POLITECHNIKA KOSZALIŃSKA

Wydział Elektroniki i Informatyki
Katedra Podstaw Informatyki i Zarządzania

Metoda wykrywania konfliktów zasobowych w aplikacjach wielowątkowych

Rozprawa doktorska

Autor:
mgr inż. Damian GIEBAS

Promotor:
dr hab. inż. Grzegorz BOCEWICZ prof. PK

Promotor pomocniczy:
dr inż. Rafał WOJSZCZYK

Koszalin, 11 kwietnia 2022

Oświadczenie

Ja, Damian GIEBAS, zaświadczam, że niniejsza dysertacja o tytule, „Metoda wykrywania konfliktów zasobowych w aplikacjach wielowątkowych" i praca w niej zawarta jest moją własnością. Potwierdzam, że:

- Ta praca została wykonana w całości lub głównie w trakcie ubiegania się o stopień naukowy na Politechnice Koszalińskiej.
- W przypadku, gdy jakakolwiek część tej pracy została wcześniej złożona w celu uzyskania stopnia naukowego lub jakichkolwiek innych kwalifikacji na tej uczelni lub jakiejkolwiek innej instytucji, zostało to wyraźnie stwierdzone.
- Wszędzie tam, gdzie odnoszę się do opublikowanych prac innych, zawsze jest to wyraźnie oznaczone.
- Wszystkie cytaty pochodzące z innych prac, są wyraźnie oznaczone i spisane w bibliografii. Wyjątkiem jest przedstawiona w niniejszej pracy teza, która jest całkowicie moją własną pracą.
- Uznałem wszystkie główne źródła pomocy.
- Tam, gdzie teza opiera się na pracy wykonanej przeze mnie wspólnie z innymi, wyjaśniłem dokładnie to, co zostało zrobione przez innych i co sam wniosłem.

Podpis:

_____

Data:

_____

Streszczenie

Metoda wykrywania konfliktów zasobowych w aplikacjach wielowątkowych

mgr inż. Damian GIEBAS

Prawie każdy z powszechnie dzisiaj stosowanych typów procesorów wspiera technologię wielowątkowości. Dzięki tej technologii programiści otrzymali możliwość równoległego wykonywania zadań w ramach jednej aplikacji. Aby do tego doszło należało jednak rozszerzyć istniejące języki programowania o odpowiednie biblioteki, dzięki którym programista może wykorzystać wspomnianą technologię. Skutkiem tego wiele języków, w tym język C, umożliwia tworzenie aplikacji wielowątkowych, pomimo że języki te nie zaprojektowano w tym celu. Efektem takiego stanu rzeczy jest pojawienie się nowej kategorii błędów. Polega ona na możliwości tworzenia struktur w kodzie programu, których skutkiem są wzajemne (zwykle nieprzewidziane) interakcje między jednostkami logicznymi nazywanymi wątkami. Struktury te powodują zwiększenie kosztów wytwarzania oprogramowania, a to oznacza, że tworzenie aplikacji wielowątkowych staje się w wielu przypadkach nieopłacalne. W takiej sytuacji poszukiwana jest odpowiedź na pytanie: czy zadana aplikacja wielowątkowa napisana w języku C z użyciem biblioteki *pthread* jest wolna od błędów strukturalnych prowadzących do konfliktów zasobowych w szczególności: szkodliwej rywalizacji, zakleszczenia, naruszenia niepodzielności i naruszenia porządku?

W celu lokalizowania błędów prowadzących do konfliktów zasobowych opracowano wiele metod i narzędzi. Wiele z tych rozwiązań wspiera lokalizowanie od jednego do trzech typów konfliktów zasobowych, a więc ich użycie nie pozwala odpowiedzieć na wcześniej postawione pytanie. Powodem tego są m. in. pierwotne założenia istniejących rozwiązań (np. wykorzystanie prostych modelów), które powodują, że nie są one na tyle elastyczne, aby można było wykorzystać je w celu lokalizowania czterech typów konfliktów zasobowych w aplikacjach wielowątkowych pisanych za pomocą języka C z użyciem biblioteki *pthread*. Innym istotnym czynnikiem, który nie spełniało żadne inne rozwiązanie jest możliwość jego wykorzystania on-line tzn. czas potrzebny na lokalizowanie błędów wynosił nawet kilka godzin. Oznacza to brak rozwiązań wspierających programistów języka C w procesie wytwarzania kodu źródłowego aplikacji wielowątkowych, w sposób umożliwiający szybką analizę kodu źródłowego, w celu zlokalizowania ww. błędów strukturalnych prowadzących do konfliktów zasobowych.

Analiza literatury tematu wykazała, że aby osiągnąć cel jakim jest opracowanie metody umożliwiającej lokalizowanie konfliktów zasobowych w trybie on-line należy wykorzystać metody analizy statycznej. Aby osiągnąć ten cel należało opracować model umożliwiający wyznaczanie warunków pozwalających na wykrywania błędów prowadzących do konfliktów zasobowych pozwalających na obniżenie kosztów wykorzystania metod analizy statycznej. Opracowany model kodu źródłowego aplikacji wielowątkowej zorientowany na lokalizowanie czterech typów konfliktów zasobowych zakłada wykorzystanie języka C i biblioteki *pthread* w trybie on-line. Model ten pozwala na analizę kodu źródłowego w ujęciu strukturalno-czasowym.

W dysertacji przedstawiono problem lokalizacji błędów w aplikacjach wielowątkowych. Przegląd literatury tematu pozwolił doprecyzować założenia, składające

się na podstawy modelu kodu źródłowego i warunki wykrywania błędów. Umożliwiło to opracowanie metody, jej zaimplementowanie w prototypowym narzędziu i przeprowadzenie eksperymentów w celu weryfikacji opracowanej metody.

Abstract

Method for detecting resource conflicts in multithreaded applications

Damian GIEBAS, M. Sc. Eng.

Almost each and every one of contemporarily utilised processors supports the technology of concurrency. Thanks to said technology, programmers have attained the capability of simultaneous task execution in the scope of one application. In order to enable this technology, it was required to extend the existent programming languages with proper libraries, thanks to which programmers may use the aforementioned technology. As a result, a substantial number of languages, with C included, support the creation of multithreaded applications, even despite the fact that these languages were not designed to do so. The side effect of introducing concurrency is the emergence of a new kind of errors. The errors stem from the fact that a programmer may create structures resulting in unpredicted interactions between logical units called threads. These structures cause the rise of costs involved in software development, therefore rendering the creation of multithreaded applications unprofitable. In such a situation, the answer to the following question is sought: Is given multithreaded application written in C with pthread library free from structural errors leading to resource conflicts, such as race conditions, deadlocks, atomicity violations and order violations.

Numerous methods and tools have been developed so that the localization of errors leading to resource conflicts is more attainable. A significant number of these solutions supports the localization of one to three types of resource conflicts, therefore the usage of these tools does not allow one to answer the hereabove question. The original assumptions such as the usage of simple models are among the main reasons for the fact that these tools are not agile enough to be utilised to localize four types of resource conflicts in multithreaded applications written in C with pthread library. Another important factor is the inability of any of the tools to be used online because of the time needed for the localization of the errors extending to several hours. It denotes the lack of solutions supporting C programmers in the process of development of multithreaded applications in a way enabling swift source code analysis targeted at localizing the aforementioned structural errors leading to resource conflicts.

The analysis of literature in the subject matter showed that in order to achieve the objective of creation of a method capable of localization of resource conflict in online mode, one is required to use the method of static analysis. For that end, it was essential to design a model enabling the designation of conditions not only allowing to localize errors leading to resource conflicts but also decreasing the cost of the usage of static analysis method. The designed model of multithreaded application source code oriented at localization of four types of resource conflicts assumes the utilisation of C language and pthread library in online mode. Said model enables source code analysis on a structural and temporal level.

In this dissertation the problem of localization of errors in multithreaded applications is discussed. The literature review enabled the clarification of assumptions in the area of the basis of the source code model and error localization conditions. The design of the method, its implementation in a prototype solution and conducting experiments in order that the method is properly verified were thus rendered possible.

Podziękowania

Pragnę złożyć serdeczne podziękowania Panu dr hab. inż. Grzegorzowi Bocewiczowi i Panu dr inż. Rafałowi Wojszczykowi za nieocenioną pomoc i wsparcie udzielone w trakcie całych moich studiów doktoranckich. Bez cierpliwości i wyrozumiałości, którą zostałem obdarowany, ta praca nigdy by nie powstała.

Szczególne podziękowania chciałbym także złożyć Panu prof. dr hab. inż. Zbigniewowi Banaszakowi za pomoc w redagowaniu pracy oraz motywację niezbędną do jej ukończenia.

Chciałby także złożyć szczerą wdzięczność wszystkim pracownikom Katedry Podstaw Informatyki i Zarządzania, którzy pomagali mi podnieść moje umiejętności prezentacji.

Szczególne wyrazy wdzięczności dla mojej Narzeczonej za nieustanne wspieranie oraz motywowanie mnie do działania.

Niniejszą pracę pragnę zadedykować mojemu tacie Zbigniewowi oraz nieobecnym już mamie Jadwidze i dziadkowi Mieczysławowi.

Spis treści

Spis rysunków

Spis tablic

Wykaz skrótów

BWD	**b**ack**w**ar**d** - skrót określający grupę krawędzi relacji niepodzielności wstecz
AST	**A**bstract **S**yntax **T**ree
AIX	**A**dvanced **I**nteractive **E**xecutive
BSD	**B**erkeley **S**oftware **D**istribution
C	skrótowo język C
C++	skrótowo język C++
C#	skrótowo język C# dostępny na platformie .NET
CLR	**C**ommon **L**anguage **R**untime - środowisko uruchomieniowe platformy .NET
CSP	**C**ommunicating **S**equential **P**rocesses
CI/CD	**C**ontinuous **I**ntegration / **C**ontinuous **D**elivery lub **C**ontinuous **D**eployment
DRY	**D**on't **R**epead **Y**ourself
ECO	**E**wha **CO**ncurrency Detector
FWD	**f**or**w**ar**d** - skrót określający grupę krawędzi relacji niepodzielności wprzód
GNU	**G**NU's **N**ot **U**nix
HAVE	**H**ybrid **A**tomicity **V**iolation **E**xplorer
HP-UX	**H**ewlett **P**ackard **Unix**
IoT	**I**nternet **o**f **T**hings
ISTQB	**I**nternational **S**oftware **T**esting **Q**ualifications **B**oard
JRE	**J**ava **R**untime **E**nvironment - środowisko uruchomieniowe Javy
KISS	**K**eep **I**t **S**imple, **S**tupid
lams	dead**L**ock **A**nalysis **M**odel**s**
PMD	**P**THREAD_**M**UTEX_**D**EFAULT
PME	**P**THREAD_**M**UTEX_**E**RRORCHECK
PMN	**P**THREAD_**M**UTEX_**N**ORMAL
PMR	**P**THREAD_**M**UTEX_**R**ECURSIVE
PyPI	**Py**thon **P**ackage **I**ndex
RD-L	Races and Deadlocks - Locating; nazwa metody opisanej w roz. 5
SI	**S**ztuczna **i**nteligencja
SHB	**S**tatic **H**appens-**B**efore
SYM	**sym**metric - skrót określający grupę krawędzi w symetrycznej relacji niepodzielności
STM	**S**oftware **T**ransactional **M**emory
TDD	**T**est **D**riven **D**evelopment

Spis symboli i oznaczeń

α	moc zbioru T_P
β	liczba przedziałów czasu aplikacji P
γ	liczba zasobów współdzielonych aplikacji P
ϵ	moc zbioru O_P
κ	moc zbioru Q_P
$\lambda^{P,i}$	zbiór wszystkich ścieżek zbudowanych z operacji wątku t_i aplikacji P
$\lambda_a^{P,i}$	a-ta ścieżka wątku t_i aplikacji P
$\lambda_{a,b}^{P,i}$	b-ty elementy a-tej ścieżki wątku t_i aplikacji P
ι	moc zbioru F_P
μ	moc zbioru B_P
ξ	element zbioru {FWD, BWD, SYM}, używany przy oznaczaniu zbioru B_P^ξ
B_P	sekwencja zbiorów par reprezentujący relacje niepodzielności operacji aplikacji P
B_P^ξ	zbiór par reprezentujący jedną z relacji niepodzielności
C_P	instancja modelu kodu źródłowego aplikacji wielowątkowej P
$C\lambda^{P,i}$	zbiór wszystkich ścieżek cyklicznych wchodzących w skład zbioru $\lambda^{P,i}$
E_P	zbiór krawędzi skierowanych grafu G_P, $F_P \cup B_P$
F_P	zbiór par reprezentujących relacje między elementami zbiorów O_P, R_P, Q_P
G_P	graf operacji aplikacji P
${}_i^s G_P$	podgraf grafu G_P rozpoczynający się od s-tej blokady w i-tym wątku
O_P	zbiór operacji aplikacji P
P	indeks instancji modelu wybranej aplikacji
Q_P	zbiór blokad wykorzystywanych w aplikacji P
R_P	rodzina zasobów współdzielonych pomiędzy wątkami aplikacji P
T_P	zbiór wątków aplikacji P
U_P	sekwencja przedziałów czasu realizacji wątków zbioru T_P
V_P	zbiór wierzchołków, $O_P \cup Q_P \cup R_P$
b_m	m-ta relacja niepodzielności zbioru B_P
f_n	n-ta para (krawędź) zbioru F_P
$o_{i,j}$	i-ta operacja j-tego wątku, element zbioru O_P
q_s	s-ta blokada zbioru Q_P
r_c	c-ty zasób rodziny R_P
t_i	i-ty wątek zbioru T_P
u_b	b-ty przedział czasu sekwencji U_P
$\lhd$	znak przynależności elementu do ścieżki: $a \lhd A$ oznacza, że w ścieżce A występuje element a
$\not\lhd$	znak braku przynależności elementu do ścieżki: $a \not\lhd A$ oznacza, że w ścieżce A nie występuje element a
$\prec$	znak relacji „poprzedzania się operacji”: $o_{i,a} \prec o_{i,b}$ oznacza, że operacja $o_{i,a}$ poprzedza operację $o_{i,b}$
$\prec_i^l$	znak relacji „poprzedzania się blokad” w l-tej ścieżce i-tego wątku: $q_c \prec_i^l q_d$ oznacza, że blokada q_c poprzedza blokadę q_d w ścieżce $\lambda_l^{P,i}$

Rozdział 1

Wstęp

1.1 Geneza pracy

Poprzez język programowania należy rozumieć sztuczny formalizm, który pełni dwie funkcje. Pierwszą z nich jest dostarczenie zestawu pojęć programiście, którymi może się posługiwać w celu *wyrażania myśli* [92] tj. wyrażania algorytmów [32], natomiast drugą jest przekazywanie urządzeniom elektronicznym poleceń do wykonania [92]. Języki programowania charakteryzują się składnią, semantyką i mechanizmami abstrakcji umożliwiającymi sterowanie urządzeniami elektronicznymi. Wśród tych urządzeń znajdują się mikrokontrolery, układy scalone z grup *System-on-a-chip* i *single-board computer*, ale także komputery klasy PC, konsole do gier, superkomputery, smartfony, tablety i smartwatche. Wszystkie te odmiany urządzeń elektronicznych łączy fakt, że każde z nich do pracy wykorzystuje procesor, który pozwala równolegle wykonywać instrukcje w jednostkach logicznych zwanych wątkami. W procesorach firmy Intel wielowątkowość (nazywana też współbieżnością [92]) znana jest pod nazwą Intel Hyper-Threading [22], natomiast konkurencyjne rozwiązanie firmy AMD nosi nazwę Simultaneous Multithreading [4]. Ceny urządzeń z procesorami wielowątkowymi są coraz niższe, co wpływa bezpośrednio na wzrost liczby programów wykorzystujących wielowątkowość. Programy te pisane są najczęściej z użyciem języków programowania, które powstały jeszcze zanim upowszechniła się technologia procesorów wielowątkowych. Stosownym tego przykładem jest język C [53], który został opracowany przez Dennisa Ritchie'ego w Laboratorium Bella w latach 70-tych XX wieku. Początkowo język ten nie udostępniał programistom mechanizmów umożliwiających pisanie programów wielowątkowych. Dopiero w latach 90-tych XX wieku [94] opracowana została biblioteka *pthread* implementująca standard *POSIX threads* i pozwalająca pisać programy wielowątkowe w języku C.

Program definiowany jest jako *zestaw instrukcji wykonywanych przez komputer w celu wykonania określonego zadania* [83]. Programem jest np. system operacyjny (np. Windows, Ubuntu), sterownik do urządzenia (np. nouveau), malware, a także oprogramowanie wykorzystywane bezpośrednio przez użytkownika do wykonywania zadań. Programem wielowątkowym jest z kolei program, którego architektura pozwala na równoległe wykonywanie instrukcji programu w ramach jednego procesu. Każdy z programów wielowątkowych posiada przynajmniej dwa wątki, wliczając w to wątek główny (ang. main thread). Jako wątek należy rozumieć pewną jednostkę wykonywania zarządzaną przez jądro systemu operacyjnego [89]. Stroustrup stwierdził, że wątek to reprezentacja na poziomie systemu zadania w programie [92], innymi słowy precyzuje on, że wspomniana jednostka wykonywania jest tożsama z zadaniem. Wprowadzenie mechanizmów umożliwiających współbieżne realizowanie operacji pozwoliło zmniejszyć czas obliczeń programu [22], ale jednocześnie stało się źródłem nowej kategorii błędów programów komputerowych.

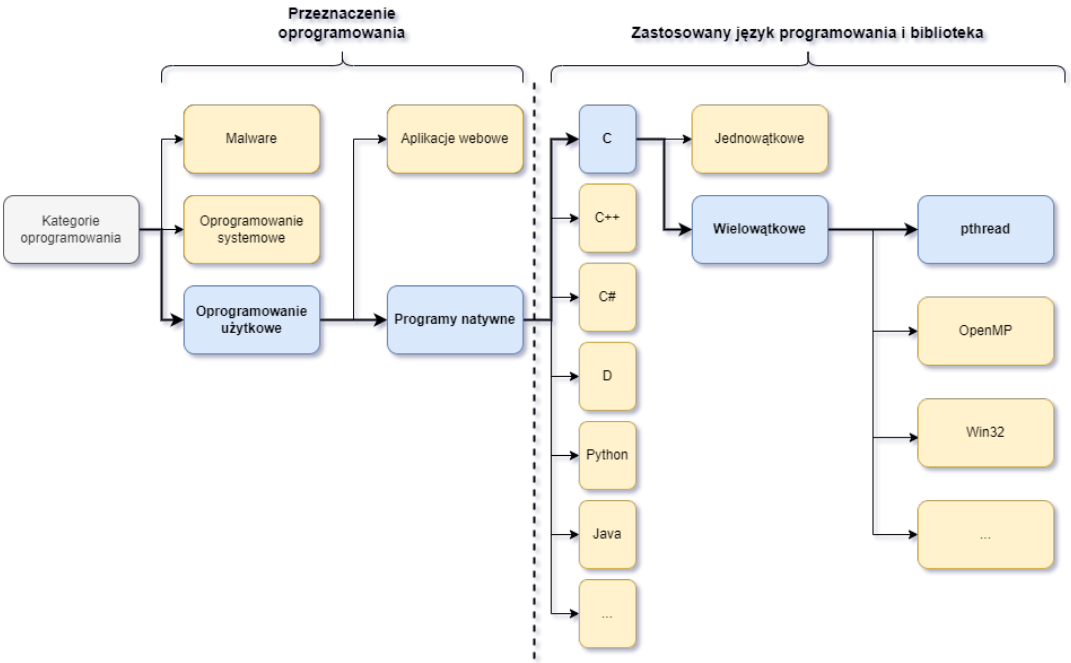
Błędy te są skutkiem wzajemnej (zwykle nieprzewidzianej) interakcji realizowanych wątków programu. Rozważany w rozprawie problem dotyczy wykrywania najczęściej występujących błędów aplikacji wielowątkowych. Należą do nich błędy skutkujące możliwością wystąpienia takich zjawisk jak konflikty zasobowe tj.: szkodliwa rywalizacja (ang. race condition), zakleszczenie (ang. deadlock), naruszenie niepodzielności (ang. atomicity violation) oraz naruszenie porządku (ang. order violation). Zjawiska te są najpopularniejszymi rodzajami konfliktów występujących w aplikacjach wielowątkowych. Zgodnie z „Concurrency bugs in multithreaded software: Modeling and analysis using Petri nets" stanowią one 96,5% spotykanych w praktyce konfliktów (30% to zakleszczenia, 49% to naruszenie niepodzielności, 10% to szkodliwa rywalizacja, 7,5% i naruszenie porządku). Z tego powodu badania prezentowane w nieniniejszej pracy ograniczają się również tylko do tych czterech przypadków. Ze względu na ich powszechność spotykane w literaturze badania koncentrują się głównie na poszukiwaniu efektywnych metod ich identyfikacji i eliminacji. Istniejące metody nie pozwalają jednak na eliminację wszystkich możliwych rodzajów błędów, m.in. takich jak omawiane w niniejszej pracy (szkodliwa rywalizacja, zakleszczenia, naruszenie porządku i niepodzielności) oraz spoza jej zakresu (jak np. żywe zakleszczenia [62]. Oznacza to, że problem identyfikacji/lokalizacji wszystkich możliwych błędów tego typu pozostaje wciąż otwarty.

Aplikacją wielowątkową nazywa się natomiast program uruchamiany w przestrzeni użytkownika (ang. user space) zadanego systemu operacyjnego, często przeznaczony do wykonywania zadań specjalnych, innych niż obsługa komputera [5]. Przykładami aplikacji są m.in.: zintegrowane środowiska programistyczne (np. Visual Studio, Anjuta), gry komputerowe (np. Doom, Tetris), odtwarzacze muzyki (np. AIMP, Spotify), komunikatory internetowe (np. Microsoft Teams, Gadu-Gadu) i wiele innych. Analogicznie aplikacją wielowątkową jest aplikacja, której architektura pozwala na równoległe wykonywanie instrukcji w ramach jednego procesu. Proces (lub zamiennie zadanie) to program, który jest wykonywany przez komputer [89].

Pisząc kod programu komputerowego takiego jak system operacyjny czy sterownik urządzenia, programista nie posiada możliwości korzystania ze wszystkich mechanizmów standardowej biblioteki dostarczanej wraz z językiem programowania. Dzieje się tak, ponieważ wiele wysokopoziomowych mechanizmów języków programowania wykorzystuje wiele mechanizmów niedostępnych np. na poziomie jądra systemu operacyjnego. Dobrym tego przykładem jest biblioteka *pthread*, która nie może być stosowana do pisania programów, które będą uruchamiane poza przestrzenią użytkownika, a co za tym idzie może być ona stosowana tylko i wyłącznie w aplikacjach. Innymi słowy każdy program, w którym możliwe jest skorzystanie z biblioteki *pthread* jest aplikacją.

Niesłabnąca popularność języka C w branży medycznej, motoryzacyjnej i energetycznej wynika z faktu, że programy pisane w tym języku są bardzo szybkie, a ich wykonywanie na urządzeniach wiąże się z niskim zużyciem energii elektrycznej. Na wysoką popularność języka C wpływ ma także rosnąca popularność dziedziny Internet of Things, która zapoczątkowała modę na urządzenia ubieralne (ang. wearables) i samodzielne budowanie inteligentnych domów (ang. smart home) przy pomocy urządzeń takich jak RaspberryPi, Arduino i innych urządzeń typu single-board computer. Wspomniane urządzenia często charakteryzują się bardzo małą mocą obliczeniową, niejednokrotnie uniemożliwiając uruchomienie na nich jakiegokolwiek systemu operacyjnego. Jednakże te urządzenia, na których możliwe staje się uruchomienie typowego systemu operacyjnego np. systemu z rodziny Windows lub systemu z jądrem Linux pozwalają także uruchamiać aplikacje pisane w języku

C z wykorzystaniem wszystkich mechanizmów tego języka w tym biblioteki *pthread*. Analogicznie w takich przypadkach można wykorzystać inne języki programowania jak Java, Python, czy językami platformy .NET, które podobnie jak język C mogą być stosowane na urządzeniach z systemem operacyjnym GNU/Linux. Stosowanie tych języków warunkowane jest jednak potrzebą skrócenia procesu pisania kodu źródłowego aplikacji. Języki te, w przeciwieństwie do języka C, wymagają do działania maszyn wirtualnych lub interpreterów, co skutkuje większym zużyciem energii przez takie urządzenia lub krótszym czasem działania urządzeń wykorzystujących baterie jako źródło energii. Powoduje to, że niejednokrotnie czas pracy urządzeń konsumenckich przekładany jest nad czas wytwarzania oprogramowania. W konsekwencji oprogramowanie na wspomniane urządzenia bardzo często pisane jest przy pomocy języka C, który ma zapewnić dłuższy czas pracy urządzeń zasilanych bateriami lub mniejsze zużycie energii urządzeń bezpośrednio podłączonych do sieci energetycznych.



RYSUNEK 1.1: Klasyfikacja programów komputerowych z zaznaczonymi istotnymi węzłami w zakresie tej pracy.

Rysunek 1.1 przedstawia diagram, na którym znajduje się podział oprogramowania ze względu na jego przeznaczenie (górna część diagramu) i zastosowany język programowania (dolna część diagramu). Do kategorii programów natywnych zaliczają się wspomniane wyżej aplikacje pisane za pomocą języka C. Standardową biblioteką opracowaną do tworzenia programów wielowątkowych we wspomnianym języku jest biblioteka *pthread*. Nie jest to jednak jedyna biblioteka opracowana w tym celu - inne stosowane w praktyce biblioteki, (np. biblioteki OpenMP, Win32, musl - rys. 1.1) nie są tak popularne i uniwersalne jak *pthread*. Najpopularniejsza z nich - OpenMP - wykorzystuje dyrektywy preprocesora, które w procesie kompilacji (odpowiednim kompilatorem, który wspiera bibliotekę OpenMP) zamieniane są na kod umożliwiający równoległe wykonanie fragmentów kodu opatrzonych wspomnianymi dyrektywami. Bibliotekę Win32 opracowano dla systemów z rodziny Windows, natomiast biblioteka musl działa tylko w systemach z jądrem Linux. Poza

wymienionymi wyżej bibliotekami istnieje jeszcze szereg innych niszowych bibliotek, które nigdy nie zyskały popularności ze względu na ich liczne ograniczenia [52]. Istotnym zagadnieniem związanym z wytwarzaniem oprogramowania jest proces jego testowania, głównie poświęcony wykrywaniu ukrytych w aplikacjach błędów. Testowanie oprogramowania na urządzeniu docelowym, na które jest ono wytwarzane bywa niejednokrotnie niemożliwe ze względu na ograniczenia techniczne (np. niewielka ilość pamięci operacyjnej komputera albo procesor o niskiej częstotliwości taktowania). W praktyce stosowane są dwa podejścia pozwalające uniknąć przeprowadzania testów na urządzeniach docelowych.

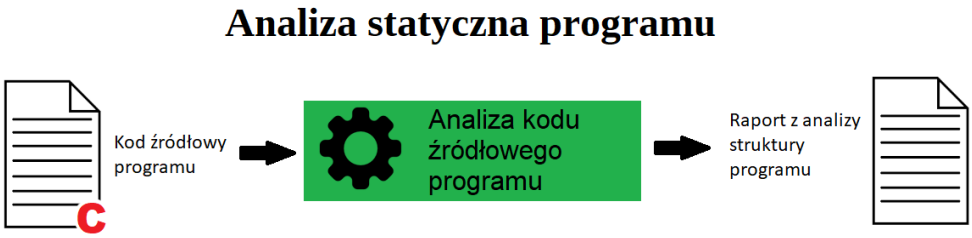
Pierwszym z nich jest testowanie oprogramowania na emulatorach lub urządzeniach zbliżonych do urządzeń docelowych. Te jednak nie zawsze mogą odzwierciedlać odpowiednie warunki fizyczne powodowane np. wpływem temperatury na elementy elektroniczne, skokami napięcia w sieci energetycznej czy też ograniczeniami wynikającymi z zastosowanych materiałów wykorzystanych do budowy docelowych urządzeń. Drugim rozwiązaniem jest stosowanie narzędzi umożliwiających analizę programów pod kątem występowania w nich błędów. Przez analizę rozumiany będzie dalej proces przeglądu kodu źródłowego (tzw. metody analizy statycznej) i/lub monitorowania działania programu wykonywany przez nadzorcę (ang. supervisor) wprowadzonego do programu (tzw. metody analizy dynamicznej).

RYSUNEK 1.2: Schemat procesu analizy dynamicznej programu.

Metody dynamicznej analizy programu polegają na analizie stanu pamięci uruchomionego programu (rys. 1.2) i/lub artefaktów wytwarzanych przez taki program w trakcie jego działania. Metody te cechują się tym, że po wykryciu błędu możliwe jest natychmiastowe działanie zapobiegające jego skutkom. Wadą ich stosowania jest natomiast fakt, że dopóki nie zostaną spełnione warunki wystąpienia błędu, nie jest możliwe jego wykrycie. Ograniczenia techniczne często powodują, że niemożliwym staje się stosowanie metod analizy dynamicznej polegających na analizie pamięci poprzez wprowadzenie nadzorców. Warto dodać, że niewłaściwie zaprojektowany nadzorca może sam być przyczyną błędów, które mogą destabilizować nadzorowany program. Dodatkowo nadzorcy bardzo często mogą być stosowani tylko w obrębie danego systemu operacyjnego bądź tylko w obrębie danej maszyny wirtualnej. Różnice pomiędzy maszynami wirtualnymi i systemami powodują, że przeniesienie danego mechanizmu nadzorcy z jednego środowiska do drugiego najczęściej jest niemożliwe. Same metody dynamiczne także poprzez swoją inwazyjność nie mogą być wykorzystywane w przypadku, gdy docelowe urządzenie nie posiada wystarczających zasobów na uruchomienie dodatkowego oprogramowania.

Metody statycznej analizy programu polegają na analizie kodu źródłowego (rys. 1.3), a nie zachowania programu, dzięki czemu czas analizy statycznej jest zwykle znacznie krótszy, niż analiza dynamiczna. Analiza statyczna kodu źródłowego jest

powszechnie używana w zintegrowanych środowiskach developerskich np. Visual Studio, CLion czy Eclipse, jednakże proces ten ogranicza się najczęściej do sprawdzania gramatyki języka i zgodności typów. W trakcie statycznej analizy kodu źródłowego możliwe jest wykrycie błędów składniowych jak i błędów strukturalnych. Zintegrowane środowiska developerskie wykorzystują te metody do informowania programistów np. o niepoprawnych nazwach zmiennych, brakujących słowach kluczowych czy „martwych" blokach kodu, gdzie przez „martwy" blok należy rozumieć kod, który nigdy nie zostanie uruchomiony. Bardzo często występują jednak sytuacje, w których analiza statyczna jest bezskuteczna. Proces optymalizacji wykonywany przez kompilator w celu wygenerowania jak najlepszej jakości plików binarnych powoduje, że kompilowany kod ulega stosownym zmianom, które z kolei prowadzić mogą do sytuacji, w której program zachowa się inaczej niż założył to programista. Oznacza to, że tak przeprowadzona analiza statyczna kodu źródłowego może się okazać bezproduktywną. Tego typu błędy są jednak możliwe do wykrycia w procesie następnie przeprowadzanej analizy dynamicznej.



RYSUNEK 1.3: Schemat procesu analizy statycznej programu.

W literaturze spotyka się także metody mieszane, w których monitoruje się zachowanie (analiza dynamiczna) fragmentu kodu źródłowego wyselekcjonowanego na podstawie analizy statycznej. Metody te, podobnie jak metody analizy dynamicznej wymagają jednak znacznych nakładów czasowych. Stosunkowo proste mechanizmy testów stosowanych w podejściach bazujących na metodach analizy dynamicznej stanowią o ich najczęstszym stosowaniu. Ich skuteczność jest jednak ograniczona brakiem możliwości przeglądu wszystkich możliwych scenariuszy działania.

Prostym przykładem ilustrującym tego typu sytuację jest fragment programu przedstawiony na rysunku 1.4. Zaprezentowany fragment kodu źródłowego programu wielowątkowego pochodzi z aplikacji opracowanej w celu badań nad wydajnością metod statycznych i dynamicznych [36] (kod źródłowy: `http://bit.ly/2ZBqhr0`). Należy zwrócić uwagę, że kod znajdujący się w wewnętrznej klauzuli *else* wykona się w momencie, gdy warunek wyrażenia *if* nie będzie spełniony (tzn. wystąpi data: niedziela, 29 lutego, godzina 23:59). Zakładając, że w klauzuli *else* znajduje się kod prowadzący do zakleszczenia, wykrycie go w procesie testowania zachowania programu może okazać się niemożliwe (jednoczesne spełnienie wszystkich wymaganych warunków jest mało prawdopodobne). W przypadku zastosowania analizy statycznej błąd ten zostanie wykryty niezależnie od spełnienia warunków w wyrażeniu *if*. Podkreślić należy fakt, że do konfliktów typu zakleszczenie często dochodzi, gdy programista chce zabezpieczyć kod przed konfliktami typu szkodliwa rywalizacja, naruszenie niepodzielności i naruszenie porządku.

Przedstawiony przykład pokazuje, że w wielu przypadkach analiza statyczna

```
void* run_job(void *args)
{
    ...
    int err_no {pthread_mutex_lock(&m)};
    ...
    if (user_job.is_day_of_week(actual_date.at(0)) &&
        user_job.is_month(actual_date.at(1)) &&
        user_job.is_month_day(actual_date.at(2)) &&
        user_job.is_time(actual_date.at(3)))
    {
        if (user_job.month_day != "29" &&
            user_job.month != "2" &&
            user_job.minutes != "59" &&
            user_job.hours != "23" &&
            user_job.day_of_week != "Sut")
        {
            ... // Regular job
        }
        else
        {
            ... // Unusual job
        }
    }
    else
    {
        pthread_mutex_unlock(&m);
    }
    ...
}
```

RYSUNEK 1.4: Kod źródłowy zawierający błąd objawiający się tylko w niedzielę 29 lutego o godzinie 23:59.

kodu źródłowego umożliwia wykrycie błędów znaczenie szybciej niż metody bazujące na analizie zachowania aplikacji (wymagające przeglądu wszystkich możliwych scenariuszy zachowania). Skuteczność tych metod warunkowana jest jednak umiejętnością analizy kodu źródłowego pod kątem występowania stosownych właściwości (cech programu), które skutkują określoną klasą błędu. Przeprowadzona analiza literatury (rozdział 2) pokazała jednak, że żadne z dotychczas opracowanych rozwiązań nie pozwala na lokalizowanie rozważanych konfliktów zasobowych przy użyciu metod opartych na statycznej analizie kodu źródłowego aplikacji wielowątkowych. Metody dynamiczne okazują się z kolei nieskuteczne w przypadku programów o dużej liczbie scenariuszy zachowań. Możliwość opracowania efektywnej metody wykrywania błędów w aplikacjach wielowątkowych, bazującej na analizie statycznej kodu programu stanowi więc motywację podjętych badań.

W przedstawionym kontekście rozważany dalej *problem lokalizacji błędów prowadzących do konfliktów zasobowych* (ang. fault localization) [106, 75] definiowany jest następująco.

Dany jest program wielowątkowy (w szczególności napisany w języku C z użyciem biblioteki *pthread*). W programie tym może dochodzić do interakcji między wątkami skutkującymi wystąpieniem konfliktów zasobowych typu: szkodliwa rywalizacja, zakleszczenie, naruszenie niepodzielności i naruszenie porządku. Poszukiwana jest odpowiedź na pytanie: *Czy w zadanym programie występują błędy prowadzące do wystąpienia zadanych konfliktów zasobowych?*

Dla tak zdefiniowanego problemu prowadzone badania koncentrowały się na

poszukiwaniach modelu, którego właściwości umożliwiają budowę efektywnej metody (tzn. uzyskującej lepsze rezultaty niż metody spotykane w literaturze) lokalizacji błędów programów wielowątkowych.

1.2 Teza

Błędy prowadzące do konfliktów zasobowych typu: szkodliwa rywalizacja, zakleszczenie, naruszenie niepodzielności i naruszenie porządku są obecne w wielu programach, niezależnie od języka jaki został wybrany do napisania ich kodu. Wynika to z faktu, że wciąż nie opracowano metod i narzędzi pozwalających wykrywać wszystkie cztery typy konfliktów zasobowych w toku jednorazowo przeprowadzanego procesu analizy. Na efektywność metod wykrywania błędów w aplikacjach wielowątkowych ma wpływ m.in.: struktura kodu źródłowego programu, mechanizmy języka i wspierane paradygmaty programowania. Przykładowo inaczej wygląda komunikacja z użytkownikiem w programach wykonywanych na poziomie jądra systemu operacyjnego, a inaczej w programach wykonywanych w przestrzeni użytkownika.

Jak to pokazał przykład z rysunku 1.4 możliwe jest [104] wykrywanie błędów w aplikacjach wielowątkowych w procesie przeglądu kodu źródłowego (ang. code review), co oznacza że proces ten można częściowo zautomatyzować. Obecnie istnieje wiele narzędzi (kompilatory, parsery i tzw. lintery (ang. linter)) bazujących na tej metodzie jednak ich zastosowania ograniczają się wyłącznie do wykrywania prostych błędów (składniowych, gramatycznych, itp.). Brak narzędzi dedykowanych do wykrywania błędów w aplikacjach wielowątkowych wynika z braku referencyjnego modelu kodu źródłowego pozwalającego opisać warunki występowania określonych błędów. Istnienie warunków, spełnienie których pozwala zlokalizować w kodzie programu napisanego w języku C błąd prowadzący do konfliktu zasobowego (szkodliwej rywalizacji, zakleszczenia, naruszenia niepodzielności lub naruszenia porządku) stanowi podstawę do budowy efektywnej metody ich wykrywania.

Formalnie konfliktem zasobowym nazywa się sytuację, w której dochodzi do równoległego wykonania operacji na współdzielonym zasobie (wbrew założonemu scenariuszowi) wprowadzając zasób w stan nieokreślony (wartość zasobu jest wypadkową operacji, których kolejność jest nieznana), co może prowadzić do awarii całego programu np. zawieszania się programu lub jego nieoczekiwanego zamknięcia. Stroustrup w swoim podręczniku do języka C++ zdefiniował zasób [92] *jako to coś, co trzeba zająć, a potem zwolnić (jawnie bądź niejawnie)*. Do przykładowych zasobów zaliczają się: pamięć, blokady, gniazda, uchwyty do wątków oraz uchwyty do plików. Każdy z zasobów niezależnie od typu może brać udział w konflikcie pomiędzy operacjami dwóch różnych wątków. Przyczyną takiego konfliktu są błędy programu, które ze względu na rodzaj inicjowanego konfliktu dzielimy na: szkodliwą rywalizację, zakleszczenie, naruszenie niepodzielności lub naruszenie porządku. Jako, że w konfliktach tych „uszkodzeniu" ulega zasób, konflikty te nazywane są konfliktami zasobowymi. Skutkiem tych konfliktów jest działanie programu odbiegające od scenariusza założonego przez programistę, czego efekty są niemożliwe do przewidzenia. Eliminacja błędów powodujących konflikty zasobowe wiąże się z procesem ich wykrywania nazywanym dalej procesem lokalizowania konfliktów zasobowych. W procesie lokalizowania konfliktów zasobowych danymi wejściowymi jest kod źródłowy programu, plik binarny programu lub dziennik zdarzeń (ang. log) programu. Wynikiem natomiast jest informacja wskazująca struktury w

kodzie programu powodujące konflikty zasobowe. W zależności od stosowanej metody lokalizowania wynikiem może być konkretna instrukcja w kodzie źródłowym, funkcja, moduł lub dowolna inna jednostka logiczna, którą jednoznacznie można zlokalizować.

Efektywne wspieranie programistów w zakresie lokalizacji błędów w programach wielowątkowych jest zagadnieniem wciąż otwartym. Wyzwaniem jest opracowanie metod, które umożliwią lokalizowanie błędów aplikacji wielowątkowych w czasie umożliwiającym ich stosowanie w praktyce. W tym ujęciu najlepszym rozwiązaniem jest opracowanie metody i jej implementacja w narzędziu, które umożliwi programistom lokalizowanie błędów powodujących konflikty zasobowe jeszcze podczas procesu pisania kodu źródłowego aplikacji wielowątkowych.

W niniejszej dysertacji rozważany jest problem lokalizacji błędów, którego celem jest zidentyfikowanie potencjalnie wadliwego kodu tzn. określenie szansy na spełnienie warunków prowadzących do konfliktu zasobowego [75, 73]. Upraszczając problem ten sprowadza się do odpowiedzi na następujące pytanie: Czy w danej aplikacji wielowątkowej występują konflikty zasobowe (szkodliwa rywalizacja, zakleszczenie, naruszenia niepodzielności i naruszenia porządku) – a jeśli tak, to które operacje w kodzie źródłowym aplikacji są ich przyczyną?

Proces statycznej analizy kodu źródłowego pozwalający odpowiedzieć na powyższe pytania będzie różnił się w zależności od tego jaki język programowania został użyty do napisania programu i jaka biblioteka dostarcza mechanizmy programowania wielowątkowego. Na potrzeby niniejszej pracy ograniczono się do języka C, jako jednego z najpopularniejszych języków programowania [95]. Natomiast wybór biblioteki *pthread* potraktowany jest czynnikami praktycznymi. Biblioteka ta jest dostępna dla wiodących systemów operacyjnych, co sprzyja tworzeniu aplikacji wieloplatformowych wykorzystując ideę „jeden kod wiele platform" (ang. one code multiple platforms). Dzięki takiemu podejściu raz napisany kod aplikacji bez zmian może zostać skompilowany dla każdego systemu operacyjnego, który posiada implementację biblioteki *pthread*.

Idea lokalizowania konfliktów zasobowych w procesie statycznej analizy kodu źródłowego powstała, gdyż proces analizy dynamicznej aplikacji wielowątkowych w praktyce jest nieefektywny. Głównym tego powodem jest konieczność odtworzenia (symulacji) wszystkich możliwych scenariuszy zachowania aplikacji. W praktyce bardzo często jest to trudne lub wręcz niemożliwe. Wykrywanie błędów prowadzących do konfliktów zasobowych jest możliwe jednak w procesie przeglądu kodu, który to proces można zautomatyzować. W przedstawionym kontekście teza niniejszej pracy brzmi następująco:

> Istnieją modele kodu źródłowego aplikacji wielowątkowych umożliwiające wyprowadzenie warunków, których spełnienie może skutkować wystąpieniem konfliktów zasobowych typu: szkodliwa rywalizacja, zakleszczenie, naruszenie niepodzielności lub naruszenie porządku.

Potwierdzenie tezy pozwoli na opracowanie metody i jej implementację w postaci narzędzia do automatycznej analizy kodu źródłowego. Celem niniejszej rozprawy jest zatem opracowanie modelu kodu źródłowego aplikacji wielowątkowej i bazującej na nim metody wykrywania błędów prowadzących do konfliktów zasobowych w procesie statycznej analizy. Opracowana metoda zostanie zaimplementowana w postaci narzędzia z interfejsem konsolowym, które umożliwi wsparcie programisty w procesie pisania kodu źródłowego aplikacji wielowątkowych.

1.3 Struktura i zakres pracy

Zakres pracy obejmuje analizę błędów aplikacji wielowątkowych prowadzących do wystąpienia: szkodliwej rywalizacji, zakleszczenia, naruszenia niepodzielności oraz naruszenia porządku. W ramach analizy poruszono zagadnienia związane z przyczynami występowania oraz zapobiegania powstawania ww. konfliktów. Omówiono również wybrane metody i narzędzia, umożliwiające ich lokalizację. W ramach pracy opracowano przykłady ilustrujące wystąpienie konfliktów i ich skutki oraz możliwe sposoby poprawy kodu źródłowego. Podano także postępowanie oraz przykłady przedstawiające sposób budowy modelu kodu źródłowego aplikacji wielowątkowej oraz bazującą na nim autorską metodę lokalizowania konfliktów zasobowych. Opracowana metoda jest dedykowana dla aplikacji wielowątkowych pisanych w języku C z wykorzystaniem biblioteki *pthread*. Jej weryfikacja przeprowadzona została w serii eksperymentów wykorzystujących dostępne w literaturze benchmarki oraz autorskie aplikacje wielowątkowe.

Na niniejszą pracę składa się 7 rozdziałów. W rozdziale drugim przedstawiono przegląd literatury, w którym przeanalizowano metody i narzędzia pozwalające lokalizować i eliminować konflikty zasobowe powodujące konflikty zasobowe takie jak szkodliwa rywalizacja, zakleszczenie, naruszenie niepodzielności i naruszenie porządku. Przedstawiono w nim także najbardziej zaawansowaną aplikację pozwalającą eliminować konflikty zasobowe o nazwie Convoider (opisana szczegółowo w rozdziale 2.5).

Rozdział trzeci opisuje czym jest model kodu źródłowego aplikacji wielowątkowej – które elementy kodu źródłowego aplikacji wielowątkowej są uwzględniane w instancji modelu, a które są zbędne. W rozdziale tym zdefiniowanych jest także kilka pojęć (np. graf operacji) wymaganych w dalszej części pracy.

W rozdziale czwartym przedstawiono warunki lokalizowania konfliktów zasobowych, które opracowano z użyciem przedstawionego w rozdziale trzecim modelu kodu źródłowego aplikacji wielowątkowej. Rozdział ten zawiera szereg warunków pozwalających lokalizować konflikty zasobowe, a także dowody wykazujące ich poprawność. Przykładowe aplikacje, w których kodzie dochodzi do konfliktów zasobowych przedstawiono w graficznej formie odpowiadającej formalnej reprezentacji instancji modelu.

Rozdział piąty zawiera opis opracowanej metody lokalizowania konfliktów zasobowych, a także odpowiada na pytanie kiedy stosować opracowaną metodę. Opisano w nim także zasady działania oraz algorytmy poszczególnych procesów i podprocesów opracowanej metody. Przedstawiono prototyp narzędzia o nazwie rdao detector, które wspiera metodę i umożliwia lokalizację konfliktów zasobowych przy użyciu analizy statycznej kodu źródłowego aplikacji wielowątkowej.

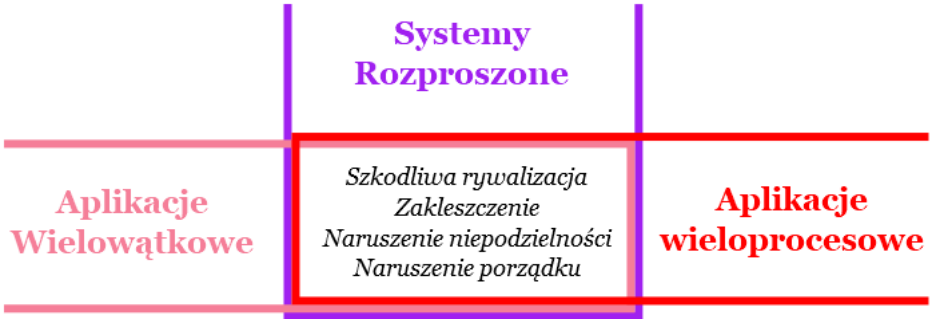
Rozdział szósty zawiera opis wykorzystanego sprzętu, procedur i wszystkich aplikacji, użytych w eksperymentach. Przedstawia również wyniki przeprowadzonych eksperymentów i ich analizę. Podsumowuje uzyskane oceny metody oraz jej implementacji.

Rozdział siódmy stanowi podsumowanie pracy. W jego skład wchodzi ocena otrzymanych wyników eksperymentów, wnioski, a także propozycja możliwych kierunków przyszłych badań.

Rozdział 2

Przegląd literatury

Celem rozdziału jest omówienie istniejących rozwiązań pozwalających na lokalizowanie błędów prowadzących do konfliktów zasobowych typu: szkodliwa rywalizacja, zakleszczenie, naruszenie niepodzielności i naruszenie porządku. Błędy tego typu wynikają z asynchronicznej natury programów wielowątkowych. Spotkać je można również, w systemach rozproszonych, czy w aplikacjach wykorzystujących wiele procesów (rys. 2.1).



RYSUNEK 2.1: Rodzina zbiorów błędów występujących w systemach asynchronicznych.

Metody lokalizowania konfliktów zasobowych dzielą się na metody statyczne (sprowadzające się do analizy kodu źródłowego) metody dynamiczne (polegające na analizie zachowania działającego programu) i metody mieszane, nazywane też metodami hybrydowymi (łączące metody dynamiczne ze statycznymi). Proces analizy statycznej często wymaga transformacji kodu źródłowego do instancji modelu umożliwiającego wykrycie określonych (błędnych) wzorów relacji między jego elementami np. instancji modelu kodu źródłowego aplikacji wielowątkowej przedstawionej na rysunku 2.2. Analiza zachowania działającego programu z kolei najczęściej sprowadza się do analizy wpisów dziennika zdarzeń takiego programu lub umieszczenia w programie nadzorcy.

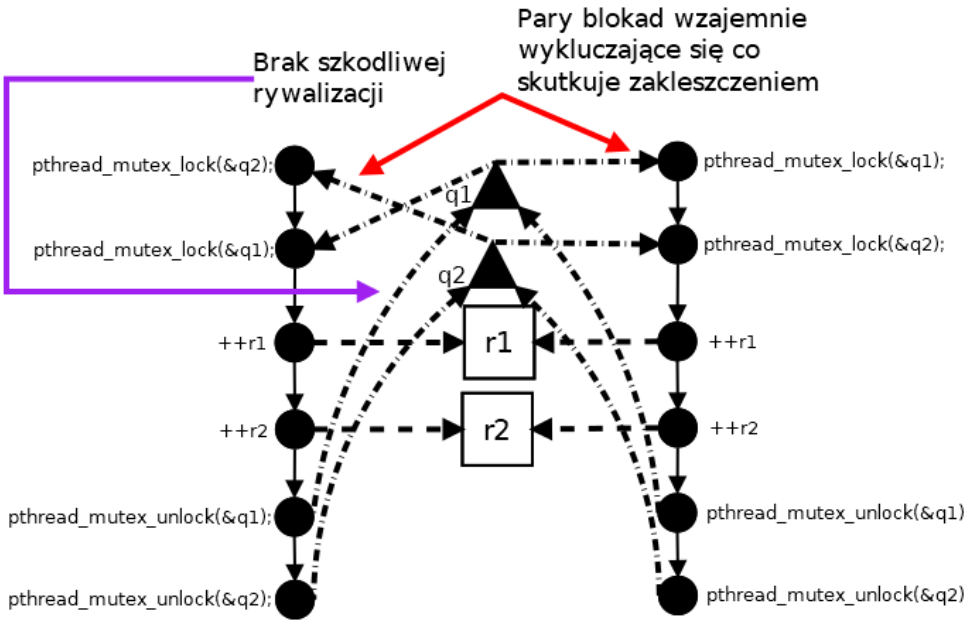
Niezależnie od rodzaju stosowanej metody zawsze istnieje prawdopodobieństwo otrzymania zgłoszeń wskazujących błąd, który w rzeczywistości nie istnieje, [3]. Zgłoszenia takie dalej będą nazywane *zgłoszeniami fałszywie pozytywnymi* (ang. false-positive errors). Zjawisko to jest szczególnie widoczne w narzędziach implementujących metody analizy statycznej. Wynika to z faktu, że metody te poszukują w kodzie źródłowym programu pewnych struktur językowych spełniających określone warunki, które to często są na tyle ogólne, że spełniają je także te fragmenty kodu, których struktura nie dopuszcza do wystąpienia konfliktów zasobowych. Jeśli jednak wyeliminuje się tę nadmiarowość poprzez takie doprecyzowanie warunków w sposób uniemożliwiający wystąpienie zgłoszeń fałszywie pozytywnych to

skuteczność opracowanej metody może okazać się niewystarczająca [109]. Lokalizowanie konfliktów zasobowych, stosując jedną z opisanych dalej metod, prawie zawsze wiąże się z występowaniem fałszywie pozytywnych zgłoszeń błędów, których całkowita eliminacja może wpłynąć negatywnie na cały proces lokalizowania błędów.

```
int main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, deposit1, NULL);
    pthread_create(&t2, NULL, deposit2, NULL);
    ...
}
```

```
void* deposit1(void *args) {
    pthread_mutex_lock(&q2);
    pthread_mutex_lock(&q1);
    ++r1;
    ++r2;
    pthread_mutex_unlock(&q1);
    pthread_mutex_unlock(&q2);
}
```

```
void* deposit2(void *args) {
    pthread_mutex_lock(&q1);
    pthread_mutex_lock(&q2);
    ++r1;
    ++r2;
    pthread_mutex_unlock(&q1);
    pthread_mutex_unlock(&q2);
}
```



RYSUNEK 2.2: Fragment instancji przykładowej aplikacji z błędem powodującym konflikt zasobowy typu zakleszczenie.

Błędy powodujące konflikty zasobowe takie jak szkodliwa rywalizacja, naruszenie niepodzielności i naruszenie porządku należą do grupy tzw. błędów o charakterze wyścigu. Poprzez błędy o charakterze wyścigu należy rozumieć te, w których uwzględnia się czas jako jedną z najważniejszych zmiennych [66, 68, 17] tzn. jeśli w pewnym (zazwyczaj) bardzo małym przedziale czasu uruchomione zostaną dwa wątki korzystające z jednego wspólnego zasobu to ich równoległe/współbieżne działanie zawsze zakończy się konfliktem na tym zasobie. Niepoprawna ich eliminacja może doprowadzić do wystąpienia zakleszczenia. W efekcie konflikty zasobowe, które powodują niepoprawne działanie programu lub nieoczekiwane zakończenie

pracy programu zostają zastąpione przez konflikty zasobowe, których skutkiem jest zawieszanie się programu. Istotnym zatem jest, aby każda poprawka eliminująca błędy klas o charakterze wyścigu była badana w taki sposób, aby wykryć potencjalne, wynikające z jej wprowadzenia, wystąpienie zakleszczeń.

