje wszystkie kroki algorytmu detekcji stanów oraz klas cykli. Wyniki w postaci binarnej można zapisać do pliku za pomocą komendy save_dump.
save_dump	Zapisuje obecny stan pamięci systemu (wczytane parametry, konfiguracja, wyniki działania algorytmów, itp.) do pliku zrzutu „dump_file".
load_dump	Polecenie wczytuje z pliku „dump_file" stan pamięci systemu oraz zastępuje obecny stan pamięci wczytanym stanem.
raw_view	Polecenie otwiera okno interfejsu graficznego bez dodatkowych adnotacji umożliwiając przeglądanie sygnału wejściowego algorytmu.
plot_view	Polecenie otwiera okno interaktywnego interfejsu graficznego pozwalając na przeglądanie wyników działania algorytmu w postaci adnotacji i graficznych oznaczeń fragmentów wykresu sygnału numerami klas cykli oraz informacjami o statystykach danego wystąpienia klasy cykli
graph_raw_view	Polecenie generuje plik pdf zawierający grafikę wektorową przedstawiającą graf stanów przed operacją sklejania stanów, a także wyświetla graf w osobnym oknie
graph_view	Polecenie generuje plik pdf zawierający grafikę wektorową przedstawiającą graf stanów po operacji sklejania stanów, a także wyświetla graf w osobnym oknie
print_cc *[all]*	Polecenie wypisuje na standardowe wyjście listę wszystkich klasów cykli wraz z ich statystykami. Jeśli flaga *all* jest podana

	jako argument, system wypisze także każde wystąpienie klasy cykli.
print_ci *[CC=argCCNumber]* *{[CI=argCINumberN]}**	Polecenie wypisuje na standardowe wyjście listę wszystkich wystąpień klas cykli zdefiniowanych przez opcjonalne argumenty CC oraz CI. Jeśli tylko argument CC jest podany, polecenie wypisze wszystkie wystąpienia klasy cyklu o numerze *argCCNumber*. Jeśli podane są argument CC oraz argument CI kilkukrotnie polecenie wypisze statystyki wystąpień o numerach zdefiniowanych przez kolejne wartości *argCINumberN* klasy cyklu o numerze *argCCNumber*.
export_ci *DEST=obj|file|<u>std</u>* CC=*argCCNumber* *{[CI=argCINumberN]}** *[FILE=argFileName]*	Polecenie generuje informacje w formacie daty i czasu początku i końca każdego wystąpienia klasy cyklu z listy zdefiniowanej przez argumenty CC i CI. Dane zapisywane są w formacie tekstowym CSV. Argument DEST determinuje czy wynik polecenia jest wypisywany na standardowe wyjście (*std*- wartość domyślna) czy do pliku (wartość *file*). Jeśli wybrany jest plik, to pole argument *FILE* także musi być podany. Wynik zapisywany jest do pliku o ścieżce podanej jako *argFileName*. Jeśli polecenie zostało wydane z DEST=*obj*, wygenerowane informacje zostaną włączone w stan pamięci systemu i mogą być wykorzystane jako dane wejściowe dla kolejnych algorytmów lub kroków skryptu. Lista instancji klasy cykli o numerze *argCCNumber* podawana jest analogicznie jak w przypadku polecenia *print_ci*.
import_ci *argFileName*	Polecenie wczytuje z pliku o ścieżce *argFileName* informacje o datach i czasach początków i końców instancji klasy cykli do stanu pamięci systemu.
correlation_execute *argFileName*	Polecenie wczytuje logi tekstowe z pliku/plików o ścieżce *argFileName*. Następnie konwertuje je do formatu worków słów i rekordów. Polecenie wykonuje algorytm korelacji listy wystąpień klasy cykli z logami tekstowymi. Lista wystąpień korelacji klasy cykli jest pobierana ze stanu pamięci i powinna

	być wcześniej wygenerowana za pomocą polecenia *export_ci* lub wczytana *import_ci.*
print_correlated_logs *DEST=file|std* [D=*argDNumber]* *{[S=argSNumberN]}** *[FILE=argFileName]*	Polecenie wypisuje wybrany fragment wyniku działania algorytmu korelacji. Wynik polecenia jest wypisywany w zależności od wartości argumentu DEST na standardowe wyjście (std) lub do pliku (file). W przypadku zapisu do pliku, ścieżka do pliku zdefiniowana jest przez argument FILE. Wypisywany fragment jest zależny od numeru szeregu sekwencji *argGNumber* oraz numerów sekwencji *argSNumberN*. Jeśli żaden z argumentów *D* i *S* nie jest skonfigurowany, polecenie wypisze wszystkie skorelowane rekordy logów.
Print_correlated_offsets	Polecenie wypisuje zakres wartości przesunięcia w czasie, generujący najlepsze dopasowanie między zdarzeniami a logami.

