

# Niezawodność i odporność na błędy systemów informatycznych

Witold Paluszyński

Katedra Cybernetyki i Robotyki

Wydział Elektroniki, Politechnika Wroclawska

<http://www.kcir.pwr.edu.pl/~witold/>

2011–2017



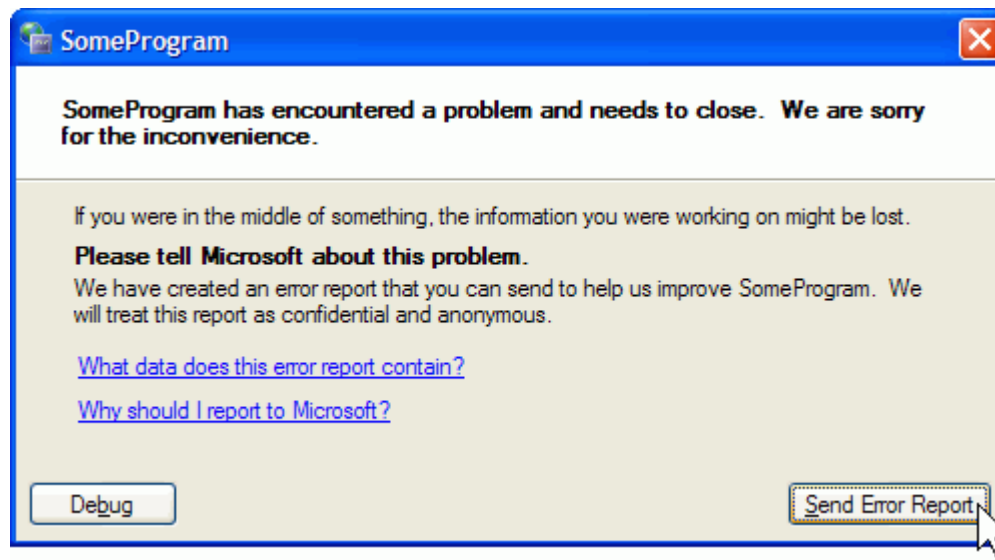
Ten utwór jest dostępny na licencji  
**Creative Commons Uznanie autorstwa-  
Na tych samych warunkach 3.0 Unported**

Utwór udostępniany na licencji Creative Commons: uznanie autorstwa, na tych samych warunkach. Udziela się zezwolenia do kopiowania, rozpowszechniania i/lub modyfikacji treści utworu zgodnie z zasadami w/w licencji opublikowanej przez Creative Commons. Licencja wymaga podania oryginalnego autora utworu, a dystrybucja materiałów pochodnych może odbywać się tylko na tych samych warunkach (nie można zastrzec, w jakikolwiek sposób ograniczyć, ani rozszerzyć praw do nich).



# Obsługa błędów

Jeśli „zwykły” program napotka błąd, którego nie potrafi rozwiązać albo naprawić, to typowym i normalnie stosowanym zachowaniem programu jest zakończenie pracy, z możliwie starannym i dokładnym poinformowaniem użytkownika (jeśli taki istnieje) o powstaniu błędu i jego okolicznościach.



Systemy czasu rzeczywistego i systemy wbudowane mają inne wymagania i inne podejście do traktowania i obsługi błędów, i powyższe podejście jest zwykle nie do przyjęcia. Na przykład, system sterujący procesem przemysłowym, po napotkaniu błędu fatalnego, nie może po prostu zatrzymać procesu, ponieważ mogłoby to być kosztowne i/lub niebezpieczne. Zamiast tego, powinien przejść do trybu podtrzymania minimalnej funkcjonalności, unikając całkowitej awarii.

# Niezawodność i odporność na błędy

Systemy czasu rzeczywistego i systemy wbudowane mają również specjalne wymagania dotyczące niezawodności. W celu ich osiągnięcia stosuje się cały zestaw technik:

- minimalizm w specyfikacji wymagań i projektowaniu systemów,
- specjalne metody projektowania i budowy oprogramowania, m.in. używanie bezpiecznych języków programowania, odpowiednie szkolenie programistów, itp.,
- weryfikacja i testowanie,
- odporność na błędy,
- efektywne usuwanie skutków awarii.