Omawiane dalej rozwiązania (zarówno metody jak i narzędzia) zostały ocenione pod względem kryteriów jakościowych i ilościowych wyrażonych w postaci następujących pytań:

I. Czy czas działania/analizy pozwala na wykorzystanie metody w trybie online? Przez tryb on-line należy rozumieć czas trwania do kilku minut w przypadku analizy w trakcie kompilacji programu lub do godziny w przypadku analizy zintegrowanej z CI/CD (ang. *continuous integration* (CI) i *continuous delivery* lub *continuous deployment* (CD)).

II. Czy liczba fałszywie pozytywnych zgłoszeń o błędach generowanych przez daną metodę nie przekracza w znacznym stopniu liczby prawdziwych (ang. true positive) zgłoszeń? Należy przyjąć, że zgłoszenia fałszywie pozytywne będą występowały, jednakże ich liczba nie powinna utrudniać poznania zgłoszeń prawdziwych.

III. Czy użycie metody warunkowane jest spełnieniem dedykowanych wymagań? Przez dedykowane wymagania należy rozumieć wymagania, które nie są powszechnie spotykane w programowaniu, np. ingerencja w kompilator.

IV. Czy możliwe jest zastosowanie danego rozwiązania w celu lokalizowania konfliktów zasobowych w aplikacjach wielowątkowych pisanych w języku C?

V. Czy dane rozwiązanie jest nadal rozwijane?

VI. Czy dane rozwiązanie może być stosowane w różnorodnych środowiskach (w systemach operacyjnych Windows, Linux i MacOS, w kontenerach, na maszynach wirtualnych oraz systemach wbudowanych)?

VII. Czy liczba lokalizowanych błędów pozwala na wykorzystanie przez programistów metody w środowisku komercyjnym?

W dobie rosnącej popularności powszechnego stosowania procesu CI/CD [20, 45], bazującego na maszynach wirtualnych i kontenerach szczególnie istotne są następujące kryteria. Po pierwsze narzędzie (np. wspomagające programistę poprzez analizę statyczną) powinno umożliwiać jego zastosowanie w środowiskach bez interfejsu graficznego, na dowolnej platformie systemowej w czasie mieszczącym się w podobnych pracach programistycznych (kryterium I). Jest to istotne, ponieważ bardzo często członkowie zespołów nie mają ograniczeń w wyborze podstawowej platformy systemowej (kryterium VI). Kolejnym kryterium wyboru jest wsparcie dla zadanego języka programowania, a w tym przypadku języka C (kryterium IV). Również istotne kryterium to dalszy rozwój metody/narzędzia (kryterium V). Wiele z metod i narzędzi m. in. ze względu na swoją niską skuteczność (kryterium VII), a także bardzo trudny (kryterium III) lub kosztowny (tzn. długi czas analizy - kryterium I) proces utrzymania (np. z powodu trudnych do spełnienia wymagań) nie znajduje wystarczająco szerokiego wykorzystania. Zbyt duża liczba fałszywie pozytywnych zgłoszeń (tzw. szum) ponadto utrudnia identyfikację zgłoszeń prawdziwych (kryterium II).

2.1 Szkodliwa rywalizacja

2.1.1 Definicja

Pierwszym z omawianych konfliktów zasobowych jest *szkodliwa rywalizacja*. Konflikty tego typu mogą występować zarówno w systemach rozproszonych, w aplikacjach wykorzystujących komunikację międzyprocesową jak i w programach wielowątkowych. W programach wielowątkowych do szkodliwej rywalizacji dochodzi, gdy jedna z operacji nie wyklucza się z innymi operacjami wykonywanymi równolegle na współdzielonym zasobie. Mechanizm wzajemnego wykluczania implementuje się w programach wielowątkowych poprzez wprowadzanie tzw. *blokad* (ang. mutex, mutual exclusion). Na *blokadę* składają się: operacja założenia blokady i operacja jej zwolnienia, pomiędzy którymi umieszczane są operacje wchodzące w skład tzw. sekcji krytycznej. Jeżeli sekcje krytyczne ze zbiorem wątków „chroni" ta sama blokada [89], wtedy tylko jeden z nich może wykonywać operacje należące do sekcji krytycznej.

W literaturze istnieje kilka definicji szkodliwej rywalizacji. Według definicji pochodzącej z podręcznika do systemów operacyjnych [89] szkodliwą rywalizacją nazywa się sytuację, w której *kilka procesów współbieżnie sięga po te same dane i wykonuje na nich działania, wskutek czego wynik tych działań zależy od porządku, w jakim następował dostęp do danych*. Definicja ta dotyczy relacji zachodzących pomiędzy procesami, jednakże jest ona także słuszna w przypadku realizacji wątków, które to w literaturze są nazywane *lekkimi procesami* [89]. Druga definicja szkodliwej rywalizacji pochodzi z książki *Encyclopedia of parallel computing* Davida Padau i cytowana jest m. in. w pracy pt. „Modelling race conditions in multithreading programs in terms of Petri nets". Według Davida Padau błąd prowadzący do szkodliwej rywalizacji oznacza sytuację, w której *prawidłowe wykonanie programu zależy od kolejności lub czasu wykonania procesów lub wątków* [74]. Definicja ta jest jednak na tyle ogólna, że opisuje wszystkie konflikty o charakterze wyścigu (szkodliwa rywalizacja, naruszenie niepodzielności i naruszenie porządku).

W patencie z roku 2010 [63] szkodliwą rywalizację zdefiniowano jako sytuację, gdy *operacje co najmniej dwóch wątków wykonawczych programu w zbliżonym czasie uzyskują dostęp do tej samej lokalizacji pamięci i co najmniej jeden modyfikuje wspomniany obszar pamięci*. W definicji tej podano mało precyzyjne stwierdzenie „w zbliżonym czasie", z tego powodu została ona zmodyfikowana do następującej postaci:

Definicja 1. *Do szkodliwej rywalizacji dochodzi, gdy struktura kodu programu umożliwia uruchomienie co najmniej dwóch operacji (co najmniej jedna z nich jest operacją zapisu) dwóch różnych wątków wykorzystujących zasób współdzielony, w taki sposób, że różnica czasu pomiędzy uruchomieniem obu operacji może być mniejsza niż czas wykonania dowolnej z nich.*

Warto podkreślić, że różnorodność definicji spotykanych w literaturze świadczy o powszechności omawianego zjawiska. Przykładem pokrewnego konfliktu zasobowego (często mylonego ze szkodliwą rywalizacją) jest *hazard* występujący w cyfrowych układach elektronicznych [44]. Różnica polega na tym, że w przypadku hazardu niepoprawny stan zasobu utrzymuje się tylko przez bardzo krótki okres czasu, natomiast [104] efekt szkodliwej rywalizacji jest zwykle nieodwracalny. Z tego powodu eliminacja tego typu konfliktów ma zwykle bardzo wysoki priorytet.

2.1.2 Metody i narzędzia do lokalizowania błędów powodujących szkodliwą rywalizację

Literatura tematu zawiera opis wielu metod lokalizowania błędów prowadzących do szkodliwej rywalizacji. Zwykle są one dedykowane dla określonych zastosowań np. dla jąder systemów operacyjnych, sterowników urządzeń, czy aplikacji uruchamianych w przestrzeni użytkownika (ang. user space). Wszystkie te metody niezależnie od tego, do jakich klas programów się je stosuje podlegają podziałowi na trzy grupy [87]:

- statyczne
- dynamiczne
  - analiza w locie (ang. on-the-fly),
  - analiza pośmiertna (ang. post-mortem).
- mieszane

W zależności od grupy, do której należy metoda, oferuje ona różne kompromisy w zakresie łatwości użycia, precyzji, wydajności i ilości lokalizowanych błędów [72].

Pierwszym narzędziem rozpoczynającym niniejszy podrozdział, które warto omówić to narzędzie wykorzystujące proces statycznej analizy RacerX [25]. Narzędzie to opracowano z myślą o lokalizowaniu błędów prowadzących do szkodliwej rywalizacji i zakleszczeń w systemie operacyjnym FreeBSD, w systemach operacyjnych z jądrem Linux i w programach pozwalających na budowanie dystrybucji z tym jądrem. Metoda zastosowana w tym narzędziu polega na testowaniu grafu przepływu sterowania, reprezentującego kod źródłowy analizowanego programu. Narzędzie to pozwala analizować kod źródłowy programu niezależnie od zastosowanej w nim biblioteki dostarczającej mechanizmy wielowątkowości. Wcześniej jednak to użytkownik musi przed zastosowaniem tego narzędzia przygotować odpowiednią tabelę funkcji opisujących użytą bibliotekę, która dostarcza mechanizm wielowątkowości. Na wysokim poziomie abstrakcji kontrola systemu za pomocą RacerX obejmuje pięć faz:

1. ustawienie narzędzia na poszukiwanie blokad specyficznych dla badanego systemu,
2. wygenerowanie grafu przepływu sterowania,
3. sprawdzenie wygenerowanego grafu pod kątem występowania błędów prowadzących do szkodliwej rywalizacji i zakleszczenia,
4. przetwarzanie końcowe i klasyfikacja wyników,
5. analiza końcowa.

Proces ten jest półautomatyczny, gdyż pierwszy i ostatni krok wykonywany jest przez użytkownika [25]. Za pomocą RacerX udało się zlokalizować pewną grupę błędów w programie SystemX i w jądrze Linuksa [25]. Blisko 40% z nich [25] stanowiły jednak fałszywie pozytywne zgłoszenia błędów. RacerX został opracowany z myślą o jądrach systemów operacyjnych i narzędzi związanych z ich budową. Niestety półautomatyczny proces lokalizowania wyklucza użycie RacerX w środowiskach bez interfejsu graficznego.

Podobnym narzędziem opracowanym w celu lokalizowania szkodliwej rywalizacji w jądrze Linuksa jest program o nazwie Relay [99]. Narzędzie to, choć bardzo proste, wykazało obecność 53 błędów w jądrze Linuksa prowadzących do szkodliwej rywalizacji, co stanowiło 70% wszystkich zgłoszeń (pozostałe 30% zgłoszeń stanowiły fałszywie pozytywne zgłoszenia błędów). Program Relay nie uwzględnia jednak arytmetyki wskaźników, co jest poważnym ograniczeniem podczas analizy niskopoziomowego kodu źródłowego (takiego jak kod jądra systemu operacyjnego). Ze względu na to ograniczenie, metoda zaimplementowana w narzędziu Relay nie jest wystarczająco dobra w celu analizy aplikacji wielowątkowych pisanych w C, gdyż każda taka aplikacja zawiera setki operacji z wykorzystaniem wskaźników. Brak obsługi arytmetyki wskaźników skutkuje pomijaniem istniejących błędów powodujących szkodliwą rywalizację.

Najczęściej cytowane prace w zakresie wykorzystania statycznej/dynamicznej analizy do lokalizowania szkodliwej rywalizacji dotyczą w większości obiektowych języków kompilowanych takich jak Java czy C# [71]. Ograniczenie tych języków tylko do paradygmatu obiektowego jest zaletą w przypadku opracowywania narzędzi do ich analizy. Sprawę ułatwia także polityka *jeden program wszystkie platformy*, dzięki której implementowane metody analizy dynamicznej są niezależne od platformy systemowej. Tego typu narzędziami są Eraser i Chord, opracowane w celu lokalizowania błędów prowadzących do szkodliwej rywalizacji [86, 71]. Eraser to narzędzie opracowane z myślą o analizie dynamicznej aplikacji wielowątkowych firmy Digital Unix, w tym silnika ich wyszukiwarki internetowej AltaVista. Metoda w nim zaimplementowana polega na analizie dostępu do pamięci współdzielonej, która chroniona jest mechanizmami wzajemnego wykluczania. Eraser do działania wymaga pliku binarnego programu pisanego w języku Java, do którego wstrzykiwany jest specjalny kod z algorytmem analizującym pamięć współdzieloną. Wynikiem działania algorytmu jest plik z raportem zawierającym informacje o konfliktach zasobowych zaobserwowanych w trakcie działania programu. Twórcy tego narzędzia w podsumowaniu pracy [86] wspominają, że optymalnym rozwiązaniem dla problemu wykrywania błędów prowadzących do szkodliwej rywalizacji byłaby metoda bazująca na statycznej analizie kodu. Wskazują oni także, że w momencie pisania swojej pracy (czyli w 1997 roku) najlepsze metody jakich można było oczekiwać to metody mieszane, co wynikało bezpośrednio z mocy obliczeniowej ówczesnych komputerów. Chord natomiast jest narzędziem do statycznej analizy kodu źródłowego programów pisanych w języku Java. Zaimplementowana w nim metoda bazuje na analizie blokad i mechanizmie o nazwie „k-object sensivity", który został opracowany z myślą o paradygmacie obiektowym. Twórcy Chorda dokonali porównania wyników z opisanym wcześniej narzędziem RacerX i choć rezultaty ich narzędzia okazały się lepsze prace nad nim zostały porzucone. Dodatkowo narzędzia te opracowane zostały w celu analizy aplikacji pisanych w języku Java, a co za tym idzie metody w nich stosowane opracowano z myślą o języku obiektowym, a to wyklucza użycie tych metod do analizy aplikacji pisanych w języku C.

W roku 2019 przedstawiono pracę pt. „Sword: A scalable whole program race detector for java", w której opisano rozszerzenie (ang. plug-in) do środowiska Eclipse o nazwie SWORD, które to pozwalała lokalizować błędy prowadzące do szkodliwej rywalizacji w programach pisanych przy pomocy języka Java. Podstawą metody jest graf nazywany „static happens-before" (SHB), na bazie którego dokonywana jest analiza statyczna z wykorzystaniem kodu bajtowego programu lub drzew składniowych (ang. Abstract Syntax Tree, AST) generowanych w środowisku Eclipse. Do działania SWORD wymagane są rozszerzenia o nazwie WALA i biblioteka

Akka. Metoda zaimplementowana w rozszerzeniu opracowana została z myślą o języku Java i paradygmacie obiektowym, co uniemożliwia jej zastosowanie do analizy programów pisanych w języku C. Dodatkowo SWORD nie może być wykorzystywany w środowiskach bez interfejsu graficznego, co uniemożliwia stosowanie tego narzędzia w procesie CI/CD.

W programie DeepRace można spotkać się z zupełnie odmiennym podejściem do lokalizowania błędów prowadzących do szkodliwej rywalizacji. Stosowana w nim metoda bazuje na konwolucyjnych sieciach neuronowych (ang. convolutional neural network) [93]. Testy narzędzia przeprowadzono na ponad 13 000 plikach zawierających programy, w których wielowątkowość dostarczał framework OpenMP (~ 5000 plików) lub biblioteka *pthread* (~ 8000 plików). Danymi wejściowymi programu były wektory zawierające węzły drzew składniowych wygenerowane przez kompilator Clang. Program ten z powodzeniem znalazł blisko 85% wszystkich znanych błędów prowadzących do szkodliwej rywalizacji. Autorzy wskazują, że jedną z wad opracowanego rozwiązania jest zależność wyników od struktury drzew składniowych, które mogą się różnić w zależności od stosowanego kompilatora. Poza wspomnianymi wyżej informacjami niewiele więcej wiadomo na temat DeepRace. Nie ukazało się więcej prac dotyczących tego narzędzia, zakłada się zatem, że dalsze badania nie były prowadzone. Bazując na wynikach prac nad programem DeepRace można założyć, że wraz z rozwojem dziedziny jaką jest sztuczna inteligencja z czasem uda się opracować takie algorytmy sztucznej inteligencji, które umożliwią lokalizowanie nie tylko błędów prowadzących do szkodliwej rywalizacji, ale także błędów prowadzących do pozostałych typów konfliktów zasobowych.

W roku 2008 do amerykańskiego urzędu patentowego trafił dokument opisujący metodę statycznej analizy kodu źródłowego nazywanej dalej metodą Berga [9]. W metodzie tej kod źródłowy pisany z użyciem paradygmatu obiektowego przekształcany jest do reprezentacji pośredniej, w której to poszukiwane są cechy wskazujące na obecność błędów prowadzących do szkodliwej rywalizacji. Autorzy rozwiązania wskazują, że działa ona z językami ANSI C i ANSI C++, zaznaczając jednak, że może zostać ona użyta z innymi językami, o ile te mogą być sprowadzone do reprezentacji pośredniej. W patencie tym wspomina się o wykorzystaniu w procesie statycznej analizy bazy danych z szeregiem wstępnie zdefiniowanych procedur, które wykorzystywane są w procesie oceny. Metoda ta zatem ograniczona jest do wykrywania tylko tych błędów, których charakterystyki wprowadzono do bazy danych. Metoda Berga posiada szereg ograniczeń, które wynikają z wykorzystania bazy danych i pozwalają domniemać niską jej skuteczność. W przypadku centralnej bazy danych niemożliwe jest stosowanie narzędzi wykorzystujących tę metodę w środowiskach bez stałego połączenia z Internetem. Jednak, gdy nie istnieje jedna centralna baza danych, i każdy użytkownik bądź każda firma utrzymuje własną bazę danych można założyć, że bazy te będą niekompletne, a stosowanie mechanizmów synchronizacji dodatkowo zwiększy koszty użycia.

W pracy pt. „Static data race detection for concurrent programs with asynchronous calls" zaproponowano metodę statycznej analizy kodu źródłowego programów pisanych w języku C wykorzystującą Współbieżny Graf Przepływu Sterowania w celu lokalizowania szkodliwej rywalizacji. Opracowano ją z myślą o programach, w których [51] wielowątkowość służyła do wykonywania różnych zadań w ramach jednego procesu, co ułatwiało komunikację pomiędzy wątkami wykonującymi te zadania. Metodę tę opatentowano w 2013 roku [50] i zaimplementowano we frameworku, w którym generowany graf programu wykorzystywany jest w

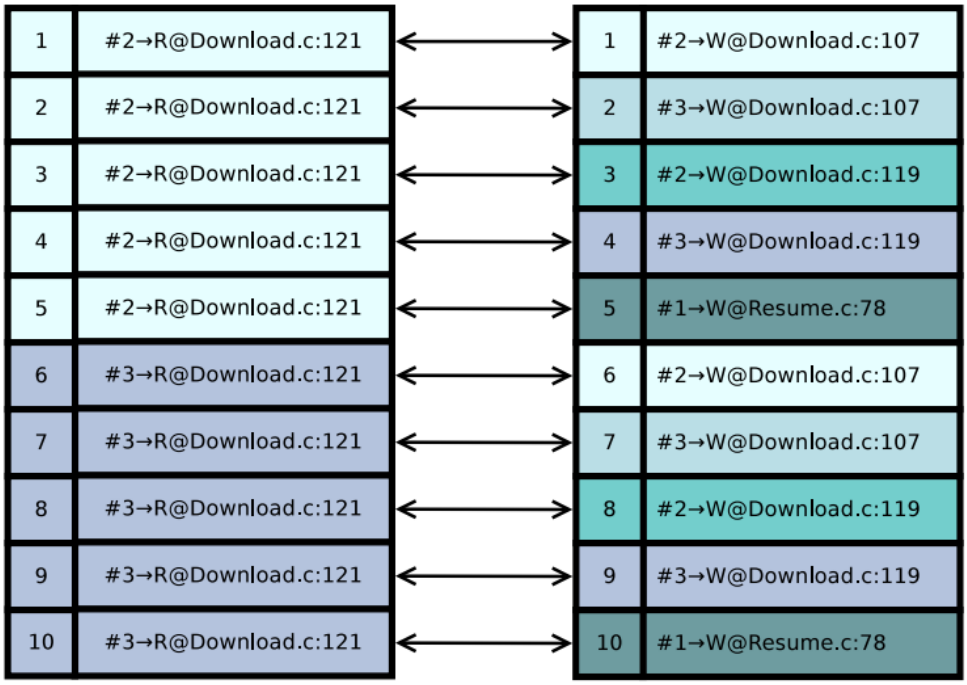
trzy-etapowym procesie analizy lokalizowania błędów prowadzących do szkodliwej rywalizacji. W pierwszym etapie lokalizowane są blokady i zasoby współdzielone, w drugim etapie budowana jest baza potencjalnych błędów, w trzecim etapie dokonywana jest analiza zawartości bazy, gdzie badane są zestawy blokad i kolejność wykonywania operacji przez wątki pracujące równolegle. Po opatentowaniu opracowanej metody dalsze prace nad nią zostały zaniechane. Implementowanie metod opracowanych w celu lokalizowania konfliktów zasobowych w postaci frameworku posiada wadę, która w wielu środowiskach uniemożliwia jest stosowanie. Otóż w procesie pisania kodu programu wielowątkowego należy wykorzystać wspomniany framework, który z definicji wnosi szereg ograniczeń i zwiększa zapotrzebowanie na zasoby komputerowe. W przypadku próby rezygnacji z takie frameworku programista zmuszony jest dokonać edycji kodu źródłowego programu, w trakcie której możliwe jest wprowadzenie do kodu nowych błędów. Rozwiązania tego typu zatem powinno się stosować tylko w momencie, gdy docelowe urządzenia posiadają wystarczającą moc obliczeniową i/lub, gdy ograniczenia narzucane przez framework nie wpływają negatywnie na proces wytwarzania kodu źródłowego.

Kolejnym narzędziem do statycznej analizy programów jest narzędzie LOCKSMITH [80]. Narzędzie to powstało z myślą o analizie programów i sterowników do urządzeń pisanych w języku C dla systemów z jądrem Linuksa (narzędzie to nie może być stosowane w innych systemach np. Windows i MacOS). Metoda zaimplementowana w LOCKSMITH bazuje na analizie kodu źródłowego programu w celu zlokalizowania zasobów współdzielonych i blokad wykluczających ich jednoczesne użycie. W tym celu kod programu sprowadzany jest do języka pośredniego, który przypomina język C i częściowo wspiera standard POSIX, z którego następnie generowany jest zestaw grafów (grafy przepływu sterowania dla operacji wątków i grafy przepływu etykiet dla zmiennych), umożliwiających lokalizowanie zasobów współdzielonych i blokad tworzących sekcje krytyczne. Analiza struktury grafów pozwala wykryć zasoby, do których potencjalnie jednoczesny dostęp posiadają operacje co najmniej dwóch wątków, a w dalszej konsekwencji błędy prowadzące do szkodliwej rywalizacji. Podobny algorytm zastosowano w autorskiej metodzie opisanej w dalszej części pracy.

W roku 2015 ukazała się praca, w której przedstawiono narzędzie do wizualizacji zjawisk o charakterze wyścigu o nazwie Bauhaus [55]. Narzędzie to okazało się jednak nieefektywne, ze względu na czasochłonny proces budowania reprezentacji graficznej jak i bardzo dużą ilość fałszywie pozytywnych zgłoszeń błędów. Przykładowa reprezentacja znajduje się na rysunku 2.3. Podkreślić należy, że dobra reprezentacja graficzna jest łatwiejsza w odbiorze dla człowieka, niż długie raporty, a w dobie szybko rozwijającej się SI może okazać się także, że taka forma reprezentacji z czasem okaże się efektywniejsza. Jednakże bardzo niska skuteczność opracowanej metody, a także brak możliwości stosowania jej w środowiskach CI/CD powoduje, że jest ona nieatrakcyjna dla potencjalnych użytkowników.

Narzędzie DR-Frame opisane w pracy pt. „Detecting Data Race in Network Applications using Static Analysis" opracowano do analizy programów sieciowych [81]. Podstawą zaimplementowanego algorytmu są drzewa składniowe generowane przez kompilator LLVM. Takie podejście pozwala w szybki i bezpieczny sposób otrzymać hierarchiczną strukturę kodu źródłowego, zawierającą wszystkie informacje niezbędne do lokalizowania błędów. W celu zmniejszenia ilości fałszywie pozytywnych zgłoszeń błędów każdy zlokalizowany błąd podlega krótkiej symulacji, co w zasadzie pozwala zaklasyfikować stosowaną metodę jako metodę mieszaną. Program ten jest w bardzo wczesnym etapie badań i jego ograniczenia do

błędów jednej klasy i programów sieciowych mocno zawęża zakres jego stosowania.



RYSUNEK 2.3: Przykład wizualizacji wykonanej przez program Bauhaus. Źródło [55].

Metoda o nazwie „Hybrid dynamic data race detection" [72], wykorzystująca dynamiczną analizę kodu aplikacji w celu lokalizowania szkodliwej rywalizacji, została opisana w roku 2003. Owa metoda analizy łączy ze sobą następujące dwie techniki dynamiczne:

- lokalizowanie bazujące na zestawie blokad (ang. lockset-based detection),
- lokalizowanie bazujące na zdarzeniach poprzedzających (ang. happens-before-based detection).

Obie techniki mogą być używane zarówno do analizy w locie jak i do analizy pośmiertnej [87].

Wśród najczęściej spotykanych metod lokalizowania błędów prowadzących do szkodliwej rywalizacji są metody dedykowane konkretnym językom lub konkretnym rodzinom języków wykorzystujących wspólne środowisko uruchomieniowe (ang. runtime environment). Przykładem są metody bazujące na systemie typów (ang. Type System) i o rozszerzenia gramatyki przeznaczone dla języka Java [2, 12, 29, 85]. Pierwszym z przeanalizowanych rozszerzeń gramatyki języka Java jest „Parameterized Race Free Java", które poszerza możliwości podzbioru języka Java o nazwie „Concurrent Java" o mechanizmy pozwalające zwiększyć bezpieczeństwo kodu programów wielowątkowych. Jako, że Java nie jest językiem w pełni statycznie typowanym to do ograniczeń tego rozwiązania należy zaliczyć brak kontroli nad procesem rzutowania odbywającego się w trakcie działania programu. Wspomniane rozszerzenie języka Java wykorzystywane jest w procesie kompilacji, stąd wyniki działania takiego rozszerzenia nie mogą zawierać zgłoszeń fałszywie pozytywnych, które skutkowały by przerwaniem poprawnych kompilacji. W efekcie tego typu rozszerzenia mogą posiadać niską skuteczność.

Drugim rozszerzeniem dla języka Java jest opracowany przez Flanagan i Freund RFJ2. Rozszerzenie to umożliwia statyczną analizę kodu przy pomocy narzędzia Rcc/Sat i lokalizowanie błędów prowadzących do szkodliwej rywalizacji [28] w czasie rzeczywistym (do minuty) dla programów posiadających po kilka tysięcy linii kodu. Programy posiadające kilkanaście lub kilkadziesiąt tysięcy linii kodu analizowane są dłużej, jednak w żadnym z badanych programów czas analizy nie był wyższy niż 50 minut [28]. Opisywane rozwiązania ograniczone są do języka Java, jednak możliwe powinno być zbudowanie analogicznych rozwiązań dla języków o budowie zbliżonej do języka Java bądź dla tych działających w tym samym środowisku uruchomieniowym.

Firma Microsoft w 2007 roku zgłosiła patent na oprogramowanie stosujące metodę dynamiczną do lokalizowania błędów prowadzących do szkodliwej rywalizacji [79]. Opracowana metoda bazuje na przeprowadzaniu szeregu testów z wykorzystaniem badanej aplikacji wielowątkowej. W trakcie tych testów narzędzie implementujące metodą wymusza bardzo częste przełączanie pracy między wątkami w celu wymuszenia wystąpienia konfliktów zasobowych. Przełączanie tych wątków występujące wielokrotnie częściej niż dzieje się to w trakcie normalnej pracy aplikacji wielowątkowej. Tego typu metoda okazała się jednak nieefektywna, a prace nad narzędziami ją stosującymi zostały porzucone.

W roku 2010 ukazał się patent, w którym przedstawiono metodę lokalizowania błędów prowadzących do szkodliwej rywalizacji, która polega na automatycznym testowaniu programów. Proces ten składał się z dwóch etapów: nagrywania pracy wątków i ponownego ich odtwarzania w celu zaobserwowania zachowania programu odbiegającego od scenariusza założonego przez programistę [63]. Automatyzacja procesu testowania posiada jednak tę wadę, że sam proces testowania składa się ze ściśle określonych scenariuszy, a na działanie programów wielowątkowych wpływ posiada bardzo wiele czynników zewnętrznych, których kontrolowanie jest niemożliwe. Metoda ta więc posiada istotne braki, gdyż przy skończonej liczbie scenariuszy i braku kontroli nad istotnymi czynnikami wpływającymi na działanie programu istnieje wysokie prawdopodobieństwo, że warunki wystąpienia konfliktu zasobowego nie zostaną spełnione. Metoda ta jest ostatnią z opisywanych metod dynamicznych opracowanych w celu lokalizowania szkodliwej rywalizacji.

Opisana wcześniej metoda „Hybrid dynamic data race detection" charakteryzuje się wykorzystaniem dwóch technik analizy dynamicznej. Rozwiązanie to dało podstawy do opracowania dwóch kolejnych narzędzi, które należą do grupy metod mieszanych. Pierwszym narzędziem jest Helgrind [48], którego rozwój porzucono ze względu na niską wydajność. Ulepszoną wersję tej metody zaimplementowano w narzędziu ThreadSanitizer [87]. Do ograniczeń ThreadSanitizera należy zaliczyć przede wszystkim możliwość pracy tylko na dystrybucjach z jądrem Linuksa i w systemie MacOS. Ze względu na to ograniczenie kod napisany w językach C i C++ kompilowany na systemie Windows nie zostanie poddany analizie. Warto także dodać, że kompilatory *clang* i *gcc* korzystające z narzędzia ThreadSanitizer nie robią tego domyślnie, a co za tym idzie mniej doświadczony programista może przez brak wiedzy/doświadczenia nie zauważyć odpowiedniego powiadomienia o błędach skutkujących konfliktem szkodliwej rywalizacji. Dodatkowo narzędzie to nie umożliwia wykrywania pozostałych konfliktów zasobowych.

W pracy dotyczącej rozszerzenia SWORD, wielokrotnie porównywano jego wyniki z wynikami pochodzącymi z narzędzia HistLock+. HistLock+ opracowany został jako narzędzie lokalizujące błędy prowadzące do szkodliwej rywalizacji przy pomocy metody mieszanej [107]. Narzędzie to jednak posiadało skuteczność niższą

niż rozszerzenie SWORD, a sam proces lokalizacji trwał dłużej. Wynika to prawdopodobnie z próby opracowania uniwersalnego narzędzia, które umożliwi lokalizowanie szkodliwej rywalizacji zarówno w języku Java jak i językach C i C++. Faktem jest jednak, że te języki bardzo się od siebie różnią, co prawdopodobnie jest przyczyną niskiej skuteczności wspomnianego narzędzia, ponieważ autorzy musieli opracować metodę, która uwzględnia wszystkie różnice pomiędzy tymi językami i bibliotekami tych języków, które umożliwiają pisanie programów wielowątkowych.

Opublikowaną w 2013 roku metodę o nazwie „access interleaving invariants" [114] opracowano z myślą o błędach o charakterze wyścigu. Jest to metoda należąca do grupy metod mieszanych i pierwotnie została zaimplementowana w narzędziu SVD [105]. Metoda ta wykorzystuje wielokrotną analizę rezultatów działań badanego programu. Skuteczność narzędzia SVD okazała się jednak niewystarczająca, jednak samą metodę rozwijano i próbowano za jej pomocą lokalizować błędy powodujące naruszenia niepodzielności.

Metodę o nazwie „Hybrid Atomicity Violation Explorer" (HAVE) [18] opracowano z myślą o języku Java i stosuje ona tzw. „Static Summary Tree". Na podstawie utworzonych drzew symulowana jest praca wybranych fragmentów programu, a efekty symulacji i utworzone drzewa są podstawą do zbudowania tzw. *hybrid trees*, które to są następnie analizowane przy użyciu autorskiego algorytmu. Wynikiem pracy algorytmu są informacje o potencjalnych błędach powodujących naruszenia niepodzielności i szkodliwą rywalizację. Autorzy tej metody wspominają o występowaniu zgłoszeń błędów fałszywie pozytywnych i dalszych pracach nad ich eliminacją. Wskazują oni także dalszy rozwój statycznej analizy kodu jako jednego z najważniejszych elementów opracowanej metody.

Odrębną grupę narzędzi stanowią metody i narzędzia pozwalające zapobiegać występowaniu konfliktów zasobowych. W pracy pt. „Automatic Detection, Validation and Repair of Race Conditions in Interrupt-Driven Embedded Software" opisana została metoda, która posłużyła do opracowania programu SDRacer w celu lokalizowania błędów prowadzących do szkodliwej rywalizacji i ich eliminacji we współczesnych systemach wbudowanych. Pod pojęciem współczesnych systemów wbudowanych może kryć się zarówno dziedzina systemów wbudowanych jak i IoT (ang. internet of things), gdyż narzędzie to było pisane z myślą o jądrze μClinux, który to został zintegrowany z główną gałęzią jądra Linuksa. SDRacer został użyty do zbadania 9 programów napisanych w języku C dla μClinux. Konflikty szkodliwej rywalizacji zlokalizowane w tych programach są spowodowane zastosowaniem mechanizmu o nazwie interrupt service routine, który to umożliwia asynchroniczne operacje na wybranych obszarach pamięci.

W roku 2009 przedstawiono narzędzie Grace, które opracowano celem eliminowania wszystkich klas błędów wynikających z zastosowania wielowątkowości poprzez eliminację samej wielowątkowości [10]. Zastosowanie tego narzędzia ograniczone było tylko do programów kompilowanych dla systemu GNU/Linux. Metoda zaimplementowana w tym narzędziu polegała na zastąpieniu wątków przez procesy potomne. Zmodyfikowane programy w większości przypadków okazały się wolniejsze [10] od swoich odpowiedników, wykorzystujących bibliotekę *pthread*. Lista wad zaimplementowanej metody jest jednak dłuższa i należą do niej:

1. zasoby, które zostały alokowane na stronie jednego podprocesu nie mogą być współdzielone z innym podprocesem,

2. konflikty dostępu do pamięci procesu nadrzędnego (procesu rodzica),

3. możliwość przekazania strony jednego podprocesu innemu podprocesowi, gdy pierwszy z nich nie zwolnił wszystkich zaalokowanych zasobów,
4. wsparcie wyłącznie 32 bitowej architektury,
5. ograniczenie działania tylko do systemów wykorzystujących jądro Linuksa,
6. znaczne spowolnienie programu w momencie zmiany zasobów współdzielonych przez wiele wątków,
7. brak wsparcia dla wątków, których celem jest nieskończone działanie np. ciągłe nasłuchiwanie komunikacji na wybranym porcie.

Ostatnia wersja narzędzia Grace została opublikowana w roku 2013 i nie jest już rozwijana, a kod programu jest dostępny publicznie na portalu GitHub.

W sekcji 2.5 opisano narzędzie Convoider opracowane w celu eliminacji błędów występujących w aplikacjach wielowątkowych, które także umożliwia lokalizowanie błędów prowadzących m. in. do konfliktów szkodliwej rywalizacji. Warto zaznaczyć, że Convoider jest obecnie najbardziej zaawansowanym narzędziem pozwalającym eliminować m. in. szkodliwą rywalizację, jednak narzędzie to może być stosowane tylko dla programów pisanych dla systemów operacyjnych z jądrem Linuksa.

Analizując ilość publikacji opisujących błędy prowadzące do szkodliwej rywalizacji można zauważyć trend, w którym największa ilość prac publikowanych w tym temacie miała miejsce w latach 2002-2011 (wg. Google Scholar). Od około roku 2011 ilość prac w tym temacie malała systematycznie do kilku prac rocznie. Większość z nich dotyczy metod dynamicznej analizy programów pisanych w językach obiektowych lub sposobów testowania programów. Jako przykład niech posłuży jedna z prac magisterskich pochodząca z 2019 roku, która omawia wykorzystanie testów funkcjonalnych, inaczej nazywanych testami czarnej skrzynki, (ang. black-box testing) jako sposobu lokalizowania błędów prowadzących do szkodliwej rywalizacji w programach web'owych [24]. Rolą przedstawionych tam narzędzi było systematyczne sprawdzanie poprawności działania programów za pomocą testów, które polegały na wywoływaniu w bardzo krótkich odstępach czasu wielu zapytań do programu. Metoda ta umożliwia określenie, czy badana funkcjonalność programu nie zawiera błędu, który prowadzi do szkodliwej rywalizacji. Jej zastosowanie ogranicza się tylko do programów web'owych podobnie jak metody stosowane w innych popularnych narzędziach np. Selenium czy Robot Framework. Ignorowany natomiast jest fakt, że kod funkcjonalności może składać się z setek bądź tysięcy linii kodu, co wiąże się z długotrwałą analizą kodu przez programistę, jego poprawieniem i ponownym testowaniem. Metodologia ta w rzeczywistości jest kolejną iteracją istniejących już rozwiązań, które są stosowanie powszechnie.

Upowszechnienie się szerokopasmowego Internetu ułatwiło komunikację pomiędzy ośrodkami badawczymi, a okres ten przypada na te same lata co wspomniane wcześniej badania nad szkodliwą rywalizacją. Możliwe zatem jest, że dzięki temu badania te mogły być prowadzone równolegle we współpracujących ze sobą ośrodkach. Późniejszą przyczyną spadku ilości badań we wspomnianym temacie może być bardzo szybki rozwój języków programowania, systemów operacyjnych i procesorów. Zmiany w procesorach wymuszały zmiany w systemach operacyjnych i językach programowania lub ich bibliotekach. Te z kolei wymuszały zmiany w kompilatorach i parserach, co bezpośrednio wpływało na istniejące już metody czyniąc je niewystarczającymi lub niezgodnymi z wprowadzonymi zmianami. Niejednokrotnie wprowadzane zmiany poszerzały zakres badań w sposób wcześniej

nieprzewidziany np. uwzględnienie nowych struktur językowych. Łatwo więc jest zauważyć, że tempo prac nad metodami lokalizacji błędów jest znacznie niższe niż tempo rozwoju technologii. Utrudnia to opracowywanie metod dostosowanych do aktualnych możliwości technicznych. Z drugiej zaś strony ilość języków programowania stale rośnie, (niektóre z nich są porzucane), a ich różnorodność nie pozwala na opracowanie jednej spójnej metody.

Z przeprowadzonego przeglądu literatury widać, że wciąż podejmuje się próby opracowania efektywnej metody lokalizowania szkodliwej rywalizacji. Skutkuje to dużą różnorodnością metod o różnej skuteczności. Wśród metod największa ich część skupia się na językach obiektowych i tylko niektóre z nich bazują na procesie statycznej analizy kodu programu, a żadne ze znalezionych rozwiązań nie pozwala na lokalizowanie szkodliwej rywalizacji w procesie statycznej analizy kodu napisanego w języku C. Literatura tematu nie dostarcza także wielu informacji o modelach stosowanych w wielu opisanych rozwiązaniach. Natomiast te dobrze opisane, jak grafy SHB w narzędziu SWORD, nie są możliwe do wykorzystania w językach, które nie są uruchamiane na maszynach wirtualnych np. JRE czy CLR.

TABLICA 2.1: Lista metod umożliwiających lokalizowanie lub zapobieganie szkodliwej rywalizacji.

Metoda/Narzędzie	Rodzaj metody	Przeznaczenie
Lokalizowanie		
RacerX [25]	statyczna	GNU/Linux, FreeBSD, *inne programy pisane w języku *C
RELAY [99]	statyczna	Jądro Linuksa
Eraser [86]	statyczna	System operacyjny Digital Unix, silnik wyszukiwania AltaVista Web, *inne oprogramowanie
Chord [71]	statyczna	Język Java
SWORD [59]	statyczna	Java w trakcie używania Eclipse
DeepRace [93]	statyczna	Programy pisane z użyciem OpenMP lub *pthread*
Metoda Berga [9]	statyczna	ANSI C ANSI C++
Współbieżny Graf Przepływu Sterowania [51]	statyczna	Język C
LOCKSMITH [80]	statyczna	Język C
Bauhaus [55]	statyczna	Język C
DR-Frame [81]	statyczna	Język C/C++
Hybrid dynamic data race detection [72]	dynamiczna	Język Java
Rozszerzenia języka Java [2, 12, 28, 29]	dynamiczna	Język Java
Rozszerzenie Sasturkara dla języka Java [85]	dynamiczna	Język Java
Narzędzia firmy Microsoft [79]	dynamiczna	*Sterowniki dla systemów z rodziny Windows
Nagrywanie i odtwarzania pracy wątków [63]	dynamiczna	*Testowanie poprzez monitorowanie platformy Windows

Metoda/Narzędzie	Rodzaj metody	Przeznaczenie
Helgrind [48]	mieszana	Język C/C++
ThreadSanitizer [87]	mieszana	Język C/C++
HistLock+ [107]	mieszana	Języki C, C++ i Java
SVD [105, 114]	mieszana	Język C
HAVE [18]	mieszana	Język Java
	Zapobieganie	
SDRacer [110]	mieszana	Programy pisane w języku C dla systemów z jądrem μClinux
Grace [10]	dynamiczna	Język C i GNU/Linux
ConFuzz [97]	mieszana	Programy skompilowane przy pomocy kompilatora llvm
Convoider [112, 113]	dynamiczna	Język C, C++ i GNU/Linux

* Pełne przeznaczenie nie jest jawnie wskazane w cytowanej pracy.

2.2 Zakleszczenie

2.2.1 Definicja

Kolejna klasa błędów, często spotykanych w programach wielowątkowych, związana jest z konfliktem zasobowym typu zakleszczenie. Według badań z 2019 roku błędy prowadzące do zakleszczeń stanowią aż 30% wszystkich błędów występujących w programach wielowątkowych [96].

W systemach współbieżnych procesów cyklicznych [11, 38, 43] zakleszczenie (tam nazywane blokadą [1]) definiowane jest następująco:

Definicja 2. *Dany jest proces P_1, zajmujący zasób R_1, żądający dostępu do zasobu R_2. Zasób ten jest zajęty przez proces P_2, który z kolei żąda dostępu do zasobu R_1, zajmowanego właśnie przez proces P_1. Powstaje zamknięty łańcuch żądań zasobowych – stan blokady [zakleszczenia], który zatrzymuje poprawne funkcjonowanie systemu.*

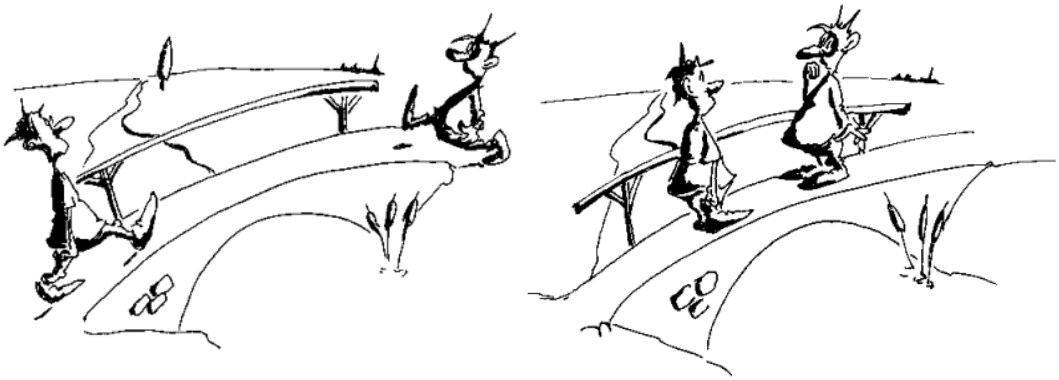
Warto zaznaczyć, że zakleszczenia występujące w naturze lub w życiu codziennym (np. sytuacja z rysunku 2.4) najczęściej są rozwiązywane naturalnie np. poprzez ustąpienie jednej ze stron lub też poprzez wprowadzenie odpowiednich regulacji prawnych np. ruch uliczny.

W książce pt. *Podstawy systemów operacyjnych* definicja zakleszczenia brzmi następująco [89]:

Definicja 3. *Sytuacja, w której kilka procesów czeka w nieskończoność na zdarzenie, które może być rozpoczęte tylko przez jeden z nich. (...) Kiedy stan taki wystąpi, wtedy o procesach, które się w nim znajdują, mówi się, że ulegają zakleszczeniu.*

W programach wielowątkowych zasobem (zgodnie z definicją 2), który jest przetrzymywany w nieskończoność (zgodnie z definicją 3) nie jest żaden z zasobów współdzielonych, a blokady [2] opisane szczegółowo w rozdz. 3.1.5. Na potrzeby niniejszej pracy zakłada się, że [36, 42] definicja zakleszczenia brzmi następująco:

Definicja 4. *Zakleszczenie jest skutkiem podjęcia nieudanej próby założenia blokady przez jeden z wątków programu, co skutkuje jego przejściem w stan oczekiwania na zwolnienie blokady. Wątek czeka w nieskończoność na zwolnienie blokady, które nigdy nie nastąpi.*



RYSUNEK 2.4: Ilustracja blokady procesów [7].

Błędy prowadzące do zakleszczenia najczęściej pojawiają się w wyniku działania programisty, gdy ten próbuje naprawić kod zawierający błędy o charakterze wyścigu [49, 65]. Do wystąpienia zakleszczenia wymagane jest spełnienie czterech warunków koniecznych [89, 88]:

1. Wzajemne **wykluczanie**: przynajmniej jeden zasób musi być niepodzielny; to znaczy, że zasobu tego może używać w danym czasie tylko jeden proces/wątek. Jeśli inny proces/wątek żąda dostępu do danego zasobu, to jego działanie musi być odłożone w czasie do momentu, aż zasób zostanie zwolniony.

2. **Przetrzymywanie i oczekiwanie**: musi istnieć proces/wątek, któremu przydzielono przynajmniej jeden zasób i który oczekuje na przydział dodatkowego zasobu, przetrzymywanego właśnie przez inny proces/wątek.

3. **Brak wywłaszczeń**: zasoby nie podlegają wywłaszczaniu, co oznacza, że zasób może zostać zwolniony wyłącznie z inicjatywy przetrzymującego go procesu/wątku, po zakończeniu pracy tego procesu/wątku.

4. **Czekanie cykliczne**: musi istnieć zbiór $P_0, P_1, P_2, ..., P_{n-1}, P_n$ czekających procesów/wątków takich że P_0 czeka na zasób przetrzymywany przez proces/wątek P_1. P_1 czeka na zasób przetrzymywany przez proces/wątek $P_2, ..., P_{n-1}$ czeka na zasób przetrzymywany przez proces/wątek P_n, a P_n czeka na zasób przetrzymywany przez proces/wątek P_0.

Aby wystąpiło zakleszczenie niezbędne jest spełnienie wszystkich czterech warunków, jednakże warunek czekania cyklicznego implikuje warunek przetrzymywania i oczekiwania, więc wymienione warunki nie są zupełnie niezależne [89]. Jeśli jednak programista zapewni, iż przynajmniej jeden z tych warunków nie będzie mógł być nigdy spełniony to do konfliktu zasobowego typu zakleszczenie nie dojdzie. Wymaga to jednak znajomości przyczyn występowania wszystkich rodzajów błędów skutkujących konfliktami zasobowymi, co w praktyce jest zwykle niemożliwe. Oznacza to, że projektowanie aplikacji wielowątkowych wiąże się z błędami strukturalnymi, których usunięcie jest bardzo trudne i/lub bardzo kosztowne.

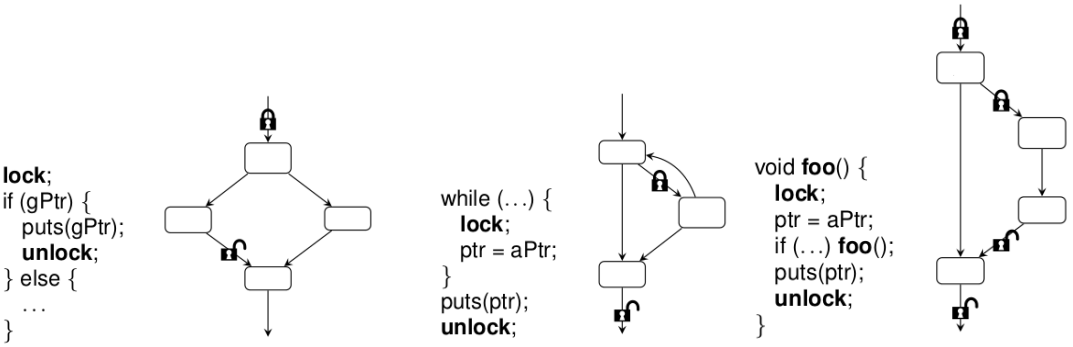
[1] Przyp. W niniejszej pracy blokadą nazywany jest mechanizm wzajemnego wykluczania (mutex)
[2] Mechanizm wzajemnego wykluczania się.

2.2.2 Przyczyny zakleszczenia

Pośród czterech klas konfliktów zasobowych rozważanych w niniejszej pracy dotychczas najlepiej zbadaną klasą są zakleszczenia. Metody pozwalające lokalizować błędy prowadzące do zakleszczeń skupiają się na lokalizowaniu blokad (lub innych mechanizmów wzajemnego wykluczania się wątków) i związanych z nimi operacji zakładania i zwalniania. Analiza prac w tym zakresie pozwoliła na dokładne określenie przyczyn błędów prowadzących do zakleszczeń, do których zaliczają się [49, 36]:

1. pary blokad wzajemnie wykluczające się (przykład programu znajduje się na listingu 2, dalej określany skrótem DL1),
2. struktura kodu pozwalająca na ominięcie operacji zwolnienia blokady (przykład programu znajduje się na listingu 3, dalej określany skrótem DL2),
3. struktura kodu, umożliwiająca wykonanie operacji ponownego założenia blokady w wyniku:
   (a) działania pętli (przykład programu znajduje się na listingu 4, dalej określany skrótem DL3),
   (b) rekurencyjnego wywołania funkcji (przykład programu znajduje się na listingu 5, dalej określany skrótem DL4).

Pierwszy wariant zakleszczenia DL1 jest najprostszy do zlokalizowania. Jeśli planista (ang. scheduler) systemu operacyjnego uruchomi wątki jednocześnie (bądź różnica czasu między uruchomieniem będzie nieznaczna) i oba wątki założą pierwszą z blokad równocześnie, to dojdzie do ich wzajemnego oczekiwania (wystąpi zakleszczenie). Warunkiem wystąpienia zakleszczenia typu DL2 i DL3 są odpowiednie dane wejściowe, z kolei warunkiem wystąpienia zakleszczenia typu DL4 jest wykorzystanie blokady innego typu niż PMR. Rysunki 2.5 i 2.6 przedstawiają cztery schematy odpowiadające fragmentom pseudokodu, których struktura prowadzi do wystąpienia zakleszczenia.

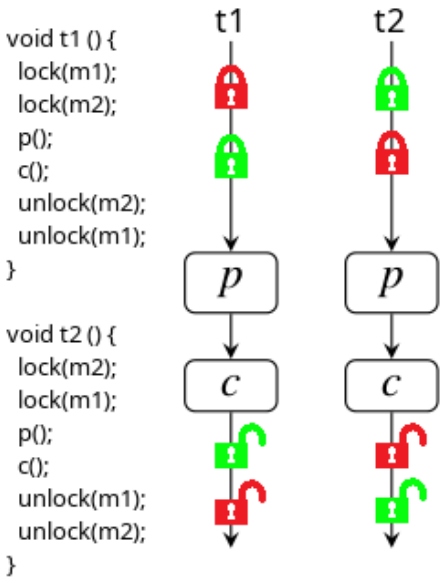


(A) Brak zwolnienia blokady; potencjalna blokada może występować później.

(B) Potencjalne podwójne blokowanie tej samej blokady lub zwolnienie blokady, bez jej uprzedniego założenia.

(C) Potencjalne podwójne blokowanie tej samej blokady.

RYSUNEK 2.5: Schematy wybranych fragmentów programów, których błędy strukturalne są przyczyną konfliktów zasobowych typu zakleszczenie. Źródło [49].



RYSUNEK 2.6: Schemat kodu źródłowego przedstawiający wzajemnie wykluczające się pary blokad.

2.2.3 Metody i narzędzia do lokalizowania błędów powodujących zakleszczenia

Pierwsze prace dotyczące lokalizowania błędów powodujących zakleszczenia pojawiały się już na początku lat sześćdziesiątych [13]. Liczba prac poświęconych temu zagadnieniu jest mniejsza niż w przypadku lokalizowania szkodliwej rywalizacji. W kolejnych akapitach znajduje się najpierw opis narzędzi i metod wykorzystujących analizę statyczną, a potem narzędzia i metody wykorzystujące analizę dynamiczną. Niektóre z narzędzi opisane w poprzednim podrozdziale (dotyczące lokalizowania błędów powodujących konflikty zasobowe typu szkodliwej rywalizacji), które umożliwiają lokalizowanie zakleszczeń nie są w niniejszym rozdziale powielane, jednak znajdują się one w ostatecznym zestawieniu w tabeli 2.2.

Jedną z najnowszych prac opisujących lokalizowanie zakleszczeń w procesie statycznej analizy opublikowano w pracy pt. „A framework for verifying deadlock and nondeterminism in UML activity diagrams based on CSP" w roku 2020. Przedstawiona tam metoda została opracowana w celu lokalizowania konfliktów w programach z domeny cloud computing i bazuje ona na diagramach aktywności UML. Autorzy rozwiązania zauważają jednak, że opracowywana przez nich metoda jest we wczesnej fazie badań, a jej użytkownicy zobowiązani są do stosowania algebry procesów jako podstawowej domeny semantycznej dla diagramów aktywności i języka CSP (ang. Communicating Sequential Processes).