Tabela 9. Komendy realizowane przez *StateEventAnalyzer*.

Nazwa parametru algorytmu	Opis
n_segment	Liczność segmentu w próbkach
max_level_recurention	Maksymalna głębokość rekurencji
avg_eps	Limit podobieństwa dla różnicy wartości średnich
min_eps	Limit podobieństwa dla różnicy wartości minimów
max_eps	Limit podobieństwa dla różnicy wartości maksimów
deviation_eps	Limit podobieństwa dla różnicy wartości odchyleń standardowych
integral_eps	Limit podobieństwa dla różnicy wartości całek oznaczonych
rms_eps	Limit podobieństwa dla różnicy wartości rms
cor_wages_word	Wartość wagi dla słowa typu l-word
cor_wages_pid	Wartość wagi dla słowa typu pid

cor_wages_source	Wartość wagi dla słowa typu source
cor_wages_non_word	Wartość wagi dla słowa typu d-word
cor_time_eps	Wartość czasu w ms, która określa wielkość odstępu czasowego dzielącego zbiór podobnych worków słów na sekwencje.
cor_neighbour_eps	Limit podobieństwa dla metryki porównującej
cor_time_tolerance	Wartość czasu w ms oznaczająca tolerancję podczas łączenia zbiorów wynikowych RES(k).

Tabela 10. Parametry konfiguracyjne systemu

```
> load_params
['load_params']
< Order received:  ['load_params']
{'n_segment': 225.0, 'max_level_recurention': 60.0, 'avg_eps': 0.4, 'min_eps': 0.5, 'max_eps': 4.5, 'deviation_eps'
: 0.35, 'integral_eps': 0.25, 'cor_wages_word': '1.0', 'cor_wages_pid': '2.0', 'cor_wages_source': '3.0', 'cor_wage
s_non_word': '0.0', 'cor_group_eps': '0.3', 'cor_seq_distance_eps': '0.3', 'cor_time_eps': 'minutes:2;miliseconds:8
000', 'cor_neighbour_eps': '0.3', 'cor_time_tolerance': 'seconds:1', 'cor_assumed_year': '2020'}
< Order executed:  ['load_params']
```

Rysunek 52. Wywołanie komendy załadowania parametrów algorytmów

Na rysunku 52 można zaobserwować wynik działania polecenia *load_params*. W wyniku działania wczytane zostają parametry konfiguracyjne algorytmu. Wczytane wartości wypisywane są na standardowe wyjście w formacie klucz: wartość.

```
> load_dump
['load_dump']
< Order received:  ['load_dump']
<converter.ResourceManager object at 0x7ffab91612e8>
< Order executed:  ['load_dump']
```

Rysunek 53. Wynik działania komendy *load_dump*

Na rysunku 53 przedstawiono działanie komendy *load_dump*. Identyczny schemat zachowania systemu dotyczy każdej innej komendy, która nie przyjmuje argumentów i jej celem nie jest wygenerowanie wyniku tekstowego. W linii „< Order received: ….” program informuje o przyjętej komendzie. Następne linie są przeznaczone na logowanie wykonanych czynności. Wykonanie zwieńcza komunikat o sukcesie wykonania: „”< Order executed: ….”.

```
> print_all_constants
['print_all_constants']
< Order received:  ['print_all_constants']
configuration_files input_file input.txt
configuration_files input_binary_file input.bin
configuration_files dump_file dump.bin
configuration_files logic_data_file logic.dat
configuration_files params_file params.ini
configuration_files conv_interval 2.0
set_params n_segment 225.0
set_params max_level_recurention 60.0
set_params avg_eps 0.4
set_params min_eps 0.5
set_params max_eps 4.5
set_params deviation_eps 0.35
set_params integral_eps 0.25
< Order executed:  ['print_all_constants']
```