# Metryki złożoności oprogramowania

Niezawodność systemu oprogramowania jest związana z jego złożonością. Bardzo złożone oprogramowanie jest kosztowne do wytworzenia, i trudno zapewnić jego niezawodność. Stosuje się różne miary złożoności systemów oprogramowania, zwane metrykami.

W czasie projektowania nowego systemu oszacowanie złożoności pomaga w przewidywaniu kosztów, niezbędnego nakładu czasu, oraz innych potrzebnych zasobów.

Po zakończeniu budowy systemu obliczenie jego metryk i innych charakterystyk pomaga w zbudowaniu bazy doświadczeń dla przyszłych projektów.

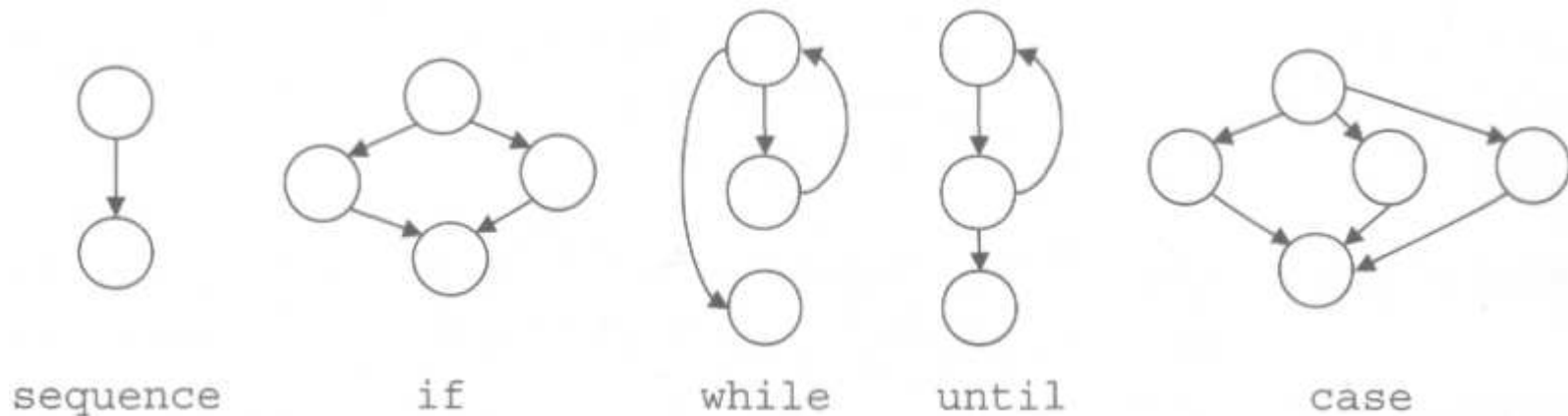
Stosowane metryki złożoności oprogramowania:

- Liczba wierszy programu (*Lines of Code, KLOC*), często liczona z pominięciem komentarzy, plików nagłówkowych, itp. W oczywisty sposób metryka ta nie bierze pod uwagę złożoności samego programu. Ponadto, często trudno obliczyć ją dla dopiero projektowanego systemu.

- Złożoność cyklomatyczna (*cyclomatic complexity*)  $C$ , obliczona na podstawie schematu blokowego programu (*flow graph*), gdzie  $e$  — liczba krawędzi grafu, a  $n$  — liczba wierzchołków:

$$C = e - n + 2$$

Ilustracją złożoności cyklomatycznej dla prostych fragmentów programów są następujące schematy blokowe:



Obliczenia złożoności cyklomatycznej można dokonać automatycznie, w trakcie kompilacji programu, lub przez analizę kodu źródłowego.

- Punkty funkcyjne (*Function Points*) jest innego rodzaju metryką, próbującą oszacować interakcje pomiędzy modułami projektowanej aplikacji, opartą o pewne jej parametry zewnętrzne. Wielką jej zaletą jest możliwość obliczenia na etapie projektowania, gdy żaden kod nie jest jeszcze napisany. Wykorzystuje takie parametry:
  - liczba źródeł wejściowych ( $I$ )
  - liczba wyjść ( $O$ )
  - liczba dialogów z użytkownikiem ( $Q$ )
  - liczba używanych plików ( $F$ )
  - liczba zewnętrznych interfejsów ( $X$ )

$$FP = 4I + 4O + 5Q + 10F + 7X$$

Istnieją bardziej rozbudowane wzory na  $FP$ , biorące pod uwagę dodatkowe aspekty projektowanej aplikacji.