W celu lokalizowania błędów prowadzących do zakleszczeń w procesie statycznej analizy kodu programów wielowątkowych pisanych z użyciem języka Java i językach do Javy podobnych opracowano i zastosowano deadLock Analysis Models (lams) [34, 57]. Model ten jest stosowany w celu lokalizowania zakleszczeń spowodowanych poprzez próbę ponownego założenia blokad. Analizowany więc jest kod w celu zlokalizowania operacji zakładania blokad znajdujących się w ciele pętli lub w funkcjach wywoływanych rekurencyjnie. W efekcie opracowana z wykorzystaniem modelu lams i Sieci Petriego metoda statycznej analizy kodu pozwala lokalizować tylko dwa z czterech przypadków, w których dochodzi do zakleszczenia.

W pracy pt. „Model checking: algorithmic verification and debugging" zaproponowano zastosowanie logiki temporalnej w celu lokalizowania zakleszczeń. Rezultatem praktycznym badań jest narzędzie ERIGONE [21], które jako wejście przyjmuje model programu opisany przy pomocy języka opracowanego na potrzeby tego narzędzia. A zatem, by użyć narzędzia ERIGONE należy znać co najmniej dwa języki, jeden do pisania kodu programu i drugi do opisania modelu. Może to spowodować, że programista pomyli się zarówno w kodzie programu jak i w modelu co może wydłużyć czas lokalizowania błędów i generować zbędne koszty. Dodatkowo należy uwzględnić koszt kursów dla programistów, którzy wykorzystywaliby ERIGONE w pracy.

Poza wymienionymi wyżej rozwiązaniami zakleszczenia można lokalizować także za pomocą narzędzia RacerX, o którym pisano w poprzednim podrozdziale opisującym narzędzia i metody lokalizacji szkodliwej rywalizacji. Lista narzędzi i metod pozwalających lokalizować zakleszczenia w procesie statycznej analizy kodu jest więc krótka. Metody i narzędzia znajdujące się na niej można uznać za ograniczone i nieefektywne, a koszt ich użycia zbyt wysoki, gdyż wymagana praca w celu ich przystosowania do aplikacji wielowątkowych pisanych w języku C jest prawie równoznaczna z opracowaniem nowej i dedykowanej metody.

Dalej opisywane metody i narzędzia wykorzystują różne metody analizy dynamicznej. Pierwszym z rozwiązań jest lokalizowanie błędów prowadzących do zakleszczenia poprzez modyfikację bibliotek dostarczających wielowątkowość [58]. Tego typu metoda przygotowana została z myślą o programach pisanych w języku C/C++ z użyciem biblioteki *pthread* lub frameworka Qt i w programach pisanych w języku Java [58]. Zastosowanie tej metody wymaga wykorzystania zmodyfikowanej biblioteki *pthread*, zmodyfikowanego OpenJDK, zmodyfikowanego frameworka Qt i zmodyfikowanego systemu bazodanowego Redis. W efekcie stosowanie tej metody jest bardzo kosztowne, gdyż wymaga utrzymania specjalnie zmodyfikowanych wersji oprogramowania. Dodatkowo, aby skorzystać z tej metody użytkownik zmuszony jest do instalacji środowisk Eclipse lub QtCreator i wtyczek współpracujących z tymi środowiskami, co wyklucza użycie tej metody w procesie CI/CD. Metoda ta w praktyce okazuje się też wysoce nieużyteczna z następujących powodów:

- Li i inni błędnie zakładają, że C i C++ to jeden język. Język C++ jest traktowany jako nadzbiór języka C, jednakże istnieje szereg poważnych różnic między wskazanymi językami [92]. Różnice te są na tyle istotne, że C++ należy traktować jako osobny język, który współdzieli z językiem C pewną część gramatyki języka. Pod koniec lat 90 oba języki otrzymały nowe rozszerzenia do swoich standardów tzn. rozszerzenie C99 dla języka C i rozszerzenie C++98 dla C++ co dodatkowo poszerzyło różnice między językami. W kolejnych latach standardy obu języków otrzymywały jeszcze kilka rozszerzeń co bywa zupełnie ignorowane. Dla przykładu język C++ wraz z rozszerzeniem C++11 otrzymał natywną bibliotekę pozwalającą pisać programy wielowątkowe z użyciem paradygmatu obiektowego [47]. W efekcie tych zmian zakładanie, że C++ jest nadzbiorem C jest błędne, gdyż oba te języki w swoich najnowszych standardach są ze sobą niekompatybilne.

- Qt według jego twórców to stworzony w języku C++ kompletny framework do tworzenia oprogramowania [103]. Ze względu na zastosowanie języka C++ framework ten posiada warstwę kompatybilności między innymi z językami Python i Java. Wraz z frameworkiem Qt dostarczony zostaje także parser metajęzyka Qt Meta Language służącego do modelowania programów, a także

parser który umożliwia pisanie logiki biznesowej w językach C++ i JavaScript. Formalnie zatem Qt nie jest językiem, gdyż jest to framework dostarczający wielu użytecznych funkcjonalności, wliczając w to wielowątkowość.

- Twórcy wymagają nieustannego rozwoju poprawek (ang. patch) dla oprogramowania Eclipse, QtCreator i Redis. Dodatkowo metoda umożliwia lokalizowanie zakleszczeń podczas pracy monitorowanego programu (analiza dynamiczna) za pośrednictwem zintegrowanych środowisk programistycznych takich jak Eclipse czy QtCreator, co zwiększa koszty użycia.
- Niemożliwe jest zastosowanie tej metody w środowiskach bez graficznego interfejsu użytkownika np. w procesie CI/CD.

Ze względu na sposób działania metodę tę zalicza się do metod dynamicznych wykorzystujących analizę pośmiertną tzn. w trakcie działania programu zbierane są informacje o jego artefaktach, które po zakończeniu działania programów podlegają analizie.

W pracy pod tytułem „A deadlock resolution strategy based on spiking neural P systems" [76] podjęto się próby zastosowania wzrostowych sieci neuronowych (ang. spiking neural networks) w celu lokalizowania błędów prowadzących do zakleszczenia. Elementem charakterystycznym tej pracy jest uwzględnienie w procesie lokalizacji błędów czasu jako jednej ze zmiennych. Warto jednak zaznaczyć, że praca ta, choć ma charakter teoretyczny, może być traktowana jako fundament kolejnych prac dotyczących zastosowania sieci neuronowych w celu lokalizowania błędów prowadzących do konfliktów zasobowych typu zakleszczenie.

W roku 2020 opublikowane pracę, w której przedstawiono metodę o nazwie Ewha COncurrency Detector [77] (ECO). Metoda ta została stworzona z myślą o poszukiwaniu konfliktów zakleszczenia, naruszenia niepodzielności i naruszenia porządku. Docelowo jej zastosowanie ograniczało się do lokalizowania błędów w systemach uzbrojenia wykorzystujących jądro Linuksa. Za pomocą tej metody możliwe jest lokalizowanie błędów nie tylko w programach wielowątkowych, ale także błędów występujących w komunikacji między procesami. Wadą rozwiązania jest pomijanie błędów powodujących szkodliwą rywalizację, a dalsze plany rozwoju metody nie uwzględniają rozszerzenia metody o ich lokalizowanie. Warto także podkreślić, że narzędzie implementujące metodę ECO nie informuje użytkownika o wystąpieniu błędów w sposób, który umożliwia łatwe ich zlokalizowanie w kodzie programu. Użytkownik natomiast dostaje informacje o błędzie w pliku binarnym, w postaci szeregu wartości zapisanych szesnastkowo, co znacząco ogranicza grupę odbiorców tego narzędzia wykluczając nawet zaawansowanych programistów.

Żadne z powyższych rozwiązań nie jest jednak na tyle skuteczne, aby przewyższały one wcześniej zaprezentowane metody statyczne. Każdą z wymienionych metod dynamicznych można określić jako kosztowną, gdyż niektóre z nich wymagającą chociażby prototypowej implementacji, a inne nie dostarczają informacji w sposób prosty do odczytu.

Jedyne rozwiązanie wykorzystujące metodę analizy mieszanej w celu lokalizowania zakleszczeń zostało opisane w patencie z 2019 roku. Metoda opisana we wspomnianym dokumencie patentowym – dalej nazywana metodą Tongpinga – łączy statyczną analizę kodu programu z technikami dynamicznymi lokalizującymi konstrukcje powodujące nieskończone cykle w programach. W pracy tej zaznaczono także, że opracowane rozwiązanie posiada wady zarówno analizy statycznej jak i dynamicznej tzn. generuje ona dużą liczbę fałszywie pozytywnych błędów i wymaga długotrwałych testów.

Podejmowano się także opracowania metod zapobiegającym zakleszczeniom i żywym zakleszczeniom (ang. livelock)[3] [62] w programach pisanych z użyciem biblioteki *pthread*, co spotkało się z dużą krytyką [90]. Przedstawiona w pracy pt. „Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability" metoda umożliwiała automatyczną poprawę kodu źródłowego. Weryfikacja poprawionych programów pokazała jednak, że wprowadzone poprawki posiadają bardzo niską jakość (tzn. powodują spadki wydajności lub powodują nowe błędy), co było efektem generowania ich kodu na podstawie ograniczonej grupy testów [90]. W efekcie porzucono dalsze badania nad metodą automatycznego generowania i nanoszenia poprawek eliminujących zakleszczenia.

Projekt o nazwie Gadara [102, 101, 56] skupia się na umieszczeniu nadzorcy w programie wielowątkowym podczas procesu kompilacji. Na podstawie kodu programu tworzona jest Sieć Petriego, która jest podstawą generowanej później maszyny stanów, która to następnie wykorzystywana jest przez nadzorcę do wymuszania poprawnego działania programu. Sposób działania nadzorcy bazuje na dyskretnym dynamicznym systemie zdarzeniowym (ang. Discrete Event Dynamic Systems), który to wykorzystywany jest do predykcji zachowania i jego korekty w celu uniknięcia niepożądanego zachowania. Pomimo zalet jakie daje umieszczenie nadzorcy w programie, wady z tym związane mogą okazać się nie do zaakceptowania w niektórych dziedzinach przemysłu. Przykładowo implementacja nadzorcy może posiadać własne błędy, które mogą wpłynąć na działanie nadzorowanego programu. W efekcie dodatkowe elementy mogą wpływać na program w sposób nieprzewidziany, co może skutkować awariami równie poważnymi w skutkach jak wystąpienie zakleszczenia. Wykorzystanie w programie wielowątkowym dodatkowego nadzorcy oznacza również większe zużycie pamięci i czasu procesora. Obecność nadzorcy w programie wciąż nie eliminuje błędu obecnego w kodzie programu. Kod ten może zostać skompilowany przy pomocy innego kompilatora niż ten z projektu Gadara, a powstały w tym procesie plik binarny programu może być używany przez nieświadomego użytkownika. Warto także podkreślić, że metody których celem jest eliminacja niepożądanych błędów wielokrotnie nie udostępniają informacji o tym, gdzie w kodzie badanego programu znajduje się błąd, który zostanie wyeliminowany. Właściwość ta powoduje, że metody eliminacji błędów nie mogą być w bezpośredni sposób porównywane z metodami bazującymi na statycznej analizie kodu programu.

Poza wspomnianymi w niniejszym podrozdziale metodami i narzędziami, do lokalizowania i zapobiegania zakleszczeniom można użyć narzędzia Convoider, którego szczegółowy opis znajduje się w podrozdziale 2.5 i narzędzia Grace opisanego w poprzednim podrozdziale.

Oczywiście poza wymienionymi w tym podrozdziale metodami istnieją także inne, które są stosowane w celu eliminacji konfliktów typu zakleszczenie m. in. w systemach operacyjnych czy w systemach automatyki. Metody te jednak bardzo często wymagają informacji specyficznych dla dziedzin, w których są stosowane stąd nie mogą one być zastosowane w dziedzinie programów wielowątkowych.

Z przeprowadzonego przeglądu literatury, którego rezultaty przedstawiono w tabeli 2.2 widać, że istniejące rozwiązania pozwalające lokalizować lub eliminować zakleszczenia są bardzo ograniczone (np. wykrywają zakleszczenia spowodowane próbą ponownego założenia blokad) albo bardzo kosztowne (np. wymagają znajomości dodatkowych języków). Model lasm opracowany w celu lokalizowania

[3] Żywe zakleszczenie to klasa błędów występująca w programach wielowątkowych, których wystąpienie powoduje cykliczną zmianę stanu aplikacji, w sposób uniemożliwiający wyjście z pętli zmiany stanów.

zakleszczeń okazuje się być ograniczony tylko do dwóch rodzajów zakleszczeń co skutecznie zawęża zakres jego stosowania. Rozwiązania wykorzystujące diagramy aktywności UML, Sieci Petriego czy logikę temporalną także nie okazały się wysoce skuteczne. Szczątkowe informacje zawarte w pracy [21] nie pozwalają określić dokładnego przeznaczenia oraz skuteczności narzędzia ERIGONE.

TABLICA 2.2: Lista metod umożliwiających lokalizowanie zakleszczeń lub ich zapobieganie.

Metoda/Narzędzie	Rodzaj metody	Przeznaczenie
Lokalizowanie		
Analiza diagramów aktywności z użyciem języka CSP [61]	statyczna	Programy w domenie cloud computing
lams [34]	statyczna	Język Java i inne podobne
RacerX [25]	statyczna	GNU/Linux, FreeBSD i inne programy pisane w języku *C
ERIGONE [21]	statyczna	Nieokreślone
Inteligentny schemat lokalizacji zakleszczeń [58]	dynamiczna	Język C/C++ i Java i programy pisane z użyciem frameworka Qt
Wzrostowe sieci neuronowe [76]	dynamiczna	Nieokreślone
Ewha COncurrency Detector [77]	dynamiczna	Programy dla systemu GNU/Linux
Metoda Tongpinga [96]	mieszana	Smalltalk, C++, C i języki im podobne
Zapobieganie		
Automatyczne naprawianie zakleszczeń i żywych zakleszczeń [62]	statyczna	Programy pisane z użyciem biblioteki *pthread*
Gadara [101]	dynamiczna	Ogólny model lokalizowania zakleszczeń
Convoider [112, 113]	dynamiczna	Język C, C++ i GNU/Linux
Grace [10]	dynamiczna	Język C i GNU/Linux

* Pełne przeznaczenie nie jest jawnie wskazane w cytowanej pracy.

2.3 Naruszenie niepodzielności

2.3.1 Definicja

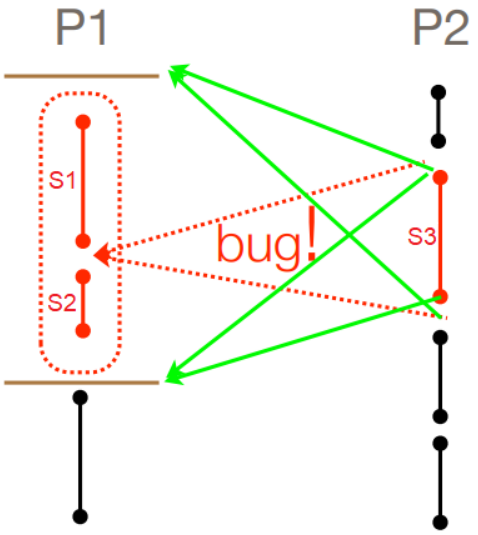
Błędy prowadzące do naruszenia niepodzielności zaliczane są do grupy błędów o charakterze wyścigu [104, 39] i stanowią one blisko 70% wszystkich zgłoszonych błędów z tej grupy [78]. W pracy pt. „Static Detection of Atomicity Violations in Object-Oriented Programs." zdefiniowano je jako błędy wynikające z „niespójnej kolejności" dostępu do danych [98]. Najczęściej odnosi się to do sytuacji, w której programista zaimplementował szeregowe wykonanie dwóch operacji na współdzielonym zasobie nie uwzględniając możliwości równoległego wykonania (w czasie pomiędzy pierwszą a drugą operacją) innej operacji zmieniającej jego stan [114]. Mówi się, że dostęp do takiego zasobu przez parę operacji powinien być niepodzielny, a

takie pary operacji traktuje się jako operacje niepodzielne (nazywane czasem atomowymi). W efekcie wcześniej wspomnianej zmiany stanu zasobu współdzielonego zakłócona zostanie relacja kolejnościowa łącząca wspomnianą parę operacji co skutkuje konfliktem zasobowym i nieprzewidzianym działaniem algorytmu. Sytuację taką przestawiono na rysunku 2.7, gdzie dwa wątki P1 i P2 równolegle wykonują swoje operacje. Kolorem czerwonym w obu wątkach zaznaczono sekcje krytyczne, zawierające operacje na wspólnym zasobie. W wątku P1 znajdują się dwie sekcje krytyczne S1 oraz S2, które ze względu na niepodzielność realizowanych w ramach nich operacji powinny stanowić całość. Tak zaprojektowany program, umożliwia uruchomienie sekcji krytycznej S1 wątku P1, następnie uruchomienie sekcji krytycznej S3 wątku P2 i na końcu uruchomienie sekcji krytycznej S2 wątku P1, skutkiem czego będzie konflikt zasobowy naruszenia niepodzielności.

W niniejszej pracy [35] definicja naruszenia niepodzielności brzmi następująco:

Definicja 5. *Do naruszenia niepodzielności dochodzi, gdy dwie operacje jednego wątku połączone relacją kolejnościową wykorzystują ten sam zasób współdzielony i relacja ta zostanie zakłócona wywołaniem operacji innego wątku z użyciem tego samego zasobu. Skutkiem takiego zakłócenia relacji jest działanie programu odbiegające od założonego przez programistę scenariusza.*

Graficzny przykład naruszenia niepodzielności znajduje się na rysunku 2.7. Wspomniana relacja kolejnościowa między dwiema operacjami O1 i O2 oznacza, że operacja O1 musi wykonać się po zakończeniu operacji O2. Szczegółowy opis relacji kolejnościowych znajduje się w podrozdziale 3.1.7.



RYSUNEK 2.7: Wizualizacja naruszenia niepodzielności. Źródło [15].

Jak już wcześniej wspomniano, błędy prowadzące do naruszenia niepodzielności zaliczane są do grupy o charakterze wyścigu. Istnieje więc podobieństwo pomiędzy naruszeniem niepodzielności i szkodliwą rywalizacją. W przypadku obu konfliktów struktura programu dopuszcza zmianę zawartości zasobu współdzielonego w sposób, którego wynik jest nieprzewidywalny. W przypadku naruszenia niepodzielności zwykle operacje wątków wykluczają się w sposób prawidłowy jednak zakłócona zostaje kolejność wykonywania sekcji krytycznych zawierających niepodzielne operacje. W pracy „Precise detection of atomicity violations" [23], naruszenie niepodzielności nazywane jest także „wysokopoziomową rywalizacją o dane"

(ang. high-level data races). W odróżnieniu od szkodliwej rywalizacji w przypadku naruszenia niepodzielności w skład sekcji krytycznych powinny wchodzić wszystkie pary niepodzielnych operacji. W obu przypadkach jednak porządek w jakim wykonywane są operacje wpływa na ostateczny wynik.

Efektem naruszenia niepodzielności jest:

1. niezdefiniowane zachowanie programu,
2. niepoprawne zachowanie programu.

Chew i Lie opisując naruszenie niepodzielności podkreślają, że skutki jego wystąpienia mogą być **wymagane**, **nieszkodliwe** albo **niepożądane** [19]. Podawany przez nich przykład **wymaganego** naruszenia niepodzielności to sytuacja, w której dwa wątki komunikują się (przekazują informacje) za pośrednictwem zasobu współdzielonego. Przykład ten nie wpisuje się jednak w żaden ze scenariuszy przedstawionych w tabeli 2.3, opisujących operacje dostępu do danych (zapis danych do pamięci/pliku i odczyt danych z pamięci/pliku) dwóch równolegle pracujących wątków. Scenariusze te zostały zidentyfikowane [19] jako podstawowe przyczyny występowania naruszenia niepodzielności w programach wielowątkowych. Przedstawione powyżej fakty pozwalają wysnuć wniosek, że twierdzenie Chew i Lie, w którym piszą i istnieniu scenariuszy, w których naruszenie niepodzielności jest wymagane jest fałszywe. Skoro dwie operacje na wskazanym zasobie mają być wykonane bez zakłóceń to zakłócenie ich pracy może być neutralne bądź niepożądane.

TABLICA 2.3: Scenariusze powodujące naruszenie niepodzielności. Źródło [19].

$odczyt_{lokalny}(A)$ $zapis_{zdalny}(A)$ $odczyt_{lokalny}(A)$	$odczyt_{lokalny}(A)$ $zapis_{zdalny}(A)$ $zapis_{lokalny}(A)$
$zapis_{lokalny}(A)$ $zapis_{zdalny}(A)$ $odczyt_{lokalny}(A)$	$zapis_{lokalny}(A)$ $odczyt_{zdalny}(A)$ $zapis_{lokalny}(A)$

2.3.2 Metody i narzędzia do lokalizowania błędów powodujących naruszenia niepodzielności

Analizę rozwiązań pozwalających na lokalizowanie błędów powodujących naruszenie niepodzielności ponownie będzie rozpoczęta od tych bazujących na statycznej analizie. Pierwsze z rozwiązań, bazujące na analizie statycznej sterowanej stanem typu (ang. typestate-guided static analysis), polega na analizie instrukcji procesora i symulacji wybranych scenariuszy [108]. Głównymi zarzutami do tej metody jest jej ograniczenie się do jednego typu mechanizmów synchronizacji i dużej ilości fałszywie pozytywnych zgłoszeń błędów [26]. Metoda wymusza na programistach stosowanie konkretnych, z góry zdefiniowanych struktur, które to wymagają jednolitych mechanizmów synchronizacji.

Wśród metod statycznej analizy kodu programu należy wyróżnić metodę umożliwiającą lokalizowanie błędów powodujących konflikty naruszenia niepodzielności bazując na rozszerzeniu gramatyki języka programowania. Metoda ta (opisana w pracy pt. „A Static Analysis for Automatic Detection of Atomicity Violations in Java Programs" [82]) bazuje na lokalizowaniu w definicji klas metod posiadających w deklaracji słowo kluczowe *synchronized*. Struktura tych metod jest następnie analizowana pod kątem występowania wzorców, które gwarantują wystąpienie konfliktów

naruszenia niepodzielności [82]. Wadą tej metody jest brak możliwości przełożenia jej na inne języki, wliczając w to język C.

Zastosowanie wspomnianych powyżej rozwiązań do lokalizowania naruszenia niepodzielności w aplikacjach pisanych w C byłoby bardzo trudne lub nawet niemożliwe, ze względu na ograniczenia tych rozwiązań. C jest językiem wykorzystującym typowanie statyczne, jednak jest to „słabe" typowanie statyczne więc analizy statyczna sterowana stanem typu może okazać się nieskuteczna. Poprzez „słabe" statyczne typowanie należy rozumieć fakt, że język ten umożliwia odczyt wartości z pamięci jako wartość innego typu niż pierwotnie zadeklarowano, co odróżnia też język C od języka C++, który jest „silnie" statycznie typowany. Poza powyższymi metodami do lokalizowania naruszenia niepodzielności w procesie statycznej analizy kodu zastosować można także omówione wcześniej narzędzia Eraser i DeepRace.

Pierwszym rozwiązaniem omówionym w niniejszej pracy wykorzystującym analizę dynamiczną w celu lokalizowania naruszenia niepodzielności jest metoda o nazwie „access interleaving invariants" [114]. Pierwotnie metodę tę zaimplementowano w narzędziu SVD (wspomniany w podrozdziale 2.1.2), a następnie w narzędziu AVIO [67] i we frameworku testowym CTrigger [78]. Metoda ta umożliwia lokalizowanie błędów prowadzących do naruszenia niepodzielności poprzez identyfikowanie scenariuszy przedstawionych w tabeli 2.3. Prace nad AVIO i CTrigger porzucono, gdyż stosowana metoda okazała się nieefektywna i kosztowna, tzn. wymaga ona dużej liczby zasobów obliczeniowych i wielu godzin pracy [78] w celu uzyskania jakichkolwiek wyników.

Park, Lu i Zhou w swojej pracy pt. „CTrigger: exposing atomicity violation bugs from their hiding places" skrytykowali także podejście lokalizacji błędów prowadzących do naruszenia niepodzielności bazujące na testach obciążeniowych (ang. stress tests). Jako wadę tego rozwiązania wskazano [78] brak możliwości odtworzenia wszystkich możliwych stanów, w jakich może znaleźć się program wielowątkowy.

Kolejnym narzędziem bazującym na dynamicznej analizie jest SideTrack opracowanym na potrzeby analizy programów wielowątkowych pisanych w Javie. Narzędzie to posiada zaimplementowany mechanizm polegający na śledzeniu zasobów współdzielonych [109]. Skuteczność tego narzędzia wynosi tylko 40%. Warto jednak podkreślić, że wśród uzyskanych wyników testów nie występują fałszywie pozytywne zgłoszenia błędów.

Podobnie jak SideTrack tak i kolejne narzędzie o nazwie RAPID opracowano w celu lokalizowania naruszeń niepodzielności w aplikacjach pisanych za pomocą języka Java. W pracy pt. „Atomicity Checking in Linear Time using Vector Clocks" [69] przedstawiono algorytm AeroDrome, który zaimplementowano w narzędziu RAPID w celu dynamicznej analizy programów pisanych dla JRE (Java Runtime Enviroment). Badania nad algorytmem są we wczesnej fazie rozwoju, a jego autorzy wskazują, że nie jest on wystarczająco skuteczny i powinien zostać użyty jako element metody mieszanej.

W pracy pt. „Trace-based symbolic analysis for atomicity violations" [100] zaproponowano metodę o nazwie „Trace-based symbolic analysis" bazującą na modelu umożliwiającym predykcję konfliktów zasobowych typu naruszenie niepodzielności. Metoda ta została opracowana zarówno dla programów pisanych w języku Java jak i tych pisanych w językach C i C++. Lokalizowanie naruszenia niepodzielności jest możliwe dzięki wykorzystaniu *teorii satysfakcji modulo* (ang. satisfiability modulo theory). Przeprowadzone eksperymenty pokazały, że użycie tej metody nie prowadzi do fałszywie pozytywnych zgłoszeń błędów. Metoda została porzucona i

nie jest dalej rozwijana. Równolegle opracowano także framework o nazwie BEST (Binary instrumentation-based Error-directed Symbolic Testing) [33], który posiadał zastosowanie podobne jak CTrigger. Metoda „Trace-based symbolic analysis" jak i framework BEST mogą być wykorzystywane dla języków C, C++ i Java ponieważ oba rozwiązania są ściśle powiązane z kompilatorami z zestawu *gcc*. Oba rozwiązania wykorzystują warstwę używaną we wszystkich kompilatorach pakietu *gcc*, a która służy do przetwarzania kodu wybranego języka do kodu pośredniego. Skutkiem tego metody te nie mogą być wykorzystywane w projektach wykorzystujących kompilatory, które nie wchodzą w skład pakietu *gcc*.

Do rozwiązań umożliwiających lokalizowanie błędów powodujących naruszenia niepodzielności należy także doliczyć wcześniej wspomniane ECO i rozszerzenia Sasturkara. Rozwiązania bazujące na analizie dynamicznej dzieli się na dwie grupy tj. rozwiązania skupione wokół języka Java lub rozwiązania o niskiej skuteczności. W efekcie żadne z zaprezentowanych rozwiązań nie nadaje się do lokalizowania naruszeń niepodzielności w aplikacjach pisanych w C.

Listę rozwiązań bazujących na metodzie mieszanej otwiera narzędzie Atomizer [27], które umożliwia lokalizowanie naruszenia niepodzielności w programach pisanych w języku Java. Pomimo obiecujących wyników autorzy postanowili skoncentrować się na metodach mieszanych (łączących statyczną analizę kodu z autorską metodą analizy "w locie"). W tym celu podjęto próby rozszerzenia podzbioru języka Java o nazwie RFJ2 i narzędzia Rcc/Sat (rozwiązanie dotychczas wykorzystywane w procesie lokalizacji szkodliwej rywalizacji). RFJ2 okazał się jednak niewystarczający i na jego podstawie opracowane kolejny podzbiór dla języka Java o nazwie AJ2, skuteczność którego wynosi 85% [30], [31]. Oba w wspomniane podzbiory są możliwe do wykorzystania tylko w języku Java.

Poza Atomizerem lista rozwiązań bazujących na metodzie mieszanej w celu lokalizowania naruszeń niepodzielności zawiera wcześniej wspomniane SVD, HAVE i ThreadSanitizer.

Cztery scenariusze opisujące dostęp do danych przez operacje dwóch wątków przedstawione w tabeli 2.3 pochodzą z pracy opisującej narzędzie o nazwie Kivati. Zostało ono opracowane z myślą o zapobieganiu konfliktów typu naruszenie niepodzielności [19]. Narzędzie to może być stosowane tylko na systemach operacyjnych z jądrem Linuksa skompilowanym dla platformy x86. Wykorzystuje ono metodę mieszaną polegającą na nadzorowaniu w trakcie działania programu, tzw. atomowych regionów (ang. atomic region - pozyskiwanych uprzednio w wyniku statycznej analizy kodu). Metoda zastosowana w programie Kivati nie uzyskała szerszego uznania i dalsze badania nad nią nie są prowadzone.

Poza narzędziem Kivati do lokalizowania naruszeń niepodzielności przy pomocy metod mieszanych można użyć także wcześniej wspomniane narzędzie Grace i opisywane dalej narzędzie Convoider (podrozdział 2.5).

Przedstawione metody i implementujące je narzędzia zostały zebrane w tabeli 2.4. Lista ta zawiera również rozwiązania opisane w poprzednich podrozdziałach (dotyczących zakleszczenia i szkodliwej rywalizacji), które pozwalają lokalizować błędy prowadzące do naruszenia niepodzielności. Dokonany przegląd literatury (obejmujący 19 rozwiązań) nie wyczerpuje tematu lokalizowania i zapobiegania błędom prowadzącym do naruszenia niepodzielności. Naruszenia niepodzielności spotkać można również m.in. w programach pisanych z wykorzystaniem paradygmatu programowania zdarzeniowego, t.j. programach serwerowych pisanych w języku JavaScript [95] z użyciem środowiska uruchomieniowego Node.js. W środowisku tym tworzy się programy z pomocą paradygmatu programowania zdarzeniowego

[16], w którym to wiele operacji wykonywanych jest równolegle w różnych wątkach [16]. Budowa języka JavaScript i środowiska Node.js uniemożliwia programiście przejęcie kontroli nad mechanizmem wątków. Dodatkowo środowisko to nie dostarcza narzędzi do oznaczania operacji jako atomowe [16]. Takiej gwarancji nie daje także stworzony z myślą o bezpiecznej wielowątkowości i opublikowany w 2010 roku język Rust [111]. Zauważyć zatem można, że metody lokalizacji jak i eliminacji konfliktów zasobowych typu naruszenie niepodzielności skupiają się prawie wyłącznie na językach obiektowych, szczególnie na języku Java. Podkreślić należy także, że żadna z metod nie została opracowana tylko z myślą o programach pisanych w języku C. Za każdym razem język C uznawany jest jako podzbiór języka C++, ignorując różnice pomiędzy tymi językami [92], w szczególności różnice między ich najnowszymi standardami.

TABLICA 2.4: Lista metod umożliwiających lokalizowanie i zapobieganie naruszeniom niepodzielności.

Metoda/Narzędzie	Rodzaj metody	Przeznaczenie
Lokalizowanie		
Analiza statyczna sterowana stanem typu [108]	statyczna	nieokreślone
Automatyczne lokalizowanie naruszenia niepodzielności przy użyciu bloków atomowych [82]	statyczna	Język Java
DeepRace [93]	statyczna	Programy pisane z użyciem OpenMP lub *pthread*
Eraser [86]	statyczna	System operacyjny Digital Unix, silnik wyszukiwania AltaVista Web, *inne oprogramowanie
AVIO [67]	dynamiczna	Język C
CTrigger[78]	dynamiczna	*Język C/C++
SideTrack [109]	dynamiczna	Język Java
RAPID [69]	dynamiczna	Programy uruchamiane w JRE
Trace-based symbolic analysis [100]	dynamiczna	Język C, C++, Java z użyciem kompilatorów z zestawu *gcc*
BEST [33]	dynamiczna	Język C, C++, Java z użyciem kompilatorów z zestawu *gcc*
Ewha COncurrency Detector [77]	dynamiczna	Programy dla systemu GNU/Linux
Rozszerzenie Sasturkara dla języka Java [85]	dynamiczna	Język Java
Atomizer [27]	mieszana	Język Java
Chang et al. [16]	mieszana	Język JavaScript

Metoda/Narzędzie	Rodzaj metody	Przeznaczenie
SVD [105, 114]	mieszana	Język C
HAVE [18]	mieszana	Język Java
ThreadSanitizer [87]	mieszana	Język C/C++
Zapobieganie		
Kivati [19]	mieszana	GNU/Linux i platforma x86
Grace [10]	dynamiczna	Język C i GNU/Linux
Convoider [112, 113]	dynamiczna	Język C, C++ i GNU/Linux

* Pełne przeznaczenie nie jest jawnie wskazane w cytowanej pracy.

2.4 Naruszenie porządku

2.4.1 Definicja

Ostatnią omawianą w niniejszej pracy klasą konfliktów zasobowych jest naruszenie porządku, polegające na zmianie kolejności wykonywania pary operacji na zasobie współdzielonym (tzn. operacja A powinna być zawsze wywołana przed B, jednak kolejność nie jest zachowana podczas wykonania) [68, 41]. Jest to kolejna klasa konfliktów zasobowych, obok szkodliwej rywalizacji i naruszenia niepodzielności, o charakterze wyścigu [66, 68, 17], których nieudana eliminacja może skutkować wystąpieniem zakleszczeń.

Na potrzeby niniejszej pracy definicja naruszenia porządku brzmi następująco:

Definicja 6. *Do naruszenia porządku dochodzi, gdy pomiędzy dwiema operacjami dwóch różnych wątków istnieje relacja kolejnościowa, której odwrócenie powoduje działanie programu odbiegające od założonego przez programistę scenariusza.*

Na rysunku 2.8 zilustrowano dwa przebiegi aplikacji za pomocą modelu kodu źródłowego aplikacji wielowątkowej opisanego szczegółowo w rozdziale 3. Pierwszy przebieg pochodzi z aplikacji, w której nie ma błędu. Pomiędzy dwiema operacjami wątków t_{11} i t_{12} istnieje relacja kolejnościowa wprzód (szczegółowy opis relacji znajduje się w podrozdziale 3.1.7), a struktura kodu aplikacji wymusza poprawną kolejność ich wykonania. W drugim przebiegu zaś struktura kodu aplikacji dopuszcza dowolną kolejność wykonania operacji wspomnianych wątków, co w odpowiednich warunkach skutkować będzie konfliktem zasobowym naruszenia porządku.

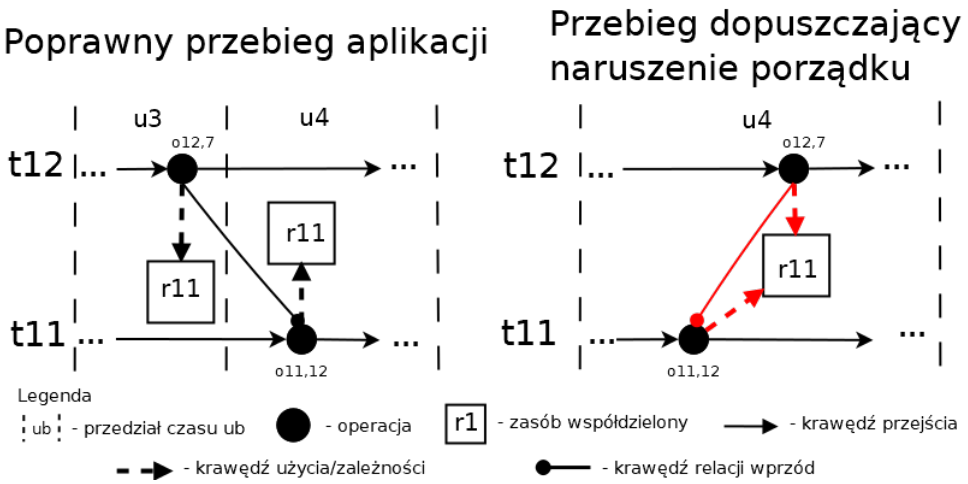
Ze wszystkich omawianych w niniejszej pracy klas konfliktów zasobowych najrzadziej badane jest naruszenie porządku. Konflikty te często są niewłaściwie klasyfikowane - naruszenia porządku najczęściej mylone są z naruszeniami niepodzielności. Z badań przeprowadzonych w roku 2017 wynika, że zarówno programiści jak i testerzy zazwyczaj nie są w stanie podać prawidłowej kolejności wykonywania par operacji wykonywanych w ramach dwóch wątków [6] tzn. znajomość scenariusza, jaki przewidział architekt bądź programista implementujący wskazaną funkcjonalność, bywa znikoma wśród innych członków zespołu. W wyniku analizy istotności błędów występujących w programach wielowątkowych [6], błędy prowadzące do naruszenia porządku ocenione zostały na poziomie 4 (gdzie 5 to błędy najmniej, a 1 to najbardziej istotne).

Rozwiązania stosowane w celu lokalizacji i usunięcia błędów prowadzących do konfliktów naruszenia porządku można podzielić na metody bazujące na [68]:

- wykorzystaniu instrukcji sterujących,

- zmianie kolejności operacji,
- zmianie struktury kodu źródłowego,
- zmianie położenia operacji zakładających i zwalniających blokady,
- innych rozwiązaniach.

Do ostatniej grupy zaliczyć można między innymi rozwiązanie polegające na wprowadzeniu zmian w kodzie programu, które powoduje przesunięcie operacji, która musi zostać wykonana jako druga do innego przedziału czasu. Rozwiązanie to znalazło zastosowanie w przypadku błędów zgłoszonych do oprogramowania fundacji Mozilla [41].



RYSUNEK 2.8: Wizualizacja naruszenia porządku przy pomocy modelu kodu źródłowego aplikacji wielowątkowej.

2.4.2 Metody i narzędzia do lokalizowania błędów naruszenia porządku

W celu lokalizowania błędów prowadzących do naruszenia porządku opracowano narzędzie ConFuzz, bazujące na metodzie wykorzystującej testy z użyciem techniki nazywaną rozmywaniem (ang. fuzzing) [97]. Poza konfliktami naruszenia porządku narzędzie to pozwala także lokalizować błędy prowadzące do szkodliwej rywalizacji i naruszenia niepodzielności. Algorytm narzędzia najpierw sprowadza kod programu do kodu pośredniego (ang. bytecode) za pomocą kompilatora *llvm*, a następnie kod ten jest poddawany procesowi rozmywania, w którym to ConFuzz zbiera zarówno informacje o przebiegu działania programu jak i informacje z narzędzia ThreadSanitizer. Na wyniki działania narzędzia ConFuzz składają się 3 zestawy danych tj. dane wejściowe programu, raport z narzędzia ThreadSanitizer i raport z przebiegu pracy programu. Proces ten kończy się raportem, w którym znajdują się informacje o zlokalizowanych błędach.

Za pomocą ConFuzz'era testowano trzy programy: pbzip2, pigz, pixz. Czas analizy każdego z tych programów to ponad 12 godzin pracy. Wyniki pracy ConFuzz'era zostały porównane z inną implementacją technik rozmywania w narzędziu DumbFuzzer. Narzędzia te wykazały zbliżone wyniki w przypadku analizy pbzip2 i pigz na korzyść ConFuzz'era. W przypadku obu narzędzi analiza pixz nie wykazała błędów, a autorzy nie podają informacji o tym, czy był to wynik oczekiwany,

czy może oba narzędzia pominęły znane błędy. Ze względu na czasochłonną analizę stosunkowo niewielkich aplikacji uznaje się, że stosowanie tego narzędzia jest wysoce kosztowne, co uniemożliwia jego stosowania np. w procesie CI/CD.

Poza ConFuzz'em istnieje jeszcze kilka metod i narzędzi wspierających lokalizowanie błędów prowadzących do naruszenia porządku. Są to wspomniane wcześniej: Eraser, DeepRace, ECO i inne wymienione w tabeli 2.5. Literatura tematu nie dostarcza informacji o innych narzędziach pozwalających lokalizować lub eliminować błędy powodujące naruszenia porządku, z wyjątkiem opisanego w kolejnym podpunkcie narzędzia Convoider. Taki stan prawdopodobnie jest spowodowany małą ilością badań nad tą klasą błędów.

TABLICA 2.5: Lista metod umożliwiających lokalizowanie lub zapobieganie naruszeniom porządku.

Metoda/Narzędzie	Rodzaj metody	Przeznaczenie
Lokalizowanie		
Eraser [86]	statyczna	System operacyjny Digital Unix, silnik wyszukiwania AltaVista Web, *inne oprogramowanie
DeepRace [93]	statyczna	Programy pisane z użyciem OpenMP lub *pthread*
Ewha COncurrency Detector [77]	dynamiczna	Programy dla systemu GNU/Linux
Rozszerzenie Sasturkara dla języka Java [85]	dynamiczna	Język Java
Trace-based symbolic analysis [100]	dynamiczna	Język C, C++, Java z użyciem kompilatorów z zestawu *gcc*
ConFuzz [97]	mieszana	Programy skompilowane przy pomocy kompilatora llvm
ThreadSanitizer [87]	mieszana	Język C/C++
Zapobieganie		
Convoider [112, 113]	dynamiczna	Język C, C++ i GNU/Linux

* Pełne przeznaczenie nie jest jawnie wskazane w cytowanej pracy.

2.5 Convoider - programowy mechanizm kontroli współbieżności

Jednym z rozwiązań umożliwiającym eliminację błędów prowadzących do wszystkich wyżej opisanych konfliktów zasobowych jest narzędzie o nazwie Convoider [112, 113]. Narzędzie to wykorzystuje mechanizm o nazwie *software transactional memory* (STM), który zdaniem jego twórców gwarantuje eliminację wszystkich (występujących w zadanym programie wielowątkowym) błędów prowadzących do konfliktów zasobowych typu: szkodliwa rywalizacja, zakleszczenie i naruszenia niepodzielności.

STM to programowy mechanizm kontroli współbieżności, za pomocą którego programiści mogą podzielić kod na transakcje i upewnić się, że każda z nich będzie uruchomiona z zachowaniem niepodzielności i w izolacji od pozostałych [113]. Poprzez programowy mechanizm kontroli współbieżności należy rozumieć taki mechanizm, który przekształca wątki w osobne procesy, eliminuje mechanizmy wzajemnego wykluczania, a także zmienia całkowicie mechanizm przydzielania i zwalniania zasobów. Rozwiązanie to jednak posiada trzy ograniczenia, które utrudniają zastosowanie tego rozwiązania powszechnie. Pierwszym ograniczeniem są wysokie koszty wprowadzenia mechanizmu transakcji przez programistów, gdyż wymaga to wprowadzenia niskopoziomowych wywołań API STM [113]. Te następnie są wykorzystywane przez Convoidera w automatycznym procesie transakcjonalizacji (ang. transactionalizes) plików binarnych. Drugim jest niska kompatybilność z wywołaniami I/O jąder systemów operacyjnych [113]. Trzecim ograniczeniem jest bardzo niska kompatybilność ze zmiennymi warunkowymi, co może prowadzić np. do błędów zagubienia sygnału (ang. lost signal) [113]. Autorzy narzędzia wspominają także tylko o implementacji tego narzędzia dla programów pisanych w C oraz C++ dla systemu GNU/Linux.

Największą wadą Convoidera jest jednak niska skuteczność w zapobieganiu przed wystąpieniem błędów prowadzących do naruszenia porządku - autorzy szacują, że ich eliminacji wynosi tylko 0.005 [112, 113]. Autorzy Convoidera nie wskazali jak należy rozumieć tę wartość, przyjmuje się więc, że wartość ta jest pomijalna, a mechanizm STM nie pozwala na eliminację naruszenia niepodzielności.

Convoider jest narzędziem rozszerzającym metodę wykorzystaną w porzuconym już narzędziu Grace [10]. Skutkiem tego Convoider podobnie jak Grace wpływa na szybkość działania programu i zwiększone zużycie pamięci, a także w chwili obecnej możliwe jest jego zastosowanie tylko i wyłącznie w systemach z jądrem Linux.

2.6 Podsumowanie

W rozdziale przedstawione zostały charakterystyki czterech konfliktów zasobowych, a także metody i narzędzia (zestawienie zbiorcze w tabeli 2.6), pozwalające na ich lokalizację i/lub eliminację. Wśród błędów identyfikowanych w programach wielowątkowych 70% z nich dotyczy występowania konfliktów o charakterze wyścigu [78] (szkodliwa rywalizacja, naruszenie niepodzielności i naruszenie porządku). Pozostałe to błędy prowadzące do zakleszczeń [96] (będących skutkiem niewłaściwej eliminacji błędów o charakterze wyścigu) oraz błędy prowadzące do niestandardowych konfliktów zasobowych [6] takich jak: zagłodzenia (ang. starvation), żywe zakleszczenia (ang. livelock), zawieszenie blokujące (ang. blocking suspension or suspension-based locking) itp. (ze względu na stosunkowo rzadkie występowanie nierozważane dalej w pracy).

Omawiane metody lokalizowania błędów można podzielić na metody: statyczne, dynamiczne oraz hybrydowe. Do zalet metod bazujących na analizie statycznej zaliczyć można ich szybkość działania. Proces analizy polega zazwyczaj na przeglądzie zupełnym kodu w poszukiwaniu struktur spełniających pewien zestaw warunków. Tego typu podejście umożliwia implementację danej metody analizy statycznej niezależnie od platformy systemowej. Do wad należy z kolei zaliczyć niską skuteczność przejawiającą się wysoką liczbą fałszywie pozytywnych zgłoszeń.

Metody bazujące na analizie dynamicznej cechują się z kolei małą ilością zgłoszeń fałszywie pozytywnych. Koszt ich stosowania jest niski dla metod analizy pośmiertnej choć wymagają one z reguły dodatkowej pracy programisty. Drogie natomiast jest stosowanie analizy w locie, gdyż wprowadzenie do aplikacji nadzorcy z reguły wymaga znikomej pracy programisty, jednak zwiększa się zapotrzebowanie na zasoby komputera w celu pracy takiej aplikacji. Należy także pamiętać, że wprowadzenie nadzorcy niesie ze sobą dodatkowe zagrożenie w postaci destabilizacji aplikacji przez błędy w kodzie źródłowym nadzorcy. Pomimo swoich zalet rozwiązania bazujące na analizie dynamicznej również charakteryzują się niską skutecznością. Wynika to z faktu, że zgłoszenie o błędzie analizowanego programu odbywa się na podstawie wystąpienia konfliktu zasobowego identyfikowanego w trakcie działania aplikacji.

Ze wszystkich wspomnianych metod, te bazujące na analizie mieszanej, posiadają najniższą liczbę fałszywie pozytywnych zgłoszeń, co jest ich najważniejszą zaletą. Do wad należy zaliczyć niską skuteczność, która wynika z faktu, że obserwowane są tylko te fragmenty działającej aplikacji, które zostały wcześniej wybrane w procesie analizy statycznej. Dodatkowo metody bazujące na analizie mieszanej, podobnie jak te bazujące na analizie dynamicznej, wymagają wystąpienia błędu w trakcie działania aplikacji, aby błąd ten został zgłoszony.

Do najważniejszych cech istniejących metod i narzędzi statycznej analizy kodu można zaliczyć:

- *Niską skuteczność*. Cecha ta dotyczy głównie metod pionierskich. Implementujące je narzędzia (np. AVIO i SVD omówione w podrozdziale 2.3.2) w kolejnych iteracjach stają się coraz bardziej skuteczne, jednak nadal nie mogą wyjść poza zakres badań.

- *Wąski zakres zastosowań*. Ze względu na wykorzystywanie specyficznych elementów języka programowania (bądź grup języków) analiza statyczna kodu programu powstałego przy użyciu języków wykorzystujących różne paradygmaty programowania jest w praktyce niemożliwe (np. metody przeznaczone dla języka Java nie mogą być zastosowane dla języka C i na odwrót). Przykładami takich rozwiązań są metody bazujące na rozszerzeniach dla języków programowania jak RFJ2 Flanagan i Freund i AJ [30] dla języka Java.

- *Ograniczenie metody do grupy mało popularnych programów takich jak systemy operacyjne czy sterowniki urządzeń.* Jako przykład można podać narzędzia RacerX, RELAY i Eraser opracowane głównie z myślą o systemach operacyjnych, czy metoda wykorzystująca diagramy aktywności dla programów w domenie cloud computing [61].

- *Zastosowanie tylko w specyficznych środowiskach.* Tutaj jako przykład można podać narzędzia SWORD i Bauhaus, których zastosowanie wymaga środowisk graficznych w systemie operacyjnym co wyklucza ich zastosowanie w procesie CI/CD, w kontenerach czy w systemach operacyjnych dostarczających tylko interfejs konsolowy.

Warto podkreślić również fakt, że przedstawione rozwiązania najczęściej umożliwiają lokalizację od jednej do trzech klas błędów. Wyjątkiem są tu narzędzia Grace i Convoider, te jednak charakteryzują się bardzo niską skutecznością. Oczywistym jest, iż opracowanie narzędzia lokalizującego wszystkie klasy błędów dla wszystkich języków programowania jest niemożliwe. Przedstawiony przegląd literatury pokazuje, że rozwój metod bazujących na statycznej analizie kodu źródłowego nie

są popularnym tematem badań. Niska skuteczność tych metod jest konsekwencją stosowania bardzo prostych modeli programów wielowątkowych. Wykorzystanie takich modeli (np. Sieci Petriego, Sieci Gadary, graf przepływu sterowania, deadLock Analysis Models) z jednej strony upraszcza analizę, z drugiej zaś prowadzi do sytuacji, gdy wiele błędów jest pomijanych. Opracowanie bardziej złożonych modeli (tzn. uwzględniających relacje konieczne do identyfikacji zadanej klasy błędów), pozwoli zwiększyć skuteczność tego typu podejść przy jednoczesnym utrzymaniu ich zalet (krótki czas analizy).

W kolejnym rozdziale pracy przedstawiony został autorski model aplikacji wielowątkowej stanowiący podstawę dla statycznej metody lokalizowania błędów prowadzących do konfliktów zasobowych typu: naruszenie niepodzielności, zakleszczenie, naruszenie porządku i naruszenie niepodzielności. Metoda jest dedykowana dla programów wielowątkowych pisanych w języku C z użyciem biblioteki *pthread*.

TABLICA 2.6: Lista metod umożliwiających lokalizowanie i zapobieganie konfliktom zasobowym w programach wielowątkowych.