Rysunek 54. Wynik działania komendy *print_all_constants*

Komenda *print_all_constants* wypisuje konfiguracje systemu oraz algorytmów w formacie skryptu- sekwencji poleceń konfiguracyjnych. Wynik komendy można wykorzystać np. w innej instancji programu podając go na standardowe wejście jako skrypt inicjalizujący.

```
> print_cc
['print_cc']
< Order received:  ['print_cc']
{CC=0}[num=1 , time_lasting(ms)=16538]
{CC=1}[num=1 , time_lasting(ms)=10798]
{CC=2}[num=2056 , time_lasting(ms)=392638]
{CC=3}[num=1 , time_lasting(ms)=1542]
{CC=4}[num=1 , time_lasting(ms)=1092]
{CC=5}[num=9 , time_lasting(ms)=15046]
{CC=6}[num=8 , time_lasting(ms)=8534]
{CC=7}[num=1 , time_lasting(ms)=192]
{CC=8}[num=1 , time_lasting(ms)=1684]
{CC=9}[num=20 , time_lasting(ms)=3852]
{CC=10}[num=1 , time_lasting(ms)=2528]
{CC=11}[num=2 , time_lasting(ms)=3316]
{CC=12}[num=1 , time_lasting(ms)=896]
{CC=13}[num=1 , time_lasting(ms)=4720]

< Order executed:  ['print_cc']
```

Rysunek 55. Wynik działania komendy *print_cc* bez dodatkowych argumentów

Na rysunku 55 zaprezentowany jest wynik działania komendy *print_cc* bez flagi *all*. Komenda wypisuje wszystkie klasy cykli bez ich instancji (CI). Każda klasa cykli identyfikowana jest numerem, a także posiada 2 sumaryczne statystyki: 1) *num*- liczba instancji/wystąpień danej klasy cykli, 2) *time_lasting*- zsumowany czas trwania/występowania wszystkich instancji klas cykli w milisekundach.

Znając wszystkie klasy cykli oraz liczności instancji każdej z nich program pozwala na wyświetlenie bardziej szczegółowych danych.

```
> print_ci CC=11
['print_ci', 'CC=11']
< Order received:  ['print_ci', 'CC=11']
{CC=11}[num=2 , time_lasting(ms)=3316]
  [CC=11, CI=0, CI_ID=1958]
    timings(ms): [B=1606215631600.0, E=1606215633340.0, T=1740]
    timestamps(ms): [B=2020-11-24 11:00:31.600000, E=2020-11-24 11:00:33.340000]
    stats: [min=1.1051775, max=2.121762, avg=1.6961291920303887, dev=0.08056208 , int= 2511.553639769554]
  [CC=11, CI=1, CI_ID=2099]
    timings(ms): [B=1606220367736.0, E=1606220369312.0, T=1576]
    timestamps(ms): [B=2020-11-24 12:19:27.736000, E=2020-11-24 12:19:29.312000]
    stats: [min=1.5743841, max=2.0777602, avg=1.6879027447319757, dev=0.06767927 , int= 2251.4853048324585]

< Order executed:  ['print_ci', 'CC=11']
```

Rysunek 56. Wynik działania komendy *print_cc* z parametrem CC=11

Rysunek 56 przedstawia wynik komendy *print_cc* dla klasy cyklu o numerze 11. Komenda wypisała szczegółowe dane dla każdego wystąpienia zadanej klasy cyklu. Dla każdej instancji polecenie podaje kolejno numer klasy cyklu (CC), numer wystąpienia w ramach klasy cyklu (CI) oraz numer identyfikacyjny wystąpienia cyklu/instancji w ramach całego szeregu czasowego (CI_ID). Każde wystąpienie zawiera trzy pola: 1) timings, 2) timestamps, 3) stats. Pole 1) zawiera informacje o czasie początku (B) i końca (E) trwania instancji w formacie znacznika czasowego UNIX timestamp. Parametr T określa czas trwania instancji w sekundach. Pole 2) prezentuje te same dane co parametry B i E pola 1) lecz w formacie daty i czasu UTC z dokładnością do mikrosekund. Pole 3) zawiera wartości statystyk obliczonych dla próbek tworzących dany cykl. W prezentowanym wypadku są to kolejno wartości: minimum, maksimum, średnia arytmetyczna, odchylenie standardowe, całka oznaczona.