Metoda/Narzędzie	Rodzaj metody	Klasa błędu	Przeznaczenie	Ocena
Lokalizowanie				
RacerX [25]	statyczna	1, 2	GNU/Linux, FreeBSD inne programy pisane w języku *C	III, V, VI
RELAY [99]	statyczna	**1	Jądro Linux	III, V, VII
Eraser [86]	statyczna	**1, 3, 4	System operacyjny Digital Unix, silnik wyszukiwania AltaVista Web, *inne oprogramowanie	IV, V
Chord [71]	statyczna	1	Język Java	IV, V
SWORD [59]	statyczna	1	Java w trakcie używania Eclipse	I, III, VII
DeepRace [93]	statyczna	**1, 3, 4	Programy pisane z użyciem OpenMP lub *pthread*	-
Metoda Berga [9]	statyczna	1	ANSI C i ANSI C++	II, III, V
Współbieżny Graf Przepływu Sterowania [51]	statyczna	**1	Język C	I, V
LOCKSMITH [80]	statyczna	**1	Język C	II, III, V
Bauhaus [55]	statyczna	**1	Język C	I, V, VI, VII
DR-Frame [81]	statyczna	**1	Język C/C++	III
Analiza diagramów aktywności z użyciem języka CSP [61]	statyczna	2	Programy w domenie cloud computing	III, IV

Metoda/Narzędzie	Rodzaj metody	Klasa błędu	Przeznaczenie	Ocena
lams [34]	statyczna	2	Język Java i inne podobne	IV, V
ERIGONE [21]	statyczna	2	Nieokreślone	I, V
Analiza statyczna sterowana stanem typu [108]	statyczna	3	nieokreślone	II, III, V
Automatyczne lokalizowanie naruszenia niepodzielności dla języka Java [82]	statyczna	3	Język Java	IV, V
Hybrid dynamic data race detection [72]	dynamiczna	1	Język Java	IV, V
Rozszerzenia języka Java [29, 12, 28, 2]	dynamiczna	1	Język Java	IV, V
Narzędzia firmy Microsoft [79]	dynamiczna	1	*Sterowniki systemów z rodziny Windows	V, VII
Nagrywanie i odtwarzania pracy wątków [63]	dynamiczna	1	*Testowanie poprzez monitorowanie platformy Windows	V, VII
Inteligentny schemat lokalizacji zakleszczeń [58]	dynamiczna	2	Język C/C++ i Java i programy pisane z użyciem frameworka Qt	I, III, VI
Wzrostowe sieci neuronowe [76]	dynamiczna	2	Nieokreślone	–
Ewha COncurrency Detector [77]	dynamiczna	2, 3, 4	Programy dla systemu GNU/Linux	I, VI
AVIO [67]	dynamiczna	3	Język C	II, V
CTrigger[78]	dynamiczna	3	*Język C/C++	II, V
SideTrack [109]	dynamiczna	3	Język Java	IV, V
RAPID [69]	dynamiczna	3	Programy uruchamiane w JRE	IV, VII
Trace-based symbolic analysis [100]	dynamiczna	**1, 3, 4	Język C, C++, Java z użyciem kompilatorów z zestawu *gcc*	III, V
BEST [33]	dynamiczna	3	Język C, C++, Java z użyciem kompilatorów z zestawu *gcc*	III, V

Metoda/Narzędzie	Rodzaj metody	Klasa błędu	Przeznaczenie	Ocena
Rozszerzenie Sasturkara dla języka Java [85]	dynamiczna	**1, 3, 4	Język Java	IV, V
Helgrind [48]	mieszana	1	Język C/C++	V, VII
ThreadSanitizer [87]	mieszana	**1, 3, 4	Język C/C++	III, V
HistLock+ [107]	mieszana	**1	Języki C, C++ i Java	VII
SVD [105, 114]	mieszana	1	Język C	V, VII
HAVE [18]	mieszana	1, 3	Język Java	II, V
Metoda Tongpinga [96]	mieszana	2	Smalltalk, C++, C i języki im podobne	I, II
Atomizer [27]	mieszana	3	Język Java	IV, V
Chang et al. [16]	mieszana	3	Język JavaScript	IV
ConFuzz [97]	mieszana	**1, 3, 4	Programy skompilowane przy pomocy kompilatora llvm	I, III
Zapobieganie				
Automatyczne naprawianie błędów klas zakleszczeń i żywych zakleszczeń [62]	statyczna	2	Programy pisane z użyciem biblioteki *pthread*	V, VII
Grace [10]	dynamiczna	1, 2, 3	Język C i GNU/Linux	I, V, VII
Convoider [112, 113]	dynamiczna	1, 2, 3, 4	Język C, C++ i GNU/Linux	I, VI
Gadara [101]	dynamiczna	2	Ogólny model lokalizowania zakleszczeń	I, III, V
SDRacer [110]	mieszana	1	Programy pisane w języku C dla systemów z jądrem μClinux	III
Kivati [19]	mieszana	3	GNU/Linux i platforma x86	V, VI

1 szkodliwa rywalizacja, 2 zakleszczenie, 3 naruszenie niepodzielności, 4 naruszenie porządku.
Litery wymienione w kolumnie uwagi oznaczają negatywny wynik oceny względem wymagań opisanych na początku niniejszego rozdziału: I - zbyt duży czas analizy/koszt użycia, II - zbyt duża liczba fałszywie pozytywnych zgłoszeń, III - dedykowane wymagania, IV - brak możliwości zastosowania do aplikacji pisanych w języku C, V - porzucony rozwój, VI - brak możliwości użycia na różnych platformach, VII - zbyt mała skuteczność, „–" - brak możliwości weryfikacji metody.
* Pełne przeznaczenie nie jest jawnie wskazane w cytowanej pracy.
** Autorzy metody używają ogólnego pojęcia *wyścig o dane*, które w zależności od kontekstu może oznaczać szkodliwą rywalizację lub pozostałe konflikty o charakterze wyścigu.

Rozdział 3

Model aplikacji wielowątkowej

Lokalizacja błędów prowadzących do konfliktów zasobowych poprzez analizę kodu źródłowego aplikacji wielowątkowej (analiza statyczna) wymaga opracowania modelu uwzględniającego niezbędne (umożliwiające identyfikację określonej klasy błędów) relacje między elementami jego struktury. W niniejszym rozdziale przedstawiono autorski model [40, 37] umożliwiający reprezentację struktury kodu źródłowego aplikacji wielowątkowej w dwóch wymiarach: wymiarze wątków i wymiarze czasu. Uwzględnienie wymiaru czasu (spotykane w literaturze modele nie uwzględniają tego wymiaru) pozwala określić, które z wątków aplikacji faktycznie pracują równolegle i są wzajemnie od siebie zależne (ich wzajemne relacje mogą prowadzić do konfliktu zasobowego). W praktyce wielokrotnie dochodzi do sytuacji, gdy zadania są realizowane w kilku wątkach jednocześnie, jednak struktura aplikacji gwarantuje ich wywołanie w taki sposób, że nigdy nie dojdzie do ich równoległego wykonania (inaczej mówiąc są one uruchamiane w różnych przedziałach czasowych). Oznacza to, że stosowanie „strukturalno-czasowej" reprezentacji umożliwi identyfikację potencjalnych interakcji między wątkami analizowanej aplikacji, i ich ocenę pod względem możliwości wystąpienia konfliktu zasobowego.

Opracowany model stanowi podstawę metody, która umożliwia lokalizowanie błędów prowadzących do konfliktów zasobowych w aplikacjach wielowątkowych. Jego wykorzystanie posiada następujące zalety:

- możliwość analizy kodu aplikacji niezależnie od docelowej platformy systemowej, na której ma ona działać,
- możliwość implementacji metody w dowolnym języku programowania.

Powyższe cechy sprawiają, że koszty procesu statycznej lokalizacji błędów są mniejsze niż w przypadku metod dynamicznych (implementację takiej metody często należy dokonać w tym samym języku co analizowana aplikacja). Możliwość implementacji metody w dowolnym języku programowania pozwala wykorzystać tzw. „proste" języki programowania (Python, Java, itp.) i tym samym zmniejszyć koszty utrzymania narzędzia.

Opracowany model jest dedykowany dla języków programowania spełniających następujące założenia:

- istnieją słowa kluczowe lub mechanizmy języka, które jawnie wskazują miejsce rozpoczęcia i miejsce zakończenia pracy wątku,
- mechanizmy wzajemnego wykluczania się jawnie są zakładane i zwalniane,
- zasoby współdzielone są dostępne poprzez mechanizm wskaźników, referencji lub zmiennych globalnych.

Przykładem języka spełniającego powyższe założenia jest język C z biblioteką *pthread*.

3.1 Model kodu źródłowego

Przyjęto, że kod źródłowy aplikacji wielowątkowej C_P jest reprezentowany w postaci następującej n-tki:

$$C_P = (T_P, U_P, R_P, O_P, Q_P, F_P, B_P) \tag{3.1}$$

gdzie:

P oznacza indeks aplikacji,

T_P oznacza zbiór wątków aplikacji C_P,

U_P oznacza sekwencję przedziałów czasu realizacji wątków zbioru T_P,

R_P oznacza rodzinę zasobów współdzielonych aplikacji C_P,

O_P oznacza zbiór operacji składających się na wątki zbioru T_P,

Q_P oznacza zbiór blokad wykorzystywanych w aplikacji C_P,

F_P oznacza zbiór par (krawędzi) reprezentujących relacje między elementami zbiorów O_P, R_P, Q_P (operacjami, zasobami i blokadami),

B_P oznacza zbiór par (krawędzi) reprezentujących relacje kolejnościowe między operacjami różnych wątków.

W kolejnych podrozdziałach szczegółowo scharakteryzowano elementy składowe zaproponowanego modelu.

3.1.1 Zbiór wątków T_P

Formalnie zbiór wątków T_P jest definiowany następująco:

$$T_P = \{t_i | i = 0...\alpha\}, \alpha \in \mathbb{N} \tag{3.2}$$

gdzie: t_i oznacza i-ty wątek aplikacji C_P.

Każda aplikacja wielowątkowa C_P (której kod napisano w języku C) składa się co najmniej z dwóch wątków: wątku głównego t_0 i wątku t_1 zainicjowanego poprzez funkcję *pthread_create*. Każdy wątek, poza wątkiem t_0, jest wątkiem potomnym, tzn. że został on uruchomiony w innym wątku aplikacji, a nie przez system operacyjny, jak ma to miejsce z wątkiem głównym. Funkcja ta wymaga czterech argumentów [64]:

- wskaźnika do obiektu typu *pthread_t*; traktowanego jako uchwyt wątku,
- stałego wskaźnika do struktury typu *pthread_attr_t*, która determinuje cechy wątku; wartość **NULL** powoduje zastosowanie domyślnych atrybutów,
- wskaźnik na funkcję typu *void**; kod wykonywany w ramach wątku,
- argumenty funkcji wykonywanej przez wątek przekazywane poprzez wskaźnik typu *void**.

Powyższe atrybuty pozwalają określić jeden z dwóch (wg. standardu **POSIX**) typów wątku tj. wątków *łączonych* (ang. joinable) ustawianych za pomocą makra PTHREAD_CREATE_JOINABLE i wątków *odłączonych* (ang. detached) ustawianych przy użyciu makra PTHREAD_CREATE_DETACHED. Ze względu na to, że występowanie konfliktów zasobowych nie jest zależne od typu wątku, w dalszej części pracy, dla uproszczenia, przyjęto że zbiór T_P zawiera tylko wątki *łączone*.

Wraz z inicjacją wątku t_i zwykle definiowany jest również algorytm szeregowania, odpowiedzialny za przydział czasu pracy procesora dla tego wątku. Algorytmy te dzielą się na dwie grupy[64]:

- *normalne* (ang. normal) - grupa algorytmów ignorujących ustawiony priorytet
  - SCHED_OTHER - karuzelowy algorytm przydziału czasu (ang. round-robin time-sharing), który przydziela wątkom czas równomiernie,
  - SCHED_IDLE - algorytm uruchamiający nieinteraktywne wątki w momencie kiedy procesor przeszedłby w stan bezczynności (ang. idle) [1],
  - SCHED_BATCH - algorytm zaprojektowany z myślą o obecności nieinteraktywnych wątków, z niskim priorytetem wykonania lecz długim czasem pracy, jednorazowo nawet do 1.5 sekundy [8],
- *czasu rzeczywistego* (ang. real-time) - grupa algorytmów uwzględniająca priorytety
  - SCHED_FIFO - algorytm przydzielający czas pracy procesora na zasadzie pierwszy na wejściu, pierwszy na wyjściu (ang. First-In-First-Out),
  - SCHED_RR - odmiana algorytmu SCHED_FIFO, która przydziela każdemu wątkowi w kolejce kwant czasu, którego przekroczenie oznacza przeniesienie wątku na koniec kolejki
  - SCHED_DEADLINE - algorytm przydzielania czasu specyficzny dla jądra Linux, który stosowany jest przy zadaniach cyklicznych, w których każde zadanie musi być uruchomione co ΔT czasu, a jego praca potrwa maksymalnie Q czasu.

Algorytmy szeregowania mają istotny wpływ na proces lokalizowania błędów przy użyciu metod dynamicznych. Podczas testów obciążeniowych może dochodzić do sytuacji gdy wydłuża się czas pracy niektórych wątków przez co zmniejsza się szansa na wykrycie zachowań świadczących o wystąpieniu konfliktu zasobowego. Tego typu zjawisko nie występuje jednak w przypadku stosowania metod analizy statycznej. Oznacza to, że rodzaj przyjętego algorytmu szeregowania nie ma wpływu na proces lokalizacji błędów przy użyciu tego typu rodzaju metody.

W przypadku metod analizy statycznej, trudne okazuje się z kolei określenie całkowitej liczby wątków α zainicjowanych w aplikacji C_P. Dzieje się tak, gdyż ich liczba jest zależna m.in. od danych otrzymanych od użytkownika aplikacji. Część omawianych w niniejszej pracy błędów np. błędy prowadzące do szkodliwej rywalizacji mogą wystąpić w momencie, gdy równolegle pracują przynajmniej dwa wątki. W sytuacji, gdy funkcja inicjująca nowe wątki jest wywoływana wielokrotnie (np. w pętli lub funkcji rekurencyjnej), przyjmuje się dla uproszczenia, że inicjowane są tylko dwa wątki.

Podsumowując zbiór T_P zawiera wątki *łączone*, dla których pomija się przyjęte algorytmy szeregowania. Liczba wątków $\alpha = |T_P|$ określana jest na podstawie zadeklarowanych w kodzie źródłowym wywołań wątków (gdzie dla każdej pętli inicjującej wątki przyjęto, że ich liczba wynosi 2).

3.1.2 Sekwencja przedziałów czasu U_P

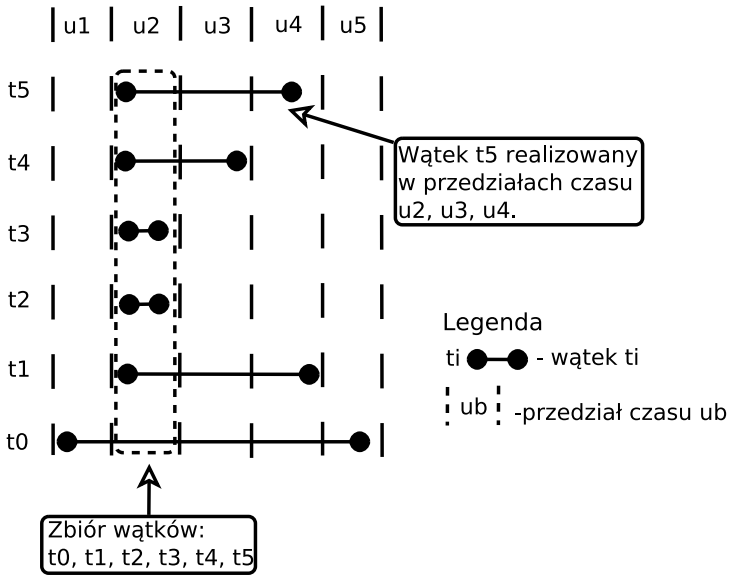
Formalnie sekwencja przedziałów czasu U_P realizacji wątków zbioru T_P definiowana jest następująco:

$$U_P = (u_1, ..., u_b, ..., u_\beta), \beta > 2, \beta \in \mathbb{N}^+ \tag{3.3}$$

gdzie: $u_b \subseteq T_P$ jest zbiorem wątków realizowanych w b-tym przedziale czasu. Przyjmuje się ponadto, że: $u_1 = \{t_0\}, u_\beta = \{t_0\}$ oraz $\bigcup_{b=1}^{\beta} u_b = T_P$.

Na sekwencję U_P składają się kolejno zbiory u_b zawierające wątki realizowane w b-tym ($b = 1...\beta$) przedziale czasu. Inaczej mówiąc, zbiór u_b zawiera wątki, między którymi może dojść do interakcji prowadzącej do konfliktów zasobowych. Przykładowo zbiór $u_2 = \{t_2, t_3\}$ sekwencji $U_P = (u_1, u_2, u_3)$ oznacza, że wątki t_2, t_3 są realizowane współbieżnie w przedziale czasu u_2 (w aplikacji C_P zidentyfikowano 3 przedziały: u_1, u_2, u_3). Możliwość identyfikacji zbiorów wątków realizowanych współbieżnie jest szczególnie istotna w przypadku lokalizacji błędów prowadzących do szkodliwej rywalizacji. Konflikt ten może wystąpić tylko między wątkami należącymi do tego samego zbioru u_b. W praktyce oznacza to, że jeśli dwa wątki t_a, t_b wykorzystujące wspólny zasób r_c, nie są realizowane w tym samym przedziale czasu (tzn. $t_a \in u_h$, $t_b \in u_g$), wtedy nie jest konieczne stosowanie mechanizmów synchronizacji (np. blokad) tych wątków w dostępie do zasobu r_c. Inaczej mówiąc, architektura kodu źródłowego aplikacji gwarantuje wzajemne wykluczanie się tych wątków w dostępie do zasobu r_c.

Przyjmuje się, że sekwencja U_P zawiera zawsze dwa przedziały czasu u_1 i u_β, w ramach których wykonywane są tylko operacje wątku głównego t_0. Założenie to wynika z praktyki. W aplikacjach wielowątkowych zwykle wątek główny jest odpowiedzialny za inicjację pozostałych wątków (stąd zbiór u_1) oraz realizację instrukcji końcowych, gdy wszystkie wątki zostały już zakończone (stąd zbiór u_2).



RYSUNEK 3.1: Strukturalno-czasowy układ wątków T_P aplikacji wielowątkowej C_P.

Rysunek 3.1 przedstawia graficzną reprezentację zbioru wątków T_P (3.1.1) oraz sekwencji U_P (3.1.2) przykładowej aplikacji C_P. Wiersze przedstawiają przedziały

czasu $u_1, u_2, ..., u_5$, kolumny kolejne wątki $t_0, t_1, t_2, .., t_5$, w ramach których wykonywane są operacje.

Ze względu na potencjalnie liczne interakcje między wątkami aplikacji C_P proces wyznaczania przedziałów czasu sekwencji U_P jest zadaniem dość złożonym i zależnym od wybranego języka programowania. W przypadku języka C identyfikacja przedziałów odbywa się na podstawie wywołań funkcji *pthread_create* oraz funkcji *pthread_join*. Funkcja *pthread_create* rozpoczyna nowy przedział czasu i inicjuje nowy wątek, z kolei funkcja *pthread_join* jawnie kończy pracę wątku (i kończy przedział), którego wskaźnik przekazywany jest jako parametr.

3.1.3 Zasoby współdzielone R_P

Formalnie rodzina zasobów współdzielonych aplikacji C_P definiowana jest następująco:

$$R_P = \{r_c | c = 1...\gamma\}, r_c = \{v_1, ..., v_\gamma\}, \gamma \in \mathbb{N}^+ \tag{3.4}$$

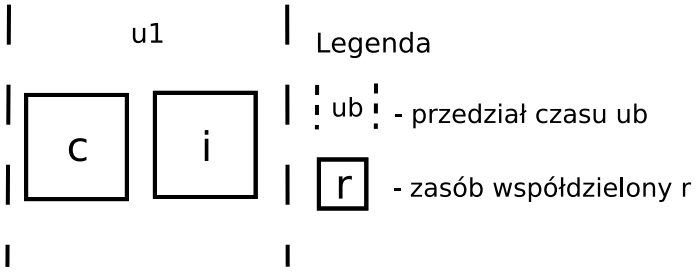
gdzie r_c oznacza c-ty zasób definiowany jako zbiór, na który składają się elementy v_d oznaczające nazwy zmiennych (w tym zmiennych wskaźnikowych) i makrodefinicji, które odnoszą się do zasobu r_c (język C dopuszcza dostęp do zasobu poprzez zmienne wskaźnikowe, które mogą posiadać różne nazwy).

W tym ujęciu rodzina zasobów współdzielonych R_P reprezentuje zbiór wszystkich zmiennych, stałych i definicji preprocesora do stałych wartości, których używa aplikacja. Do zasobów mogą być wliczane także standardowe wejście, standardowe wyjście i standardowe wyjście błędów.

Każdy zasób r_c jest niepodzielny. Dotyczy to również zasobów typu *struktura*. Wyjątkiem w przypadku struktur jest jawne przekazanie wskaźnika do składowej struktury innym wątkom niż wątek macierzysty.

Zasoby współdzielone mogą być *niemodyfikowalne*. Poprzez niemodyfikowalne zasoby należy rozumieć stałe i definicje preprocesora do stałych. Ze względu na to, że tego typu zasoby nie wpływają na występowanie konfliktów zasobowych, nie zostały one wyróżnione w opracowanym modelu.

Na rysunku 3.2 przedstawiono graficzną reprezentację zasobu r_c. Zasób reprezentowany jest poprzez prostokąt umieszczany w przedziale u_k, w którym wykonywany jest wątek wykorzystujący ten zasób. W danym przedziale czasu u_k może umieszczony być tylko jeden symbol zasobu r_c.



RYSUNEK 3.2: Graficzna reprezentacja zasobów umieszczonych w przedziale czasu u_1.

3.1.4 Zbiór operacji O_P

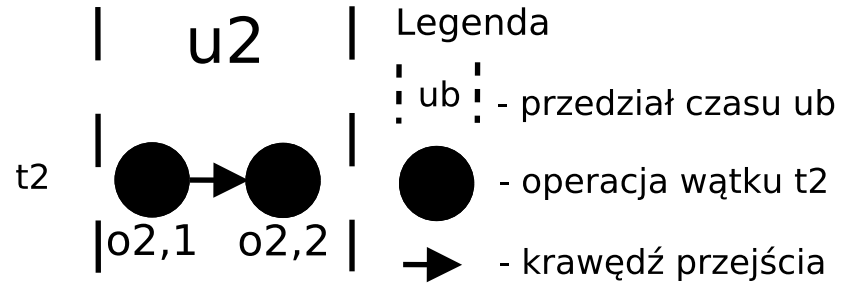
Zbiór operacji aplikacji wielowątkowej C_P definiowany jest następująco:

$$O_P = \{o_{i,j} | i = 1...\alpha, j = 1...\epsilon\}, \alpha = |T_P|, \epsilon \in \mathbb{N}^+ \tag{3.5}$$

gdzie $o_{i,j}$ oznacza j-tą operację wątku t_i.

Zbiór operacji O_P reprezentuje zbiór instrukcji, procedur i funkcji, które znajdują się w kodzie aplikacji wielowątkowej C_P. Operacje $o_{i,j}$ na przyjętym poziomie abstrakcji są operacjami atomowymi tzn. nie da się ich podzielić na mniejsze. Do operacji atomowych należy zaliczyć wszystkie instrukcje języka C i funkcje dostarczone wraz z biblioteką standardową. Jako operacje atomowe należy traktować również funkcje, instrukcje i procedury pochodzące z bibliotek niestandardowych lub rozszerzeń do języka. Założenie to wynika z faktu, że istnieje szereg komercyjnych bibliotek, których kod nie jest publicznie dostępny.

Na rysunku 3.3 przedstawiono graficzną reprezentację dwóch kolejnych operacji jednego z wątków. Graficznie operacje przedstawiane są w postaci czarnego koła z etykietą $o_{i,j}$, gdzie i to indeks wątku, w którym dana operacja jest wykonywana, a j to liczba porządkowa.



RYSUNEK 3.3: Graficzna reprezentacja dwóch kolejnych operacji wątku t_2 w przedziale czasu u_2.

3.1.5 Zbiór blokad Q_P

Formalnie zbiór blokad definiowany jest następująco:

$$Q_P = \{q_s | s = 1...\kappa\}, q_s = (w_s, x_s), \kappa \in \mathbb{N}^+ \tag{3.6}$$

gdzie q_s oznacza s-tą blokadę definiowaną jako para *zmienna, typ blokady*.

Blokada (ang. mutex) to jeden z mechanizmów synchronizacji wątków. W praktyce każda poprawna aplikacja wielowątkowa wykorzystuje mechanizmy synchronizacji wątków w celu uniknięcia konfliktów zasobowych o charakterze wyścigu. Tak zwane *założenie* blokady na zasobie współdzielonym r_c przez wątek t_i zapewnia wykluczenie dostępu do tego zasobu innych wątków. Blokady wykorzystywane są również do zabezpieczenia bibliotek przed ich współbieżną realizacją. Dzieje się tak, gdy zaimplementowany w bibliotece algorytm wykorzystuje szereg zmiennych, pomiędzy którymi istnieją silne relacje.

Biblioteka *pthread* w języku C udostępnia trzy funkcje umożliwiające założenie blokady na zasobie współdzielonym r_c: *pthread_mutex_lock*, *pthread_mutex_trylock* i *pthread_mutex_timedlock*. Funkcje te różnią się zachowaniem w sytuacji, gdy zasób r_c jest niedostępny (ze względu na to, że jest on już zajęty przez inną blokadę). Pierwsza z nich czeka aż blokada zostanie zwolniona przez wątek, który ją aktualnie zajmuje. Druga funkcja w momencie niepowodzenia założenia blokady nie czeka aż ta będzie dostępna, tylko zwraca wartość **EBUSY** w celu kontynuowania pracy wątku, który ją wywołał. Wywołanie ostatniej funkcji skutkuje oczekiwaniem na dostęp do zasobu przez czas określony parametrem wejściowym tej funkcji. Gdy operacja założenia blokady się nie powiedzie, zwracana jest wartość **ETIMEDOUT** i wznawiana jest praca wątku, który oczekiwał na blokadę. Każdy z wątków, który założył

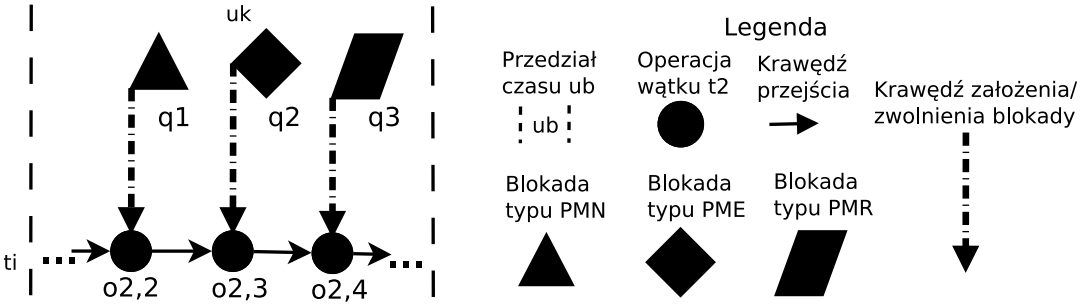
blokadę, musi ją zwolnić poprzez użycie funkcji *pthread_mutex_unlock* niezależnie od tego, jaka funkcja została użyta do założenia blokady.

W niniejszej pracy zakłada się, że blokady zawsze są zakładane z użyciem funkcji *pthread_mutex_lock*. Pozostałe funkcje służące do zakładania blokad tzn. *pthread_mutex_trylock* i *pthread_mutex_timedlock* w momencie, gdy nie mogą wykonać swojego zadania wznawiają dalszą pracę wątku. Takie zachowanie nie prowadzi do rozważanych w pracy konfliktów zasobowych (może być jednak przyczyną innych konfliktów np. żywych zakleszczeń).

Wyróżnia się cztery typy blokad:

- PTHREAD_MUTEX_NORMAL
- PTHREAD_MUTEX_ERRORCHECK
- PTHREAD_MUTEX_RECURSIVE
- PTHREAD_MUTEX_DEFAULT

Typ blokady nie wpływa na współpracę między wątkami. Ma jednak on znaczenie w realizacji pojedynczego wątku w sytuacjach, gdy dochodzi do próby ponownego założenia blokady lub próby zwolnienia blokady już zwolnionej. Uwzględnienie typu blokad ma więc znaczenie w przypadku lokalizacji błędów prowadzących do zakleszczeń.



RYSUNEK 3.4: Graficzna reprezentacja blokad w przedziale czasu u_K.

Każdy typ blokady posiada swój własny symbol w reprezentacji graficznej, co przedstawiono na rysunku 3.4. Blokada typu PMN reprezentowana jest przez symbol trójkąta, blokadę typu PME reprezentuje symbol rombu, zaś blokadę typu PMR przedstawia się za pomocą symbolu trapezu. W implementacji biblioteki *pthread* dla systemu GNU/Linux typ PMD jest aliasem typu PMN, dlatego w niniejszej pracy typ PMD będzie równoznaczny z PMN. W przypadku korzystania z innej implementacji należy wyraźnie zaznaczyć jaki symbol reprezentuje blokady typu PMD.

3.1.6 Zbiór relacji F_P

Zbiór F_P reprezentuje relacje występujące między operacjami zbioru O_P, zasobami zbioru R_P i blokadami Q_P. Formalnie jest on definiowany następująco:

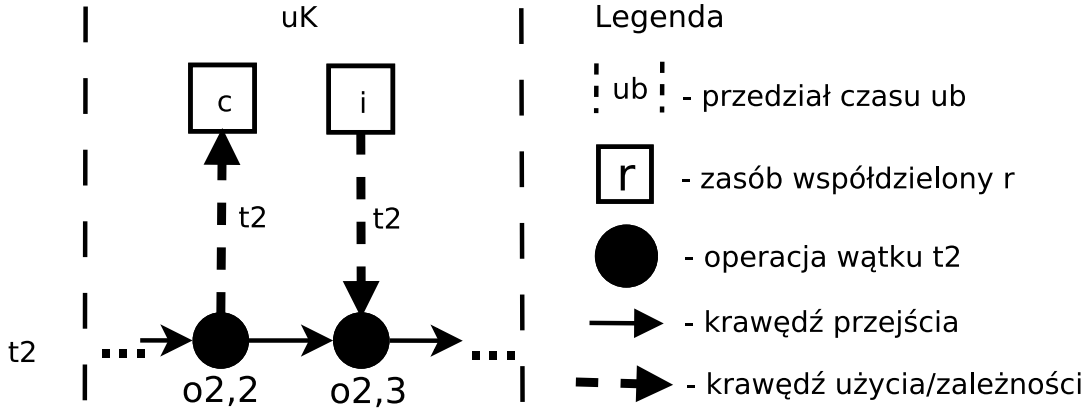
$$\begin{aligned} &F_P = \{f_n | n = 1...\iota\}, \\ &F \subseteq (O_P \times O_P) \cup (O_P \times R_P) \cup (R_P \times O_P) \cup (O_P \times Q_P) \cup (Q_P \times O_P), \\ &\iota \in \mathbb{N}^+ \end{aligned} \tag{3.7}$$

gdzie f_n jest parą oznaczającą n-tą relację między elementami zbiorów O_P, R_P, Q_P. Wyróżnia się cztery rodzaje relacji:

1. relacje przejścia
2. relacje użycia
3. relacje zależności
4. relacje założenia blokady
5. relacje zwolnienia blokady

Ad.1. Relacje przejścia. Relacje przejścia reprezentują porządek wykonywania operacji jednego wątku. Definiowane są one w postaci pary $f_n = (o_{i,j}, o_{i,k})$, której elementami są dwie następujące po sobie operacje $o_{i,j}, o_{i,k} \in O_P$ wykonywane w ramach wątku t_i. W reprezentacji graficznej relacja ta przedstawiana jest w postaci krawędzi skierowanej rysowanej linią ciągłą z pełnym grotem, (rys. 3.3 i 3.4).

Relacja przejścia może wystąpić tylko między operacjami jednego wątku. Język C dopuszcza sytuacje kiedy tego typu relacja obejmuje operacje różnych dwóch wątków. Wymaga ona jednak stosowania do synchronizacji wątków tzw. zmiennych warunkowych (ang. conditional variable). Mechanizmy synchronizacji wykorzystujące blokady i zmienne warunkowe prowadzą do błędów klasy zagubienia sygnału [52], które nie są omawiane w niniejszej pracy.



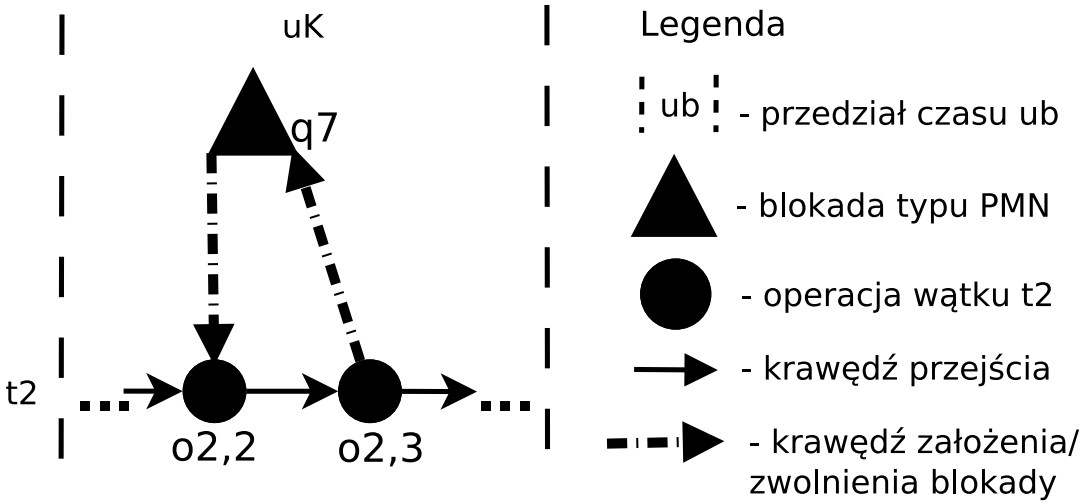
RYSUNEK 3.5: Graficzna reprezentacja krawędzi użycia i krawędzi zależności.

Ad.2. Relacje użycia. Relacja ta wiąże ze sobą operację $o_{i,j} \in O_P$ i zasób $r_c \in R_P$, którego wartość jest zmieniana w wyniku realizacji tej operacji. Relacja ta jest definiowana w postaci pary $f_n = (o_{i,j}, r_c)$ i reprezentowana graficznie w postaci krawędzi skierowanej rysowanej linią przerywaną zakończoną pełnym grotem (rys. 3.5). Przyjmuje się, że krawędź tego typu skierowana jest od symbolu operacji $o_{i,j}$ (koło) do symbolu zasobu r_c (kwadrat). Należy podkreślić, że zasoby r_c będące elementem tej relacji zawsze są zmiennymi (stałe i definicje preprocesora do stałych z zasady nie mogą podlegać zmianie).

Ad.3. Relacje zależności. Relacja ta wiąże ze sobą operację $o_{i,j} \in O_P$ i zasób $r_c \in R_P$, wymagany do jej realizacji. Formalnie jest ona definiowana jako para $f_n = (r_c, o_{i,j})$. Zasoby r_c będące elementem tej relacji nie są modyfikowane, a więc mogą to być zarówno zmienne, stałe jak i definicje preprocesora do stałych. Graficznie relacja ta reprezentowana jest przez krawędź skierowaną (tak jak w przypadku relacji użycia) od zasobu r_c do operacji $o_{i,j}$ (rys. 3.5).

Ad.4. Relacje założenia blokady. Relacja ta wiąże ze sobą operację $o_{i,j}$ założenia blokady dla wątku t_i z blokadą q_s (blokadą, która ma zapewnić brak dostępu

innych w wątków do zasobów wykorzystywanych przez wątek t_i). Formalnie relacja ta definiowana jest następująco: $f_n = (q_s, o_{i,j})$, gdzie $q_s \in Q_P$ i $o_{i,j} \in O_P$. W języku C (biblioteka *pthread*) operacja założenia blokady sprowadza się do wywołania jednej z następujących funkcji: *pthread_mutex_lock*, *pthread_mutex_trylock* i *pthread_mutex_timedlock*. Relacja założenia blokady reprezentowana jest przez krawędź skierowaną (linia -.- zakończona pełnym grotem) prowadzącą od symbolu blokady q_s do symbolu operacji $o_{i,j}$ - rysunek 3.6.



RYSUNEK 3.6: Graficzna reprezentacja krawędzi założenia blokady i krawędzi zwolnienia blokady.

Ad.5. Relacje zwolnienia blokady. Relacja ta (analogicznie jak poprzednia) wiąże ze sobą operację $o_{i,j}$ zwolnienia blokady wątku t_i z blokadą q_s. Relacja ta definiowana jako para: $f_n = (o_{i,j}, q_s)$, gdzie $q_s \in Q_P$ i $o_{i,j} \in O_P$. Relacja zwolnienia blokady reprezentowana jest przez krawędź skierowaną (linia -.- zakończona pełnym grotem) prowadzącą od symbolu operacji $o_{i,j}$ do symbolu blokady q_s - rys. 3.6.

Operacje wątku t_i występujące pomiędzy operacjami założenia i zwolnienia blokady składają się na *sekcję krytyczną*. Oznacza to że dostęp do zasobów przez operacje sekcji krytycznej jest możliwy w trybie wzajemnego wykluczania.

3.1.7 Sekwencja zbiorów relacji niepodzielności B_P

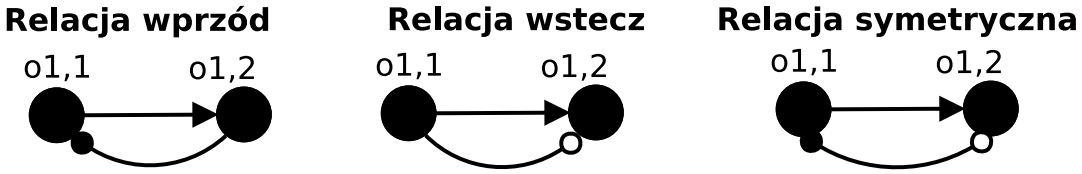
W aplikacjach wielowątkowych relacje pomiędzy dwiema operacjami $o_{i,j}, o_{i,k}$, jakie można odczytać z kodu źródłowego, ograniczają się do kolejności ich wykonywania (relacje przejścia $(o_{i,j}, o_{i,k}) \in O_P$). W dokumentacji języka C (jak i w dostępnych bibliotekach) znajdują się informacje o tym, że niektóre operacje są powiązane dodatkowymi relacjami niepodzielności (tzn. pomiędzy wywołaniami tych funkcji nie może dojść do wykonania operacji innych wątków). Przykładem takich operacji są funkcje **malloc** i **calloc**, których liczba wywołań musi być równa liczbie wywołań funkcji **free** w innym wypadku dojdzie do tzw. wycieków pamięci (ang. memory

leak). Przykłady innych funkcji dla standardowej biblioteki języka C zostały przedstawione w tabeli 3.1. Formalnie zbiór relacji niepodzielności jest definiowany następująco:

$$\begin{aligned} &B_P = (B_P^{FWD}, B_P^{BWD}, B_P^{SYM}), \\ &B_P^{\xi} = \{b_m | m = 1...\mu\}, \xi \in \{FWD, BWD, SYM\}, \\ &B_P^{\xi} \subseteq (O_P \times O_P), \\ &\mu \in \mathbb{N}^+ \end{aligned} \tag{3.8}$$

gdzie b_m oznaczają m-tą relację niepodzielności. Wyróżnia się trzy rodzaje relacji b_m (rysunek 3.7).

- **wprzód** (ang. forward, FWD) - relacja, w której po wykonaniu operacji $o_{i,j}$ zawsze musi nastąpić wykonanie operacji $o_{k,l}$: $b_m = (o_{i,j}, o_{k,l}) \in B_P^{FWD}$;
- **wstecz** (ang. backward, BWD) - relacja, w której operacja $o_{i,j}$ zawsze musi zostać wykonana nim wykonana zostanie operacja $o_{k,l}$: $b_m = (o_{k,l}, o_{i,j}) \in B_P^{BWD}$;
- **symetryczna** (ang. symmetric, SYM) - relacja, w której wykonanie operacji $o_{i,j}$ zawsze musi poprzedzać wykonanie operacji $o_{k,l}$, a operacja $o_{k,l}$ zawsze musi następować po operacji $o_{i,j}$: $b_m = (o_{k,l}, o_{i,j}) \in B_P^{SYM}$;



RYSUNEK 3.7: Krawędzie odzwierciedlające relację między operacjami.

TABLICA 3.1: Przykładowe relacje opracowane na podstawie opisu funkcji biblioteki standardowej języka C [46].

Wprzód	Wstecz	Symetryczna
	fgetpos i strerror	
	fsetpos i strerror	
	ftell i strerror	
	atof i strerror	
calloc i free	strtod i strerror	va_start i va_arg
malloc i free	strtol i strerror	va_arg i va_end
	strtoul i strerror	
	calloc i realloc	
	malloc i realloc	
	srand i rand	

W przeprowadzonym przeglądzie literatury, uwzględniającym również przegląd dokumentacji technicznych, nie znaleziono dotychczas mechanizmu, za pomocą którego możliwe jest zadeklarowanie w kodzie aplikacji przedstawionych powyżej relacji. Informację o tym, że dana para operacji jest powiązana relacją niepodzielności, można znaleźć w dokumentacji bibliotek. Potocznie mówi się, że są

to relacje, których *nie widać*. Biblioteka standardowa języka C posiada wiele zdefiniowanych funkcji, które mogą wchodzić między sobą w jedną z trzech wymienionych wcześniej relacji. Większość relacji przedstawionych w tabeli 3.1 wynika bezpośrednio z dokumentacji języka C. W przypadku par *calloc, free* i *malloc, free* relacja wynika ze sposobu zarządzania pamięcią w języku C. Czasami relacja niepodzielności może wynikać z kontekstu i nie jest ona regułą np. ustawienie nowej wartości jednemu zasobowi współdzielonemu, wiąże się z wyzerowaniem innego zasobu współdzielonego będącego licznikiem, ale tylko wtedy kiedy wartość ta jest ustawiana w wybranym z wątków.

3.1.8 Grafowa reprezentacja modelu C_P

Na potrzeby wprowadzonych w kolejnym rozdziale warunków umożliwiających lokalizację błędów prowadzących do konfliktów zasobowych wprowadzono graficzną reprezentację opracowanego modelu C_P. W tym kontekście wraz z modelem C_P skojarzone jest pojęcie *grafu operacji* G_P, odzwierciedlającym relacje między operacjami wątków zbioru T_P, a zbiorami R_P i Q_P. Definicja grafu G_P prezentuje się następująco:

$$G_P = (V_P, E_P) \tag{3.9}$$

gdzie:

$V_P = O_P \cup Q_P \cup R_P$ oznacza zbiór wierzchołków, na który składają się operacje aplikacji O_P, blokady aplikacji Q_P oraz zasoby współdzielone R_P,

$E_P = F_P \cup B_P$ oznacza zbiór krawędzi skierowanych grafu G_P, na który składają się relacje zdefiniowane w ramach zbiorów F_P i B_P.

Poprzez ${}^{s}_{i}G_P$ rozumie się podgraf grafu G_P, który budowany jest na skutek usunięcia:

- wszystkich wierzchołków zbioru O_P i związanych z nimi krawędzi z wykluczeniem operacji wątku t_i,
- wszystkich wierzchołków zbioru Q_P i związanych z nimi krawędzi z wyłączeniem wierzchołka q_s,
- wszystkich wierzchołków zbioru R_P i związanych z nimi krawędzi z wyłączeniem krawędzi między operacjami wątku t_i, a zasobami przez nie wykorzystywanymi,
- krawędzi przejścia $(o_{i,j}, o_{i,k})$ dla wierzchołka $o_{i,j}$ gdy dla tego wierzchołka istnieje krawędź zwolnienia blokady $(o_{i,j}, q_s)$,
- krawędzi niepodzielności zbioru B_P.

W grafie operacji G_P wyróżnia się ścieżki zbudowane z wierzchołków operacji O_P i blokad Q_P. Niech zatem $\lambda^{P,i}$ jest zbiorem wszystkich ścieżek wątku t_i rozpoczynając z operacji $o_{i,1}$. Formalnie a-tą *ścieżkę* zbioru $\lambda^{P,i}$ definiuje się jako sekwencję:

$$\lambda_a^{P,i} = (\lambda_{a,1}^{P,i}, \lambda_{a,2}^{P,i}, ..., \lambda_{a,b}^{P,i}, ..., \lambda_{a,q}^{P,i}),$$
$$\text{gdzie}$$
$$\lambda_{a,1}^{P,i} = o_{i,1}, \lambda_{a,b}^{P,i} \in O_P \cup Q_P,$$
$$(\lambda_{a,k}^{P,i}, \lambda_{a,k+1}^{P,i}) \in F_P, \lambda_{a,k}^{P,i} \neq \lambda_{a,l}^{P,i}$$
$$\text{dla}$$
$$k \neq l,\ l,k = 1...q$$

W dalszej części pracy przyjęto zapis:

- $\lambda_{a,b}^{P,i} \lhd \lambda_a^{P,i}$ jeśli element $\lambda_{a,b}^{P,i}$ składa się na ścieżkę $\lambda_a^{P,i}$
- $\lambda_{a,b}^{P,i} \not\lhd \lambda_a^{P,i}$ jeśli element $\lambda_{a,b}^{P,i}$ nie składa się na ścieżkę $\lambda_a^{P,i}$.

Ponadto przyjmuje się, że ścieżka $\lambda_a^{P,i}$ jest *ścieżką cykliczną* jeśli spełniony jest warunek: $\exists_{\lambda_{a,b}^{P,i} \lhd \lambda_a^{P,i}} \lambda_{a,b}^{P,i} = \lambda_{a,q}^{P,i}, b \neq q$. Zbiór zawierający wszystkie ścieżki cykliczne wątku t_i oznaczany jest przez $C\lambda^{P,i} \subseteq \lambda^{P,i}$

Z pojęciem ścieżki skojarzona jest następująca definicja:

Definicja 7. *Niech w ścieżce $\lambda_l^{P,i}$ istnieją dwie takie operacje $o_{i,a} \lhd \lambda_l^{P,i}$ oraz $o_{i,b} \lhd \lambda_l^{P,i}$ w ramach, których dochodzi do założenia odpowiednio blokady q_c oraz q_d (tzn. w F_P istnieją krawędzie $(o_{i,a}, q_c)$ oraz $(o_{i,b}, q_d)$). Mówi się, że blokada q_c poprzedza blokadę q_d w ścieżce $\lambda_l^{P,i}$, jeśli $o_{i,a}$ poprzedza $o_{i,b}$ ($o_{i,a} \prec o_{i,b}$). Relację tę oznacza się $q_c \prec_i^l q_d$.*

3.2 Problem lokalizacji błędów prowadzących do konfliktów zasobowych

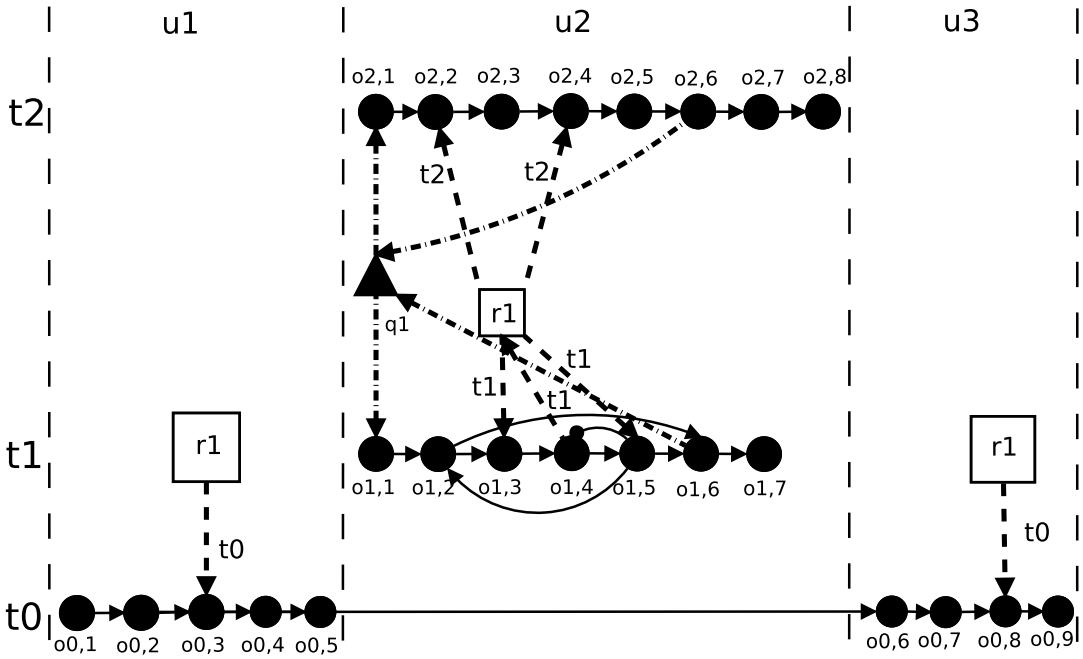
Przykład modelu kodu źródłowego C_P dla aplikacji wielowątkowej *EG* z rys. 3.8 został przedstawiony na rys. 3.9. W aplikacji tej występują trzy wątki. Poza wątkiem głównym (t_0 wykonywanym w przedziałach u_1 i u_3) w aplikacji równolegle wykonywane są operacje wątków potomnych (t_1 i t_2 w przedziale czasu u_2). Wątek główny składa się tylko z kilku operacji:

- deklaracji wskaźników do wątków (operacje $o_{0,1}$ i $o_{0,2}$),
- zaprezentowania wartości początkowej zasobu współdzielonego r_1 (operacja $o_{0,3}$),
- uruchomienia wątków potomnych (utworzenie przedziału czasu u_2) w celu wykonania zdefiniowanych zadań na zasobie współdzielonym r_1 (operacje $o_{0,4}$ i $o_{0,5}$),
- odebrania informacji z wątków potomnych i zakończenie ich pracy (operacje $o_{0,6}$ i $o_{0,7}$, utworzenie przedziału czasu u_3),
- zaprezentowania wartości końcowej zasobu współdzielonego r_1 (operacja $o_{0,8}$),
- zakończenia pracy aplikacji (operacja $o_{0,9}$).

```
static volatile int r1 = 0;
pthread_mutex_t q1;

void* thread_1_code(void *args) {
    pthread_mutex_lock(&q1);
    for(;r1 < 10; ++r1) {
        printf("Resource value: %d\r\n", r1);
    }
    pthread_mutex_unlock(&q1);
    return NULL;
}

void* thread_2_code(void *args) {
    pthread_mutex_lock(&q1);
    int y = 2*pow(r1, 2) + r1;
    printf("Calculation result: %d\r\n", y);
    pthread_mutex_unlock(&q1);
    return NULL;
}

int main() {
    pthread_t t1, t2;
    printf("App start work with r1 = %d\r\n", r1);

    pthread_create(&t1, NULL, thread_1_code, NULL);
    pthread_create(&t2, NULL, thread_2_code, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("App finish work with r1 = %d\r\n", r1);
    return 0;
}
```

RYSUNEK 3.8: Kod źródłowy przykładowej aplikacji wielowątkowej EG.

Wątki t_1 i t_2 wykonują swoje operacje, wykorzystując zasób współdzielony r_1. Pierwszą operacją wątku t_1 (operacja $o_{1,1}$) jest operacja założenia blokady (blokada q_1 typu PMD). Następną operacją jest pętla (operacja $o_{1,2}$), której warunkiem zakończenia jest osiągniecie przez zasób wartości 10 (operacja $o_{1,3}$), który to zasób zwiększany jest co krok o jeden (operacja $o_{1,4}$). W ciele pętli co krok wyświetlana jest aktualna wartość zasobu (operacja $o_{1,5}$). Pomiędzy operacjami $o_{1,4}$ i $o_{1,5}$ istnieje relacja kolejnościowa wprzód. W kodzie aplikacji *EG* nie jest zawarta informacja o tej relacji, jednak programista aplikacji wie o niej i zaznaczył ją tworząc model instancji. Relacja ta wynika niejawnie z wymagań, jakie musi spełniać aplikacja. Dalej poza ciałem pętli znajduje się operacja $o_{1,6}$ zwalniająca wcześniej założoną blokadę i operacja $o_{1,7}$ będąca operacją wyjścia. Warto także zauważyć, że operacje $o_{1,3}$ i $o_{1,5}$ nie zmieniają zasobu, tylko odczytują jego wartość. Fakt ten powoduje, że strzałka prowadzi od zasobu do operacji. Wątek t_2 podobnie jak wątek t_1 rozpoczyna się od założenia blokady q_1. Dalej w kodzie źródłowym znajduje się wyrażenie matematyczne „$y = 2 * r1^2 + r1$” składające się z kilku operacji, a którego wynik przypisywany jest do deklarowanej zmiennej „y”. Interpretacja tego wyrażenia zaczyna się od operacji potęgowania zasobu r_1 ($o_{2,2}$), następnie wykonywane jest mnożenie otrzymanej wartości przez dwa (operacja $o_{2,3}$) i dodawanie wartości r_1 (operacja $o_{2,4}$). Wynik tych operacji zapisywany jest do zmiennej (operacja $o_{2,6}$), która uprzednio musi zostać zadeklarowana (operacja $o_{2,5}$). Ostatnią operacją $o_{2,8}$ jest operacja wyjścia.



RYSUNEK 3.9: Graficzna reprezentacja instancji modelu kodu źródłowego aplikacji EG.

Powyższy przykład pokazuje jak wykorzystać zaproponowany model C_P do zilustrowania wzajemnych relacji (czasowych na poziomie przedziałów U_P i strukturalnych na poziomie wątków T_P) występujących w aplikacji wielowątkowej P. Model ten może zostać wykorzystany w procesie lokalizacji błędów prowadzących do rozważanej klasy konfliktów zasobowych. W tym kontekście sformułowany na wstępie pracy problem badawczy wyrażony w reprezentacji modelu C_P jest formułowany następująco:

Dana jest aplikacja wielowątkowa P, której kod napisano w języku C z użyciem biblioteki *pthread*. Znany jest zbiór wątków T_P, zbiór operacji O_P składających się na te wątki, rodzinę współdzielonych zasobów R_P oraz zbiory relacji F_P i B_P. Synchronizacja pracy wątków T_P odbywa się przy użyciu blokad zdefiniowanych w zbiorze Q_P. Poszukiwane są warunki, spełnienie których pozwala na identyfikację w aplikacji P błędów prowadzących do konfliktów zasobowych typu: szkodliwa rywalizacja, zakleszczenie, naruszenie porządku i naruszenie niepodzielności.

3.3 Podsumowanie

W rozdziale przedstawiono model kodu źródłowego aplikacji wielowątkowej. Zaproponowana reprezentacja „strukturalno-czasowa" umożliwia identyfikację potencjalnych interakcji między wątkami analizowanej aplikacji, a następnie ich ocenę pod względem możliwości wystąpienia konfliktu zasobowego. Na opracowany model C_P składa się siedem elementów reprezentujących niezbędne relacje między operacjami O_P wątków T_P, blokadami Q_P, zasobami współdzielonymi R_P, itd. Przyjętą strukturę modelu można w sposób naturalny wyrazić w postaci grafu operacji. Tego typu reprezentacja pozwala na ilustrowanie często złożonych aplikacji z pominięciem żmudnej analizy zapisu formalnego.

Model ten w kolejnym rozdziale posłuży jako podstawa do opracowania warunków lokalizowania konfliktów zasobowych w aplikacjach wielowątkowych.