Polecenie *print_cc* pozwala również wypisać dane związane tylko z wybranymi instancjami zadanej klasy cyklu, co ukazane jest na rysunku 57.

```
> print_ci CC=2 CI=5
['print_ci', 'CC=2', 'CI=5']
< Order received:  ['print_ci', 'CC=2', 'CI=5']
{CC=2}[num=2056 , time_lasting(ms)=392638]
  [CC=2, CI=5, CI_ID=20]
    timings(ms): [B=1606149766930.0, E=1606149767136.0, T=206]
    timestamps(ms): [B=2020-11-23 16:42:46.930000, E=2020-11-23 16:42:47.136000]
    stats: [min=0.10869231, max=8.734462, avg=1.2480132947317684, dev=2.2242804 , int= 672.7987169502303]

< Order executed:  ['print_ci', 'CC=2', 'CI=5']
```

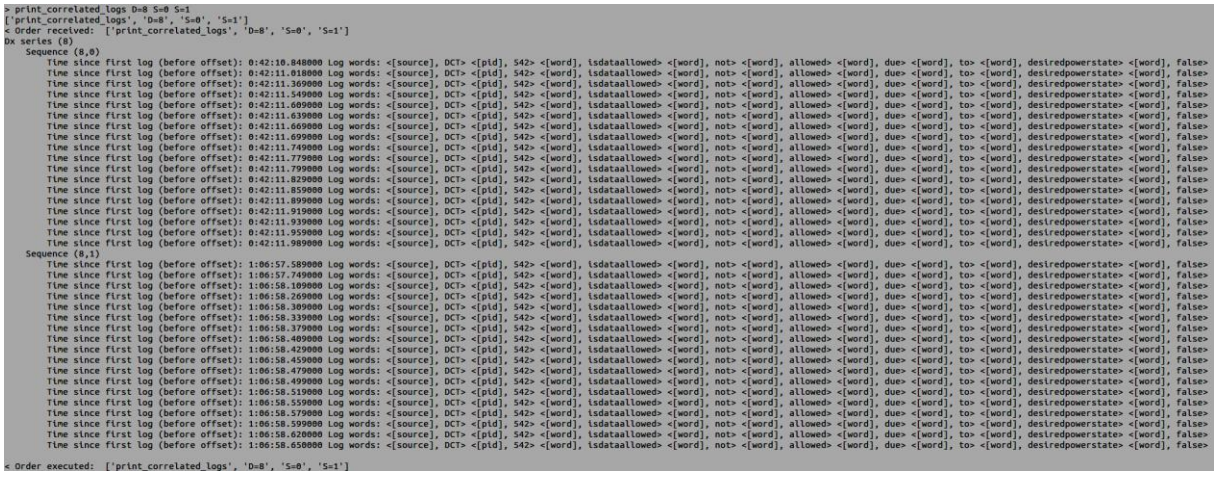
Rysunek 57. Wynik działania komendy *print_cc* przy CC=2 i CI=5

Operator przygotowując dane dla algorytmu korelacji może załadować listę par znaczników czasu początków i końców zdarzeń/stanów (które mogły zostać wykryte jako instancje klasy cykli) za pomocą polecenia *import_ci*. Opisywany scenariusz jest przedstawiony na rysunku 58. Po wczytaniu par znaczników czasu polecenie wypisuje w każdym wierszu kolejne pary.