Rozdział 4

Warunki lokalizowania konfliktów zasobowych

Proces lokalizowania błędów prowadzących do konfliktów zasobowych typu: szkodliwa rywalizacja, zakleszczenie, naruszenie niepodzielności i naruszenie porządku wymaga opracowania warunków ich występowania. W tym celu wykorzystany został, zaprezentowany w poprzednim rozdziale, model aplikacji wielowątkowej C_P.

Opracowany zbiór warunków umożliwia identyfikację cech strukturalnych modelowanego kodu źródłowego aplikacji, które potencjalnie mogą prowadzić do zachowań aplikacji skutkujących wystąpieniem konfliktów zasobowych. Opracowane warunki są *konieczne* dla wystąpienia rozważanej klasy błędów, ale nie są *wystarczające*. Ich spełnienie nie zawsze oznacza więc wystąpienie błędu. W praktyce własność ta objawia się możliwością występowania zgłoszeń błędów fałszywie pozytywnych. Innymi słowy metoda implementująca opracowane warunki umożliwia lokalizację rozważanych błędów z pewną nadmiarowością.

4.1 Szkodliwa rywalizacja

W dodatku A (listing 1) znajduje się kod źródłowy aplikacji wielowątkowej o nazwie RC1 (której kod napisano w języku C z użyciem biblioteki *pthread*). W aplikacji występuje błąd prowadzący do szkodliwej rywalizacji. Konsekwencją tego błędu jest niepoprawna wartość zmiennej r_1, cyklicznie zwiększana przez operacje inkrementacji wątków t_1 i t_2.

Graf reprezentujący kod źródłowy tej aplikacji znajduje się na rysunku 4.1. Łatwo zauważyć, że wartość zasobu r_1, jest równolegle zmieniana przez operacje $o_{1,4}$ i $o_{2,4}$ (inaczej mówiąc operacje $o_{1,4}$ i $o_{2,4}$ powiązane są z r_1 relacją użycia). Obie te operacje wzajemnie się nie wykluczają, gdyż znajdują się w tym samym przedziale czasu u_2, a także żadnej z nich nie poprzedza operacja założenia blokady (zapewniająca wzajemne wykluczanie operacji). Taka struktura aplikacji umożliwia zatem, aby obie operacje zostały wykonywane równolegle i jedna z nich nadpisała wynik działania drugiej.

Bazując na tym przykładzie oraz wprowadzonej w podrozdziale 2.1 definicji szkodliwej rywalizacji (definicja 1) można opracować warunek, spełnienie którego świadczy o wystąpieniu w kodzie źródłowym aplikacji błędu prowadzącego do tego typu konfliktu.

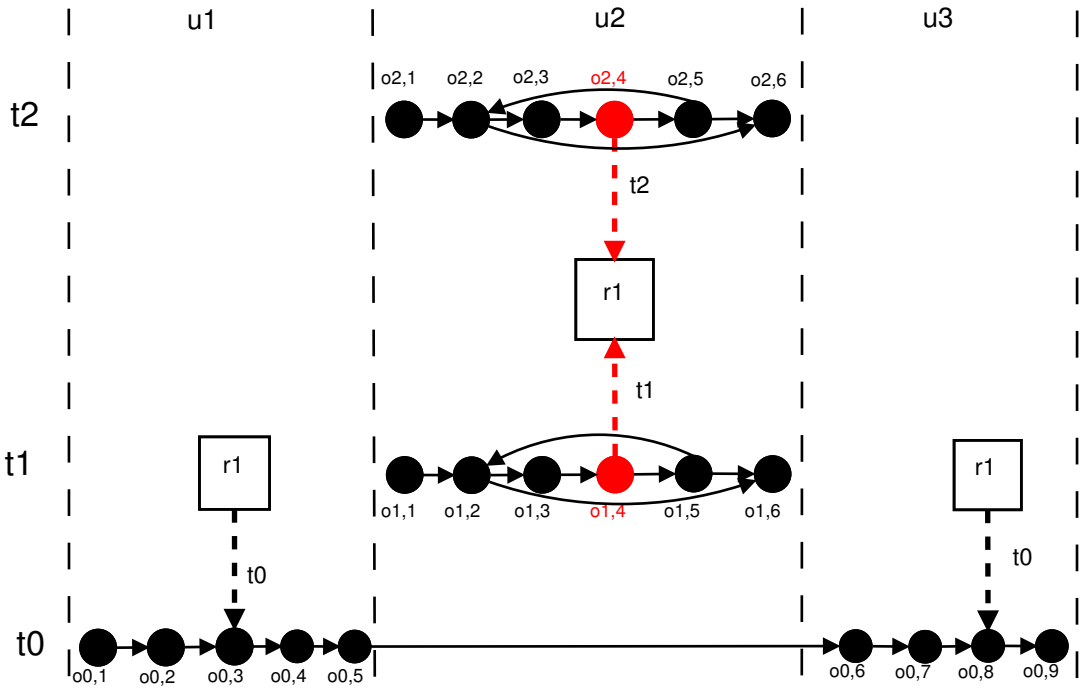
Twierdzenie 1. *Niech $O_P = \{o_{m,j}, ..., o_{i,j}, ..., o_{a,b}\}$ oznacza zbiór operacji wątków realizowanych w przedziale $u_b \in U_P$, współdzielących zasób: $r_c \in R_P$ (tzn. istnieją w grafie G krawędzie użycia $(o_{m,j}, r_c) \in F_P$ lub krawędzie zależności $(r_c, o_{m,j}) \in F_P$, ..., lub $(o_{a,b}, r_c) \in F_P$ lub $(r_c, o_{a,b}) \in F_P$).*

Jeżeli istnieje taka operacja $o_{i,j} \in O_P$ wykorzystująca zasób r_c, która:

(i) nie jest elementem ścieżki cyklicznej $\lambda_a^{P,i} \in C\lambda^{P,i}$ *(tzn.* $o_{i,j} \notin \lambda_a^{P,i}$*) lub*

(ii) jest elementem ścieżki cyklicznej $\lambda_a^{P,i} \in C\lambda^{P,i}$ *(tzn.* $o_{i,j} \lhd \lambda_a^{P,i}$*), ale nie poprzedza jej operacja założenia blokady* q_n

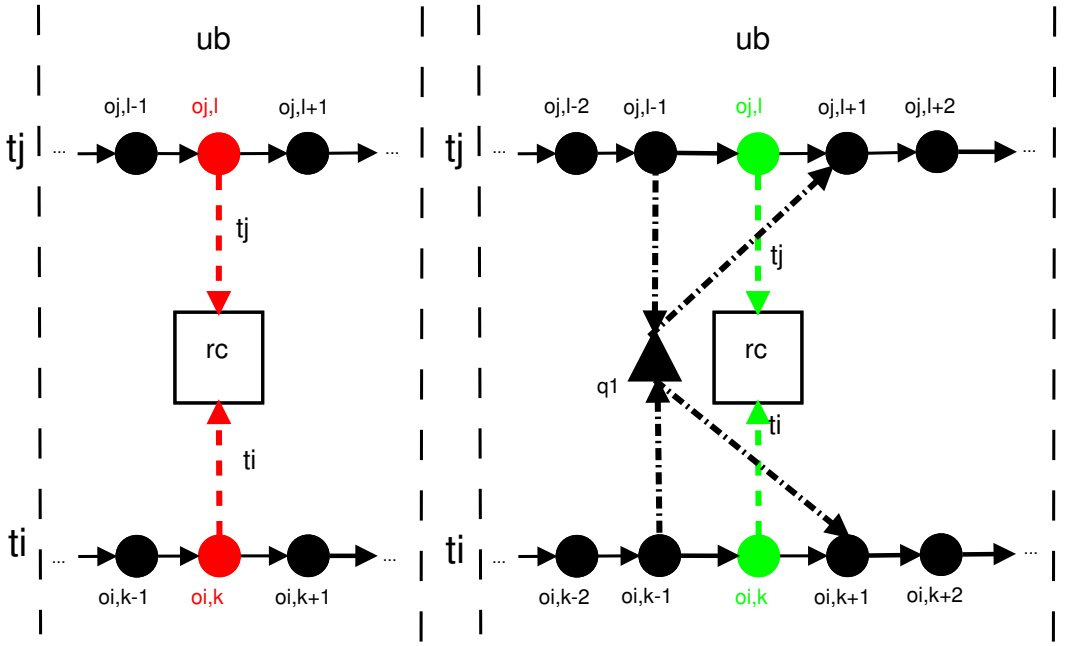
to realizacja tej operacji może doprowadzić do szkodliwej rywalizacji.



RYSUNEK 4.1: Graf operacji aplikacji RC1.

Dowód. Niech aplikacja P posiada dwa wątki – t_i i t_j realizowane w przedziale czasu u_b. W ramach wątku t_i realizowana jest operacja $o_{i,k}$. Analogicznie w ramach wątku t_j realizowana jest operacja $o_{j,l}$. Obie operacje korzystają z zasobu współdzielonego r_c. Szkodliwa rywalizacja między wątkami t_i i t_j wystąpi, gdy operacjom $o_{i,k}$ i $o_{j,l}$ umożliwi się swobodny dostęp do zasobu współdzielonego r_c (patrz rys. 4.2 (A)). Wzajemne wykluczanie się operacji jest gwarantowane poprzez umieszczenie blokad (dla każdego wątku współdzielonego zasób r_c), których sekcje krytyczne zawierają operacje $o_{i,k}$ i $o_{j,l}$. W opracowanym modelu sytuacja ta ma miejsce gdy operacja $o_{i,k}$ (analogicznie $o_{j,l}$) jest elementem ścieżki cyklicznej zawierającej blokadę (patrz rys. 4.2 (B)). Swobodny dostęp (szkodliwa rywalizacja) do zasobu współdzielonego r_c przez wątki t_i i t_j jest więc możliwy, gdy co najmniej jedna operacja ($o_{i,k}$ lub $o_{j,l}$) nie jest elementem ścieżki cyklicznej (i) lub jest elementem ścieżki cyklicznej, który nie zawiera operacji założenia blokady (ii). c.k.d.

Podsumowując, do błędu prowadzącego do szkodliwej rywalizacji może dojść tylko między wątkami realizowanymi we wspólnym przedziale u_b. Jest to możliwe, gdy operacje tych wątków nie wykluczają się wzajemnie tzn. istnieje co najmniej jedna operacja, która nie jest „chroniona" przez blokadę. Pamiętać należy przy tym, że stosowanie blokad umożliwia uniknięcie błędów szkodliwej rywalizacji, ale ich nadmiar może prowadzić do błędów skutkujących zakleszczeniem wątków.



(A) Struktura wątków prowadząca do szkodliwej rywalizacji.

(B) Struktura wątków wykluczająca wystąpienie szkodliwej rywalizacji.

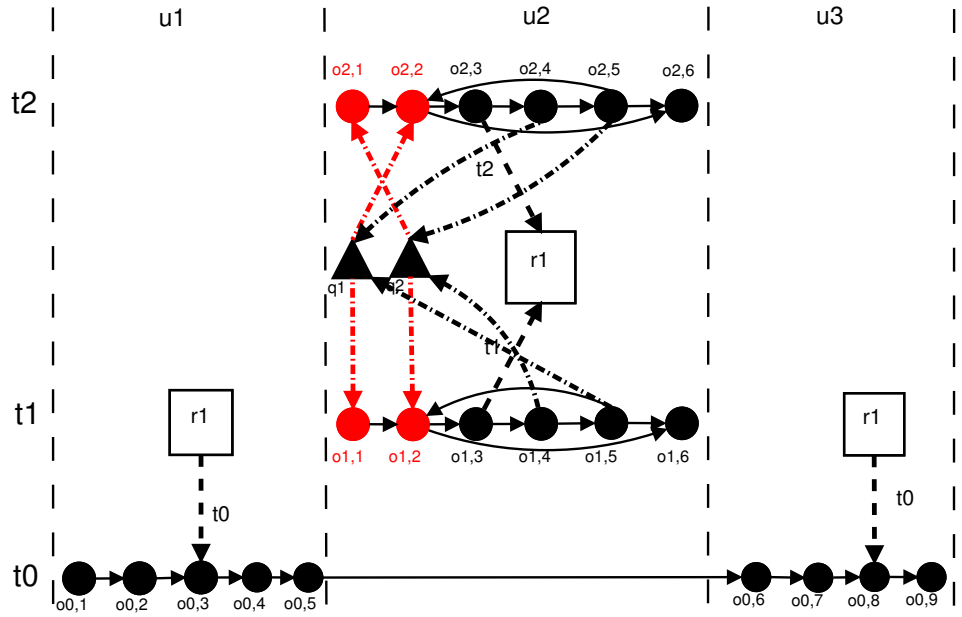
RYSUNEK 4.2: Graficzna reprezentacja błędu prowadzącego do szkodliwej rywalizacji (A) i poprawki wykluczające wystąpienie tego zjawiska (B).

4.2 Zakleszczenie

Błędy prowadzące do zakleszczeń (definicja 4, podrozdział 2.2) są znacznie bardziej złożone niż błędy prowadzące do szkodliwej rywalizacji. W podrozdziale 2.2 wyróżniono cztery ich rodzaje: DL1, DL2, DL3 i DL4 [36]. Przykłady aplikacji, w których dochodzi do błędów skutkujących zakleszczeniem, umieszczono w dodatku A kolejno na listingach 2, 3, 4 i 5. Grafy operacji reprezentujące te aplikacje znajdują się z kolei na rysunkach 4.3, 4.5, 4.6 i 4.9. Przyjęta reprezentacja umożliwia lokalizację wszystkich rozważanych błędów:

- W aplikacji nr 2 błąd DL1 spowodowany jest wzajemnie wykluczającymi się parami blokad (rys. 4.3).
- Aplikacja nr 3 dopuszcza taki przebieg, w którym operacja zwolnienia blokady zostaje pominięta w jednym z wątków, przez co drugi z nich będzie oczekiwać na jej zwolnienie w nieskończoność (błąd DL2 - rys. 4.5).
- Aplikacja nr 4 została zaprojektowana w taki sposób, że możliwa jest ponowna próba wykonania operacji założenia blokady w wyniku działania pętli bez uprzedniego jej zwolnienia przez wątek, w skutek czego wątek oczekuje sam na siebie (błąd DL3 - rys. 4.6).
- W aplikacji nr 5 także dochodzi do ponownej próby wykonania operacji założenia blokady, z tą różnicą, że operacja ta znajduje się w ciele funkcji rekurencyjnej (błąd DL4 - rys. 4.9). Skutkiem tego wątek, którego funkcja wywołuje samą siebie, także oczekuje w nieskończoność.

Ze względu na to, że rozważane błędy różnią się przyczyną ich wystąpienia dla każdego z nich opracowano oddzielny warunek ich lokalizacji.



RYSUNEK 4.3: Graf kodu źródłowego aplikacji DL1 z konfliktem spowodowanym wzajemnie wykluczającymi się parami blokad.

Warunek wystąpienia błędu DL1

Lemat 1. *Niech $T_P = \{t_0, ..., t_\alpha\}$ oznacza zbiór wątków wykorzystujących blokady $Q_P = (q_1, ..., q_\kappa)$ aplikacji wielowątkowej P. Dane są dwa wątki $t_i \in T_P$ oraz $t_j \in T_P$, których znane są zbiory $\lambda^{P,i}$, $\lambda^{P,j}$ ścieżek zbudowanych z operacji tych wątków. W obu wątkach istnieją operacje zakładające blokady $q_c \in Q_P$ oraz $q_d \in Q_P$.*

Jeśli istnieje taka para ścieżek: $\lambda_l^{P,i} \in \lambda^{P,i}, \lambda_k^{P,j} \in \lambda^{P,j}$ dla których spełniony jest warunek: $(q_c \prec_i^l q_d) \wedge (q_d \prec_j^k q_c)$ lub $(q_c \prec_j^k q_d) \wedge (q_d \prec_i^l q_c)$, to w kodzie źródłowym aplikacji P może wystąpić błąd prowadzący do zakleszczenia DL1 przy udziale wątków t_i i t_j.

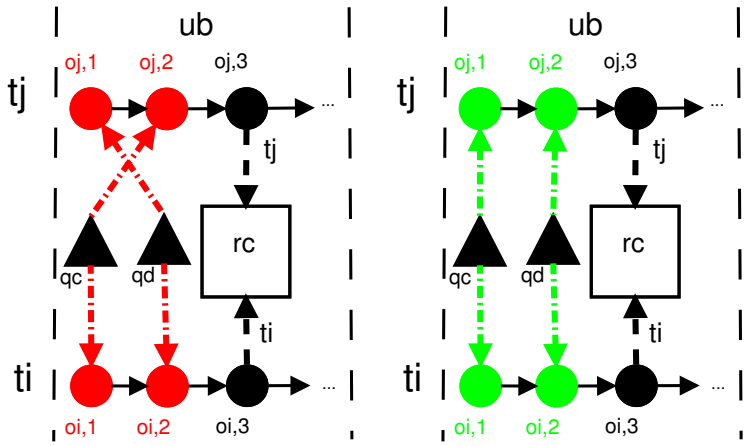
Dowód. Bazując na instancji modelu aplikacji nr 2, warunkiem koniecznym do wystąpienia błędu DL1 prowadzącego do zakleszczenia z udziałem wątków t_i i t_j jest istnienie sekwencji operacji zakładających blokady q_c and q_d w odwrotnej kolejności w każdym z tych wątków. Warunek ten jest więc spełniony, gdy w wątku t_i blokada q_c poprzedza blokadę q_b (zgodnie z def. 7), a w wątku t_j blokada q_b poprzedza blokadę q_c tzn. $(q_c \prec_i^l q_d) \wedge (q_d \prec_j^k q_c)$ lub odwrotnie $(q_c \prec_j^k q_d) \wedge (q_d \prec_i^l q_c)$ (patrz rys. 4.4 (A)). c.k.d.

Jak łatwo zauważyć, dla instancji modelu z rys. 4.3, Lemat 1 jest spełniony dla wątków t_1 i t_2. Operacje $o_{2,1}$ i $o_{2,2}$ zakładają blokady w porządku odwrotnym niż operacje $o_{1,1}$ i $o_{1,2}$.

Warunek wystąpienia błędów DL2 i DL3

Lemat 2. *Dany jest graf operacji G_P aplikacji P, znany jest również zbiór wątków $T_P = \{t_0, ..., t_\alpha\}$ oraz zbiór blokad $Q_P = (q_1, ..., q_\kappa)$. Jeśli w podgrafie ${}_i^s G_P$ (podgraf dla wątku $t_i \in T_P$ i blokady $q_s \in Q_P$) istnieje ścieżka, która rozpoczyna się od blokady q_s i nie zawiera tej*

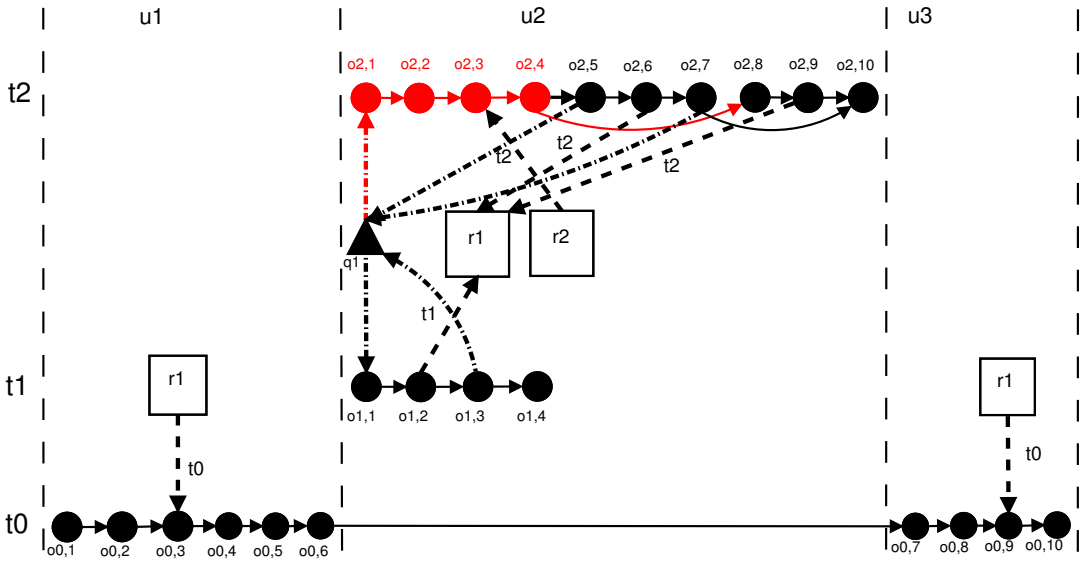
samej liczby operacji założenia i zwolnienia blokady q_s, to w aplikacji P może dojść do błędu, którego skutkiem jest zakleszczenia DL2/DL3 przy udziale wątku t_i.



(A) Struktura wątków prowadząca do zakleszczenia DL1.

(B) Struktura wątków wykluczająca wystąpienie zakleszczenia DL1.

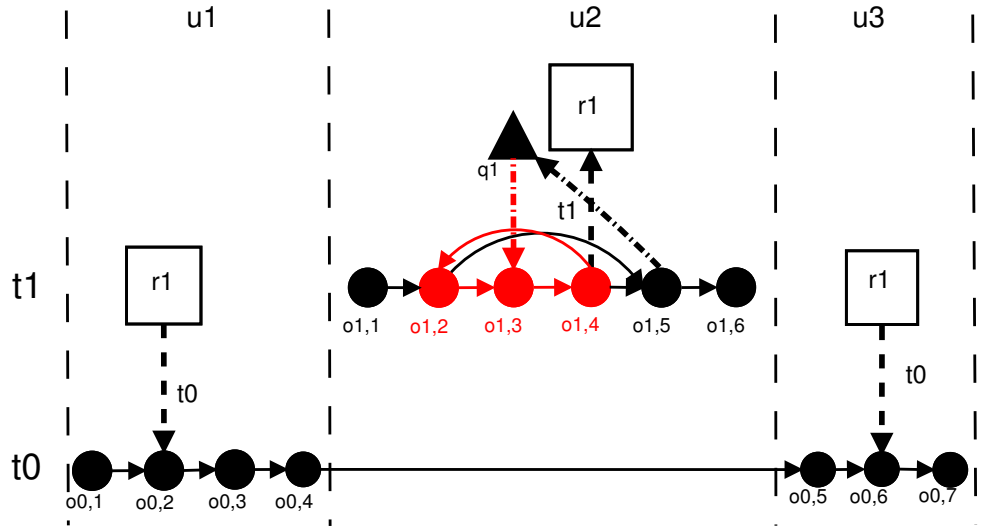
RYSUNEK 4.4: Graficzna reprezentacja błędu prowadzącego do zakleszczenia DL1 (A) i poprawki wykluczające wystąpienie tego zjawiska (B).



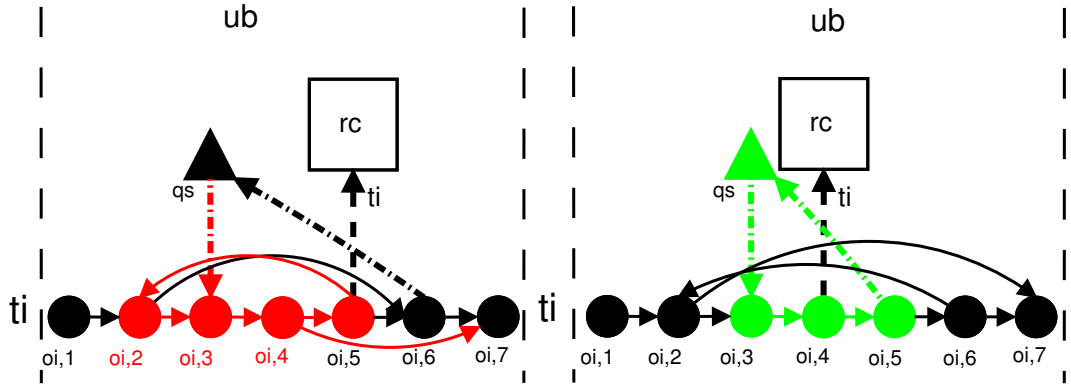
RYSUNEK 4.5: Graf aplikacji DL2 z konfliktem spowodowanym pominięciem operacji zwolnienia blokady.

Dowód. Warunkiem koniecznym do wystąpienia konfliktu zasobowego prowadzącego do błędu DL2 oraz DL3 w wątku t_i jest istnienie sekwencji operacji, których skutkiem jest brak zwolnienia wcześniej założonej blokady q_s. Warunek ten jest więc spełniony, gdy w zbiorze $\lambda^{P,i}$ aplikacji P istnieje ścieżka rozpoczynająca się od q_s (założenie blokady) i niezawierająca operacji zwolnienia blokady q_s (patrz rys. 4.7 (A)). c.k.d.

W modelu z rys. 4.5, Lemat 2 jest spełniony dla wątku t_2, w którym istnieje ścieżka rozpoczynająca się od blokady q_1, która nie zawiera operacji zwolnienia blokady $o_{2,5}$. Analogicznie w modelu z rys. 4.6.



RYSUNEK 4.6: Graf aplikacji DL3 z konfliktem spowodowanym próbą ponownego założenia blokady w wyniku działa pętli.



(A) Struktura wątków prowadząca do zakleszczeń DL2 i DL3.

(B) Struktura wątków wykluczająca wystąpienie zakleszczeń DL2 i DL3.

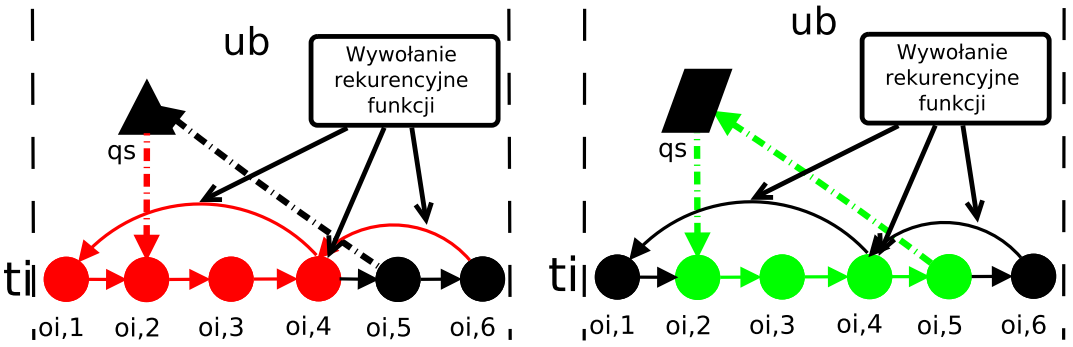
RYSUNEK 4.7: Graficzna reprezentacja błędu prowadzącego do zakleszczeń rodzaju DL2 i DL3 (A) i poprawek wykluczających wystąpienie tych konfliktów (B).

Warunek wystąpienia błędu DL4

Lemat 3. *Niech $T_P = \{t_0, ..., t_\alpha\}$ oznacza zbiór wątków wykorzystujących blokady $Q_P = (q_1, ..., q_\kappa)$ aplikacji wielowątkowej P. Jeśli wątek $t_i \in T_P$ wywołuje funkcję rekurencyjną, w której nie są stosowane blokady typu PMR, to może wystąpić błąd typu DL4 (patrz rys. 4.8 (A)).*

Dowód. Lemat jest bezpośrednią konsekwencją przyjętych założeń wywołania funkcji rekurencyjnych. c.k.d.

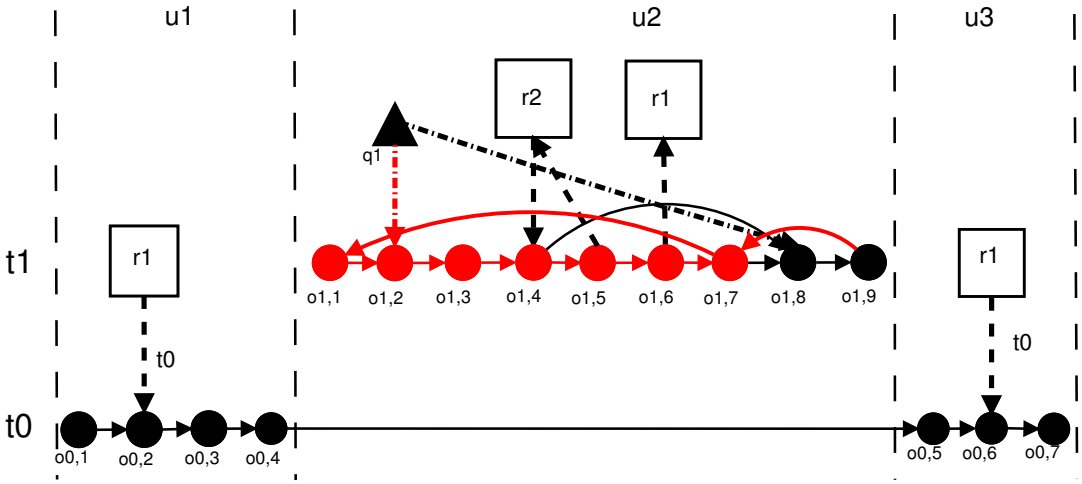
W przykładzie z rys. 4.9, Lemat 3 jest spełniony dla wątku t_1, w którym istnieje ścieżka cykliczna, której składową jest operacja założenia blokady typu PMD, a której składową nie jest operacja zwolnienia blokady.



(A) Struktura wątków prowadząca do zakleszczenia DL4.

(B) Struktura wątków wykluczająca wystąpienie zakleszczenia DL4.

RYSUNEK 4.8: Graficzna reprezentacja błędu prowadzącego do zakleszczenia DL4 (A) i poprawki wykluczające wystąpienie tego zjawiska (B).



RYSUNEK 4.9: Fragment grafu aplikacji DL4 z konfliktem spowodowanym próbą ponownego założenia blokady w wyniku wywołania rekurencyjnego funkcji.

Twierdzenie o zakleszczeniu

Przedstawione powyżej lematy prowadzą do następującego twierdzenia:

Twierdzenie 2. *Niech $T_P = \{t_0, ..., t_\alpha\}$ oznacza zbiór wątków wykorzystujących blokady $Q_P = (q_1, ..., q_\kappa)$ aplikacji wielowątkowej P. Błąd prowadzący do zakleszczenia może wystąpić jeśli istnieje taka para wątków t_i, $t_j \in T_P$, dla której spełniony jest Lemat 1 lub istnieje wątek $t_k \in T_P$ dla którego spełniony jest dowolny z Lematów 2 i 3.*

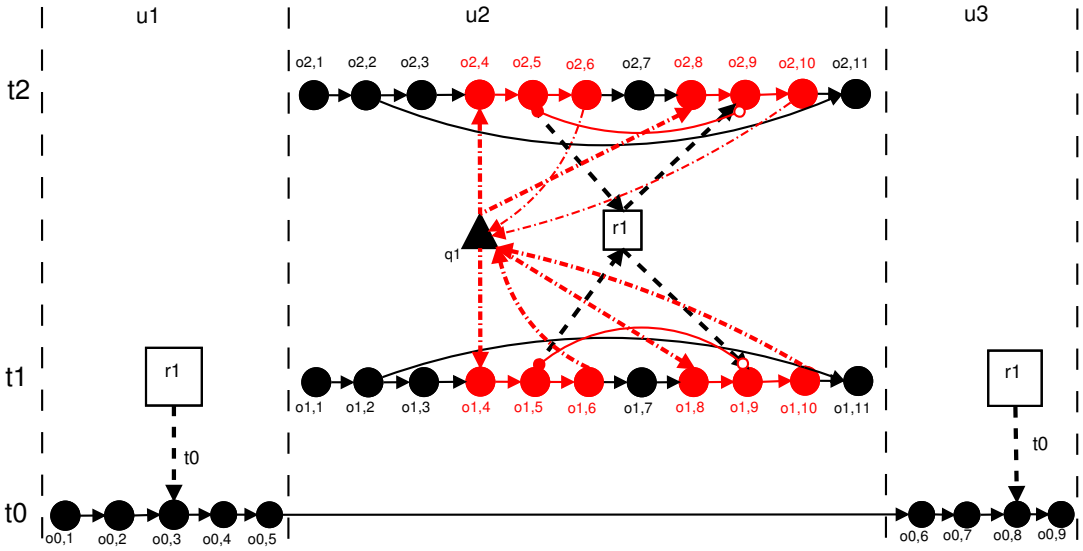
Dowód. Dowód wynika z przedstawionych powyżej Lematów 1, 2, 3. c.k.d.

Przedstawione w niniejszym podrozdziale warunki (Lematy 1, 2, 3) pozwalają lokalizować struktury kodu źródłowego skutkujące zakleszczeniem wątków analizowanej aplikacji. Proces ten sprowadza się do przeglądu ścieżek grafu operacji pod

względem wykrywania: kolejności występowania blokad (błąd DL1), cykliczności ścieżek (błędy DL2 i DL3) oraz rodzaju wykorzystywanych blokad (DL4).

4.3 Naruszenie niepodzielności

W dodatku A (listing 6) przedstawiono [35] kod źródłowy aplikacji wielowątkowej o nazwie AV1. Model C_P dla tej aplikacji przedstawiono na rys. 4.10. Aplikacja ta jest wolna od błędów prowadzących do szkodliwej rywalizacji i zakleszczenia. Gdy w aplikacji dochodzi do wykonania kodu funkcji *deposit* tylko przez jeden wątek to w każdym cyklu pętli zwiększana jest wartość zasobu współdzielonego r1, a następnie wyświetlana jest jego nowa wartość. Obie te operacje znajdują się w



RYSUNEK 4.10: Graf operacji aplikacji AV1.

dwóch różnych sekcjach krytycznych, co eliminuje ryzyko wystąpienia szkodliwej rywalizacji w momencie, gdy kod ten uruchomiony zostanie równolegle w innym wątku. Wywołanie funkcji *deposit* równolegle prowadzi jednak do konfliktu naruszenia niepodzielności. Objawiać się on będzie np. podwójnym wyświetleniem wartości zmiennej.

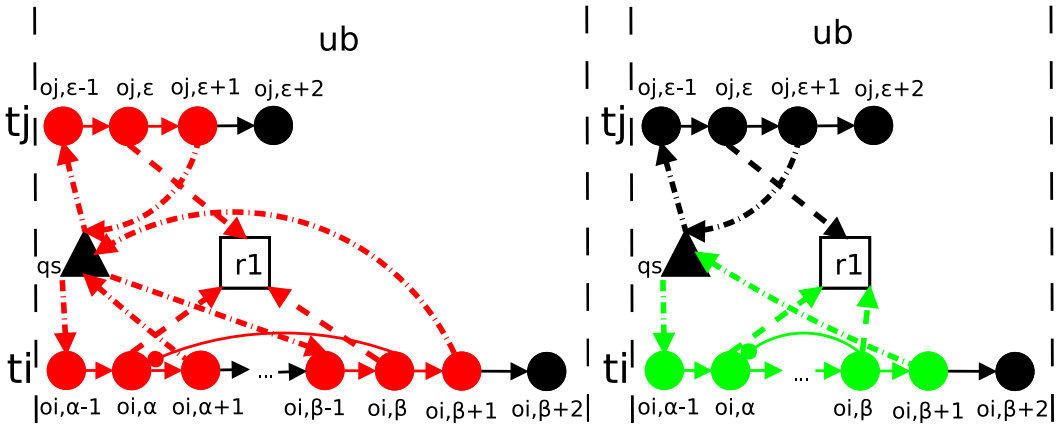
Na rysunku 4.10 znajduje się graf operacji dla aplikacji AV1. Błąd naruszenia niepodzielności wynika z faktu, że operacja $o_{1,5}$ (operacje zwiększenia zasobu) i wydrukowanie wartości na standardowym wyjściu powinny być wykonane jedna po drugiej. Architektura aplikacji natomiast pozwala na zmianę zasobu współdzielonego przez inny wątek pomiędzy wykonaniem tych operacji. Wspomniane operacje powinny być wykonane niepodzielnie (relacja symetryczna - podrozdział 3.1.7).

Przedstawiony przykład aplikacji AV1 oraz wprowadzona definicja naruszenia niepodzielności (definicja 5) pozwoliły opracować warunek, spełnienie którego świadczy o możliwości wystąpienia w kodzie źródłowym aplikacji błędu prowadzącego do tego typu konfliktu.

Twierdzenie 3. *Niech O_P oznacza zbiór operacji wątków T_P wykorzystujących blokady Q_P aplikacji wielowątkowej P. Dany jest wątek $t_i \in T_P$, dla którego znany jest zbiór $\lambda^{P,i}$ ścieżek zbudowanych z operacji tego wątku. Ponadto dane są operacje $o_{i,\alpha} \in O_P$ oraz $o_{i,\beta} \in O_P$, powiązane ze sobą jedną z relacji niepodzielności B_P^{ξ} (patrz podrozdział 3.1.7).*

Jeśli w zbiorze $\lambda^{P,i}$ aplikacji P istnieje ścieżka cykliczna, która zawiera blokadę $q_s \lhd \lambda^{P,i}$ i tylko jedną operację ze zbioru $\{o_{i,\alpha}, o_{i,\beta}\}$ to w kodzie źródłowym aplikacji P może wystąpić błąd skutkujący naruszeniem niepodzielności.

Dowód. Dowód wynika bezpośrednio z definicji błędu klasy naruszenia niepodzielności. Jeśli w zbiorze $\lambda^{P,i}$ aplikacji P istnieje ścieżka cykliczna, która zawiera blokadę $q_s \lhd \lambda^{P,i}$ i tylko jedną operację ze zbioru $\{o_{i,\alpha}, o_{i,\beta}\}$ to znaczy, że operacje $o_{i,\alpha}, o_{i,\beta}$ nie należą do jednej sekcji krytycznej. Oznacza to, że dopuszczalne jest wykonanie operacji $o_{j,\epsilon}$ innego wątku (t_j) na tym samym zasobie pomiędzy operacjami $o_{i,\alpha}, o_{i,\beta}$ tzn: kolejności $o_{i,\alpha} \prec o_{j,\epsilon} \prec o_{i,\beta}$ (patrz rys. 4.11). Dopuszczalne jest więc naruszenie niepodzielności operacji $o_{i,\alpha}, o_{i,\beta}$ przez wykonanie operacji $o_{j,\epsilon}$. c.k.d



(A) Struktura wątków prowadząca do naruszenia niepodzielności.

(B) Struktura wątków wykluczająca wystąpienie naruszenie niepodzielności.

RYSUNEK 4.11: Graficzna reprezentacja błędu prowadzącego do naruszenia niepodzielności (A) i poprawki wykluczające wystąpienie tego konfliktu zasobowego (B).

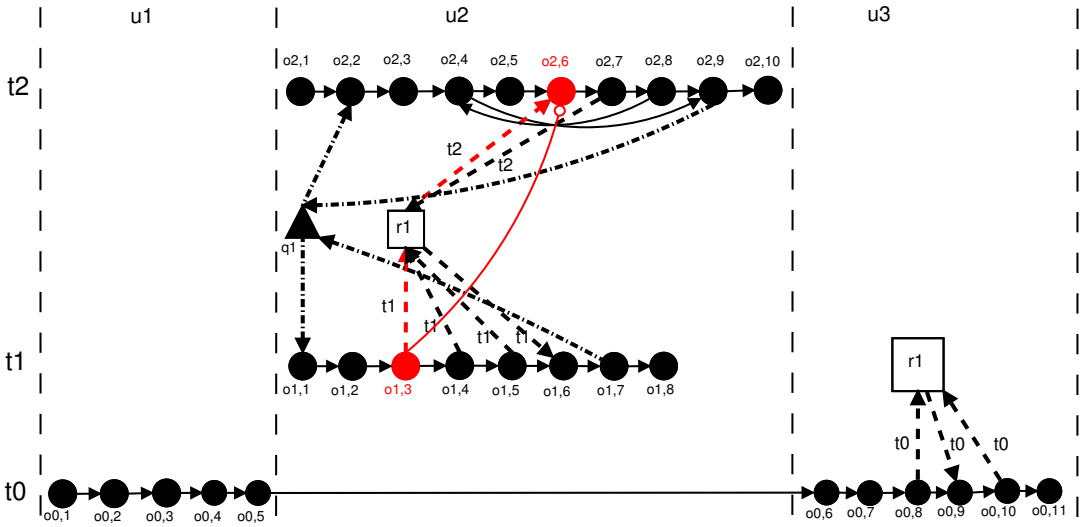
Jak łatwo zauważyć, Twierdzenie 3 jest spełnione również dla modelu z rys. 4.10. W prezentowanym modelu w ramach wątku t_1 istnieje ścieżka cykliczna, której składową jest operacja $o_{1,5}$, a która połączona jest relacją niepodzielności z operacją, $o_{1,9}$, która składową tej ścieżki cyklicznej nie jest. Identyczna sytuacja ma miejsce w przypadku pary operacji $o_{2,5}$ i $o_{2,9}$ w wątku t_2. Innymi słowy operacje wątku t_2 mogą zaburzyć relację pomiędzy operacjami wątku t_1 i odwrotnie.

Błędy klasy naruszenia niepodzielności stanowią 70% wszystkich błędów z grupy o charakterze wyścigu [78]. Ich lokalizacja przy użyciu opracowanego warunku sprowadza się do przeglądu ścieżek grafu operacji pod względem występowania sekcji krytycznych zawierających operacje powiązane relacją niepodzielności. Najtrudniejszym etapem tego procesu jest określenie, które z operacji są ze sobą w relacji niepodzielności. Ze względu na to, że informacja ta nie jest zawarta w kodzie źródłowym aplikacji zakłada się, że jest ona dostarczana przez programistę.

4.4 Naruszenie porządku

W dodatku A (listing 7) przedstawiono kod źródłowy przykładowej aplikacji OV1, której architektura dopuszcza do wystąpienia *naruszenia porządku* realizacji operacji. Zgodnie z definicją 6 naruszenie porządku wystąpi w sytuacji, w której jedna z par operacji będących ze sobą w jednej z relacji niepodzielności zostanie wykonana w

sposób ignorujący porządek ich wykonania. Na rysunku 4.12 znajduje się graf operacji reprezentujący aplikację OV1. W grafie występują dwie operacje $o_{1,2}, o_{2,6}$ połączone relacją wstecz tzn. $b_n = (o_{1,2}, o_{2,6})$, $b_n \in B^{BWD}_{OV1}$. Obie operacje znajdują się w dwóch różnych wątkach t_1 oraz t_2, przedziale czasu u_2. Istnieje więc szansa, że do wykonania operacji $o_{2,6}$ dojdzie jeszcze przed wykonaniem operacji $o_{1,3}$ co będzie skutkować naruszeniem zadanego porządku (może to doprowadzić do nieoczekiwanego zakończenia pracy aplikacji).



RYSUNEK 4.12: Graf operacji aplikacji OV1.

Bazując na przykładzie aplikacji OV1 i wprowadzonej definicji naruszenia porządku (definicja 6) opracowano warunek którego spełnienie, świadczy o wystąpieniu w kodzie źródłowym aplikacji błędu prowadzącego do tego typu konfliktu.

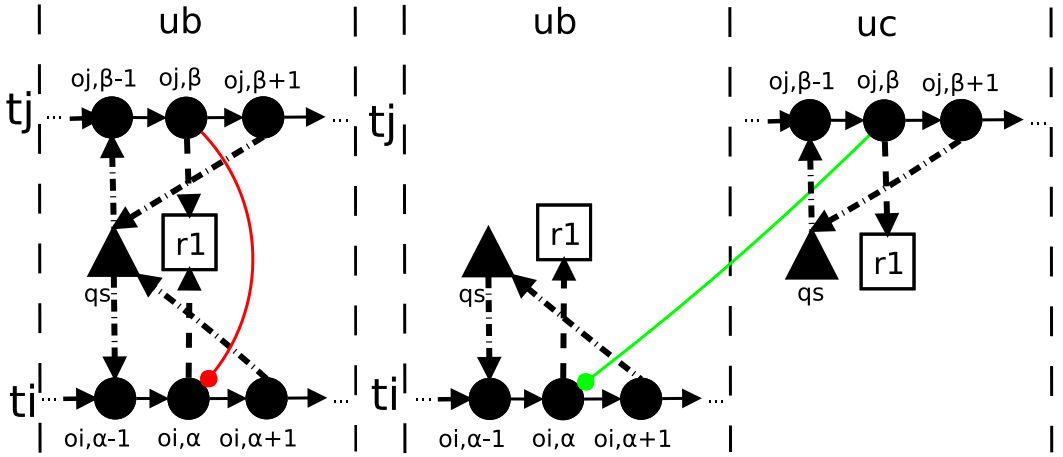
Twierdzenie 4. *Niech O_P oznacza zbiór operacji wątków T_P aplikacji wielowątkowej P. Ponadto dane są operacje $o_{i,\alpha} \in O_P$ oraz $o_{j,\beta} \in O_P$, powiązane ze sobą jedną z relacji niepodzielności B_P (patrz podrozdział 3.1.7). Niech ${}^{i,j}B^{\xi}_P \subseteq B^{\xi}_P$ ($\xi \in \{FWD, BWD, SYM\}$) będzie podzbiorem zwierającym takie pary operacji $(o_{i,\alpha}, o_{j,\beta})$, z których pierwsza wykonywana jest w wątku t_i, a druga w wątku t_j. Jeśli $\{t_i, t_j\} \subseteq u_b \in U_P$ to może dojść do naruszenia porządku realizacji operacji $(o_{i,\alpha}, o_{j,\beta})$.*

Dowód. Dowód jest bezpośrednią konsekwencją definicji błędu klasy naruszenia porządku. Jeśli wątki $\{t_i, t_j\}$ są wykonywane we wspólnym przedziale czasu, tj. $\{t_i, t_j\} \subseteq u_b \in U_P$ to dopuszczalna jest tym samym współbieżna realizacja operacji $(o_{i,\alpha}, o_{j,\beta})$ (patrz rys. 4.13). Oznacza to jednocześnie, że możliwy jest dowolny porządek realizacji operacji. Dopuszczalne jest zatem naruszenie zadanego porządku operacji $(o_{i,\alpha}, o_{j,\beta})$. c.k.d.

Jak łatwo zauważyć, Twierdzenie 4 jest spełnione również dla modelu z rys. 4.12. W przedziale czasu u_2 swoje operacje wykonują wątki t_1 i t_2. W ramach tych dwóch wątków istnieje para operacji $(o_{1,3}, o_{2,6})$, która połączona jest relacją niepodzielności wstecz. Taka struktura aplikacji dopuszcza więc przebieg, w którym może dojść do naruszenia porządku.

Błędy prowadzące do naruszenia porządku są podobne do błędów prowadzących do naruszenia niepodzielności. W obu przypadkach dochodzi do zakłócenia zadanej z góry kolejności ich wykonywania. Różnicą jest fakt, że w przypadku naruszenia porządku operacje znajdują się w dwóch różnych wątkach. Wykorzystanie

opracowanego warunku w procesie lokalizacji błędów prowadzących do naruszenie porządku sprowadza się do przeglądu operacji wątków realizowanych we wspólnym przedziale czasu u_b.



(A) Struktura wątków prowadząca do naruszenia porządku.

(B) Struktura wątków wykluczająca wystąpienie naruszenie porządku.

RYSUNEK 4.13: Graficzna reprezentacja błędu prowadzącego do naruszenia porządku (A) i poprawki wykluczające wystąpienie tego konfliktu zasobowego (B).

4.5 Podsumowanie

W rozdziale zdefiniowano warunki konieczne, których spełnienie oznacza, że analizowana aplikacja wielowątkowa zawiera błędy prowadzące do szkodliwej rywalizacji, zakleszczenia, naruszenia niepodzielności i naruszenia porządku. Przedstawione Twierdzenia 1, 2, 3 i 4 pozwoliły na opracowanie metody statycznej analizy aplikacji wielowątkowych. Wykorzystanie zaproponowanych warunków w procesie lokalizacji błędów aplikacji wielowątkowych sprowadza się do przeglądu grafu operacji G_P reprezentującego model C_P pod względem występowania:

- ścieżek cyklicznych zawierających (lub nie) operacje założenia blokady (szkodliwa rywalizacja),
- właściwej kolejności występowania blokad i ich odpowiedniego rodzaju (zakleszczenia),
- ścieżek cyklicznych zawierających operacje powiązane relacją niepodzielności (naruszenie niepodzielności),
- przedziałów czasu dla których realizowane są wątki zawierające operacje, dla których zdefiniowano porządek ich wykonania (naruszenie porządku),

W ogólności proces ten wymaga przeglądu wszystkich ścieżek grafu operacji G_P. Ze względu na wysoką złożoność obliczeniową (liczba ścieżek grafu rośnie wykładniczo wraz z jego rozmiarem) takiego postępowania, skala problemów (rozmiar grafów) ogranicza się do niewielkich instancji. Niemniej jednak, jak to pokazały wyniki eksperymentów, z rozdziału 6 opracowana metoda pozwala analizować aplikacje wielowątkowe o rozmiarze spotykanym w praktyce.

Rozdział 5

Metoda lokalizowania konfliktów zasobowych

5.1 Przeznaczenie

Zaproponowana metoda lokalizowania konfliktów zasobowych (ang. Races and Deadlocks - Locating, (RD-L)) pozwala na identyfikowanie „miejsc" (tj. fragmentów kodu źródłowego) występowania błędów aplikacji wielowątkowych. Rezultatem jej stosowania jest informacja (reprezentowana w postaci stosownych raportów) o możliwościach występowania nieprawidłowych interakcjach między operacjami wykorzystującymi zasoby współdzielone. Ich skutkiem może być konflikt zasobowy prowadzący do niepoprawnego zachowania aplikacji lub jej awarii np. nieoczekiwanego wyłączenia się. W celu identyfikacji tego typu błędów wykorzystano zaprezentowany w poprzednim rozdziale model C_P i wyprowadzone na jego bazie Twierdzenia 1–4.

Główną grupą odbiorców niniejszej metody są programiści. Warto podkreślić, że nie posiadają oni obecnie wielu narzędzi wspomagających proces pisania kodu aplikacji wielowątkowych. Weryfikacja kodu aplikacji jest realizowana przez programistę w dwóch etapach: etapie wstępnego przeglądu kodu (zwykle realizowanego ręczenie) oraz procesie testowania. W tym celu wykorzystywane są zwykle narzędzia i metody opisane w rozdziale 2 (o ile te wspierają wykorzystany język programowania). Narzędzia te najczęściej charakteryzują się niską skutecznością, są trudne w użyciu i często ograniczają się do wybranej klasy błędów (np. identyfikacja wyłącznie szkodliwej rywalizacji [25, 99, 59]). Oznacza to potrzebę efektywnego wsparcia etapów weryfikacji projektowanych aplikacji wielowątkowych. Jest to szczególnie istotne, gdy fragmenty kodu są pisane przez różnych członków zespołu projektowego.

Metoda RD-L przeznaczona jest do wspomagania pracy programisty w procesie pisania kodu aplikacji wielowątkowej pozbawionej błędów prowadzących do wybranej klasy konfliktów zasobowych: szkodliwej rywalizacji, zakleszczenia, naruszenia niepodzielności i naruszenia porządku (patrz rozdział 2).

Przyjęto, że metoda RD-L jest dedykowana dla wsparcia zespołów pracujących w tzw. metodykach zwinnych np. Scrum, Nexus, Kanban, wspomagających wytwarzanie aplikacji wielowątkowych [84, 91]. Zgodnie z metodykami zwinnymi w trakcie wytwarzania oprogramowania stosuje się wiele procesów inicjowanych cyklicznie w celu sprawdzenia czy w projektowanej/rozwijanej aplikacji nie występują błędy. Do procesów tych zalicza się:

- proces statycznej analizy kodu polegający na sprawdzaniu czy kod spełnia wymagany standard np.:

  - czy nie istnieją powtarzające się bloki kodu, tzn. spełniona jest reguła **nie powtarzaj się** (ang. Don't Repead Yourself, DRY)?
  - czy kod nie zawiera bloków kodu, które nigdy nie zostaną uruchomione (tzw. martwych bloków kodu)?
  - czy wybrane jednostki są odpowiednio małe, np. funkcja posiada nie więcej niż 50 linii kodu lub plik posiada nie więcej niż 500 linii kodu?
- proces testowania, np.:
  - testy jednostkowe;
  - testy komponentowe;
  - testy end-to-end;
- proces dynamicznej analizy aplikacji.

Zgodnie z wytycznymi organizacji ISTQB (ang. International Software Testing Qualifications Board) [14] dwa z powyższych procesów powinny być zawsze stosowane w trakcie pisania kodu aplikacji. Stosowanie wymienionych wytycznych pozwala uzyskać następujące korzyści:

- Statyczna analiza stosowana w celu sprawdzenie wymaganych standardów jest procesem szybkim i powinna być wykonywana zawsze tam gdzie analizowany kod przechodzi wszystkie testy jednostkowe. Spełnianie standardów jakości kodu często wymaga wprowadzania zmian w architekturze np. poprzez hermetyzację wybranych fragmentów kodu (wymagane przy zachowaniu reguły DRY). Zmiany można wykonywać bezpiecznie, ponieważ programista dysponuje testami jednostkowymi, pozwalającymi na łatwe wykrycie niekorzystnych zmian w implementacji.
- Uruchamianie testów jednostkowych jest bardzo często wykonywanym procesem w zespołach stosujących metodyki zwinne, a szczególnie w tych, które dodatkowo stosują technikę wytwarzania oprogramowania polegającą na wytwarzaniu oprogramowania sterowanym testami (ang. test driven development, TDD). Częste uruchamianie testów gwarantuje, że wprowadzone zmiany nie wpłyną na już działający kod. Natomiast stosowanie techniki TDD gwarantuje, że wprowadzone zmiany spełniają wymagania napisanych testów tzn. napisany kod działa zgodnie z oczekiwaniami.

Pozostałe procesy (np. analiza bezpieczeństwa użytych bibliotek) zazwyczaj uruchamiane są automatycznie w procesie CI/CD [20, 45] lub przez innych członków zespołu odpowiedzialnych za jakość dostarczanego oprogramowania, np. testerów.

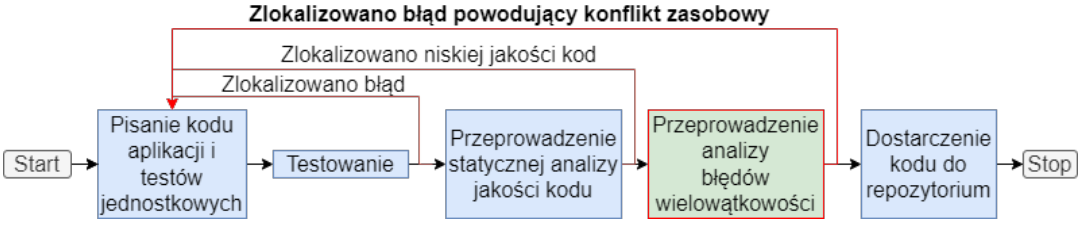
Powszechnie programiści kończą swoją pracę nad daną funkcjonalnością w momencie, gdy statyczna analiza kończy się sukcesem, a testy jednostkowe nie raportują błędów w napisanym przez nich kodzie. Jednak kiedy aplikacja ta jest wielowątkowa, to należy przeprowadzić proces lokalizowania konfliktów zasobowych.

Wykonanie testów jednostkowych tzn. pokrycie kodu (ang. code coverage) testami wynosi 100% i spełnienie wymaganych norm pozwala na usunięcie z aplikacji większości błędów prowadzących do zakleszczenia np. zakleszczenie typu DL4 (wynikającego z zastosowania nieodpowiedniego typu blokady w funkcji rekurencyjnej). Istnieje również wiele dobrych praktyk, których użycie przez doświadczonego programistę zwiększa szansę na uniknięcie błędów prowadzących do konfliktów zasobowych. Nie ma jednak gwarancji, że stosowanie tych praktyk pozwoli

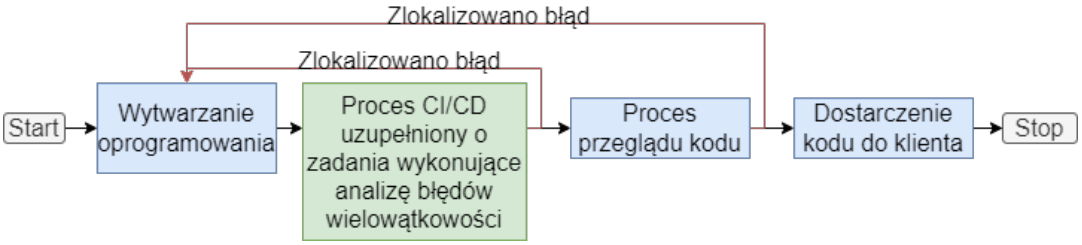
uniknąć błędów powodujących konflikty zasobowe. Dlatego, aby taką gwarancję uzyskać, programista powinien posiadać narzędzia, umożliwiające uzyskanie takiej gwarancji.

W tym kontekście, proces lokalizowania konfliktów zasobowych powinien być częścią procesu wytwarzania oprogramowania. Zgodnie z rysunkiem 5.1 proces lokalizowania konfliktów należy przeprowadzać po napisaniu kodu aplikacji i po jego dostarczeniu do repozytorium w procesie CI/CD (patrz rys. 5.3). Takie podejście (dwuetapowa weryfikacja) pozwala uniknąć sytuacji, w których programista wprowadza do repozytoriom niepoprawny kod. W trakcie procesu CI/CD automatycznie przeprowadzona zostaje analiza kodu aplikacji. W przypadku wykrycia błędów prowadzących do konfliktów zasobowych, proces dostarczania oprogramowania zostanie zatrzymany. Zatrzymanie procesu CI/CD (i zgłoszenie błędu) na odpowiednio wczesnym etapie zmniejsza koszty wytwarzania oprogramowania. Dzięki temu pozostali członkowie zespołu wykonują mniej pracy ponieważ nie muszą wykonywać przeglądu kodu, w którym błędy zostały wykryte wcześniej (patrz rys. 5.2).

Jak łatwo zauważyć na rys. 5.3, lokalizowanie konfliktów odbywa się na etapie „Statyczna analiza". Pozytywny rezultat przeprowadzonej analizy należy traktować jako formalne potwierdzenie, że zmiany wprowadzone przez programistę nie prowadzą do błędów skutkujących wystąpieniem konfliktów zasobowych.

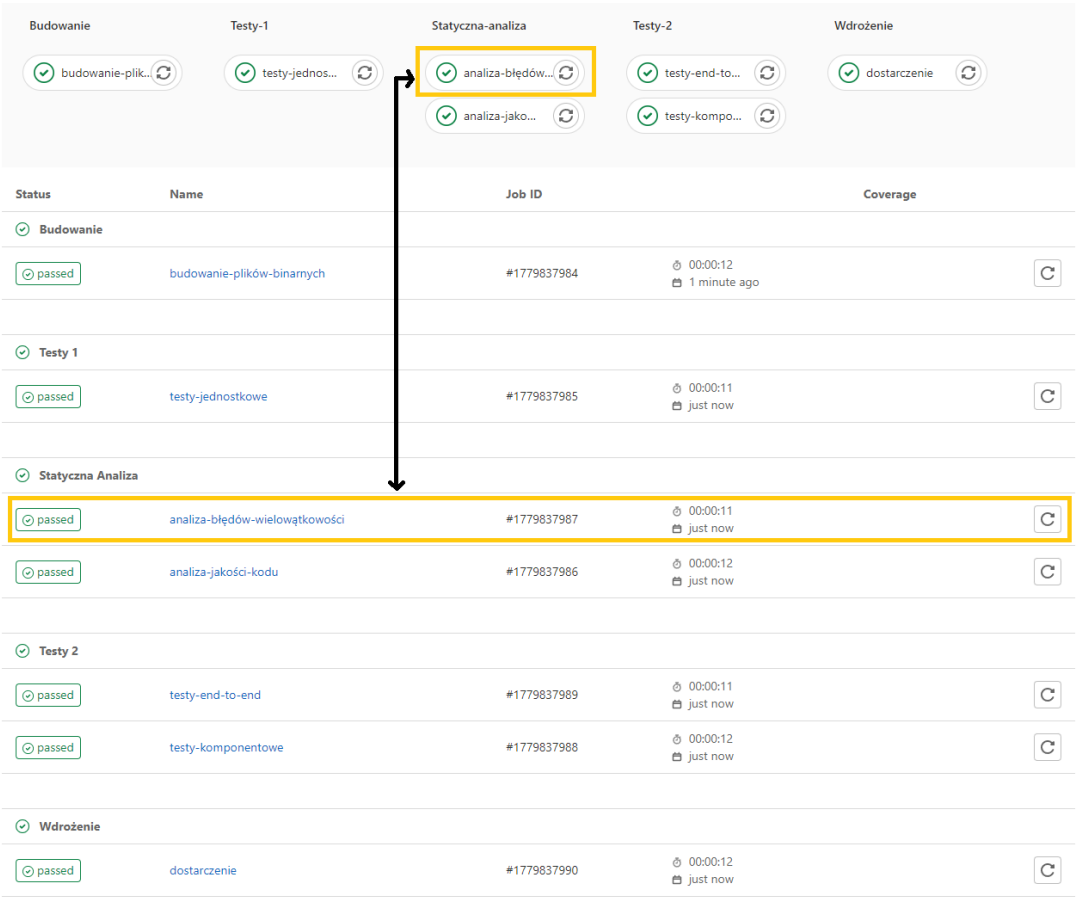


RYSUNEK 5.1: Algorytm wytwarzania oprogramowania rozszerzony o proces analizy pozwalającej wykrywać błędy powodujące konflikty zasobowe.



RYSUNEK 5.2: Algorytm procesu dostarczania oprogramowania.

Przykład procesu zaprojektowanego dla prostej aplikacji wielowątkowej znajduje się na rysunku 5.3. Proces CI/CD podzielony jest na etapy, w ramach których równolegle wykonywane są zdefiniowane zadania. Warunkiem uruchomienia kolejnego etapu jest zakończenie powodzeniem wszystkich zadań poprzedniego etapu (chyba, że definicja zadania zakłada inną możliwość, co stanowi wyjątek). W przedstawionym procesie CI/CD z rysunku 5.3 zaproponowano równoległe wykonanie dwóch zadań w etapie statycznej analizy. Oba te zadania zostaną wykonane wtedy, gdy kod aplikacji wielowątkowej uda się skompilować, a zbudowana aplikacja przechodzi wszystkie testy jednostkowe.

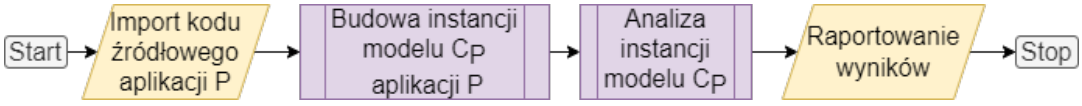


RYSUNEK 5.3: Przykładowy proces CI/CD dla aplikacji wielowątkowej utworzony w aplikacji GitLab.

5.2 Zasada działania

Na metodę lokalizowania konfliktów zasobowych składają się z cztery etapy przedstawione na schemacie blokowym z rysunku 5.4:

1. Import kodu źródłowego aplikacji P.
2. Budowa instancji modelu C_P dla aplikacji P.
3. Analiza instancji modelu C_P pod względem spełnienia warunków wystąpienia błędów prowadzących do konfliktów zasobowych (Twierdzenia 1–4).
4. Raportowanie błędów.



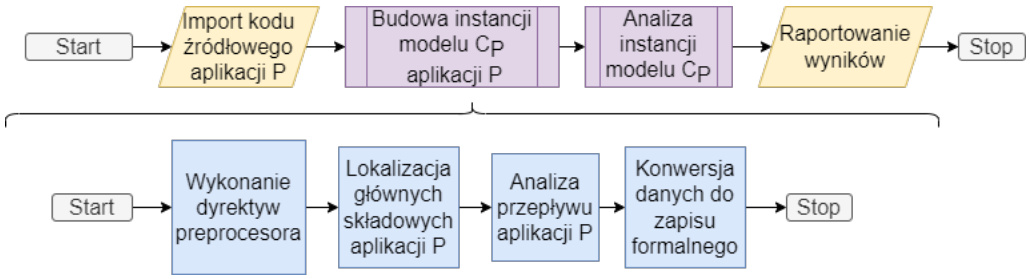
RYSUNEK 5.4: Schemat blokowy metody lokalizowania konfliktów zasobowych.

Import kodu źródłowego aplikacji P oraz *Raportowanie błędów* są etapami technicznymi związanymi z przygotowaniem danych wejściowych i wyjściowych. *Budowa instancji modelu* C_P oraz *Analiza instancji modelu* C_P to dwa główne etapy opracowanej metody, szerzej opisane w kolejnych punktach.

5.2.1 Budowa instancji modelu C_P

Budowa instancji modelu to proces wieloetapowy (patrz rys. 5.5), w którym należy dokonać analizy kodu źródłowego aplikacji, zidentyfikować istotne (ze względu na poszukiwane błędy) elementy jego struktury, a następnie na ich podstawie zbudować instancję modelu C_P (zgodnie z terminologią wprowadzoną w rozdziale 3). Proces ten można sprowadzić do następujących etapów:

1. Wykonanie dyrektyw preprocesora języka C;
2. Lokalizacja składowych aplikacji tzn.
   - Lokalizacja definicji funkcji;
   - Lokalizacja definicji struktur;
   - Lokalizacja zasobów i blokad zadeklarowanych w postaci zmiennych globalnych;
3. Analiza przepływu aplikacji;
   - Lokalizacja punktu wejścia tzw. funkcji **main**;
   - Lokalizowanie wątków aplikacji;
   - Budowanie drzewa operacji wątków;
   - Lokalizacja zasobów i blokad zadeklarowanych w postaci zmiennych lokalnych, których wskaźnik przekazywany jest do wątków;
   - Lokalizacja relacji;
   - Przydział wątków do przedziałów czasu;
4. Transformacja danych do zapisu formalnego;



RYSUNEK 5.5: Algorytm budowy instancji modelu C_P.

Wykonanie dyrektyw preprocesora języka C

Dyrektywy preprocesora nie są elementem języka C i są wykorzystywane w celu przetworzenia kodu źródłowego w tzw. kod wyjściowy, czyli kod który może zostać poddany dalszej analizie, kompilacji i konsolidacji. W przypadku języka C dyrektywami nazywamy wyrażenia, które zapoczątkowane są znakiem „#" czyli tzw. hash'a i kończą się znakiem końca linii. W zależności od stosowanego kompilatora lista dyrektyw może się znacznie od siebie różnić, niemniej istnieje zbiór wspólnych dyrektyw, do których należą m.in.: include, define, assert czy warning.

Wykonanie dyrektyw preprocesora przed rozpoczęciem lokalizowania głównych elementów aplikacji P jest niezbędne, gdyż ich realizacja determinuje strukturę kodu

źródłowego. W zależności od parametrów wejściowych preprocesora wygenerowany kod wyjściowy może się różnić dla różnych platform docelowych. Przykładowo kod aplikacji dla systemu GNU/Linux będzie wykorzystywać bibliotekę *pthread*, natomiast dla platformy Windows kod będzie wykorzystywać bibliotekę Win32.

Najprostszym sposobem wykonania dyrektyw preprocesora jest wykorzystanie jednego z preprocesów dostępnych na rynku i dostarczanych wraz z kompilatorami. Przykładowo kompilator gcc posiada flagę -E, której użycie kończy działanie kompilatora po etapie wykonania dyrektyw preprocesora. W ten sposób otrzymany kod wynikowy, pozbawiony jest „uciążliwych" dyrektyw i może zostać dalej wykorzystany do budowy instancji modelu C_P kodu źródłowego aplikacji wielowątkowej P.

Lokalizowanie głównych składowych aplikacji P

Kod źródłowy każdej komercyjnej aplikacji podzielony jest zwykle na wiele plików źródłowych. Zawartość każdego z tych plików musi zostać poddana walidacji pod kątem poprawności składni. W tym celu (podobnie jak w przypadku dyrektyw preprocesora) można skorzystać z gotowego narzędzia dostarczanego wraz z kompilatorem języka C. Przykładowo wykorzystanie kompilatora gcc (po użyciu flagi -fsyntax-only) zwróci informacje, w którym miejscu kodu źródłowego składnia jest niepoprawna.

Kolejnym krokiem jest ustalenie, gdzie znajdują się główne elementy analizowanego kodu (tzw. składowe aplikacji P), tj. definicje funkcji, definicje struktur, deklaracje zmiennych globalnych w tym deklaracje blokad. Identyfikacja tych elementów pozwoli na realizację dalszych etapów procesu budowy instancji modelu C_P.

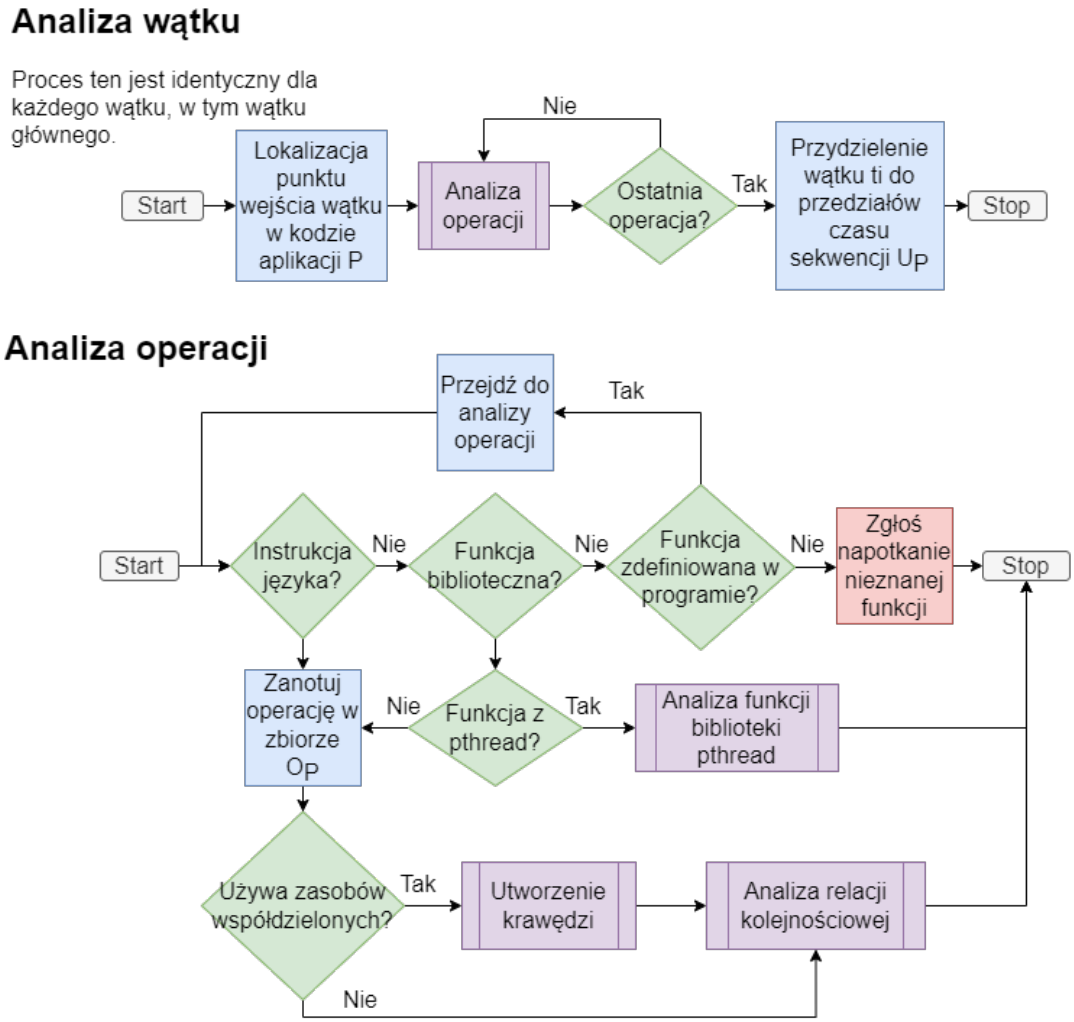
Analiza przepływu aplikacji P

Analiza przepływu aplikacji to najbardziej skomplikowany etap budowy instancji modelu C_P. Proces ten jest bardzo podobny do procesu analizy złożoności cyklomatycznej kodu źródłowego programu [70], a jego algorytm przedstawiono na rysunkach 5.6 i 5.7.

Pierwszym krokiem analizy przepływu aplikacji jest lokalizacja punktu wejścia aplikacji, którym najczęściej jest funkcja **main**. Następnie rozpoczyna się przegląd kodu źródłowego, rozpoczynając od pierwszej operacji funkcji **main**. Operacją może być:

- deklaracja zasobu,
- wykonanie instrukcji języka C,
- wywołanie innej funkcji.

Deklaracja dowolnego zasobu (np. zmiennej) wiąże się z przechowaniem informacji o tym zasobie, ponieważ użycie zasobu przez co najmniej dwa wątki kwalifikuje go jako zasób współdzielony (zasób zostaje dodany do R_P). Język C umożliwia przekazywanie dostępu do zasobów poprzez mechanizm wskaźników. Oznacza to, że zmienna lokalna funkcji **main** może być wykorzystana przez operacje dowolnego wątku, o ile adres tej zmiennej zostanie podany jako parametr wywołania funkcji *pthread_create*. W tym kontekście każdy zasób, którego adres zostanie przekazany do innego wątku, należy traktować jako zasób współdzielony.



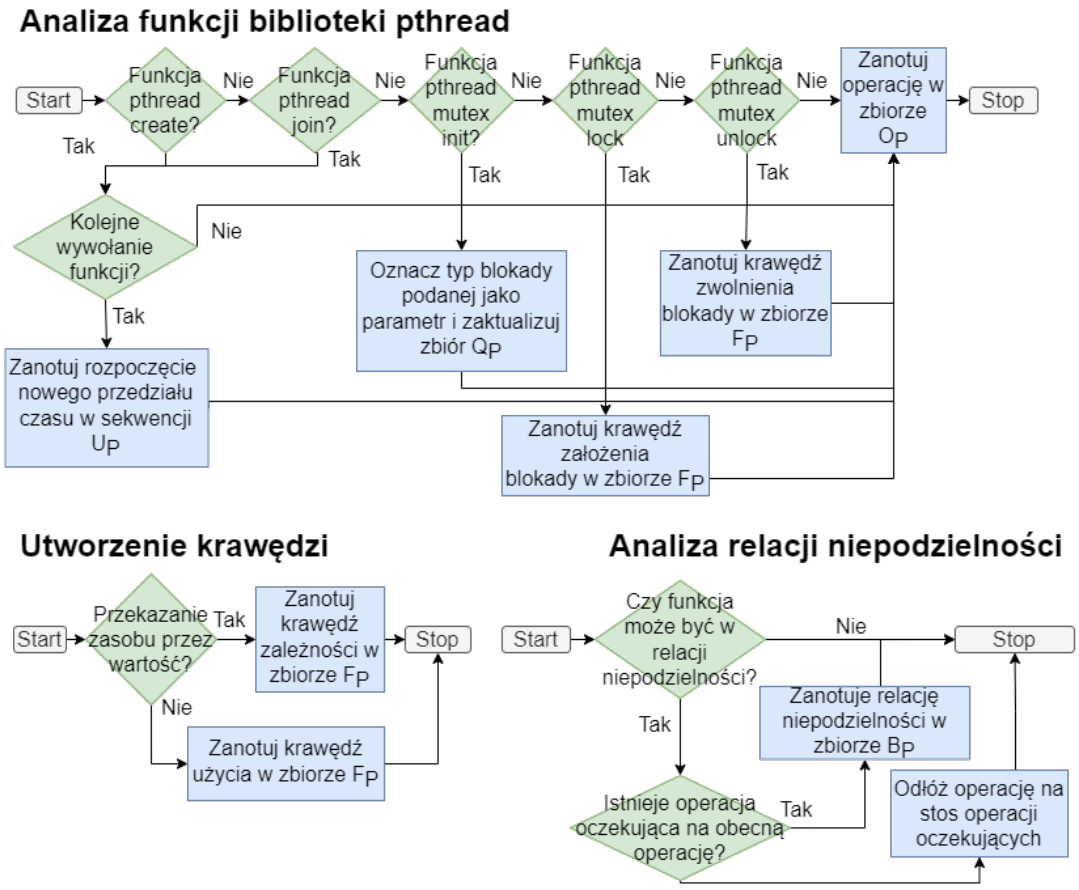
RYSUNEK 5.6: Algorytm analizy wątku i algorytm podprocesu analizy operacji.

Jeśli operacja jest wywołaniem funkcji, wtedy, w zależności od jej rodzaju, należy wybrać odpowiedni algorytm postępowania:

- Jeśli jest to funkcja z biblioteki standardowej języka C, to zbiór operacji O_P uzupełniany jest o operację reprezentującą tę funkcję. Tak samo należy postąpić w przypadku napotkania instrukcji języka (np. while lub if).
- Jeśli jest to funkcja zadeklarowana w kodzie aplikacji, to należy zlokalizować definicję tej funkcji, odłożyć aktualny proces analizy i przejść do analizy nowo wywołanej funkcji, a po jej zakończeniu kontynuować poprzedni proces analizy.
- Jeśli jest to funkcja *pthread_create* to uzupełnia się zbiór O_P, a zbiór U_P uzupełnia się o nowy przedział u_b. Następnie należy wykonać analizę parametrów przekazywanych poprzez wskaźnik, zanotować nowy wątek (zbiór T_P uzupełnić o wątek inicjowanych przez tę funkcję) i kontynuować dalszy przegląd kodu źródłowego. Wystąpienie funkcji *pthread_create* nie przerywa procesu przeglądu kodu aktualnej funkcji, tak jak ma to miejsce w przypadku wywołań funkcji zadeklarowanych w kodzie aplikacji. W przypadku, gdy

jest to kolejne wywołanie funkcji *pthread_create* pod rząd (jawne lub w skutek działania pętli) i pomiędzy operacjami wywołania tej funkcji nie była wykonywana żadna operacja (z wyłączeniem operacji wyłuskania wskaźnika do zmiennej, który przekazywany jest jako parametr wejściowy), to nie aktualizuje się zbioru przedziałów czasu.

- Jeśli jest to funkcja *pthread_join* (funkcja oczekująca na zakończenie pracy wskazanego wątku), to należy zaktualizować zbiór operacji O_P oraz oraz zbiór przedziałów czasu U_P. Jeśli następuje po sobie kilka wywołań funkcji *pthread_join*, to podobnie jak w przypadku funkcji *pthread_create* nie aktualizuje się zbioru przedziałów czasu U_P.
- Jeśli jest to funkcja *pthread_mutex_init*, to aktualizuje się typ blokady (tzn. zbiór Q_P jest uzupełniany lub aktualizowany o tę informację). W tym przypadku nie jest dodawana żadna para do zbioru F_P. Drugi parametr (będący zestawem atrybutów) funkcji *pthread_mutex_init* determinuje m.in. typ blokady. Gdy typ ten nie jest jawnie podany, to blokada ta jest typu PTHREAD_MUTEX_DEFAULT.



RYSUNEK 5.7: Algorytmy podprocesów używanych w analizie wątków.

Powyższy proces należy przeprowadzić dla wątku głównego rozpoczynającego się od funkcji **main** oraz każdego wątku potomnego rozpoczętego wywołaniem funkcji *pthread_create*.

W trakcie analizy przepływu aplikacji notowane są operacje zbioru O_P, ale także relacje (krawędzie) zbioru F_P składające się na model C_P tzn. relacje przejścia, użycia, zależności, założenia blokady oraz zwolnienia blokady.

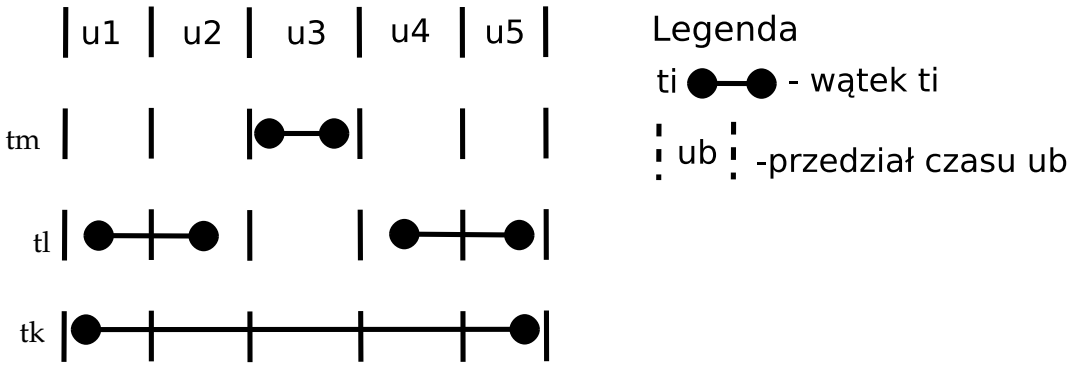
Relacje zbioru F_P są określane na podstawie analizy parametrów wywołania funkcji:

- Jeśli funkcja przyjmuje parametr poprzez wartość, to jest relacją zależności, natomiast każdy inny przypadek oznaczany jest jako relacja użycia.
- Wywołanie funkcji *pthread_mutex_lock* określa relację założenia blokady.
- Wywołanie funkcji *pthread_mutex_unlock* określa relację zwolnienia blokady.

W trakcie przeglądu kodu aplikacji P należy również sprawdzić, czy dana operacja powiązana jest z inną operacją relacją niepodzielności (należy do jednego ze zbiorów sekwencji B_P) tzn. lokalizowane są relacje wprzód, wstecz i symetryczne (patrz podrozdział 3.1.7).

W ostatnim kroku odpowiednie wątki składające się na zbiór T_P przydzielane są do przedziałów czasu określonych w zbiorze U_P. Przyjęta procedura bazuje na zasadach:

- Każdy wątek, poza wątkiem głównym, zaczyna się wywołaniem funkcji *pthread_create* i kończy wywołaniem funkcji *pthread_join*.
- Niech przykładowa aplikacja P posiada kolejno trzy wątki t_K, t_L, t_M i pięć przedziałów czasu u_1, u_2, u_3, u_4, u_5. Jeśli wywołanie funkcji *pthread_create* dla wątków t_K i t_L miało miejsce w przedziale czasu u_1, natomiast wywołanie funkcji *pthread_join* dla tych samych wątków miało miejsce w przedziale czasu u_5, to zakłada się, że wątki te pracowały także w przedziałach realizacji wątków u_2, u_3 i u_4. Od tej reguły istnieje wyjątek. Jeśli w kodzie wątku t_L i przedziale u_2 wywołano funkcję *pthread_create* dla wątku t_M, po czym wywołano funkcję *pthread_join* dla tego wątku, to zakłada się, że wątek t_M działał w przedziale u_3. W przedziale tym swoją pracę zawiesza natomiast wątek t_L, który nie wykonuje żadnej operacji równolegle do wątku t_M, natomiast wznawia pracę po zakończeniu pracy wątku potomnego (patrz rys. 5.8).



RYSUNEK 5.8: Wizualizacja tworzenia wątku.

Transformacja danych do zapisu formalnego

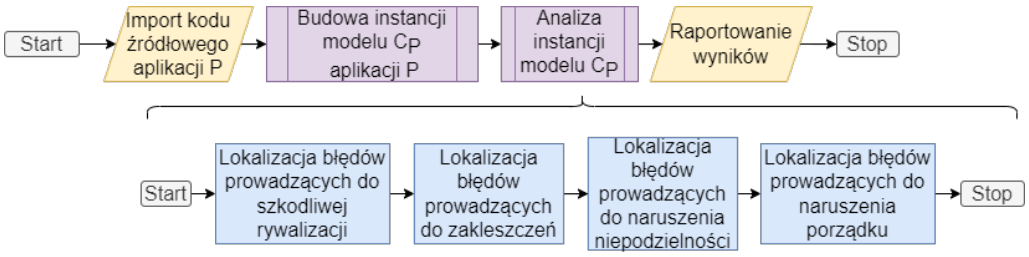
Ostatnim krokiem jest transformacja zidentyfikowanych danych do zapisu formalnego. Wynikiem przeglądu kodu aplikacji P jest wiele danych, które po zakończonym procesie są zbędne np. informacje o zmiennych lokalnych. Na tym etapie ze

zbioru zgromadzonych danych, wyselekcjonowane są te, niezbędne do przedstawienia instancji modelu zgodnie z jego definicją z rozdziału 3. Utworzona w ten sposób instancja modelu C_P może zostać poddana dalszej analizie w celu zlokalizowania błędów prowadzących do konfliktów zasobowych.

5.2.2 Analiza instancji modelu C_P

Proces analizy instancji modelu obejmuje lokalizację błędów prowadzących do (patrz rys. 5.9):

1. szkodliwej rywalizacji.
2. zakleszczeń.
3. naruszeń niepodzielności.
4. naruszeń porządku.



RYSUNEK 5.9: Algorytm procesu analizy instancji modelu C_P.

Lokalizacja błędów prowadzących do szkodliwej rywalizacji

Proces lokalizowania błędów prowadzących do szkodliwej rywalizacji zaczyna się od przeglądu przedziałów czasu u_b (dla $b = 1, ..., \beta$) sekwencji U_P. Dla każdego wątku t_i przedziału u_b ($t_i \in u_b$) określany jest zbiór ścieżek $\lambda^{P,i}$. Spośród operacji składających się na ścieżki zbioru $\lambda^{P,i}$ identyfikowane są te, które wykorzystują zasoby współdzielone $r_c \in R_P$. Następnie dla każdej takiej operacji $o_{i,j}$ sprawdzane jest, czy operacja ta w niekontrolowany sposób zmienia stan zasobu współdzielonego r_c tzn. weryfikowane jest, czy spełnione jest Twierdzenie 1. Istnienie takich operacji świadczy o błędzie prowadzącym do szkodliwej rywalizacji.

Lokalizacja błędów prowadzących do zakleszczeń

Proces lokalizowania błędów prowadzących do zakleszczeń wygląda podobnie jak w przypadku lokalizowania błędów szkodliwej rywalizacji. W tym przypadku dla każdego przedziału czasu u_b budowane są podgrafy ${}^s_i G_P$ grafu operacji G_P (tj. podgrafy reprezentujące operacje wątków $t_i \in u_b$ i związanych z nimi blokad $q_s \in Q_P$). Następnie dla każdego podgrafu ${}^s_i G_P$ weryfikowane są warunki określone w lematach 1, 2 i 3 tzn. weryfikowane jest Twierdzenie 2. Jeśli, co najmniej jeden z ww. warunków jest spełniony oznacza to, że w aplikacji występuje błąd prowadzący do zakleszczenia.

Lokalizacja błędów prowadzących do naruszenia niepodzielności i naruszenia porządku

Proces lokalizowania błędów prowadzących do naruszenia niepodzielności i naruszenia porządku jest realizowany analogicznie jak poprzednie. W pierwszej kolejności sprawdza się, czy zbiory relacji niepodzielności w sekwencji B_P aplikacji P nie są puste. Jeśli tak nie jest, to dla operacji składających się na relacje dowolnego zbioru B_P^{ξ} weryfikowany jest warunek naruszenia:

- niepodzielności tzn. sprawdzane jest Twierdzenie 3,
- porządku tzn. sprawdzane jest Twierdzenie 4.

Jeżeli w aplikacji zachodzi którykolwiek z ww. warunków oznacza to, że może w niej wystąpić błąd prowadzący do naruszenia niepodzielności i/lub naruszenia porządku.

5.3 Prototyp narzędzia implementującego metodę RD-L

Implementacja metody RD-L może zostać wykonana w dowolnym języku programowania. Warto zaznaczyć, że niektóre etapy metody mogą być wykonywane równolegle w wątkach. Prowadzi to do sytuacji, w której narzędzie z zaimplementowaną metodą RD-L mogłoby dokonać sprawdzenia swojego kodu źródłowego (o ile kod aplikacji zostałby napisany w języku C z użyciem biblioteki *pthread*).

Aplikacja implementująca metodę lokalizowania konfliktów zasobowych składa się z dwóch głównych modułów. Pierwszy moduł odpowiada za budowę instancji modelu opisanego w podrozdziale 5.2.1. Drugi moduł natomiast odpowiada za analizę instancji modelu i lokalizowanie konfliktów zasobowych opisanych w podrozdziale 5.2.2. Zgodnie z tym założeniem narzędzie *rdao detector* (`https://github.com/PKPhdDG/rdao_detector`) implementujące metodę RD-L (napisane w języku Python) dzieli się na dwa moduły:

- *mascm_generator* - odpowiedzialny za transformację kodu źródłowego aplikacji P do instancji modelu C_P,
- *rdao_detector* - odpowiedzialny za analizę instancji modelu C_P.

Język Python został wybrany do implementacji z kilku powodów. Pierwszym z nich jest bogata biblioteka standardowa języka, w której skład wchodzą narzędzia wspierające przetwarzanie tekstu. Drugim powodem jest składnia języka Python, która wspiera modelowanie skomplikowanych abstrakcji (grafów, ścieżek itp.). Trzecim powodem jest bogaty zestaw otwarto-źródłowych (ang. open-source) bibliotek dostępny w repozytorium Python Package Index (w skrócie PyPI). Poprzez otwarto-źródłowe biblioteki należy rozumieć oprogramowanie, którego kod źródłowy jest rozpowszechniany publicznie z myślą o współpracy otwartej dla wszystkich przy jego rozwijaniu i kolektywnej produkcji. Brak podobnych rozwiązań sprawił, że zrezygnowano z innych popularnych języków programowania takich jak C, C++, C# czy Java. Dodatkowo warto zwrócić uwagę, że czas wytwarzania oprogramowania w tych językach zwykle jest dłuższy niż w przypadku języka Python. Docelowo narzędzie *rdao detector* powinno być możliwe do zastosowania na jak największej liczbie systemów operacyjnych, włącznie z systemami GNU/Linux. Aby to osiągnąć język programowania nie może ograniczać się tylko do platform Windows, MacOS i GNU/Linux. Istnieje szereg innych i mniej popularnych systemów

operacyjnych np. FreeBSD, AIX, RISC OS, Solaris czy HP-UX, które są stosowane w przemyśle, dla których istnieją porty interpretera języka Python, a które nie są oficjalnie wspierane przez twórców takich języków jak Java i C#.

5.3.1 Moduł *mascm_generator*

W kodzie modułu *mascm_generator* zaimplementowano kroki procedury opisanej w podrozdziale 5.2.1. Do wykonania dyrektyw preprocesora i lokalizacji głównych elementów aplikacji użyto wspomniany już kompilator języka C z pakietu GCC. Kompilator *gcc* wykorzystano także do wygenerowania drzew składniowych kodu źródłowego badanej aplikacji P, tzn. do transformacji kodu źródłowego do drzew, w których węzły reprezentują pewne konstrukcje języka C. Dzięki temu zlokalizowanie głównych składowych aplikacji P sprowadza się do znalezienia odpowiednich węzłów w wygenerowanym drzewie. Wyróżnia się trzy grupy węzłów drzew składniowych:

- Węzły funkcji - wykorzystywane w analizie przebiegu aplikacji.
- Węzły struktur i unii - wykorzystywane w procesie lokalizowania zasobów podczas przeglądu zmiennych lokalnych.
- Węzły zasobów
  - Węzły blokad - deklaracja stosowanych blokad;
  - Węzły zmiennych - deklaracja zasobów współdzielonych;

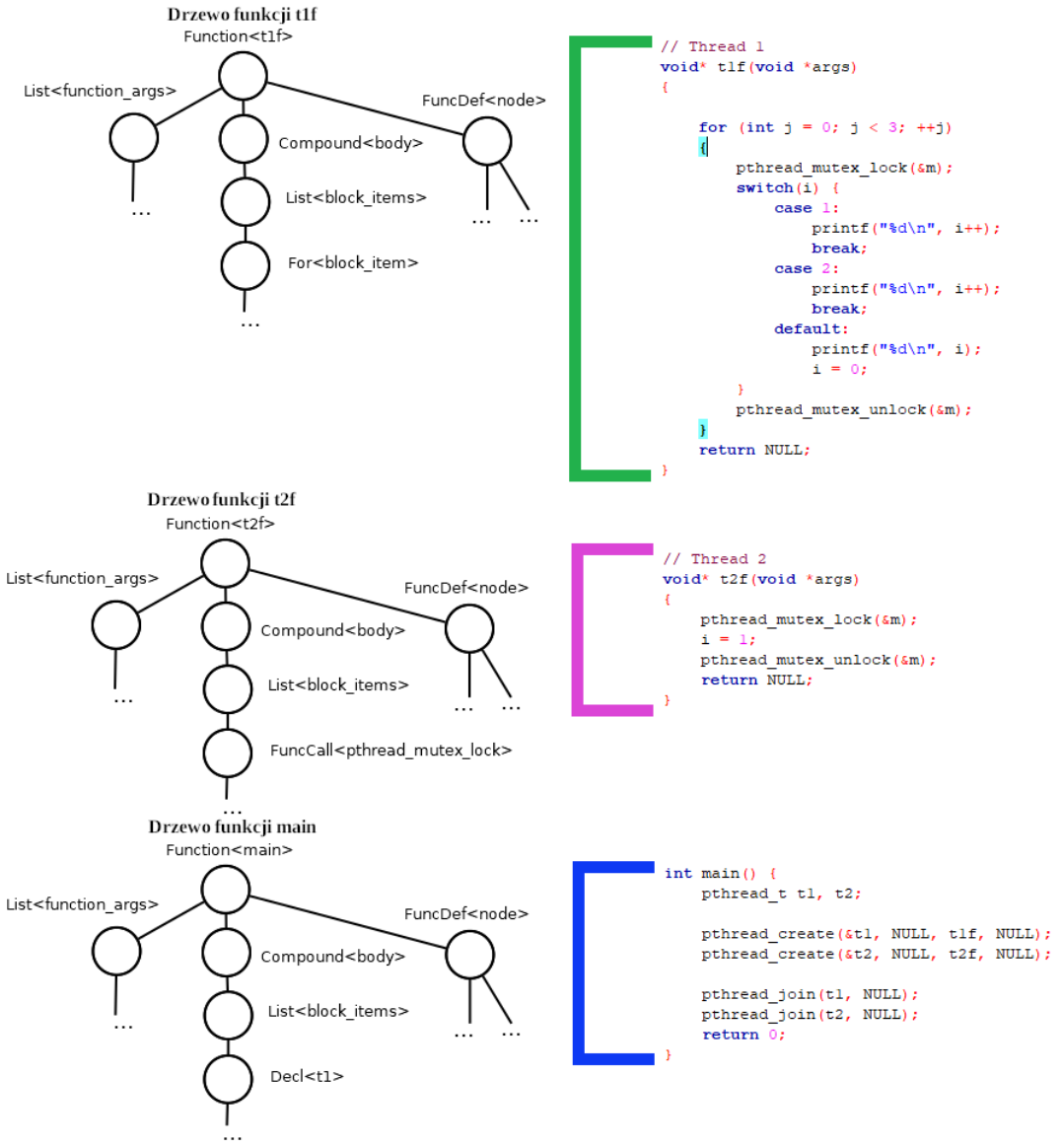
Należy pamiętać, że nie istnieje jeden standard drzew składniowych i mogą się one różnić w zależności od stosowanego kompilatora. Wspólną cechą wszystkich kompilatorów jest fakt, że każda definicja funkcji to pojedyncze drzewo składniowe, co pokazano na rysunku 5.10. Analiza przepływu aplikacji sprowadza się więc do przeglądu drzewa składniowego zgodnie z tym, jak to opisano w podrozdziale 5.2.1. W efekcie cały proces budowy modelu C_P polega na przeglądzie i transformacji wszystkich drzew składniowych.

5.3.2 Moduł *rdao_detector*

Analiza instancji modelu wyróżnia cztery kroki - zgodnie z opisem przedstawionym w podrozdziale 5.2.2. Odizolowanie od siebie etapów lokalizowania błędów konkretnych konfliktów zasobowych upraszcza proces testowania aplikacji. Zaletą takiego podejścia jest również uproszczenie procesu rozwoju takiego kodu np. doprecyzowanie warunków w celu zmniejszenia liczby zgłoszeń fałszywie pozytywnych. Jednoczesne lokalizowanie błędów wszystkich klas mogłoby doprowadzić do powstania tzw. „kodu spaghetti" (kodu nienadającego się do dalszego rozwoju).

5.4 Podsumowanie

Metoda RD-L umożliwia automatyzację procesu lokalizacji błędów, który powszechnie jest wykonywany ręcznie przez programistę (w ramach etapu przeglądu kodu wykonywanego w procesie wytwarzania oprogramowania). Automatyzacja tego procesu pozwala uniknąć wpływu tzw. czynnika ludzkiego (choroba, rozkojarzenie, brak wystarczającej wiedzy, itp.).



RYSUNEK 5.10: Przykład budowy drzew składniowych na podstawie struktury kodu źródłowego.

Metoda RD-L i implementujące tę metodę prototypowe narzędzie *rdao detector* opracowano w celu wspomagania programistów w procesie wytwarzania aplikacji wielowątkowych przy pomocy języka C i biblioteki *pthread*. Metoda może być wykorzystywana iteracyjnie w procesie pisania kodu źródłowego (co przedstawiono na rys. 5.1), a także stanowić składową procesu CI/CD, co pokazano na rys. 5.2.

Programiści języka C nie doczekali się narzędzi, które wspierałyby ich w procesie wytwarzania aplikacji wielowątkowych. Jest to o tyle istotne, że język C ze względu na swoją szybkość (tzn. czas wykonania tego samego algorytmu zaimplementowanego w innych językach jest dłuższy niż w C) wykorzystywany jest w branżach, gdzie czas reakcji ma ogromne znaczenie tj. branża energetyczna, branża medyczna czy motoryzacyjna. Aplikacje wykorzystywane w tego typu obszarach muszą być niezawodne. Opracowana metoda RD-L (oraz *rdao detector*) wypełnia więc lukę w obszarze technologii wspierających programistów wytwarzających tego typu oprogramowanie. Metoda została zweryfikowana w serii eksperymentów, których wyniki przedstawiono w kolejnym rozdziale.

Rozdział 6

Eksperymentalna ocena metody

6.1 Plan eksperymentów

W rozdziale przedstawiono wyniki eksperymentalnej oceny metody RD-L implementowanej w aplikacji *rdao detector*. W pierwszej kolejności omówiono założenia i charakterystykę obiektu badań. Następnie dla każdej z rozważanej klasy błędów przedstawiono przeprowadzone eksperymenty oraz scharakteryzowano otrzymane wyniki.

Celem eksperymentów było potwierdzenie lub falsyfikacja postawionej tezy tzn. poszukiwano odpowiedzi na pytanie: Czy metoda RD-L bazująca na opracowanym modelu aplikacji wielowątkowej (wraz z wyprowadzonymi dla niego warunkami) umożliwia lokalizowanie błędów, które prowadzą do konfliktów zasobowych? Analiza otrzymanych wyników pozwoliła także określić kryteria umożliwiające efektywne wsparcie procesu weryfikacji poprawności aplikacji wielowątkowych. Pierwszym weryfikowanym kryterium było zapotrzebowanie na pamięć operacyjną narzędzia *rdao detector*. Drugim weryfikowanym kryterium był czas analizy aplikacji wielowątkowych. Warunkiem użycia narzędzia *rdao detector* w trybie on-line było osiągnięcie czasu analizy aplikacji (na przeciętnej klasy komputerze) poniżej jednej godziny.

Eksperymenty obejmowały weryfikację 19 aplikacji (wszystkie napisane w języku C i biblioteki *pthread*). Dla każdej z nich przeprowadzono proces lokalizowania konfliktów zasobowych. Zgodnie ze z schematem przedstawionym na rysunku 5.4 sprowadzał się on głównie do:

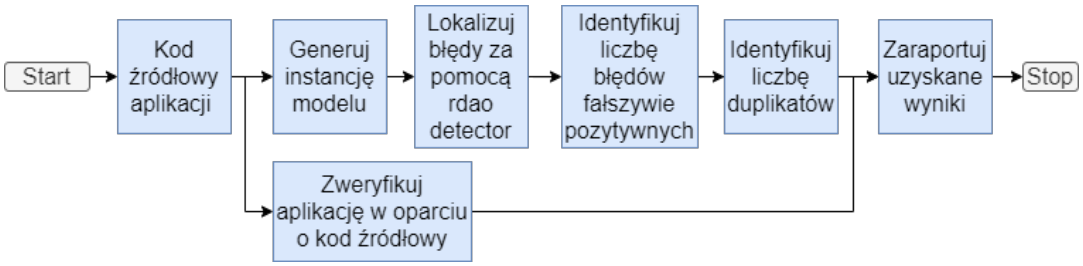
1. budowy instancji modelu C_P analizowanej aplikacji,
2. weryfikacji instancji modelu C_P pod względem spełnienia warunków wystąpienia błędów prowadzących do konfliktów zasobowych.

Proces ten realizowany był automatycznie przy użyciu aplikacji *rdao detector*. Otrzymane wyniki (raporty zawierające informacje o zlokalizowanych błędach) zostały zweryfikowane przez zespół ekspertów (programistów języka C), którzy dla każdej aplikacji dodatkowo przeprowadzali ten sam proces ręcznie (przegląd kodu źródłowego). Przebieg eksperymentów obejmujący automatyczną (aplikacja *rdao detector*) i manualną (zespół ekspertów) lokalizację błędów analizowanych aplikacji przedstawiono na rysunku 6.1.

Porównanie otrzymanych wyników pozwoliło określić m.in. liczbę niewykrytych błędów oraz liczbę fałszywie pozytywnych zgłoszeń. Każde zgłoszenie wygenerowane przez *rdao detector* zostało ocenione przez eksperta. Jeśli zgłoszenie wskazywało błąd, który także został znaleziony podczas analizy manualnej, to potwierdzało to skuteczność metody. W sytuacji, gdy zespół ekspertów nie znalazł zgłoszonego błędu, dokonywano ponownej oceny wskazanego fragmentu kodu. Jeśli

fragment ten okazywał się poprawny, zgłoszenie oznaczano jako fałszywie pozytywne.

Pełne raporty zawierające wyniki przeprowadzanych eksperymentów umieszczono w repozytorium pod adresem `https://github.com/PKPhdDG/mascm-expirements-docs` (ich najważniejsze fragmenty zawarto w dodatku B).



RYSUNEK 6.1: Algorytm przeprowadzonych eksperymentów.

6.2 Założenia

Metoda RD-L została opracowana z myślą o języku C i nie posiada żadnych ograniczeń co do standardu języka, w jakim napisany jest kod źródłowy weryfikowanej aplikacji. Na potrzeby eksperymentów przyjęto jednak, że ma on spełniać standard ISO C99. Inaczej mówiąc, kod aplikacji musi być możliwy do skompilowania w kompilatorze języka C pakietu GCC. Jako bibliotekę dostarczającą wielowątkowość wybrano bibliotekę *pthread* (gdzie mechanizm wzajemnego wykluczania gwarantowany jest poprzez stosowanie blokad - funkcja *pthread_mutex_lock*). Ponadto przyjęto, że w aplikacji występują tylko wątki łączone (patrz podrozdział 3.1.1).

Do eksperymentów wybrano 19 aplikacji, na które składają się:

- cztery autorskie aplikacje opracowane w trakcie prac nad narzędziem *rdao detector* (dostępne w repozytorium: `https://github.com/PKPhdDG/rdao_detector`),
- osiem aplikacji pochodzących z pakietu Phoenix (dostępne w repozytorium: `https://github.com/kozyraki/phoenix/tree/master/sample_apps`),
- siedem wersji aplikacji *s-mptcp* (dostępne w repozytorium: `https://github.com/PKPhdDG/smptcp`).

Aplikacje projektu Phoenix nie zawierają wszystkich możliwych wariantów błędów prowadzących do konfliktów zasobowych. Aby usunąć tę lukę wykorzystano aplikację *s-mptcp* pochodzącą ze zbiorów aplikacji o otwartych źródłach. Aplikacja ta została zweryfikowana przez zespół ekspertów (programistów C). Wyniki analizy pokazały, że jest ona wolna od błędów i może zostać użyta w eksperymentach, w których celowo wprowadzany jest błąd powodujący konflikt zasobowy. Na potrzeby tego typu eksperymentów opracowano 7 wersji aplikacji *s-mptcp*.

W skład przyjętego zbioru wchodzą aplikacje, które umożliwiają weryfikację metody RD-L pod względem wszystkich rozważanych w pracy błędów (tzn. prowadzących do: szkodliwej rywalizacji, zakleszczenia, naruszenia niepodzielności i naruszenia porządku). Warto pokreślić, że część z nich (autorskich aplikacji) pozwala na manualną analizę występujących w nich błędów.

Do eksperymentów wykorzystano komputer o parametrach:

- Procesor: AMD Ryzen 5 1500X, 3.5 GHz,

- RAM: Corsair Vengeance LPX, DDR3, 32GB, 3000MHz, CL15,
- Dysk: SSD Samsung 970 Evo 1 TB M.2 2280, Prędkość odczytu 3400 MB/s.

Na komputerze kontrolowanym przez system operacyjny Windows 10 zainstalowano parser języka Python w wersji 3.9.5., który wykorzystano do uruchomienia aplikacji *rdao detector*. Pomiary czasu pracy aplikacji oraz pomiary zużycia pamięci wykonano z użyciem wewnętrznego mechanizmu aplikacji *rdao detector*.

6.3 Charakterystyka obiektu

W repozytorium aplikacji *rdao detector* (w wersji oznaczonej numerem v1.1.2) w katalogu *example_c_sources* znajdują się pliki z kodem źródłowym czterech autorskich aplikacji wielowątkowych *RC1*, *DL1*, *AV1* i *OV1* (kod źródłowy tych aplikacji znajduje się również w dodatku A). Rozmiar tych aplikacji nie przekracza długości 50 wierszy kodu. Aplikacje te zostały celowo ograniczone do jak najmniejszej liczby wierszy kodu, tak aby uzyskać możliwie największą przejrzystość zaimplementowanego scenariusza, a ich wykorzystanie umożliwiało ocenę generowanej przez metodę RD-L liczby fałszywie pozytywnych zgłoszeń.

Pozostałe 15 aplikacji tworzy grupę, która zawiera różne błędy prowadzące do konfliktów zasobowych np. brak blokad chroniących współdzielone zasoby, wielokrotne zakładanie blokady w ciele pętli, niewłaściwy typ blokady, itp. Aplikacje z tej grupy zostały wykorzystane w eksperymentach umożliwiających ocenę liczby i rodzaju błędów niewykrywalnych przez metodę RD-L.

W skład analizowanych aplikacji wchodzi osiem aplikacji projektu Phoenix, który jest implementacją modelu MapReduce firmy Google opracowanego dla zadań intensywnego przetwarzania danych (ang. model for data-intensive processing tasks). W repozytorium Phoenix znajdują się trzy różne implementacje wspomnianego modelu:

- Phoenix 1 (w katalogu phoenix-1.0);
- Phoenix 2 (w katalogu phoenix-2.0);
- Phoenix++ (w katalogu phoenix++-1.0).