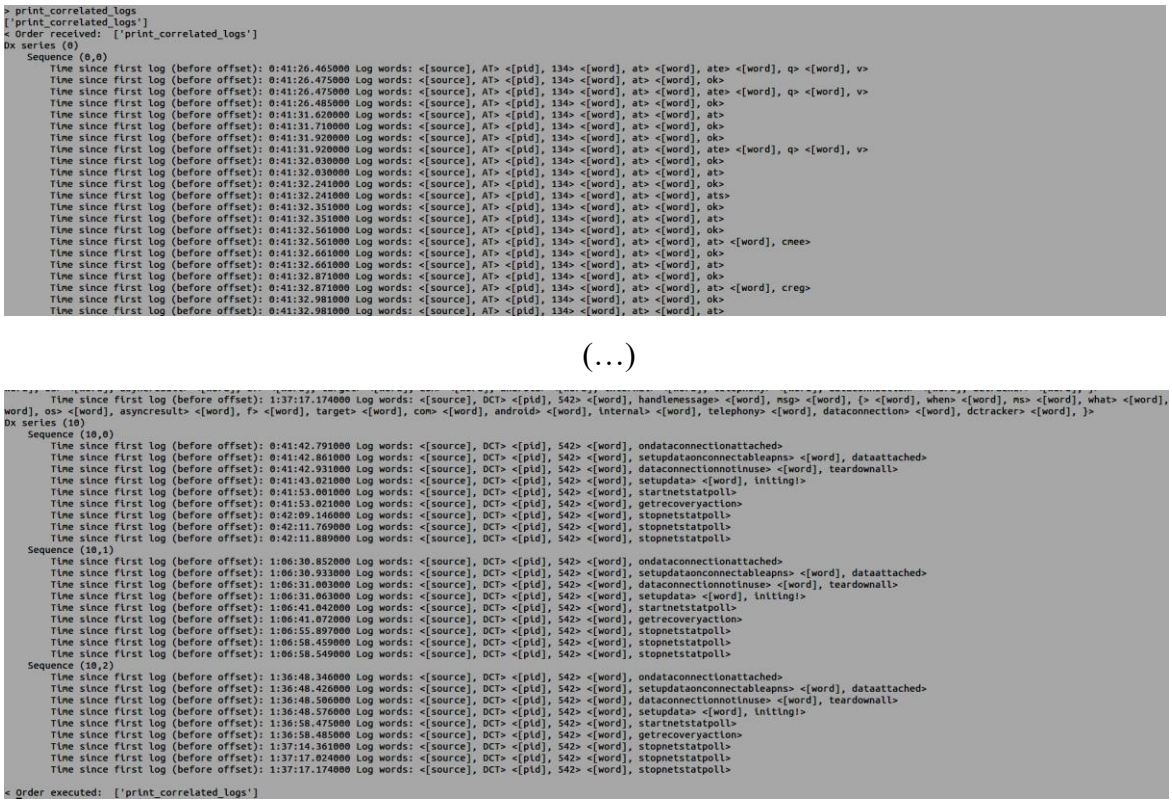
```
> import_ci correlation_input_orig.txt
['import_ci', 'correlation_input_orig.txt']
< Order received:  ['import_ci', 'correlation_input_orig.txt']
2019-10-01 13:43:10.100000;2019-10-01 13:45:10.100000
2019-10-01 14:07:20.100000;2019-10-01 14:09:20.100000
2019-10-01 14:36:40.100000;2019-10-01 14:40:00.100000

< Order executed:  ['import_ci', 'correlation_input_orig.txt']
```

Rysunek 58. Wynik działania komendy *import_ci*

Polecenie *print_correlated_logs* generuje listę sekwencji składających się z przetworzonych rekordów logów do worków słów. Gdy komenda nie zawiera argumentów precyzujących numer szeregu oraz listę numerów sekwencji, program wypisuje na wyjście lub do zadanego pliku kolejne rekordu podzielone na sekwencje i zgrupowane w ramach szeregów D_x. Przykład takiego działania polecenia jest pokazany na rysunku 60. Rysunek 59 przedstawia przykład wyniku polecenia *print_correlated_logs* z parametrem definiującym indeks szeregu sekwencji (D=8) oraz listę sekwencji, których rekordy powinny być wypisane (S=0 i S=1). Każdy wypisany element sekwencji zawiera czas od pierwszego rekordu oraz zbiór sklasyfikowanych worków słów wypisanych w kolejności zgodnej z ich występowaniem w rekordzie logu. Dodatkowe dane można uzyskać korzystając z obrazu binarnego (*load_dump, save_dump*) modelu. Model umożliwia uzyskanie danych dla każdego rekordu takich jak: 1) numer linii z oryginalnego pliku z logami, 2) wartość metryki *silimanity* oraz 3) liczności sekwencji i szeregu.