W eksperymentach prowadzonych na potrzeby niniejszej pracy stosowana jest implementacja Phoenix 2, ze względu na ogólnie podniesioną jakość kodu (tzn. poprawiono liczne błędy) jak i możliwość jej wykorzystania na systemach z jądrem Linux.

Zestaw badanych aplikacji składa się z następujących aplikacji:

- *histogram* - aplikacja generująca histogram częstotliwości wartości pikseli w kanałach czerwonym, zielonym i niebieskim obrazu bitmapowego;
- *kmeans* - aplikacja grupująca kilka n-wymiarowych punktów danych w określoną przez użytkownika liczbę grup;
- *linear regression* - aplikacja generująca statystyki zbiorcze punktów w celu liniowej aproksymacji wszystkich punktów;
- *matrix multiply* - aplikacja obliczająca iloczyn macierzy o tym samym rozmiarze;

- *pca* - aplikacja obliczająca wartość średnią i macierz kowariancji dla losowo wygenerowanej macierzy;
- *reverse index* - aplikacja wyodrębniająca z plików HTML tzw. *anchor tag* i generująca na jego podstawie indeks linków do stron;
- *string match* - aplikacja przeglądająca listę zaszyfrowanych słów w celu zlokalizowania kluczy, dostarczanych w postaci pliku;
- *word count* - aplikacja zliczająca częstotliwość występowania każdego unikalnego słowa w dokumencie tekstowym.

Ostatnią aplikacją wykorzystywaną w badaniach jest *s-mptcp*. Jest to aplikacja kliencka do komunikacji/transferu danych z wykorzystaniem protokółu MPTCP. Kod aplikacji opublikowano na portalu GitHub i dostępny jest na licencji GNU/GPL3. Aplikacja ta, podobnie jak wiele innych aplikacji o otwartych źródłach, napisana została z wykorzystaniem reguły „zrób to prosto, głupcze" (ang. Keep It Simple, Stupid lub KISS). Jednym z efektów stosowania tej reguły jest czytelny i łatwy w analizie kod źródłowy. Z tego właśnie powodu aplikacja *s-mptcp* została wybrana do eksperymentów. Kod tej aplikacji nie posiada błędów prowadzących do szkodliwej rywalizacji, zakleszczenia, naruszenia niepodzielności i naruszenia porządku. Została ona wykorzystana jako wzorzec, do którego „wstrzykiwano" różnego rodzaju błędy celem ich późniejszej identyfikacji. I tak powstało następujących 7 wersji tej aplikacji:

- *s-mptcp RC1* - wersja aplikacji z błędem prowadzącym do szkodliwej rywalizacji,
- *s-mptcp DL1* - wersja aplikacji z błędem prowadzącym do zakleszczenia DL1,
- *s-mptcp DL2* - wersja aplikacji z błędem prowadzącym do zakleszczenia DL2,
- *s-mptcp DL3* - wersja aplikacji z błędem prowadzącym do zakleszczenia DL3,
- *s-mptcp DL4* - wersja aplikacji z błędem prowadzącym do zakleszczenia DL4,
- *s-mptcp AV1* - wersja aplikacji z błędem prowadzącym do naruszenia niepodzielności,
- *s-mptcp OV1* - wersja aplikacji z błędem prowadzącym do naruszenia porządku.

Wstrzykiwane w aplikację *s-mptcp* błędy zostały przygotowane jako zestaw poprawek udostępnionych publicznie pod adresem `https://github.com/PKPhdDG/smptcp`). Eksperymenty prowadzone na tak przygotowanym zbiorze pozwoliły ocenić skuteczność metody RD-L dla aplikacji o rozmiarach spotykanych w praktyce.

6.4 Lokalizowanie szkodliwej rywalizacji

Na listingu 1 w dodatku A przedstawiono kod aplikacji zawierającej błąd prowadzący do szkodliwej rywalizacji. Jest to najprostszy przykład aplikacji, której skompilowanie i uruchomienie zakończy się działaniem odbiegającym od założonego scenariusza. Zaimplementowany scenariusz zakłada, że każdy z wątków zwiększy wartość zasobu współdzielonego r_1 (zmienna $r1$) o 1 milion. Instancja modelu C_{RC1} kodu źródłowego aplikacji *RC1* przedstawia się następująco [37]:

$T_{RC1} = \{t_0, t_1, t_2\}$

$U_{RC1} = (\{t_0\}, \{t_1, t_2\}, \{t_0\})$

$R_{RC1} = \{\{r1\}\}$

$O_{RC1} = \{o_{0,1}, o_{0,2}, o_{0,3}, o_{0,4}, o_{0,5}, o_{0,6}, o_{0,7}, o_{0,8}, o_{0,9}, o_{1,1}, o_{1,2}, o_{1,3}, o_{1,4}, o_{1,5}, o_{1,6}, o_{2,1}, o_{2,2}, o_{2,3}, o_{2,4}, o_{2,5}, o_{2,6}\}$

$Q_{RC1} = \varnothing$

$F_{RC1} = \{(o_{0,1}, o_{0,2}), (o_{0,2}, o_{0,3}), (r_1, o_{0,3}), (o_{0,3}, o_{0,4}), (o_{0,4}, o_{0,5}), (o_{0,5}, o_{0,6}), (o_{0,6}, o_{0,7}), (o_{0,7}, o_{0,8}), (r_1, o_{0,8}), (o_{0,8}, o_{0,9}), (o_{1,1}, o_{1,2}), (o_{1,2}, o_{1,6}), (o_{1,2}, o_{1,3}), (o_{1,3}, o_{1,4}), (o_{1,4}, r_1), (o_{1,4}, o_{1,5}), (o_{1,5}, o_{1,2}), (o_{1,5}, o_{1,6}), (o_{2,1}, o_{2,2}), (o_{2,2}, o_{2,6}), (o_{2,2}, o_{2,3}), (o_{2,3}, o_{2,4}), (o_{2,4}, r_1), (o_{2,4}, o_{2,5}), (o_{2,5}, o_{2,2}), (o_{2,5}, o_{2,6})\}$

$B_{RC1} = \varnothing$

Wynikiem analizy powyższego modelu (raport wygenerowany przez *rdao detector* - listing 8) są dwa zgłoszenia o błędach skutkujących szkodliwą rywalizacją. Oba zgłoszenia wskazują na tą samą operację w kodzie źródłowym - dwa wątki aplikacji t_1 oraz t_2 wykonują ten sam kod, stąd duplikat zgłoszenia. W raporcie z analizy nie pojawiły się zgłoszenia fałszywie pozytywne. Zlokalizowany błąd pokrywa się z wynikiem uzyskanym w procesie ręcznej weryfikacji aplikacji RC1 przez zespół ekspertów.

Analogiczne eksperymenty przeprowadzono dla aplikacji z projektu Phoenix oraz aplikacji *s-mptcp RC1* (wersja z błędem prowadzącym do szkodliwej rywalizacji). Wyniki eksperymentów zostały zebrane w tabeli 6.1. Tabela zawiera kolumny z:

1. nazwą aplikacji, której kod był analizowany,
2. liczbą błędów zlokalizowanych w procesie manualnej weryfikacji kodu aplikacji przez ekspertów,
3. wynikami weryfikacji automatycznej, na którą składają się kolumny z liczbą:
   (a) konfliktów zgłoszonych przez aplikację *rdao detector*,
   (b) zgłoszeń, które okazały się fałszywie pozytywne w procesie ręcznej analizy kodu źródłowego przez ekspertów – weryfikacja manualna przez zespół ekspertów,
   (c) zgłoszeń będących duplikatami wcześniej przeanalizowanych zgłoszeń – weryfikacja manualna przez zespół ekspertów,
   (d) poprawnie zlokalizowanych konfliktów zasobowych – weryfikacja manualna przez zespół ekspertów.

Raport z analizy kodu źródłowego aplikacji *histogram* (listing 9) zawierał 81 zgłoszeń, z czego 15 okazało się zgłoszeniami fałszywie pozytywnymi, a 45 z nich to duplikaty wcześniej występujących zgłoszeń. Natomiast 21 zgłoszeń dotyczyło błędów, które rzeczywiście prowadzą do wystąpienia szkodliwej rywalizacji.

W przypadku aplikacji *kmeans* (listing 10) raport zawierał aż 141 zgłoszeń. Ponad połowa, bo aż 73 zgłoszeń to zgłoszenia fałszywie pozytywne, a 47 zgłoszeń okazało się duplikatami. Tylko 21 ze wszystkich zgłoszeń dotyczyło błędów, które rzeczywiście prowadzą do wystąpienia szkodliwej rywalizacji.

Trzecią aplikacją, w której udało zlokalizować się błędy prowadzące do szkodliwej rywalizacji jest *pca*. W raporcie z wykonanej analizy (listing 13) znalazło się 226 zgłoszeń, z czego 60 to zgłoszenia fałszywie pozytywne, a 108 to duplikaty. Potwierdzonych zostało 58 zgłoszeń, w których wskazano miejsca z błędami prowadzącymi do szkodliwej rywalizacji.

Ostatnią aplikacją projektu Phoenix, w której udało się zlokalizować błędy skutkujące szkodliwą rywalizacją jest aplikacja *reverse index*. Raport z przeprowadzonej analizy (listing 14) zawierał 147 zgłoszeń, 85 to zgłoszenia fałszywie pozytywne, 43 to duplikaty, a 19 zgłoszeń wskazywało na miejsca, w których błędy prowadzą do szkodliwej rywalizacji.

Analiza kodu źródłowego pozostałych czterech aplikacji tzn. *linear regression*, *matrix multiply*, *string match* i *word count* nie wykazała w kodzie obecności tego typu błędów (listingi 11, 12, 15, 16).

Duplikaty obecne w raportach najczęściej są wynikiem uruchamiania w aplikacjach wielu wątków, które wykonują ten sam kod, w którym dochodzi do konfliktów zasobowych. Należy także pamiętać, że wśród duplikatów znajdują się zarówno zgłoszenia rzeczywistych błędów jak i zgłoszenia fałszywie pozytywne.

TABLICA 6.1: Wyniki eksperymentu lokalizowania błędów skutkujących konfliktami zasobowymi typu szkodliwa rywalizacja.

Nazwa aplikacji	Weryfikacja ręczna – zespół ekspertów	Weryfikacja automatyczna – rdao detector			
	Liczba błędów	Liczba zgłoszonych konfliktów	Liczba zgłoszeń fałszywie pozytywnych (ocena ekspertów)	Liczba duplikatów (ocena ekspertów)	Liczba potwierdzonych błędów (ocena ekspertów)
RC1	1	2	0	1	1
histogram	21	81	15	45	21
kmeans	21	141	73	47	21
linear regression	0	0	0	0	0
matrix multiply	0	0	0	0	0
pca	58	226	60	108	58
reverse index	19	147	85	43	19
string match	0	0	0	0	0
word count	0	0	0	0	0
s-mptcp RC1	2	218	215	1	2
Suma	122	815	448	245	122

Ostatnią aplikacją, której kod poddano analizie jest *s-mptcp*. Na potrzeby tego eksperymentu do kodu aplikacji *s-mptcp* wprowadzono zmiany (błędy), skutkujące wystąpieniem szkodliwej rywalizacji (patrz repozytorium *race_condition.patch*). Taki kod aplikacji został poddany analizie z użyciem aplikacji *rdao detector*. Wyniki przedstawiono w tabeli 6.1 (wiersz *s-mptcp RC1*). Raport zawierał 218 zgłoszeń z czego 215 to zgłoszenia fałszywie pozytywne. Wśród trzech pozostałych zgłoszeń jedno było duplikatem a dwa wskazywały fragmenty kodu dopuszczającego niekontrolowany dostęp do zasobu współdzielonego. Zgłoszenia te dotyczyły kodu, który został dodany do kodu aplikacji na potrzeby eksperymentu.

Przeprowadzone eksperymenty pokazały, że metoda RD-L (implementowana w narzędziu *rdao detector*) wykrywa tę samą liczbę błędów co zespół ekspertów analizujących kod aplikacji ręcznie. Metoda ta charakteryzuje się jednak wysoką liczbą fałszywie pozytywnych zgłoszeń. Wyniki eksperymentów (tabela 6.1) pokazały, że blisko 55% wszystkich zgłoszeń stanowiły zgłoszenia fałszywie pozytywne. Ponadto, co trzecie zgłoszenie okazało się duplikatem.

6.4.1 Sposoby eliminacji błędów prowadzących do szkodliwej rywalizacji

Aby nie dopuścić do błędów prowadzących do szkodliwej rywalizacji, operacje wykorzystujące zasoby współdzielone powinny zostać poprzedzone operacją założenia blokady. Należy przy tym pamiętać, że operacje mające się wykluczać, powinny stosować tę samą blokadę. Najczęściej stosowanym rozwiązaniem jest założenie blokad w liczbie równej liczbie zasobów współdzielonych. Innymi słowy, pojedyncza blokada „pilnuje" wszystkich operacji wykonywanych na pojedynczym zasobie. Wadą tego rozwiązania jest to, że jeśli w aplikacji występują zasoby, których stan jest ściśle powiązany i zmiana wartości jednego zasobu wymusza zmianę stanu pozostałych zasobów to liczba blokad jest nadmiarowa. Wszystkie operacje wykonywane na dowolnym zasobie należącym do zbioru mogą zostać zabezpieczone pojedynczą blokadą, dzięki czemu redukuje się szansę wystąpienia zakleszczenia. Stosowanie tej zasady skutkuje zwiększeniem szybkości aplikacji i zmniejszeniem ilości używanej pamięci operacyjnej.

6.4.2 Podsumowanie

Przeprowadzone eksperymenty pokazały, że metoda RD-L bazująca na opracowanym modelu C_P kodu źródłowego aplikacji wielowątkowej, zaimplementowana w aplikacji *rdao detector*, pozwala lokalizować błędy, które powodują konflikty zasobowe szkodliwej rywalizacji. Należy jednak zaznaczyć, że raporty generowane przez aplikację *rdao detector* zawierają zgłoszenia fałszywie pozytywne. Zgłoszenia te uznano za fałszywie pozytywne w procesie manualnej analizy instancji modelów badanych aplikacji wykonanej przez zespół ekspertów. Występowanie zgłoszeń fałszywie pozytywnych jest skutkiem wykorzystania w procesie lokalizowania błędów warunków koniecznych (Twierdzenie 1), których spełnienie oznacza, że dany fragment kodu może zawierać błąd prowadzący do konfliktu zasobowego. Innymi słowy, metoda RD-L charakteryzuje się nadmiarowością liczby zgłoszeń (część z nich jest więc zgłoszeniami fałszywie pozytywnymi). Należy jednak podkreślić, że sytuacja odwrotna, czyli gdy Twierdzenie 1 nie jest spełnione dla żadnego fragmentu kodu aplikacji wielowątkowej oznacza, że jest ona wolna od błędów prowadzących do szkodliwej rywalizacji. W praktyce oznacza to (co potwierdziły również eksperymenty), że metoda RD-L wykrywa wszystkie występujące w kodzie błędy prowadzące do szkodliwej rywalizacji.

6.5 Lokalizowanie zakleszczenia

Poniżej przedstawiono instancję modelu C_{DL1} kodu źródłowego aplikacji *DL1*, który można znaleźć w dodatku A (listing 2) i w repozytorium aplikacji *rdao detector* w pliku *deadlock1.c*.

$T_{DL1} = \{t_0, t_1, t_2\}$

$U_{DL1} = (\{t_0\}, \{t_1, t_2\}, \{t_0\})$

$R_{DL1} = \{\{r1\}\}$

$O_{DL1} = \{o_{0,1}, o_{0,2}, o_{0,3}, o_{0,4}, o_{0,5}, o_{0,6}, o_{0,7}, o_{0,8}, o_{0,9}, o_{1,1}, o_{1,2}, o_{1,3}, o_{1,4}, o_{1,5}, o_{1,6}, o_{1,7}, o_{1,8}, o_{1,9}, o_{1,10}, o_{2,1}, o_{2,2}, o_{2,3}, o_{2,4}, o_{2,5}, o_{2,6}, o_{2,7}, o_{2,8}, o_{2,9}, o_{2,10}\}$

$Q_{DL1} = \{q_1 = (m, PMN), q_2 = (n, PMN)\}$

$F_{DL1} = \{(o_{0,1}, o_{0,2}), (o_{0,2}, o_{0,3}), (r_1, o_{0,3}), (o_{0,3}, o_{0,4}), (o_{0,4}, o_{0,5}), (o_{0,5}, o_{0,6}), (o_{0,6}, o_{0,7}), (o_{0,7}, o_{0,8}), (r_1, o_{0,8}), (o_{0,8}, o_{0,9}), (q_2, o_{1,1}), (o_{1,1}, o_{1,2}), (q_1, o_{1,2}), (o_{1,2}, o_{1,3}), (o_{1,3}, o_{1,4}), (o_{1,4}, o_{1,8}), (o_{1,4}, o_{1,5}), (o_{1,5}, o_{1,6}), (o_{1,6}, r_1), (o_{1,6}, o_{1,7}), (o_{1,7}, o_{1,4}), (o_{1,7}, o_{1,8}), (o_{1,8}, q_1), (o_{1,8}, o_{1,9}), (o_{1,9}, q_2), (o_{1,9}, o_{1,10}), (q_1, o_{2,1}), (o_{2,1}, o_{2,2}), (q_2, o_{2,2}), (o_{2,2}, o_{2,3}), (o_{2,3}, o_{2,4}), (o_{2,4}, o_{2,8}), (o_{2,4}, o_{2,5}), (o_{2,5}, o_{2,6}), (o_{2,6}, r_1), (o_{2,6}, o_{2,7}), (o_{2,7}, o_{2,4}), (o_{2,7}, o_{2,8}), (o_{2,8,q_2}), (o_{2,8}, o_{2,9}), (o_{2,9}, q_1), (o_{2,9}, o_{2,10})\}$

$B_{DL1} = \varnothing$

Kolejność zakładania par blokad q_1, q_2 w wątkach t_1, t_2 aplikacji *DL1* jest różna, co może skutkować ich zakleszczeniem. Jak wspomniano w rozdziale 2, jeśli planista systemu operacyjnego uruchomi wątki jednocześnie (bądź różnica czasu między uruchomieniem będzie nieznaczna) i oba wątki założą pierwszą z blokad równocześnie, to dojdzie do ich wzajemnego oczekiwania na siebie, czyli dojdzie do zakleszczenia. Raport z analizy instancji modelu C_{DL1} (wygenerowany przez *rdao detector* - listing 18) zawiera tylko jedno zgłoszenie opisujące przyczynę zakleszczenia. Aplikacja *rdao detector* właściwie zlokalizowała tę część kodu źródłowego, która zawiera błąd prowadzący do zakleszczenia. Zlokalizowany błąd pokrywa się z wynikiem manualnej weryfikacji przeprowadzonej przez zespół ekspertów.

Analogiczne eksperymenty przeprowadzono dla aplikacji z projektu Phoenix oraz aplikacji: *s-mptcp DL1*, *s-mptcp DL2*, *s-mptcp DL3*, *s-mptcp DL4* (wersje z błędami prowadzącymi do zakleszczeń typu DL1-DL4).

Analiza kodu źródłowego przykładowych aplikacji pochodzących z projektu Phoenix nie wykazała obecności konfliktów zasobowych skutkujących błędami klasy zakleszczenia w żadnej z aplikacji (listingi 9, 10, 11, 12, 13, 14, 15, 16).

W przypadku aplikacji *s-mptcp* lokalizowanie zakleszczeń wykonywano cztery razy. Za każdym razem lokalizowano zakleszczenie, które posiadało inną przyczynę zgodnie ze scenariuszami opisanymi w podrozdziale 4.2. Wyniki wszystkich przeprowadzonych eksperymentów znajdują się w tabeli 6.2, natomiast fragmenty raportów z tych eksperymentów można znaleźć na listingach 19, 20, 21, 22.

Z tabeli 6.2 wynika, że we wszystkich scenariuszach DL1-DL4 zlokalizowano błędy skutkujące zakleszczeniem. W sumie zlokalizowano 4 błędy. Scenariusz implementujący przypadek wykluczających się par blokad (rezultat *s-mptcp DL1*) posiadał dodatkowo duplikat poprawnie zlokalizowanego błędu.

Analogicznie jak poprzednio, przeprowadzone eksperymenty pokazały, że wyniki metody RD-L pokrywają się z oceną Ekspertów analizujących kod aplikacji manualnie. Na raport składający się z 94 zgłoszeń 94% z nich (88 zgłoszeń) stanowiły zgłoszenia fałszywie pozytywne.

TABLICA 6.2: Wyniki eksperymentu lokalizowania błędów skutkujących konfliktami zasobowymi typu zakleszczenie.

Nazwa aplikacji	Weryfikacja ręczna – zespół ekspertów	Weryfikacja automatyczna – rdao detector			
	Liczba błędów	Liczba zgłoszonych konfliktów	Liczba zgłoszeń fałszywie pozytywnych (ocena ekspertów)	Liczba duplikatów (ocena ekspertów)	Liczba potwierdzonych błędów (ocena ekspertów)
DL1	1	1	0	0	1
histogram	0	0	0	0	0
kmeans	0	0	0	0	0
linear regression	0	0	0	0	0
matrix multiply	0	0	0	0	0
pca	0	0	0	0	0
reverse index	0	0	0	0	0
string match	0	0	0	0	0
word count	0	0	0	0	0
s-mptcp DL1	1	24	22	1	1
s-mptcp DL2	1	23	22	0	1
s-mptcp DL3	1	23	22	0	1
s-mptcp DL4	1	23	22	0	1
Suma	5	94	88	1	5

6.5.1 Sposoby eliminacji błędów prowadzących do zakleszczenia

Błędy, których skutkiem są zakleszczenia, wynikają z oczekiwania na zwolnienie zasobu, jakim jest blokada. „Naiwne" rozwiązania, polegające na usunięciu wszystkich blokad, są przyczyną błędów skutkujących szkodliwą rywalizacją. Z kolei rozwiązania odwrotne, sprowadzające się do stosowania tej samej blokady do każdej operacji wykorzystującej zasób współdzielony, redukuje możliwość wielowątkowej realizacji aplikacji (aplikacja sprowadzana jest do aplikacji jednowątkowej).

Z tego względu stosowanych jest wiele „dobrych" praktyk m.in. wspomniana wcześniej zasada przypisywania blokad do konkretnego zasobu lub konkretnych grup zasobów. Dzięki temu podejściu redukuje się liczbę stosowanych blokad, co prowadzi do zmniejszenia szansy na zakleszczenia typu DL1. Rozwiązanie takie nie zapobiega jednak przed scenariuszami prowadzącymi do zakleszczeń typu DL2 oraz DL3. Aby zapobiec tego typu zakleszczeniom, należałoby zawsze zakładać i zwalniać blokady poza ciałem wszelkich instrukcji sterujących. Nie zawsze jednak możliwe jest zaprojektowanie aplikacji tak, aby spełnić ten warunek. W zakleszczeniach typu DL4 istotny jest z kolei typ blokady. Jeśli w ciele funkcji rekurencyjnej zakładana jest blokada, musi być ona typu PMR, inaczej dojdzie do zakleszczenia.

Podsumowując, zapobieganie błędom, których skutkiem są zakleszczenia, jest procesem złożonym. Zabezpieczając operacje na zasobach współdzielonych, należy uwzględniać: typ blokady, to czy funkcja, w której blokada jest zakładana, jest funkcją rekurencyjną, a także czy operacja ta znajduje się w ciele innej instrukcji sterującej. Należy zaznaczyć, że nieodpowiednie zabezpieczenie operacji może powodować błędy o charakterze wyścigu.

6.5.2 Podsumowanie

Przeprowadzone eksperymenty pokazały, że metoda RD-L bazująca na opracowanym modelu C_P kodu źródłowego aplikacji wielowątkowej, zaimplementowana w aplikacji *rdao detector*, pozwala lokalizować błędy, które powodują zakleszczenia. Liczba fałszywie pozytywnych zgłoszeń jest większa niż w przypadku lokalizacji błędów prowadzących do szkodliwej rywalizacji. Wynika to głównie z większej złożoności struktur kodu źródłowego, składających się na błąd prowadzący do zakleszczenia.

Należy podkreślić, że liczba wykrytych błędów jest taka sama jak w przypadku manualnej weryfikacji przeprowadzonej przez zespół ekspertów.

6.6 Lokalizowanie naruszenia niepodzielności

Lokalizowanie błędów będących przyczyną naruszenia niepodzielności jest procesem złożonym, ale bardzo podobnym do procesu lokalizowania błędów prowadzących do zakleszczenia. Przykład kodu źródłowego aplikacji *AV1*, w której dochodzi do naruszenia niepodzielności znajduje się na listingu 6 (repozytorium aplikacji *rdao detector*, plik atomicity_violation1.c). Instancja modelu C_{AV1} tej aplikacji prezentuje się następująco:

$$T_{AV1} = \{t_0, t_1, t_2\}$$

$$U_{AV1} = (\{t_0\}, \{t_1, t_2\}, \{t_0\})$$

$$R_{AV1} = \{\{r1\}\}$$

$$O_{AV1} = \{o_{0,1}, o_{0,2}, o_{0,3}, o_{0,4}, o_{0,5}, o_{0,6}, o_{0,7}, o_{0,8}, o_{0,9}, o_{1,1}, o_{1,2}, o_{1,3}, o_{1,4}, o_{1,5}, o_{1,6}, o_{1,7}, o_{1,8}, o_{1,9}, o_{1,10}, o_{1,11}, o_{1,12}, o_{2,1}, o_{2,2}, o_{2,3}, o_{2,4}, o_{2,5}, o_{2,6}, o_{2,7}, o_{2,8}, o_{2,9}, o_{2,10}, o_{2,11}, o_{2,12}\}$$

$$Q_{AV1} = \{(m, PMN)\}$$

$$F_{AV1} = \{(o_{0,1}, o_{0,2}), (o_{0,2}, o_{0,3}), (r_1, o_{0,3}), (o_{0,3}, o_{0,4}), (o_{0,4}, o_{0,5}), (o_{0,5}, o_{0,6}), (o_{0,6}, o_{0,7}), (o_{0,7}, o_{0,8}), (r_1, o_{0,8}), (o_{0,8}, o_{0,9}), (o_{1,1}, o_{1,2}), (o_{1,2}, o_{1,12}), (o_{1,2}, o_{1,3}), (o_{1,3}, o_{1,4}), (q_1, o_{1,4}), (o_{1,4}, o_{1,5}), (o_{1,5}, r_1), (o_{1,5}, o_{1,6}), (o_{1,6}, q_1), (o_{1,6}, o_{1,7}), (o_{1,7}, o_{1,8}), (q_1, o_{1,8}), (o_{1,8}, o_{1,9}), (r_1, o_{1,9}), (o_{1,9}, o_{1,10}), (o_{1,10}, q_1), (o_{1,10}, o_{1,11}), (o_{1,11}, o_{1,2}), (o_{1,11}, o_{1,12}), (o_{2,1}, o_{2,2}), (o_{2,2}, o_{2,12}), (o_{2,2}, o_{2,3}), (o_{2,3}, o_{2,4}), (q_1, o_{2,4}), (o_{2,4}, o_{2,5}), (o_{2,5}, r_1), (o_{2,5}, o_{2,6}), (o_{2,6}, q_1), (o_{2,6}, o_{2,7}), (o_{2,7}, o_{2,8}), (q_1, o_{2,8}), (o_{2,8}, o_{2,9}), (r_1, o_{2,9}), (o_{2,9}, o_{2,10}), (o_{2,10}, q_1), (o_{2,10}, o_{2,11}), (o_{2,11}, o_{2,2}), (o_{2,11}, o_{2,12})\}$$

$$B_{AV1}^{SYM} = \{(o_{1,5}, o_{1,9}), (o_{2,5}, o_{2,9})\}$$

Raport z analizy instancji modelu C_{AV1} (wygenerowany przez *rdao detector* - listing 23) zawiera dwa zgłoszenia dotyczące błędów naruszenia niepodzielności. Oba zgłoszenia wskazują ten sam fragment kodu źródłowego, w którym występuje błąd prowadzący do konfliktu zasobowego. Ten sam fragment kodu źródłowego został wskazany przez ekspertów przeprowadzających manualną weryfikację aplikacji. Poza zgłoszeniami dotyczącymi błędów klasy naruszenia niepodzielności w raporcie znajdują się także zgłoszenia fałszywie pozytywne dotyczące błędów powodujących zakleszczenia.

Niewłaściwa identyfikacja błędów (wskazywanie błędów prowadzących do zakleszczeń jako błędy naruszenia porządku) wynika z pewnego podobieństwa ich struktury. W tym przypadku zlokalizowane fragmenty kodu źródłowego spełniają także warunki wystąpienia błędów prowadzących do zakleszczenia.

TABLICA 6.3: Wyniki eksperymentu lokalizowania błędów skutkujących konfliktami zasobowymi typu naruszenie niepodzielności.

Nazwa aplikacji	Weryfikacja ręczna – zespół ekspertów	Weryfikacja automatyczna – rdao detector			
	Liczba błędów	Liczba zgłoszonych konfliktów	Liczba zgłoszeń fałszywie pozytywnych (ocena ekspertów)	Liczba duplikatów (ocena ekspertów)	Liczba potwierdzonych błędów (ocena ekspertów)
AV1	1	2	0	1	1
histogram	0	0	0	0	0
kmeans	0	0	0	0	0
linear regression	0	0	0	0	0
matrix multiply	0	0	0	0	0
pca	0	0	0	0	0
reverse index	0	0	0	0	0
string match	0	0	0	0	0
word count	0	0	0	0	0
s-mptcp AV1	1	2	0	1	1
Suma	2	4	0	2	2

Analogiczne eksperymenty przeprowadzono dla aplikacji z projektu Phoenix oraz aplikacji *s-mptcp AV1*, (wersja z błędem prowadzącym do naruszenia niepodzielności). Wyniki eksperymentów przedstawiono w tabeli 6.3. Jak łatwo zauważyć, wszystkie aplikacje projektu Phoenix są wolne od błędów naruszenia niepodzielności. Manualna weryfikacja tych aplikacji potwierdza te wyniki.

Raport z analizy kodu aplikacji *s-mptcp* (w tabeli 6.3 jako *s-mptcp AV1*, fragment raportu znajduje się na listingu 24) zawiera dwa zgłoszenia. Podobnie jak w przypadku aplikacji *AV1*, tak i tutaj oba zgłoszenia dotyczą tego samego błędu.

Fałszywie pozytywne zgłoszenia, które znajdują się w raportach aplikacji *AV1* oraz *s-mptcp AV1* posiadają tę samą przyczynę. W kodzie źródłowym obu aplikacji uruchamiane są dwa wątki wykonujące ten sam kod, a w każdym z tych wątków istnieją po dwie sekcje krytyczne, w których znajduje się para operacji będących ze sobą w relacji kolejnościowej. W efekcie druga operacja pary wykonywana w pierwszym wątku może zostać uruchomiona przed pierwszą operacją pary w drugim wątku. Struktura ta spełnia warunki wystąpienia naruszenia porządku, pomimo że do tego naruszenia nie dojdzie.

Przeprowadzone eksperymenty pokazały, że wyniki metody RD-L pokrywają się z oceną Ekspertów analizujących kod aplikacji ręcznie. Metoda ta charakteryzuje się brakiem fałszywie pozytywnych zgłoszeń.

6.6.1 Sposoby eliminacji błędów prowadzących do naruszenia niepodzielności

Eliminowanie błędów prowadzących do naruszenia niepodzielności jest wyjątkowo trudne, gdyż podstawą konfliktu zasobowego jest zakłócenie relacji niepodzielności między dwiema operacjami. Język C nie posiada żadnych mechanizmów, które

pozwalałyby na zdefiniowanie tego typu relacji, przez co nie można ich zidentyfikować poprzez przegląd kodu źródłowego aplikacji. Pierwszym krokiem eliminacji tego typu błędów jest zatem zidentyfikowanie relacji niepodzielności bazując na dokumentacji i wiedzy programistów. Następnie należy zlokalizować w kodzie źródłowym każdą operację składającą się na niepodzielną parę. Dla każdej tego typu pary należy upewnić się, czy obie operacje składające się na tę parę znajdują się w tej samej sekcji krytycznej.

6.6.2 Podsumowanie

Przeprowadzone eksperymenty pokazały, że metoda RD-L zaimplementowana w aplikacji *rdao detector* pozwala lokalizować błędy, które powodują naruszenie niepodzielności.

Podobnie jak poprzednio, w uzyskanych raportach wystąpiły zgłoszenia fałszywie pozytywne, żadne z nich nie wskazywało jednak na błąd powodujący naruszenie niepodzielności. Zgłoszenia te pojawiają się zawsze gdy w aplikacji uruchamiane są co najmniej dwa wątki, w kodzie których operacje tworzące parę operacji (będących ze sobą w relacji niepodzielności) znajdują się w dwóch różnych sekcjach krytycznych. Należy zaznaczyć, że lokalizowanie tego typu błędów warunkowane jest posiadaniem informacji o tym, które operacje są powiązane relacją niepodzielności.

6.7 Lokalizowanie naruszenia porządku

Kod ostatniej przykładowej aplikacji *OV1* znajduje się na listingu 7 w dodatku A (a także w pliku order_violation1.c, repozytorium aplikacji *rdao detector*). Aplikacja *OV1* dopuszcza taki przebieg wątków, w którym dochodzi do naruszenia porządku operacji $o_{1,3}$ oraz $o_{2,7}$. Instancja modelu C_{OV1} kodu tej aplikacji wygląda następująco.

$T_{OV1} = \{t_0, t_1, t_2\}$

$U_{OV1} = (\{t_0\}, \{t_1, t_2\}, \{t_0\})$

$R_{OV1} = \{\{account, args\}\}$

$O_{OV1} = operations = \{o_{0,1}, o_{0,2}, o_{0,3}, o_{0,4}, o_{0,5}, o_{0,6}, o_{0,7}, o_{0,8}, o_{0,9}, o_{0,10}, o_{0,11}, o_{1,1}, o_{1,2}, o_{1,3}, o_{1,4}, o_{1,5}, o_{1,6}, o_{1,7}, o_{1,8}, o_{2,1}, o_{2,2}, o_{2,3}, o_{2,4}, o_{2,5}, o_{2,6}, o_{2,7}, o_{2,8}, o_{2,9}, o_{2,10}\}$

$Q_{OV1} = \{(m, PMN)\},$

$F_{OV1} = \{(o_{0,1}, o_{0,2}), (o_{0,2}, o_{0,3}), (o_{0,3}, o_{0,4}), (o_{0,4}, o_{0,5}), (o_{0,5}, o_{0,6}), (o_{0,6}, o_{0,7}), (o_{0,7}, o_{0,8}), (o_{0,8}, r_1), (o_{0,8}, o_{0,9}), (r_1, o_{0,9}), (o_{0,9}, o_{0,10}), (o_{0,10}, r_1), (o_{0,10}, o_{0,11}), (q_1, o_{1,1}), (o_{1,1}, o_{1,2}), (o_{1,2}, o_{1,3}), (o_{1,3}, r_1), (o_{1,3}, o_{1,4}), (o_{1,4}, r_1), (o_{1,4}, o_{1,5}), (o_{1,5}, r_1), (o_{1,5}, o_{1,6}), (r_1, o_{1,6}), (o_{1,6}, o_{1,7}), (o_{1,7}, q_1), (o_{1,7}, o_{1,8}), (o_{2,1}, o_{2,2}), (q_1, o_{2,2}), (o_{2,2}, o_{2,3}), (o_{2,3}, o_{2,4}), (o_{2,4}, o_{2,9}), (o_{2,4}, o_{2,5}), (o_{2,5}, o_{2,6}), (o_{2,6}, r_1), (o_{2,6}, o_{2,7}), (o_{2,7}, r_1), (o_{2,7}, o_{2,8}), (o_{2,8}, o_{2,4}), (o_{2,8}, o_{2,9}), (o_{2,9}, q_1), (o_{2,9}, o_{2,10})\}$

$B^{BWD}_{OV1} = \{(o_{1,3}, o_{2,7})\}$

Raport z przeprowadzonej analizy instancji modelu C_{OV1} (wygenerowanych przez *rdao detector* - listing 25) zawiera zgłoszenie jednego błędu. Aplikacja *rdao detector* prawidłowo wskazała, że istnieje możliwość realizacji operacji $o_{1,3}$ oraz $o_{2,7}$ w odwrotnym porządku, niż to wynika z relacji określonej w zbiorze B^{BWD}_{OV1}. Wynik ten

pokrywa się z rezultatem manualnej weryfikacji przeprowadzonej przez zespół ekspertów.

Analogiczne eksperymenty przeprowadzono dla aplikacji z projektu Phoenix oraz aplikacji *s-mptcp OV1*, (wersja z błędem prowadzącym do naruszenia porządku). Wyniki eksperymentów przedstawiono w tabeli 6.4.

Raporty weryfikacji aplikacji projektu Phoenix zawierają wyłącznie zgłoszenia fałszywie pozytywne. Ręczna weryfikacja potwierdziła, że aplikacje projektu Phoenix są wolne od tego typu błędów.

Weryfikacja aplikacji *s-mptcp OV1* wykazała obecność w kodzie źródłowym jednego błędu skutkującego naruszeniem niepodzielności. Fragment raportu z przeprowadzonej analizy można znaleźć na listingu o numerze 26. Podobnie jak w przypadku lokalizowania błędów naruszenia niepodzielności, tak i tutaj raport zawiera dwa zgłoszenia, z których jedno to duplikat.

Przeprowadzone eksperymenty pokazały, że wyniki metody RD-L pokrywają się z oceną Ekspertów analizujących kod aplikacji manualnie. Liczba fałszywie pozytywnych zgłoszeń stanowi 34% wykrytych błędów.

TABLICA 6.4: Wyniki eksperymentu lokalizowania błędów skutkujących konfliktami zasobowymi typu naruszenie porządku.

Nazwa aplikacji	Weryfikacja ręczna – zespół ekspertów	Weryfikacja automatyczna – rdao detector			
	Liczba błędów	Liczba zgłoszonych konfliktów	Liczba zgłoszeń fałszywie pozytywnych (ocena ekspertów)	Liczba duplikatów (ocena ekspertów)	Liczba potwierdzonych błędów (ocena ekspertów)
OV1	1	1	0	0	1
histogram	0	0	0	0	0
kmeans	0	26	8	18	0
linear regression	0	0	0	0	0
matrix multiply	0	0	0	0	0
pca	0	2	2	0	0
reverse index	0	0	0	0	0
string match	0	0	0	0	0
word count	0	1	1	0	0
s-mptcp OV1	1	2	0	1	1
Suma	2	32	11	19	2

6.7.1 Sposoby eliminacji błędów prowadzących do naruszenia porządku

Eliminacja błędów, prowadzących do naruszenia porządku, sprowadza się do przeniesienia tego z wątków do następnego przedziału czasu, którego operacja będąca w relacji niepodzielności ma być wykonana jako druga. Nim jednak dojdzie do procesu eliminacji błędów powodujących naruszenia porządku, należy upewnić się, czy do wskazanego konfliktu zasobowego faktycznie dochodzi. W eksperymencie dotyczącym lokalizowania błędów powodujących naruszenia niepodzielności udało się zaobserwować pewną zależność, w której błąd powodujący naruszenie niepodzielności spełnia także warunki, na podstawie których zgłaszane są błędy powodujące

naruszenie porządku. W pierwszej kolejności należy więc wyeliminować błędy prowadzące do naruszenia niepodzielności. Ich usunięcie spowoduje także usunięcie części zgłoszeń fałszywie pozytywnych dotyczących naruszenia porządku.

6.7.2 Podsumowanie

Przeprowadzone eksperymenty pokazały, że metoda RD-L zaimplementowana w aplikacji *rdao detector* pozwala lokalizować błędy, które powodują naruszenie porządku.

Proces ten jest równie złożony, co proces lokalizowania błędów powodujących naruszenia niepodzielności. Podobieństwa między strukturami tych błędów sprawiają, że niedoświadczeni programiści bardzo łatwo mogą się pomylić i podjąć niewłaściwe działania naprawcze, (np. których skutkiem może być spadek wydajności aplikacji).

Należy dodatkowo podkreślić, że podobnie jak w przypadku lokalizowania błędów powodujących konflikty naruszenia niepodzielności, poprawna lokalizacja błędów naruszenia porządku jest warunkowana posiadaniem informacji o istniejących relacjach pomiędzy operacjami realizowanych wątków.

6.8 Ocena implementacji metody

Do istotnych dla programistów cech aplikacji wykorzystywanych w procesie lokalizowania błędów zalicza się zużycie pamięci operacyjnej i czas, jaki należy poświęcić na dokonanie analizy. Z tego powodu przeprowadzono serię eksperymentów pozwalającą ocenić wydajność opracowanej aplikacji *rdao detector*. Ich wyniki zostały przedstawione w tabeli 6.5. Zawiera ona pomiary czasu i zużycia pamięci dla:

- procesu generowania instancji modelu C_P,
- procesu lokalizowania błędów prowadzących do konfliktów zasobowych.

Analiza otrzymanych wyników prowadzi do następujących spostrzeżeń:

- Proces generowania instancji modelu C_P kodu źródłowego aplikacji wielowątkowej P charakteryzuje się najwyższym zużyciem pamięci operacyjnej. Wynika to z faktu, że w trakcie trwania tego procesu w pamięci przechowywane jest wiele informacji niezbędnych do zbudowania instancji modelu C_P, takich jak: pełne drzewo składniowe kodu źródłowego aplikacji, składowe modelu C_P oraz inne informacje pomocnicze niezbędne do ukończenia procesu. Podkreślić należy też fakt, że najwyższe zużycie pamięci występujące w procesie generowania instancji modelu kodu źródłowego aplikacji wielowątkowej *pca* wynosi 30,87 MB. Wartość tą można uznać za niewielką w dobie komputerów posiadających gigabajty pamięci operacyjnej.
- Czas generowania instancji modelu C_P i czas analizy tej instancji jest silnie zależny od rozmiaru aplikacji. W przypadku bardzo małych aplikacji takich jak *RC1*, *DL1*, *AV1* i *OV1* większość czasu procesu lokalizowania konfliktów zasobowych stanowił czas generowania instancji modelu C_P (nieprzekraczający 1 s). Wraz ze wzrostem poziomu złożoności aplikacji czas analizy instancji modelu rośnie bardzo szybko a czas generowania instancji modelu stanowi tylko ułamek całości. Najdłuższy czas analizy instancji modelu to prawie 25 minut i dotyczy aplikacji *pca*. Jest to także aplikacja, w której raporcie znalazło się najwięcej zgłoszeń błędów powodujących konflikty zasobowe.

- Szczegółowa analiza pokazała, że parametrem wpływającym na czas analizy jest liczba jednocześnie pracujących wątków. W każdym wariancie aplikacji *s-mptcp* liczba ta wynosi 4, w aplikacjach *histogram*, *kmeans* i *reverse index* liczba wątków wynosi 19, zaś w aplikacji *pca* liczba wątków wynosi aż 37. W aplikacjach *RC1*, *DL1*, *AV1* i *OV1* liczba wątków nie przekracza 3.

Uzyskane wartości spełniają warunki do wykorzystania metody RD-L i implementującej ją aplikację *rdao detector* jako narzędzia wspierającego zespół programistów w procesie lokalizacji błędów aplikacji wielowątkowych. W przypadku aplikacji *rdao detector* czas analizy jest dostatecznie krótki w porównaniu z manualną analizą kodu źródłowego aplikacji, a ilość wymaganej pamięci operacyjnej jest niewielka w stosunku do ilości pamięci, jakie są instalowane w dzisiejszych komputerach. Wartym podkreślenia jest też fakt, że czas analizy kodu źródłowego aplikacji przez dowolnego eksperta (tabela 6.5) jest znacznie wyższy niż czas analizy wykonanej przez *rdao detector*. Z powodu zgłoszeń fałszywie pozytywnych osoba używająca *rdao detector* wciąż jest zmuszona do przeglądu wszystkich zgłoszeń. Jednakże czas przeglądu wszystkich zgłoszeń wciąż powinien być znacznie krótszy niż analiza całego kodu źródłowego.

TABLICA 6.5: Zużycie zasobów przez proces generowania instancji modelu i proces lokalizowania konfliktów zasobowych.

Nazwa aplikacji	Czas analizy eksperta* [s]	Generowanie instancji modelu		Proces lokalizowania konfliktów zasobowych	
		Czas generowania instancji modelu [s]	Najwyższe zużycie pamięci [MB]	Czas analizy instancji modelu [s]	Najwyższe zużycie pamięci [MB]
RC1	60	0.19	1.49	0.19	1.49
DL1	60	0.19	1.63	0.21	1.63
AV1	60	0.19	1.63	0.23	1.63
OV1	60	0.20	1.68	0.23	1.68
histogram	1800	10.56	19.45	318.42	19.46
kmeans	2700	22.21	19.88	389.94	19.89
linear regression	4500	1.09	4.37	1.26	4.37
matrix multiply	4500	1.20	5.24	1.90	5.24
pca	5400	42.44	30.87	1371.74	30.87
reverse index	2700	20.32	20.17	376.84	20.17
string match	1800	1.40	5.08	2.06	5.08
word count	2700	1.68	6.12	2.97	6.12
s-mptcp RC1	21 600	15.46	13.89	64.26	13.89
s-mptcp DL1	21 600	15.69	13.92	77.48	13.92
s-mptcp DL2	21 600	16.30	13.88	81.32	13.88
s-mptcp DL3	21 600	15.91	13.92	73.59	13.92
s-mptcp DL4	21 600	20.03	13.91	83.58	13.92
s-mptcp AV1	21 600	28.09	13.97	91.60	13.97
s-mptcp OV1	21 600	19.63	13.92	76.38	13.92

* - Czas analizy eksperta jest orientacyjny i zależny od poziomu doświadczenia.

Za wykorzystaniem metody RD-L w praktyce przemawiają również uzyskane wskaźniki jakościowe. Na rysunku 6.2 przedstawiono sumarycznie dane z tabeli

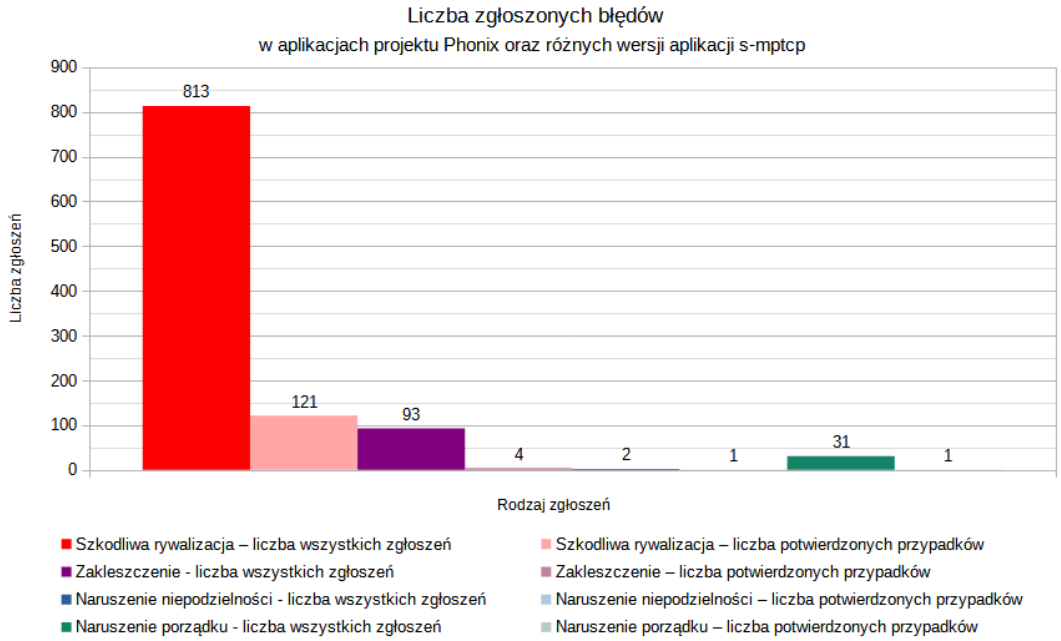
6.5 dotyczące lokalizowania konfliktów zasobowych w aplikacjach *RC1*, *DL1*, *AV1* i *OV1*. W żadnej z tych aplikacji nie pojawił się ani jeden wynik fałszywie pozytywny, co wynika z prostej budowy tych aplikacji i niewielkiej liczby linii kodu. W dwóch przypadkach pojawiły się duplikaty. Jest to jednak efekt uproszczeń stosowanych podczas implementacji algorytmu wykorzystywanego do analizy instancji modelu opisanego szczegółowo w rozdziale 5, tzn. jeśli w aplikacji uruchamiane są dwa wątki o identycznym przebiegu, to porównywane są one ze sobą dwa razy.

RYSUNEK 6.2: Wynik weryfikacji aplikacji *RC1*, *DL1*, *AV1* i *OV1*.

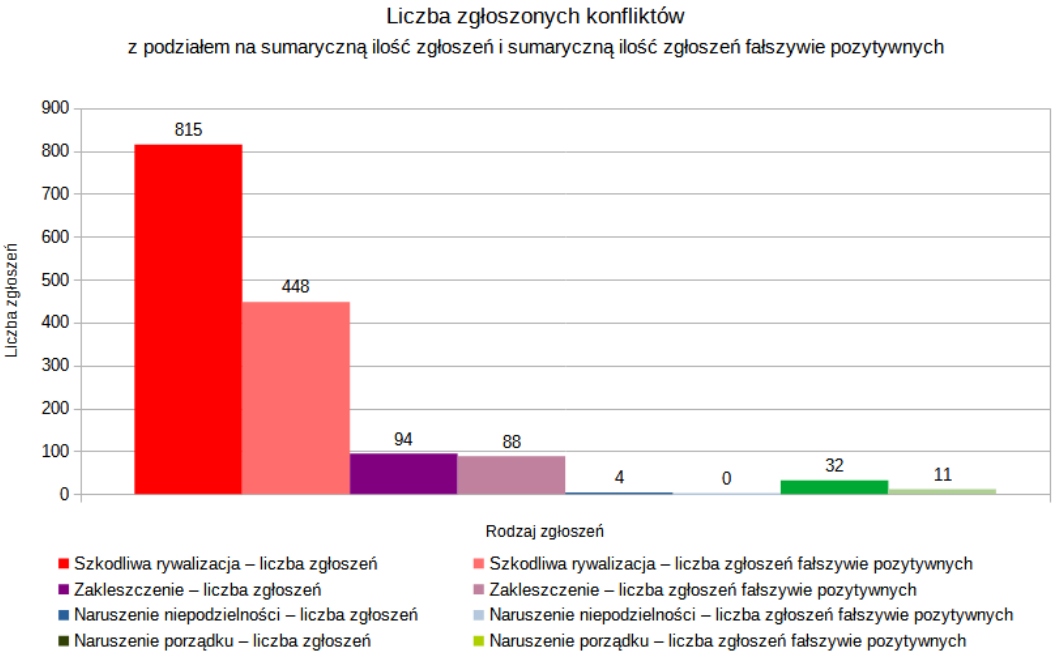
Dla pozostałych aplikacji sumaryczna liczba zgłoszonych błędów danej klasy znacząco przewyższa rzeczywistą liczbę błędów, co przedstawiono na rysunku 6.3. Wśród badanych konfliktów zasobowych największą różnicą pomiędzy liczbą zgłoszeń a faktyczną liczbą błędów charakteryzuje się szkodliwa rywalizacja. Tylko 121 z 813 zgłoszeń rzeczywiście wskazywało błąd, natomiast 692 zgłoszenia (85% wszystkich zgłoszeń) stanowiły duplikaty i zgłoszenia fałszywie pozytywne. Dzieje się tak, gdyż struktury kodu źródłowego powodujące konflikty zasobowe typu szkodliwa rywalizacja są mniej złożone niż w przypadku pozostałych konfliktów. Najmniejszą różnicę zaobserwować można w przypadku błędów prowadzących do naruszenia niepodzielności – w przypadku tego typu konfliktów zasobowych nie zarejestrowano żadnych zgłoszeń fałszywie pozytywnych.

Bazując na przedstawionych danych można błędnie założyć, że najwięcej zgłoszeń fałszywie pozytywnych generuje algorytm lokalizujący błędy powodujące szkodliwą rywalizację. Sytuacja prezentuje się inaczej, gdy porówna się liczbę wszystkich zgłoszeń do liczby zgłoszeń zaklasyfikowanych jako fałszywie pozytywne, co przedstawiono na rysunku 6.4. Na przedstawionych danych widać, że liczba zgłoszeń fałszywie pozytywnych stanowi od 30% (w przypadku naruszenia porządku) do nawet 95% (w przypadku zakleszczeń) wszystkich zgłoszeń. Należy pamiętać, że jeśli zgłoszenie fałszywie pozytywne występowało więcej niż jeden raz, to uznawane było za duplikat, dzięki czemu liczba zgłoszeń fałszywie pozytywnych wskazuje jednocześnie, ile różnych zgłoszeń fałszywie pozytywnych znalazło się w raporcie. Na podstawie przedstawionych danych można zauważyć, że w celu zmniejszenia liczby zgłoszeń fałszywie pozytywnych najlepiej podjąć się prac nad uszczegółowieniem warunków pozwalających lokalizować szkodliwą rywalizację i zakleszczenia. Brak zgłoszeń fałszywie pozytywnych występujących w procesie

lokalizowania naruszeń niepodzielności w przeprowadzonym eksperymencie nie daje gwarancji, że takie zgłoszenia nie pojawią się podczas analizy innych aplikacji.



RYSUNEK 6.3: Wyniki weryfikacji aplikacji projektu Phoenix oraz s-mptcp.



RYSUNEK 6.4: Liczba zgłoszeń vs liczba zgłoszeń fałszywie pozytywnych.

6.9 Podsumowanie

Wyniki przeprowadzonych eksperymentów pozwalają na stwierdzenie, że narzędzie *rdao detector*, w którym zaimplementowano metodę RD-L może stanowić podstawę do budowy komercyjnego środowiska wspierającego proces lokalizacji błędów aplikacji wielowątkowych (w szczególności zintegrowanego z procesami CI/CD, które są stosowane w metodykach zwinnych).

Manualna weryfikacja otrzymanych wyników pokazała, że wśród zgłoszeń występuje duża liczba duplikatów. Taki stan rzeczy powoduje, że w raportach pojawia się wiele zduplikowanych zgłoszeń fałszywie pozytywnych. Zbiorczo liczby te zestawiono tabeli 6.6. Łatwo zauważyć, że potwierdzone zgłoszenia stanowią od 5% do 50% liczby wszystkich zgłoszeń. Duża liczba fałszywie pozytywnych zgłoszeń jest konsekwencją szeregu przyjętych uproszczeń implementacyjnych. Ich eliminacja (zastosowana w kolejnych wersjach aplikacji *rdao detector*) może znacznie ograniczyć występowanie zgłoszeń fałszywie pozytywnych.

Do głównych zalet opracowanego narzędzia należy zaliczyć jego skuteczność tzn. narzędzie zgłosiło wszystkie błędy prowadzące do oczekiwanych konfliktów zasobowych. Ponadto czas potrzebny na analizę aplikacji wielowątkowej liczony jest w minutach, a najważniejszym czynnikiem wpływającym na jego wzrost jest liczba równolegle pracujących wątków. W rzeczywistych aplikacjach wielowątkowych liczba działających równolegle wątków zazwyczaj jest niewielka tzn. najczęściej spotyka się ich nie więcej niż 6. Programiści niechętnie korzystają z dużej liczby wątków ze względu na fakt, że nie istnieją narzędzia, które w krótkim czasie pozwalają zweryfikować tego typu aplikacje. Czas analizy większości przemysłowych aplikacji wielowątkowych nie powinien odbiegać znacznie od średniej uzyskanych czasów analizy różnych wersji aplikacji *s-mptcp*. Uzyskane czasy analizy świadczą o konkurencyjności aplikacji *rdao detector*. Podobnie na korzyść aplikacji *rdao detector* świadczy również niewielkie zużycie pamięci operacyjnej w procesie analizy. Krótki czas analizy i niewielkie zużycie pamięci pozwalają zatem na potencjalne wykorzystanie aplikacji *rdao detector* w procesie CI/CD, w którym to aplikacje uruchamiane są w celu wykonania krótkich zadań w niewielkich kontenerach o niewielkiej pamięci operacyjnej.

Podsumowując, metodę RD-L jak i narzędzie *rdao detector*, w którym ją zaimplementowano, należy ocenić pozytywnie, ponieważ umożliwiają one lokalizowanie on-line czterech typów konfliktów zasobowych na podstawie instancji modelu kodu źródłowego aplikacji wielowątkowej.

TABLICA 6.6: Porównanie liczby zgłoszonych błędów prowadzących do konfliktów zasobowych z liczbą potwierdzonych błędów.

Nazwa aplikacji	Liczba konfliktów							
	Szkodliwa rywalizacja		Zakleszczenie		Naruszenie niepodzielności		Naruszenie porządku	
	Zgłoszono	Potwierdzono	Zgłoszono	Potwierdzono	Zgłoszono	Potwierdzono	Zgłoszono	Potwierdzono
RC1	2	1	–	–	–	–	–	–
DL1	–	–	1	1	–	–	–	–
AV1	–	–	–	–	2	1	–	–
OV1	–	–	–	–	–	–	1	1
histogram	81	21	0	0	0	0	0	0
kmeans	141	21	0	0	0	0	26	0
linear regression	0	0	0	0	0	0	0	0
matrix multiply	0	0	0	0	0	0	0	0
pca	226	58	0	0	0	0	2	0
reverse index	147	19	0	0	0	0	0	0
string match	0	0	0	0	0	0	0	0
word count	0	0	0	0	0	0	1	0
s-mptcp RC1	218	2	–	–	–	–	–	–
s-mptcp DL1	–	–	24	1	–	–	–	–
s-mptcp DL2	–	–	23	1	–	–	–	–
s-mptcp DL3	–	–	23	1	–	–	–	–
s-mptcp DL4	–	–	23	1	–	–	–	–
s-mptcp AV1	–	–	–	–	2	1	–	–
s-mptcp OV1	–	–	–	–	–	–	2	1
Suma	815	122	94	5	4	2	32	2

Rozdział 7

Podsumowanie

7.1 Przebieg badań

W pierwszym rozdziale pracy omówiono zagadnienia związane z zastosowaniem języka C we współczesnym świecie, oraz powstaniem standardowej biblioteki *pthread* umożliwiającej pisanie kodu aplikacji wielowątkowych. Następnie przedstawiono problem badawczy, który dotyczy wykrywania błędów wynikających z zastosowania wielowątkowości. W tym ujęciu sformułowano pojęcia programu i aplikacji wielowątkowej. Obszar rozważań ograniczono do lokalizowania błędów związanych z występowaniem czterech najbardziej popularnych konfliktów zasobowych: szkodliwa rywalizacja, zakleszczenie, naruszenie niepodzielności i naruszenie porządku. Dla tej klasy błędów przedstawiono podstawowe podejścia do ich lokalizacji: statyczną, dynamiczną oraz mieszaną metodę analizy aplikacji wielowątkowych. Zauważono, że najczęściej stosowane w praktyce metody, bazujące na podejściu dynamicznym, charakteryzują się niską efektywnością (tzn. czasochłonnością). Metody bazujące na podejściach statycznych charakteryzują się niską czasochłonnością, ale wymagają z kolei stosowania złożonych modeli aplikacji (odwzorowujących niezbędne interakcje między realizowanymi wątkami). Opracowanie modelu aplikacji wielowątkowej pozwalającego na efektywną lokalizację ww. błędów stanowi podstawę sformułowanej tezy pracy.

W drugim rozdziale przedstawiono wyniki krytycznej analizy literatury przedmiotu w zakresie rozważanych rodzajów błędów i konfliktów zasobowych. Opracowana taksonomia metod i narzędzi wspierających procesy wykrywania (tzn. lokalizacji i identyfikowania) oraz usuwania błędów prowadzących do konfliktów zasobowych wskazuje na lukę badawczą. Związana jest ona z liczbą rozwiązań umożliwiających jednoczesną lokalizację rozważanych czterech rodzajów błędów. W szczególności wskazuje również na brak rozwiązań dedykowanych dla aplikacji pisanych z użyciem języka C (przy użyciu biblioteki *pthread*).

W dalszych rozdziałach przedstawiono przebieg i wyniki badań własnych. W rozdziale trzecim opisano autorski model kodu źródłowego aplikacji wielowątkowej C_P i zdefiniowano szereg pojęć (np. graf operacji G_P, ścieżka wątku $\lambda_a^{P,i}$, itp.) niezbędnych do opracowania warunków lokalizacji rozważanej klasy błędów. Rozdział ten stanowił podstawę do opracowania warunków lokalizowania błędów prowadzących do konfliktów zasobowych. Warunki te, wyrażone w postaci Twierdzeń 1–4, przedstawiono w rozdziale 4. Twierdzenia 1–4 pełnią rolę warunków *koniecznych*, których spełnienie może skutkować wystąpieniem konfliktu zasobowego.

Model C_P wraz z wykazanymi przy jego założeniach Twierdzeniami 1–4 stanowi podstawę opracowanej metody lokalizowania konfliktów zasobowych (metoda RD-L) przedstawionej w rozdziale piątym. Metoda R-DL sprowadza się do czterech etapów:

1. Import kodu źródłowego aplikacji P.
2. Budowa instancji modelu C_P dla aplikacji P.
3. Analiza instancji modelu C_P pod względem spełnienia warunków wystąpienia błędów prowadzących do konfliktów zasobowych (Twierdzenia 1–4).
4. Raportowanie błędów.

Efektywność metody RD-L, implementowanej w aplikacji *rdao detector*, zweryfikowano eksperymentalnie dla 19 aplikacji wielowątkowych pochodzących z otwartych źródeł (np. projekt aplikacji s-mptcp) - rozdział 6.

7.2 Rezultaty pracy

Do najważniejszych rezultatów prowadzonych badań należy zaliczyć:

- opracowanie modelu C_P kodu źródłowego aplikacji wielowątkowej;
- sformułowanie i wykazanie czterech Twierdzeń, spełnienie których może prowadzić do konfliktów zasobowych typu:
  - szkodliwa rywalizacja;
  - zakleszczenie;
  - naruszenie niepodzielności;
  - naruszenie porządku;
- opracowanie autorskiej metody RD-L umożliwiającej lokalizowanie błędów prowadzących do konfliktów w instancji modelu C_P z wykorzystaniem opracowanych warunków lokalizowania konfliktów zasobowych;
- opracowanie prototypu aplikacji (*rdao detector*) wspierającej programistę w zakresie wykrywania błędów w wytwarzanych aplikacjach wielowątkowych;
- przeprowadzenie serii eksperymentów weryfikujących poprawność i przydatność metody RD-L w procesie wytwarzania aplikacji wielowątkowych.

Wyniki badań potwierdzają, że metoda RD-L skutecznie lokalizuje błędy występujące w aplikacjach wielowątkowych. Niskie zapotrzebowanie na pamięć operacyjną (< 31MB) oraz niska czasochłonność (< 25 min.) pozwala na stosowanie opracowanej metody w procesie CI/CD. Oznacza to, że metoda ta może z powodzeniem zostać wykorzystana w środowiskach produkcyjnych.

Postawiona w rozdziale 1 teza, przyjmująca, że istnieją modele kodu źródłowego aplikacji wielowątkowych umożliwiające wyprowadzenie warunków, których spełnienie może skutkować wystąpieniem konfliktów zasobowych, została zweryfikowana pozytywnie (wykazana).

Osiągnięte rezultaty stanowią autorski wkład w rozwój informatyki w obszarze teorii inżynierii oprogramowania, teorii języków programowania, przetwarzania współbieżnego i równoległego, systemów współbieżnych procesów, itp. Praktycznym wynikiem pracy jest prototyp narzędzia *rdao detector*, bazujący na opracowanej metodzie RD-L, wspierający programistów w zakresie lokalizacji błędów wytwarzanych aplikacji wielowątkowych.

7.3 Kierunki dalszych badań

Przedstawione wyniki mogą być punktem wyjścia do innych pochodnych badań w zakresie weryfikacji aplikacji wielowątkowych. Z zakresu prowadzonych badań wyłączono pewne kierunki związane m.in. z lokalizacją błędów prowadzących do żywych zakleszczeń i zagłodzenia. Błędy te stanowią około 3,5% wszystkich spotykanych błędów, jednak ich lokalizacja może być istotna dla aplikacji branży motoryzacyjnej, czy lotniczej gdzie wymagana jest szczególnie wysoka jakość dostarczanego oprogramowania. W tym kontekście opracowany model C_P należy rozszerzyć o składowe umożliwiające identyfikację tego typu błędów.

Jednym z kierunków dalszych badań są prace zmierzające do wzmocnienia opracowanych warunków koniecznych, tak aby skutkowały one mniejszą liczbą zgłoszeń fałszywie pozytywnych. W tym kontekście dalsze prace winny być ukierunkowane na uszczegółowienie warunków pozwalających na lokalizacje konfliktów zasobowych. Wiąże się to z rozszerzeniem modelu o możliwość lokalizowania błędów wynikających ze stosowania mechanizmów biblioteki *pthread*, które zostały pominięte w niniejszej pracy (tzn. uwzględnienie wątków odłączonych, jak i innych mechanizmów synchronizacji wątków - obsługa funkcji *pthread_mutex_trylock* i *pthread_mutex_timedlock*).

Kolejny kierunek badań związany jest z bardzo popularnym trendem wykorzystania rozwiązań sztucznej inteligencji. Wykorzystanie SI do identyfikacji struktur modelu C_P prowadzących do konfliktów zasobowych, może pozwolić na opracowanie nowych (bardziej szczegółowych) warunków ich wystąpienia. Należy dodatkowo podkreślić, że aplikacja *rdao detector* napisana została w języku Python, który jest najpopularniejszym językiem używanym w dziedzinie sztucznej inteligencji - ułatwi to prowadzenie tego typu badań.

Istotnym ograniczeniem opracowanej metody jest możliwość jej zastosowania wyłącznie dla aplikacji pisanych w języku C. Dalsze badania mogą być ukierunkowane na rozszerzenie opracowanego modelu o możliwość reprezentacji kodu źródłowego aplikacji pisanych w innych językach. W przypadku języków obiektowych takich jak C++, C#, Java czy Python podczas transformacji kodu źródłowego do instancji modelu C_P kodu źródłowego aplikacji wielowątkowych, należy sprowadzić kod obiektowy do kodu strukturalnego, który przypominałby język C. W przypadku stosowania innych bibliotek dostarczających wielowątkowość istotnym jest, aby móc określić sposoby leżące u podstaw działania mechanizmów wzajemnego wykluczania oraz sposoby lokalizacji miejsca startu/zakończenia wątków w celu identyfikacji przedziałów czasu, w których są realizowane.

Dalsze prace skoncentrowane zostaną więc na budowie aplikacji możliwej do zastosowań w środowisku produkcyjnym, zgodnie z panującymi standardami pracy programistów (w tym CI/CD). Planowane jest również opracowanie aplikacji minimalizującej liczbę fałszywie pozytywnych zgłoszeń i umożliwiającej weryfikację aplikacji wielowątkowych pisanych w różnych językach.

Dodatek A

Kod źródłowy aplikacji

LISTING 1: Kod aplikacji *RC1* zawierający konflikt zasobowy powodujący błąd szkodliwej rywalizacji.

```
#include <stdio.h>
#include <pthread.h>

static volatile int r1 = 0;

void* deposit(void *args) {
    for (int i=0; i<1000000; i++)
    {
        ++r1;
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    printf("App start work with r1 = %d\n", r1);

    pthread_create(&t1, NULL, deposit, NULL);
    pthread_create(&t2, NULL, deposit, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("App finish work with r1 = %d\n", r1);
    return 0;
}
```

LISTING 2: Kod aplikacji *DL1* zawierający konflikt zasobowy wynikający z wzajemnie wykluczających się par blokad powodujący błąd zakleszczenia.

```
#include <stdio.h>
#include <pthread.h>

static volatile int counter = 0;
pthread_mutex_t m, n;

```

```
7  void* t1f(void *args) {
8      pthread_mutex_lock(&m);
9      pthread_mutex_lock(&n);
10     ++counter;
11     pthread_mutex_unlock(&n);
12     pthread_mutex_unlock(&m);
13     return NULL;
14 }
15
16 void* t2f(void *args) {
17     pthread_mutex_lock(&n);
18     pthread_mutex_lock(&m);
19     ++counter;
20     pthread_mutex_unlock(&m);
21     pthread_mutex_unlock(&n);
22     return NULL;
23 }
24
25 int main() {
26     pthread_t t1, t2;
27     printf("App start work with counter = %d\n", counter);
28
29     pthread_create(&t1, NULL, t1f, NULL);
30     pthread_create(&t2, NULL, t2f, NULL);
31
32     pthread_join(t1, NULL);
33     pthread_join(t2, NULL);
34     printf("App finish work with counter = %d\n", counter);
35     return 0;
36 }
```

LISTING 3: Kod aplikacji *DL2* zawierający konflikt zasobowy wynikający z pominięcia operacji zwolnienia powodujący błąd zakleszczenia.

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  static volatile int counter = 0;
5  pthread_mutex_t m;
6
7  void* t1f(void *args) {
8      pthread_mutex_lock(&m);
9      ++counter;
10     pthread_mutex_unlock(&m);
11     return NULL;
12 }
13
14 void* t2f(void *args) {
15     pthread_mutex_lock(&m);
16     int i = *((int*)args);
```

```
17
18      if (i) {
19          ++counter;
20          pthread_mutex_unlock(&m);
21      }
22      else {
23          counter += 2;
24      }
25      return NULL;
26  }
27
28  int main() {
29      pthread_t t1, t2;
30      printf("App start work with counter = %d\n", counter);
31      int t2arg = 0;
32
33      pthread_create(&t1, NULL, t1f, NULL);
34      pthread_create(&t2, NULL, t2f, &t2arg);
35
36      pthread_join(t1, NULL);
37      pthread_join(t2, NULL);
38      printf("App finish work with counter = %d\n", counter);
39      return 0;
40  }
```

LISTING 4: Kod aplikacji *DL3* zawierający konflikt zasobowy wynikający z próby wielokrotnego wykonywania operacji założenia blokady w ciele pętli powodujący błąd zakleszczenia.

```
1   #include <stdio.h>
2   #include <pthread.h>
3
4   static volatile int counter = 0;
5   pthread_mutex_t m;
6
7   void* t1f(void *args) {
8       int i = *((int*)args);
9       while(i--) {
10          pthread_mutex_lock(&m);
11          ++counter;
12      }
13      pthread_mutex_unlock(&m);
14      return NULL;
15  }
16
17  int main() {
18      pthread_t t1;
19      printf("App start work with counter = %d\n", counter);
20      int t1arg = 100;
21
22      pthread_create(&t1, NULL, t1f, &t1arg);
```

```
23
24      pthread_join(t1, NULL);
25      printf("App finish work with counter = %d\n", counter);
26      return 0;
27  }
```

LISTING 5: Kod aplikacji *DL4* zawierający konflikt zasobowy wynikający z próby wielokrotnego wykonywania operacji założenia blokady w wyniku wywołania rekurencyjnego funkcji powodujący błąd zakleszczenia.

```
1   #include <stdio.h>
2   #include <pthread.h>
3
4   static volatile int counter = 0;
5   pthread_mutex_t m;
6
7   void* t1f(void *args) {
8       int *i = ((int*)args);
9       pthread_mutex_lock(&m);
10      if(*i) {
11          --*i;
12          ++counter;
13          t1f(args);
14      }
15      pthread_mutex_unlock(&m);
16
17      return NULL;
18  }
19
20  int main() {
21      pthread_t t1;
22      printf("App start work with counter = %d\n", counter);
23      int t1arg = 100;
24
25      pthread_create(&t1, NULL, t1f, &t1arg);
26
27      pthread_join(t1, NULL);
28      printf("App finish work with counter = %d\n", counter);
29      return 0;
30  }
```

LISTING 6: Kod aplikacji *AV1* zawierający konflikt zasobowy powodujący błąd naruszenia niepodzielności.

```
1   #include <stdio.h>
2   #include <pthread.h>
3   #include <unistd.h>
4
5   volatile int r1 = 0;
```

```
6  pthread_mutex_t m;
7
8  void* deposit(void *args) {
9
10     for (int i=0; i<10; i++)
11     {
12         pthread_mutex_lock(&m);
13         ++r1;
14         pthread_mutex_unlock(&m);
15         usleep(100);
16         pthread_mutex_lock(&m);
17         printf("%d\n", r1);
18         pthread_mutex_unlock(&m);
19     }
20     return NULL;
21 }
22
23 int main() {
24     pthread_t t1, t2;
25     printf("App start work with r1 = %d\n", r1);
26
27     pthread_create(&t1, NULL, deposit, NULL);
28     pthread_create(&t2, NULL, deposit, NULL);
29
30     pthread_join(t1, NULL);
31     pthread_join(t2, NULL);
32     printf("App finish work with r1 = %d\n", r1);
33     return 0;
34 }
```

LISTING 7: Kod aplikacji *OV1*, zawierający konflikt zasobowy powodujący błąd klasy naruszenia porządku.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  pthread_mutex_t m;
6
7  void* create_account(void *args) {
8      pthread_mutex_lock(&m);
9      int **account = (int**)args;
10     *account = (int*)malloc(sizeof(int));
11     **account = 0;
12     printf("Account created. Current balance: %d\r\n", **account);
13     pthread_mutex_unlock(&m);
14     return NULL;
15 }
16
17 void* deposit(void *args) {
18     int *account = (int*)args;
```

```
19      pthread_mutex_lock(&m);
20      for (int i=0; i<1000000; i++)
21      {
22          ++*account;
23      }
24      pthread_mutex_unlock(&m);
25      return NULL;
26  }
27
28  int main() {
29      pthread_t t1, t2;
30      int *account;
31      pthread_create(&t1, NULL, create_account, (void*)&account);
32      pthread_create(&t2, NULL, deposit, (void*)account);
33
34      pthread_join(t1, NULL);
35      pthread_join(t2, NULL);
36      printf("App finish work with account balance = %d\r\n", *account);
37      free(account);
38      return 0;
39  }
```

Dodatek B

Fragmenty raportów z przeprowadzonych eksperymentów

Pełny raport z przeprowadzonych eksperymentów można znaleźć w repozytorium pod adresem `https://github.com/PKPhdDG/mascm-expirements-docs`. Niniejszy dodatek przedstawia tylko fragmenty raportów w celu umożliwienia analizy otrzymanych wyników.

LISTING 8: Raport z analizy aplikacji *AV1*.

```
Race conditions:
        ...

Race condition was reported 2 times.
============================================================
Deadlocks:
Deadlock was reported 0 times.
============================================================
Atomicity violations:
Atomicity violation was reported 0 times.
============================================================
Order violations:
Order violation was reported 0 times.
============================================================
Resource usage:
        Current memory usage is 1.40MB
        Peak was 1.49MB
        Time: 0.19s
```

LISTING 9: Fragment raportu z analizy aplikacji *histogram*.

```
Race conditions:
        ...

Race condition was reported 81 times.
============================================================
Deadlocks:
Deadlock was reported 0 times.
============================================================
Atomicity violations:
Atomicity violation was reported 0 times.
============================================================
Order violations:
Order violation was reported 0 times.
============================================================
Resource usage:
        Current memory usage is 9.63MB
        Peak was 19.46MB
        Time: 318.42s
```

LISTING 10: Fragment raportu z analizy aplikacji *kmeans*.

```
Race conditions:
        ...

Race condition was reported 141 times.
============================================================
Deadlocks:
Deadlock was reported 0 times.
============================================================
Atomicity violations:
Atomicity violation was reported 0 times.
============================================================
Order violations:
        ...

Order violation was reported 26 times.
============================================================
Resource usage:
        Current memory usage is 9.74MB
        Peak was 19.89MB
        Time: 389.94s
```

LISTING 11: Fragment raportu z analizy aplikacji *linear regression*.

```
Race conditions:
Race condition was reported 0 times.
============================================================
Deadlocks:
Deadlock was reported 0 times.
============================================================
Atomicity violations:
Atomicity violation was reported 0 times.
============================================================
Order violations:
Order violation was reported 0 times.
============================================================
Resource usage:
        Current memory usage is 3.39MB
        Peak was 4.37MB
        Time: 1.26s
```

LISTING 12: Fragment raportu z analizy aplikacji *matrix multiply*.

```
Race conditions:
Race condition was reported 0 times.
============================================================
Deadlocks:
Deadlock was reported 0 times.
============================================================
Atomicity violations:
Atomicity violation was reported 0 times.
============================================================
Order violations:
Order violation was reported 0 times.
============================================================
Resource usage:
        Current memory usage is 4.22MB
        Peak was 5.24MB
        Time: 1.90s
```

LISTING 13: Fragment raportu z analizy aplikacji pca.

```
Race conditions:
        ...
```

```
Race condition was reported 226 times.
============================================================
Deadlocks:
Deadlock was reported 0 times.
============================================================
Atomicity violations:
Atomicity violation was reported 0 times.
============================================================
Order violations:
        ...

Order violation was reported 2 times.
============================================================
Resource usage:
        Current memory usage is 15.78MB
        Peak was 30.87MB
        Time: 1371.74s
```

LISTING 14: Fragment raportu z analizy aplikacji *reverse index*.

```
Race conditions:
        ...

Race condition was reported 147 times.
============================================================
Deadlocks:
Deadlock was reported 0 times.
============================================================
Atomicity violations:
Atomicity violation was reported 0 times.
============================================================
Order violations:
Order violation was reported 0 times.
============================================================
Resource usage:
        Current memory usage is 9.68MB
        Peak was 20.17MB
        Time: 376.84s
```

LISTING 15: Fragment raportu z analizy aplikacji *string match*.

```
Race conditions:
Race condition was reported 0 times.
============================================================
Deadlocks:
Deadlock was reported 0 times.
============================================================
Atomicity violations:
Atomicity violation was reported 0 times.
============================================================
Order violations:
Order violation was reported 0 times.
============================================================
Resource usage:
        Current memory usage is 4.15MB
        Peak was 5.08MB
        Time: 2.06s
```

LISTING 16: Fragment raportu z analizy aplikacji *word count*.

```
Race conditions:
Race condition was reported 0 times.
============================================================
Deadlocks:
Deadlock was reported 0 times.
============================================================
Atomicity violations:
Atomicity violation was reported 0 times.
```

```
============================================================
Order violations:
        ...

Order violation was reported 1 times.
============================================================
Resource usage:
        Current memory usage is 4.85MB
        Peak was 6.12MB
        Time: 2.97s
```

LISTING 17: Fragment raportu z analizy aplikacji *s-mptcp RC1*.

```
Race conditions:
        ...

Race condition was reported 218 times.
============================================================
Deadlocks:
        ...

Deadlock was reported 22 times.
============================================================
Atomicity violations:
Atomicity violation was reported 0 times.
============================================================
Order violations:
Order violation was reported 0 times.
============================================================
Resource usage:
        Current memory usage is 6.34MB
        Peak was 13.89MB
        Time: 64.26s
```

LISTING 18: Fragment raportu z analizy aplikacji *DL1*.

```
Race conditions:
Race condition was reported 0 times.
============================================================
Deadlocks:
        ...

Deadlock was reported 1 times.
============================================================
Atomicity violations:
Atomicity violation was reported 0 times.
============================================================
Order violations:
Order violation was reported 0 times.
============================================================
Resource usage:
        Current memory usage is 1.46MB
        Peak was 1.63MB
        Time: 0.21s
```

LISTING 19: Fragment raportu z analizy aplikacji *s-mptcp DL1*.

```
Race conditions:
        ...

Race condition was reported 215 times.
============================================================
Deadlocks:
        ...

Deadlock was reported 24 times.
============================================================
Atomicity violations:
```

```
Atomicity violation was reported 0 times.
============================================================
Order violations:
Order violation was reported 0 times.
============================================================
Resource usage:
        Current memory usage is 6.35MB
        Peak was 13.92MB
        Time: 77.48s
```

LISTING 20: Fragment raportu z analizy aplikacji *s-mptcp DL2*.

```
Race conditions:
        ...

Race condition was reported 215 times.
============================================================
Deadlocks:
        ...

Deadlock was reported 23 times.
============================================================
Atomicity violations:
Atomicity violation was reported 0 times.
============================================================
Order violations:
Order violation was reported 0 times.
============================================================
Resource usage:
        Current memory usage is 6.33MB
        Peak was 13.88MB
        Time: 81.32s
```

LISTING 21: Fragment raportu z analizy aplikacji *s-mptcp DL3*.

```
Race conditions:
        ...

Race condition was reported 215 times.
============================================================
Deadlocks:
        ...

Deadlock was reported 23 times.
============================================================
Atomicity violations:
Atomicity violation was reported 0 times.
============================================================
Order violations:
Order violation was reported 0 times.
============================================================
Resource usage:
        Current memory usage is 6.35MB
        Peak was 13.92MB
        Time: 73.59s
```

LISTING 22: Fragment raportu z analizy aplikacji *s-mptcp DL4*.

```
Race conditions:
        ...

Race condition was reported 215 times.
============================================================
Deadlocks:
        ...

Deadlock was reported 23 times.
============================================================
```

```
Atomicity violations:
Atomicity violation was reported 0 times.
============================================================
Order violations:
Order violation was reported 0 times.
============================================================
Resource usage:
        Current memory usage is 6.34MB
        Peak was 13.92MB
        Time: 83.58s
```

LISTING 23: Fragment raportu z analizy aplikacji *AV1*.

```
Race conditions:
Race condition was reported 0 times.
============================================================
Deadlocks:
        ...

Deadlock was reported 4 times.
============================================================
Atomicity violations:
        ...

Atomicity violation was reported 2 times.
============================================================
Order violations:
        ...

Order violation was reported 2 times.
============================================================
Resource usage:
        Current memory usage is 1.48MB
        Peak was 1.63MB
        Time: 0.23s
```

LISTING 24: Fragment raportu z analizy aplikacji *s-mptcp AV1*.

```
Race conditions:
        ...

Race condition was reported 215 times.
============================================================
Deadlocks:
        ...

Deadlock was reported 22 times.
============================================================
Atomicity violations:
        ...

Atomicity violation was reported 2 times.
============================================================
Order violations:
        ...

Order violation was reported 2 times.
============================================================
Resource usage:
        Current memory usage is 6.37MB
        Peak was 13.97MB
        Time: 91.60s
```

LISTING 25: Fragment raportu z analizy aplikacji *OV1*.

```
Race conditions:
Race condition was reported 0 times.
============================================================
```

```
Deadlocks:
Deadlock was reported 0 times.
============================================================
Atomicity violations:
Atomicity violation was reported 0 times.
============================================================
Order violations:
        ...

Order violation was reported 1 times.
============================================================
Resource usage:
        Current memory usage is 1.49MB
        Peak was 1.68MB
        Time: 0.23s
```

LISTING 26: Fragment raportu z analizy aplikacji *s-mptcp OV1*.

```
Race conditions:
        ...

Race condition was reported 219 times.
============================================================
Deadlocks:
        ...

Deadlock was reported 22 times.
============================================================
Atomicity violations:
Atomicity violation was reported 0 times.
============================================================
Order violations:
        ...

Order violation was reported 2 times.
============================================================
Resource usage:
        Current memory usage is 6.34MB
        Peak was 13.92MB
        Time: 76.38s
```

Bibliografia

[1] *A safe SCHED_IDLE implementation*. https://lwn.net/Articles/4073/. Odwiedzono: 2019-08-08.

[2] Martin Abadi, Cormac Flanagan i Stephen N Freund. „Types for safe locking: Static race detection for Java". W: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28.2 (2006), strony 207–255. DOI: 10.1145/1119479.1119480.

[3] Rahul Agarwal i inni. „Detection of deadlock potentials in multithreaded programs". W: *IBM Journal of Research and Development* 54.5 (2010), strony 3–1. DOI: 10.1147/JRD.2010.2060276.

[4] *AMD ŹenĆore Architecture*. https://www.amd.com/en/technologies/zen-core. Odwiedzono: 2021-06-22.

[5] *Application Software in Oxford Online Dictionary*. https://en.oxforddictionaries.com/definition/application_software. Odwiedzono: 2021-06-22.

[6] Sara Abbaspour Asadollah i inni. „Concurrency bugs in open source software: a case study". W: *Journal of Internet Services and Applications* 8.1 (2017), strony 1–15. DOI: 10.1186/s13174-017-0055-2.

[7] Zbigniew Banaszak, Paweł Majdzik i Robert Wójcik. *Procesy współbieżne: modele efektywności funkcjonowania*. Wydawnictwo Uczelniane Politechniki Koszalińskiej, 2008.

[8] *batch/idle priority scheduling, SCHED_BATCH*. https://lwn.net/Articles/3866/. Odwiedzono: 2019-08-08.

[9] Ryan James Berg i inni. *Method and system for detecting race condition vulnerabilities in source code*. U.S. Patent Us 7,398,516 B2. Lip. 2008. URL: https://patentimages.storage.googleapis.com/2b/ca/ca/746ba6a3ed9914/US7398516.pdf.

[10] Emery D Berger i inni. „Grace: Safe multithreaded programming for C/C++". W: *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. Association for Computing Machinery, 2009, strony 81–96. DOI: 10.1145/1640089.1640096.

[11] Grzegorz Bocewicz. *Modele multimodalnych procesów cyklicznych*. Wydawnictwo Uczelniane Politechniki Koszalińskiej, 2013.

[12] Chandrasekhar Boyapati i Martin Rinard. „A parameterized type system for race-free Java programs". W: *ACM SIGPLAN Notices*. Tom 36. 11. ACM. 2001, strony 56–69. DOI: 10.1145/504282.504287.

[13] Eric J. Braude. *An algorithm for the detection of system deadlocks*. Sprawozdanie techniczne. IBM Technical Report: TROO. 791 IBM Data Systems Division, Poughkeepsie, NY, 1961.

[14] *Certyfikowany tester. Sylabus poziomu podstawowego ISTQB*. Wersja v3.1. International Software Testing Qualifications Board, 2018.

[15] Luis Ceze. *Detecting and Avoiding Atomicity Violations*. https://www.nii.ac.jp/userimg/lectures/LuisCeze/nii-lecture4.pdf. Odwiedzono: 2019-10-09. University of Washington.

[16] Xiaoning Chang i inni. „Detecting atomicity violations for event-driven Node.js applications". W: *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press. 2019, strony 631–642. DOI: 10.1109/ICSE.2019.00073.

[17] Dongjie Chen i inni. „Testing multithreaded programs via thread speed control". W: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2018, strony 15–25. DOI: 10.1145/3236024.3236077.

[18] Qichang Chen i inni. „HAVE: Detecting atomicity violations via integrated dynamic and static analysis". W: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2009, strony 425–439. DOI: 10.1007/978-3-642-00593-0_30.

[19] Lee Chew i David Lie. „Kivati: fast detection and prevention of atomicity violations". W: *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, strony 307–320. DOI: 10.1145/1755913.1755945.

[20] *CI/CD Best Practices*. Odwiedzono: 2022-04-01. JetBrains s.r.o. URL: https://www.jetbrains.com/teamcity/ci-cd-guide/ci-cd-best-practices/.

[21] Edmund M Clarke, Allen Emerson i Joseph Sifakis. „Model checking: algorithmic verification and debugging". W: *Communications of the ACM* 52.11 (2009), strony 74–84. DOI: 10.1145/1592761.1592781.

[22] *Czym jest Hyper-Threading*. https://www.intel.pl/content/www/pl/pl/gaming/resources/hyper-threading.html. Odwiedzono: 2021-06-22.

[23] Ricardo J Dias, Vasco Pessanha i João M Lourenço. „Precise detection of atomicity violations". W: *Haifa Verification Conference*. Springer. 2012, strony 8–23. DOI: 10.1007/978-3-642-39611-3_8.

[24] Rob J. van Emous. „Towards systematic black-box testing for exploitable race conditions in web apps". Praca magisterska. University of Twente, czer. 2019. URL: http://essay.utwente.nl/78020/.

[25] Dawson Engler i Ken Ashcraft. „RacerX: effective, static detection of race conditions and deadlocks". W: *ACM SIGOPS Operating Systems Review*. Tom 37. 5. ACM. 2003, strony 237–252.

[26] Jan Fiedor i inni. „A uniform classification of common concurrency errors". W: *International Conference on Computer Aided Systems Theory*. Springer. 2011, strony 519–526. DOI: 10.1007/978-3-642-27549-4_67.

[27] Cormac Flanagan i Stephen N Freund. „Atomizer: a dynamic atomicity checker for multithreaded programs". W: *ACM SIGPLAN Notices*. Tom 39. 1. ACM. 2004, strony 256–267. DOI: 10.1145/982962.964023.

[28] Cormac Flanagan i Stephen N Freund. „Type inference against races". W: *International Static Analysis Symposium*. Springer. 2004, strony 116–132. DOI: 10.1016/j.scico.2006.03.006.

[29] Cormac Flanagan i Stephen N Freund. „Type-based race detection for Java". W: *ACM Sigplan Notices*. Tom 35. 5. ACM. 2000, strony 219–232. DOI: 10.1145/349299.349328.

[30] Cormac Flanagan, Stephen N Freund i Marina Lifshin. „Type inference for atomicity". W: *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*. 2005, strony 47–58. DOI: 10.1145/1040294.1040299.

[31] Cormac Flanagan i inni. „Types for atomicity: Static checking and inference for Java". W: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30.4 (2008), strony 1–53. DOI: 10.1145/1377492.1377495.

[32] Maurizio Gabbrielli i Simone Martini. *Programming languages: Principles and Paradigms*. Springer Science & Business Media, 2006.

[33] Malay K Ganai i inni. „BEST: A symbolic testing tool for predicting multi-threaded program failures". W: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE. 2011, strony 596–599. DOI: 10.1109/ASE.2011.6100134.

[34] Elena Giachino i Cosimo Laneve. „Deadlock detection in linear recursive programs". W: *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer. 2014, strony 26–64. DOI: 10.1007/978-3-319-07317-0_2.

[35] Damian Giebas i Rafał Wojszczyk. „Atomicity violation in multithreaded applications and its detection in static code analysis process". W: *Applied Sciences* 10.22 (2020), strona 8005. DOI: 10.3390/app10228005.

[36] Damian Giebas i Rafał Wojszczyk. „Deadlocks Detection in Multithreaded Applications Based on Source Code Analysis". W: *Applied Sciences* 10.2 (2020), strona 532. DOI: 10.3390/app10020532.

[37] Damian Giebas i Rafał Wojszczyk. „Detection of Concurrency Errors in Multithreaded Applications Based on Static Source Code Analysis". W: *IEEE Access* 9 (2021), strony 61298–61323. DOI: 10.1109/ACCESS.2021.3073859.

[38] Damian Giebas i Rafał Wojszczyk. „Graphical representations of multithreaded applications". W: *Applied Computer Science* 14 (2018). DOI: 10.23743/acs-2018-10.

[39] Damian Giebas i Rafał Wojszczyk. „Lokalizacja błędów wielowątkowych w aplikacji serwera systemu kontrolno-pomiarowego". W: *Innowacje w elektronice, informatyce i inżynierii produkcji, Monografia nr 399* 2 (2021), strony 215–227.

[40] Damian Giebas i Rafał Wojszczyk. „Multithreaded application model". W: *International Symposium on Distributed Computing and Artificial Intelligence*. Springer. 2019, strony 93–103. DOI: 10.1007/978-3-030-23946-6_11.

[41] Damian Giebas i Rafał Wojszczyk. „Order violation in multithreaded applications and its detection in static code analysis process". W: *Applied Computer Science* 16.4 (2020). DOI: 10.23743/acs-2020-32.

[42] Damian Giebas i Rafał Wojszczyk. „Rules in Detection of Deadlocks in Multithreaded Applications". W: *International Symposium on Distributed Computing and Artificial Intelligence*. Springer. 2020, strony 15–24. DOI: 10.1007/978-3-030-53829-3_2.

[43] Damian Giebas i Rafał Wojszczyk. „Zastosowanie wybranych reprezentacji graficznych do analizy aplikacji wielowątkowych". W: *Zeszyty Naukowe Wydziału Elektroniki i Informatyki Politechniki Koszalińskiej* 13 (2018).

[44] Wojciech Głodzki. *Układy cyfrowe*. Wydawnictwa Szkolne i Pedagogiczne, 2010.

[45] *How to keep up with CI/CD best practices*. Odwiedzono: 2022-04-01. GitLab B.V. URL: https://about.gitlab.com/blog/2022/02/03/how-to-keep-up-with-ci-cd-best-practices/.

[46] Eric Huss. *The C Library Reference Guide*. https://web.archive.org/web/20150118141700/http://www.acm.uiuc.edu/webmonkeys/book/c_guide/index.html. Odwiedzono: 2019-11-20.

[47] *Programming Languages – C++*. Standard Międzynarodowy. Geneva, CH: International Organization for Standardization, maj 2013. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf.

[48] Ali Jannesari i Walter F Tichy. „On-the-fly race detection in multi-threaded programs". W: *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging*. 2008, strony 1–10. DOI: 10.1145/1390841.1390847.

[49] Guoliang Jin i inni. „Automated atomicity-violation fixing". W: *ACM Sigplan Notices*. Tom 46. 6. ACM. 2011, strony 389–400. DOI: 10.1145/1993498.1993544.

[50] Vineet Kahlon i inni. *Fast and accurate data race detection for concurrent programs with asynchronous calls*. US Patent 8,539,450. Wrz. 2013.

[51] Vineet Kahlon i inni. „Static data race detection for concurrent programs with asynchronous calls". W: *Proceedings of the 7th joint meeting of the European software engineering conference and the acm sigsoft symposium on the foundations of software engineering*. 2009, strony 13–22. DOI: 10.1145/1595696.1595701.

[52] Krishna M Kavi, Alireza Moshtaghi i Deng-Jyi Chen. „Modeling multithreaded applications using Petri nets". W: *International Journal of Parallel Programming* 30.5 (2002), strony 353–371. DOI: 10.1023/A:1019917329895.

[53] Brian W. Kernighan i Dennis M. Ritchie. *Język ANSI C*. Wydawnictwo Naukowo-Techniczne, 1994.

[54] Dmitrii Ivanovich Kharitonov i Daria Odyakova. „Modelling race conditions in multithreading programs in terms of Petri nets". W: *IOP Conference Series: Materials Science and Engineering*. Tom 734. 1. IOP Publishing. 2020, strona 012030. DOI: 10.1088/1757-899X/734/1/012030.

[55] Nikolaos Koutsopoulos i inni. „Advancing data race investigation and classification through visualization". W: *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*. IEEE. 2015, strony 200–204. DOI: 10.1109/VISSOFT.2015.7332437.

[56] Stéphane Lafortune, Yin Wang i Spyros Reveliotis. „Eliminating concurrency bugs in multithreaded software: An approach based on control of petri nets". W: *International Conference on Applications and Theory of Petri Nets and Concurrency*. Springer. 2013, strony 21–28. DOI: 10.1007/978-3-642-38697-8_2.

[57] Cosimo Laneve. „A lightweight deadlock analysis for programs with threads and reentrant locks". W: *Science of Computer Programming* 181 (2019), strony 64–81. DOI: 10.1016/j.scico.2019.06.002.

[58] Jiaqi Li i inni. „An Intelligent Deadlock Locating Scheme for Multithreaded Programs". W: *Proceedings of the 2019 3rd International Conference on Intelligent Systems, Metaheuristics & Swarm Intelligence*. ACM. 2019, strony 14–18. DOI: 10.1145/3325773.3325781.

[59] Yanze Li, Bozhen Liu i Jeff Huang. „Sword: A scalable whole program race detector for java". W: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE. 2019, strony 75–78. DOI: 10.1109/ICSE-Companion.2019.00042.

[60] Hongwei Liao i inni. „Concurrency bugs in multithreaded software: Modeling and analysis using Petri nets". W: *Discrete Event Dynamic Systems* 23.2 (2013), strony 157–195. DOI: 10.1007/s10626-012-0139-x.

[61] Lucas Lima, Amaury Tavares i Sidney C Nogueira. „A framework for verifying deadlock and nondeterminism in UML activity diagrams based on CSP". W: *Science of Computer Programming* (2020), strona 102497. DOI: 10.1016/j.scico.2020.102497.

[62] Yiyan Lin i Sandeep S Kulkarni. „Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability". W: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 2014, strony 237–247. DOI: 10.1145/2610384.2610398.

[63] Jonathan Lindo i Jeffrey Daudel. *Detecting race conditions in computer programs*. US Patent 7,673,181. Mar. 2010.

[64] *Linux man pages online*. http://man7.org/linux/man-pages/index.html. Odwiedzono: 2019-08-08.

[65] Peng Liu i Charles Zhang. „Axis: Automatically fixing atomicity violations through solving control constraints". W: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, strony 299–309. DOI: 10.1109/ICSE.2012.6227184.

[66] Carmen Torres Lopez i inni. „A study of concurrency bugs and advanced development support for actor-based programs". W: *Programming with Actors*. Springer, 2018, strony 155–185. DOI: 10.1007/978-3-030-00302-9_6.

[67] Shan Lu i inni. „AVIO: detecting atomicity violations via access interleaving invariants". W: *ACM SIGOPS Operating Systems Review*. Tom 40. 5. ACM. 2006, strony 37–48. DOI: 10.1145/1168917.116886.

[68] Shan Lu i inni. „Learning from mistakes: a comprehensive study on real world concurrency bug characteristics". W: *ACM SIGARCH Computer Architecture News*. Tom 36. 1. ACM. 2008, strony 329–339. DOI: 10.1145/1346281.1346323.

[69] Umang Mathur i Mahesh Viswanathan. „Atomicity Checking in Linear Time using Vector Clocks". W: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, strony 183–199. DOI: 10.1145/3373376.3378475.

[70] Thomas J McCabe. „A complexity measure". W: *IEEE Transactions on software Engineering* 4 (1976), strony 308–320. DOI: 10.1109/TSE.1976.233837.

[71] Mayur Naik, Alex Aiken i John Whaley. „Effective static race detection for Java". W: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2006, strony 308–319. DOI: 10.1145/1133981.1134018.

[72] Rober O'Callahan i Jong-Deok Choi. „Hybrid dynamic data race detection". W: *Acm Sigplan Notices*. Tom 38. 10. ACM. 2003, strony 167–178. DOI: 10.1145/1165389.945468.

[73] Alessandro Orso. „Don't forget the developers!(and be careful with your assumptions)". W: *Perspectives on Data Science for Software Engineering*. Elsevier, 2016, strony 271–275. DOI: 10.1016/B978-0-12-804206-9.00049-0.

[74] David Padua. *Encyclopedia of parallel computing*. Springer Science & Business Media, 2011. DOI: 10.1007/978-0-387-09766-4.

[75] Hsin Pan i Eugene H Spafford. „Heuristics for automatic localization of software faults". W: *World Wide Web* (1992).

[76] Shanchen Pang i inni. „A deadlock resolution strategy based on spiking neural P systems". W: *Journal of Ambient Intelligence and Humanized Computing* (2019), strony 1–12. DOI: 10.1007/s12652-019-01223-3.

[77] Jihyun Park, Byoungju Choi i Seungyeun Jang. „Dynamic Analysis Method for Concurrency Bugs in Multi-process/Multi-thread Environments". W: *International Journal of Parallel Programming* 48 (2020), strony 1032–1060. DOI: 10.1007/s10766-020-00661-3.

[78] Soyeon Park, Shan Lu i Yuanyuan Zhou. „CTrigger: exposing atomicity violation bugs from their hiding places". W: *ACM SIGARCH Computer Architecture News*. Tom 37. 1. ACM. 2009, strony 25–36. DOI: 10.1145/1508244.1508249.

[79] Kenneth Bryant Pierce i Ho-Yuen Chau. *Tools and methods for discovering race condition errors*. U.S. Patent US 7,174,554 B2. Lut. 2007. URL: https://patentimages.storage.googleapis.com/80/e3/9c/9e41ad7da9b31e/US7174554.pdf.

[80] Polyvios Pratikakis, Jeffrey S Foster i Michael Hicks. „LOCKSMITH: Practical static race detection for C". W: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33.1 (2011), strony 1–55. DOI: 10.1145/1889997.1890000.

[81] Dongsheng Qi, Naijie Gu i Junjie Su. „Detecting Data Race in Network Applications using Static Analysis". W: *2019 International Conference on Networking and Network Applications (NaNA)*. IEEE. 2019, strony 313–318. DOI: 10.1109/NaNA.2019.00061.

[82] Michael Roberson i Chandrasekhar Boyapati. „A Static Analysis for Automatic Detection of Atomicity Violations in Java Programs". W: (grud. 2010).

[83] Marc J Rochkind. *Advanced UNIX programming*. Pearson Education, 2004.

[84] Stefan Rozmus i Jerzy Stanik. „Metodyka Scrum–Moda czy konieczność". W: (paź. 2019), strony 288–303.

[85] Amit Sasturkar i inni. „Automated type-based analysis of data races and atomicity". W: *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM. 2005, strony 83–94. DOI: 10.1145/1065944.1065956.

[86] Stefan Savage i inni. „Eraser: A dynamic data race detector for multithreaded programs". W: *ACM Transactions on Computer Systems (TOCS)* 15.4 (1997), strony 391–411. DOI: 10.1145/265924.265927.

[87] Konstantin Serebryany i Timur Iskhodzhanov. „ThreadSanitizer: Data race detection in practice". W: *Proceedings of the workshop on binary instrumentation and applications*. ACM. 2009, strony 62–71. DOI: 10.1145/1791194.1791203.

[88] Chia-Shiang Shih i John A Stankovic. „Survey of deadlock detection in distributed concurrent programming environments and its application to real-time systems and Ada". W: *University of Massachusetts, Technical report UM-CS-1990-069* (1990).

[89] Abraham Silberschatz, Peter Baer Galvin i Greg Gagne. *Podstawy systemów operacyjnych*. Wydawnictwo Naukowo-Techniczne, 2003.

[90] Edward K Smith i inni. „Is the cure worse than the disease? overfitting in automated program repair". W: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, strony 532–543. DOI: 10.1145/2786805.2786825.

[91] Hanna Soroka-Potrzebna. „Zarządzanie projektami - podejście tradycyjne czy zwinne?" W: (2019), strony 89–98. DOI: 10.5604/01.3001.0013.2423.

[92] Bjarne Stroustrup. *The C++ Programming Language, 4th Edition*. Addison-Wesley, 2013.

[93] Ali Tehrani i inni. „DeepRace: Finding Data Race Bugs via Deep Learning". W: *arXiv preprint arXiv:1907.07110* (2019).

[94] *The Single UNIX®Specification*. http://www.unix.org/what_is_unix/single_unix_specification.html. Wersja 2. Odwiedzono: 2019-05-11.

[95] *TIOBE Index*. https://www.tiobe.com/tiobe-index/. Odwiedzono: 2019-10-12.

[96] Liu Tongping i inni. *Defeating deadlocks in production software*. US Patent App. 16/159,234. Kw. 2019.

[97] Nischai Vinesh i inni. „Confuzz—a concurrency fuzzer". W: *First International Conference on Sustainable Technologies for Computational Intelligence*. Springer. 2020, strony 667–691. DOI: 10.1007/978-981-15-0029-9_53.

[98] Christoph Von Praun i Thomas R Gross. „Static Detection of Atomicity Violations in Object-Oriented Programs." W: *Journal of Object Technology* 3.6 (2004), strony 103–122.

[99] Jan Wen Voung, Ranjit Jhala i Sorin Lerner. „RELAY: static race detection on millions of lines of code". W: *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM. 2007, strony 205–214. DOI: 10.1145/1287624.1287654.

[100] Chao Wang i inni. „Trace-based symbolic analysis for atomicity violations". W: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2010, strony 328–342. DOI: 10.1007/978-3-642-12002-2_27.

[101] Yin Wang i inni. „Gadara nets: Modeling and analyzing lock allocation for deadlock avoidance in multithreaded software. Decision and Control". W: *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*. IEEE, 2009, strony 4971–4976. DOI: 10.1109/CDC.2009.5399950.

[102] Yin Wang i inni. „Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs." W: *OSDI*. Tom 8. 2008, strony 281–294. URL: https://www.usenix.org/legacy/event/osdi08/tech/full_papers/wang/wang_html/.

[103] *What is Qt*. https://www.qt.io/what-is-qt/?utm_campaign=Navigation%202019&utm_source=megamenu. Odwiedzono: 2019-09-22.

[104] Rafał Wojszczyk i Damian Giebas. „Repair of Multithreaded Errors in the Control and Measurement System". W: *International Symposium on Distributed Computing and Artificial Intelligence*. Springer. 2021, strony 41–50. DOI: 10.1007/978-3-030-86887-1_4.

[105] Min Xu, Rastislav Bodík i Mark D Hill. „A serializability violation detector for shared-memory server programs". W: *ACM Sigplan Notices* 40.6 (2005), strony 1–14. DOI: 10.1145/1064978.1065013.

[106] Jifeng Xuan i Martin Monperrus. „Test case purification for improving fault localization". W: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, strony 52–63. DOI: 10.1145/2635868.2635906.

[107] Jialin Yang, Bo Jiang i Wing-Kwong Chan. „Histlock+: precise memory access maintenance without lockset comparison for complete hybrid data race detection". W: *IEEE Transactions on Reliability* 67.3 (2018), strony 786–801. DOI: 10.1109/TR.2018.2832226.

[108] Yue Yang i inni. *Detecting data race and atomicity violation via typestate-guided static analysis*. US Patent 8,510,722. Sierp. 2013.

[109] Jaeheon Yi, Caitlin Sadowski i Cormac Flanagan. „SideTrack: generalizing dynamic atomicity analysis". W: *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. 2009, strony 1–10. DOI: 10.1145/1639622.1639630.

[110] Wang Yu i inni. „Automatic Detection, Validation and Repair of Race Conditions in Interrupt-Driven Embedded Software". W: *IEEE Transactions on Software Engineering* (2020). DOI: 10.1109/TSE.2020.2989171.

[111] Zeming Yu, Linhai Song i Yiying Zhang. „Fearless Concurrency? Understanding Concurrent Programming Safety in Real-World Rust Software". W: *arXiv preprint arXiv:1902.01906* (2019).

[112] Zhen Yu, Y. Zuo i W.C. Xiong. „Concurrency Bug Avoiding Based on Optimized Software Transactional Memory". W: *Scientific Programming* 2019 (2019). DOI: 10.1155/2019/9404323.

[113] Zhen Yu, Yu Zuo i Yong Zhao. „Convoider: A Concurrency Bug Avoider Based on Transparent Software Transactional Memory". W: *International Journal of Parallel Programming* 48.1 (2020), strony 32–60. DOI: 10.1007/s10766-019-00642-1.

[114] Yuanyuan Zhou, Shan Lu i Joseph Andrew Tucek. *Atomicity violation detection using access interleaving invariants*. US Patent 8,533,681. Paź. 2013.