Rysunek 59. Wypisanie wszystkich rekordów logów należących do szeregu 8 i sekwencji 0 oraz 1



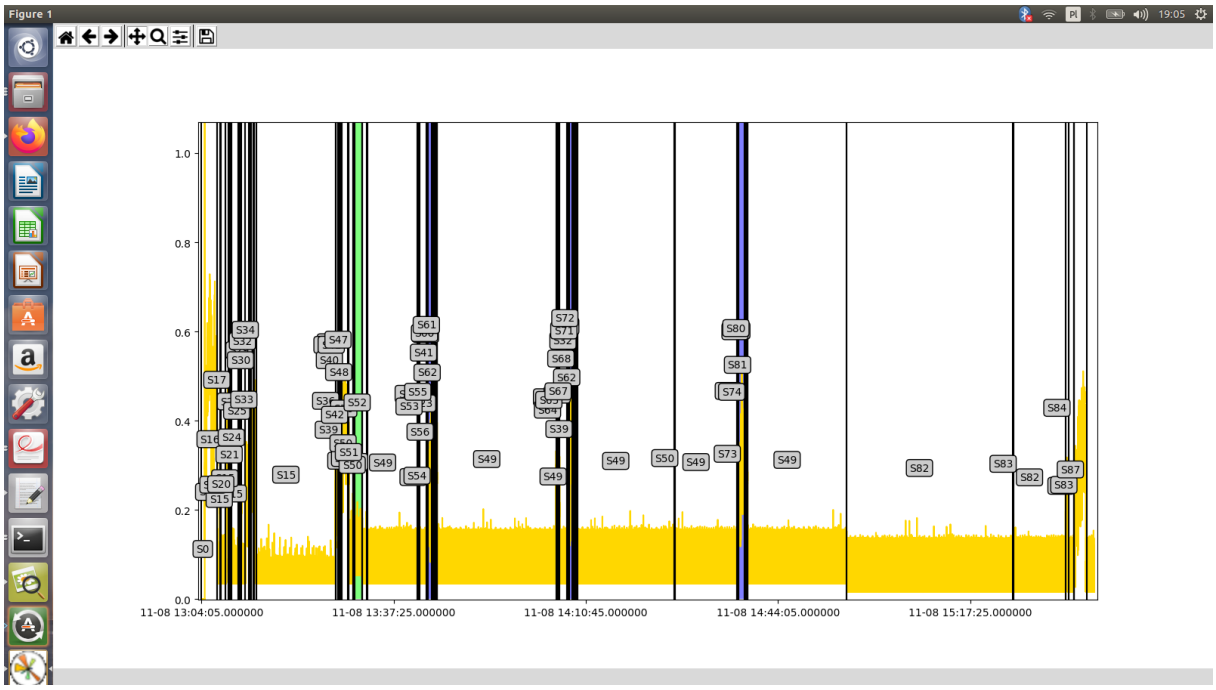
(…)

Rysunek 60. Lista skorelowanych logów

Rysunek 61. Zakres przesunięć znaczników czasu generujących najlepsze dopasowanie dla badanego systemu

Rysunek 61 przedstawia wynik działania polecenia *print_correlated_offsets.* Komenda wypisuje parę przesunięć czasowych w formacie dzielącym czas na godziny, minuty, sekundy i części tysięczne sekund. Para ta określa minimalne oraz maksymalne przesunięcie czasowe gwarantujące najlepsze dopasowanie (maksymalizujące liczbę dopasowanych szeregów sekwencji) do czasów występowania i trwania obiektów OI.

Pierwszy zaimplementowany algorytm dokonuje detekcji stanów np. energetycznych urządzenia na podstawie danych zebranych podczas pomiaru prądu. Informacje te prezentowane są na interaktywnym wykresie w postaci oznaczenia zakresów fragmentów wykresu odpowiadających wykrytym stanom odpowiednimi adnotacjami. Oś pionowa wyskalowana jest w amperach, a pozioma opisuje dokładny czas danej próbki. W prawym dolnym rogu wyświetlone są obecne współrzędne kursora myszy w amperach i jednostkach daty/czasu. Interfejs wraz z przykładowym wykresem umieszczony jest na rysunku 62.

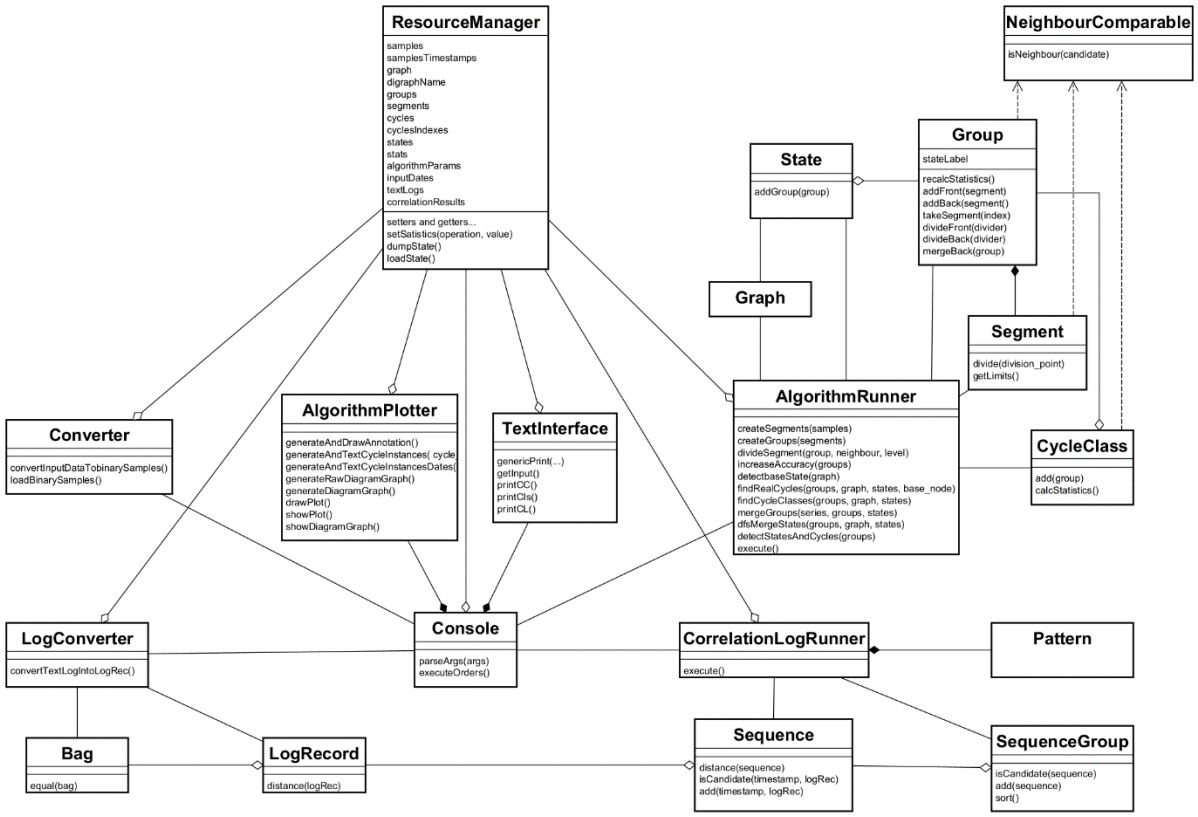


Rysunek 62. Okno interfejsu graficznego aplikacji do analizy pobieranej energii przez system

Górny pasek aplikacji służy do nawigowania wykresu oraz dostosowywania widoku do rozmiarów monitora. W kolejności od lewej znajdują się przyciski: a) powrotu do początkowego widoku, b) zmiany widoku na poprzedni, c) zmiany widoku na kolejny, d) trybu manualnego przesuwania wykresu w celu uzyskania nowego widoku, e) przybliżania i oddalania widoku w celu zwiększenia liczby szczegółów lub zmniejszenia, f) konfiguracji rozmiaru okna wykresu oraz e) zapisu obecnego widoku do pliku. Po wybraniu przycisk e) za pomocą kursora myszy można zaznaczyć obszar, dla którego chcielibyśmy wygenerować widok, a następnie program wygeneruje fragment wykresu.

8.2.2 Architektura oprogramowania systemu StateEventAnalyzer

Program został przygotowany w języku *Python* z wykorzystaniem bibliotek: a) *matplotlib*- odpowiedzialna za nawigowanie i rysowanie wykresów oraz b) *numpy*- zestaw funkcji ułatwiających obliczenia statystyczne oraz obsługę pamięci dla dużych danych. Na rysunku 63 przedstawiono diagram klas obrazujący architekturę systemu.



Rysunek 63. Architektura systemu StateEventAnalyzer

Architektura oprogramowania jest minimalistyczna i stworzona przede wszystkim w celach: 1) zaprezentowania i przetestowania opisywanych algorytmów, 2) umożliwienia zapisywanie kolejnych kroków przetwarzania oraz ich wczytywanie i wznawianie, a także 3) łatwego dodawania nowych algorytmów i umożliwiania łączenia ich w kaskady poprzez wykorzystywanie wyników działania jednego algorytmu jako danych wejściowych drugiego. Architektura oprogramowania opiera się na zmodyfikowanym obiektowym wzorcu projektowym MVC (model-widok-kontroler). Klasa interfejsu modelu odpowiedzialna za przechowywanie stanu systemu zaimplementowana została jako klasa *ResourceManger*. Obiekt tej klasy przechowuje wszystkie kolekcje i obiekty zawierające wyniki działania algorytmów, a także konfiguracje systemu i algorytmów oraz przetworzone dane wejściowe. Funkcję kontrolera pełni obiekt klasy interfejsu *Console*, która odpowiedzialna jest za przetwarzanie komend oraz ich wykonywanie poprzez odpowiednie tworzenie i wykorzystywanie innych klas systemu. System posiada dwa interfejsy użytkownika co w architekturze jest odwzorowane w postaci dwóch klas widoku: *AlgorithmPlotter* i *OutputPrinter*. *AlgorithmPlotter* jest klasą generującą elementy niezbędne dla interfejsu graficznego. Utworzony obiekt klasy ma możliwość wygenerować diagram stanów

na podstawie zagregowanych obiektów stanu w obiekcie klasy *ResourceManager*, a także wyświetlić diagram na ekranie i zapisać go do pliku. Ponadto obiekt jest w stanie uruchomić graficzny interfejs użytkownika opisany w paragrafie powyżej realizując zadania przeliczania zakresów wyświetlanego wykresu czy też rysując adnotacje stanów i statystyki odczytane z klasy modelu. Klasa *TextInterface* jest odpowiedzialna za wykonywanie zadań wejścia/wyjścia związanych z danymi tekstowymi. Obiekt tej klasy wykorzystywany jest w celu pozyskiwania komend od użytkownika. Klasa ponadto umożliwia wypisywanie w formie tekstowej do pliku lub na standardowe wyjście wyników działania algorytmów, poleceń oraz komunikatów o błędach itp.

Klasa *AlgorithmRunner* jest jedną z klas tworzących kontroler programu. Obiekt klasy jest tworzony jako rezultat działania komendy *detection_execute* przez obiekt klasy *Console*. Poprzez metodę *execute*() uruchamiana jest sekwencja czynności realizujących algorytm opisany w rozdziale 3.1. W wyniku działania metody, która wywołuje inne metody realizujące poszczególne etapy algorytmu, w obiekcie klasy *ResourceManager*, zagregowane zostają obiekty segmentów, klas cykli, stanów, a także grafu. Klasy *CycleClass*, *Group* oraz *Segment* są w relacji dziedziczenia z klasą *NeighbourComparable*, która implementuje metrykę służącą do określania podobieństwa między odpowiednimi obiektami algorytmu takimi jak segment, klasa cykli czy też grupa. W przypadku, gdyby zaszła potrzeba implementacji metryki szczególnej dla danego typu obiektu np. dla grup istnieje możliwość przeciążenia metody *isNeighbourCandidate*. Podczas tworzenia obiektu parametry algorytmu umożliwiające określenie podobieństwa są przekazywane do konstruktora klasy. Klasa *Converter* jest odpowiedzialna za przetworzenie tekstowych danych wejściowych dla algorytmu na postać binarną, a także dodania ich do stanu programu reprezentowanego przez obiekt klasy *ResourceManager*. Klasa ponadto udostępnia możliwość zapisywania i odczytywania przekonwertowanych na format binarny danych o próbkach szeregu czasowego.

Klasa *CorrelationLogRunner* umożliwia wykonanie algorytmu korelacji logów tekstowych z sekwencją zdarzeń. Obiekt tej klasy jest tworzony przez obiekt klasy *Console* w odpowiedzi na polecenie *correlation_execute*. Obiekt za pośrednictwem metody *execute*() wykonuje algorytm korelacji z rozdziału 4. Dane wejściowe dla algorytmu przygotowywane są przez obiekt klasy *LogConverter*, który wczytuje pliki z logami tekstowymi oraz przetwarza je i przedstawia w postaci obiektów klas *Bag* i *LogRecord*. Logi zagregowane w ten sposób są dodawane do modelu systemu. Podczas wykonywania algorytmu tworzone są grupy, a także sekwencje grup, które są zapisywane w pamięci w sposób umożliwiający ich export do pliku lub wypisanie na ekranie za pośrednictwem obiektu klasy *TextInterface*.