- Poprzednie metryki nie uwzględniały specyfiki programów obiektowych. Definiuje się metryki podobne do *FP*, uwzględniające w przypadku aplikacji obiektowych takie parametry jak:
  - ważona liczba metod na klasę
  - głębokość drzewa dziedziczenia
  - liczba potomków w drzewie dziedziczenia
  - związki między klasami
  - brak spójności między metodami

Należy podkreślić, że stosowanie metryk ma ograniczone zastosowanie. Na przykład, przykładanie nadmiernej wagi do metryki *KLOC* może doprowadzić do sytuacji, w której programiści, lub cała firma realizująca projekt, będą tworzyli oprogramowanie o zawyżonej *KLOC*, w celu wykazania się i podniesienia rangi swojego produktu, z oczywistą szkodą dla projektu.



# Terminologia niezawodności

**Defektem** (*fault*) nazywamy wadę programu, błędny fragment kodu, np. brak sprawdzenia wielkości bufora przed wczytaniem do niego danych nieznanego wielkości. Istnienie defektu w programie nie oznacza, że błędny kod zostanie kiedykolwiek wykonany, gdy będzie wykonany to czy nastąpi sytuacja błędna (np. dane przekroczą rozmiar bufora), a gdy wystąpi, to czy spowoduje to jakiegokolwiek negatywne konsekwencje.

**Błędem** (*error*) nazywamy sytuację, gdy program znajdzie się w stanie różnym niż stan pożądany i poprawny. Np. przypadek, odwołanie się programu do adresu spoza dozwolonego zakresu jest błędem. Błąd taki może być jednak zauważony przez system, który może wysłać programowi sygnał. Jeśli program jest wyposażony w handler obsługujący sygnał danego typu, to program ma szansę poprawnego zachowania się w przypadku takiego błędu, i podjęcia właściwych działań.

**Awarią** (*failure*) nazywamy sytuację, kiedy program nie jest w stanie realizować swojej funkcji wskutek wystąpienia błędu.

**Niezawodnością** będziemy nazywać zdolność programu takiego radzenia sobie z defektami, a także błędami, które nie dopuszczają do wystąpienia awarii.

# Błędy

Błędy można podzielić na dwie istotne kategorie:

- błędy powtarzalne — takie, dla których znana jest przynajmniej jedna ścieżka prowadząca do ich wystąpienia,
- błędy ulotne — (*transient error*) to takie, dla którego nie można precyzyjnie określić warunków jego wystąpienia, a zatem nie ma możliwości wywołania go w prosty i powtarzalny sposób.

Znaczenie powyższego rozróżnienia błędów jest takie, że procedury związane z wykrywaniem błędów są inne dla tych kategorii.

# Zapobieganie defektom

Zapobieganie defektom (*fault prevention*) sprowadza się do dwóch grup procedur:

- unikanie defektów (*fault avoidance*)
  - rygorystyczna i/lub formalna specyfikacja wymagań
  - zastosowanie sprawdzonych metod projektowania
  - użycie języków z mechanizmami wspierającymi abstrakcje, weryfikację, itp.
  - użycie narzędzi inżynierii oprogramowania
- usuwanie defektów (*fault removal*)
  - weryfikacja
  - walidacja
  - testowanie



# Weryfikacja, walidacja, i testowanie

**Weryfikacja** jest procesem realizowanym na wielu etapach cyklu rozwoju oprogramowania, w celu potwierdzenia poprawności i zgodności ze specyfikacją danego modułu, i całego systemu. Do weryfikacji można wykorzystać wiele narzędzi, w tym narzędzi analizy formalnej.

**Walidacja** jest procesem analizy ukończonego produktu, lub prototypu, dla stwierdzenia czy jest zgodny z wszystkimi wymaganiami, w tym również czy formalna specyfikacja jest zgodna z intencją i oczekiwaniami użytkownika, oraz czy uruchomiony w środowisku produkcyjnym program realizuje swoje funkcje.

Podstawowym narzędziem walidacji jest **testowanie**.

# Testowanie

Testowanie jest procesem powtarzalnego uruchamiania programu z określonymi danymi wejściowymi, w celu stwierdzenia, czy program produkuje właściwe sygnały wyjściowe.

Jakkolwiek w trakcie testowania ujawniają się defekty i błędy, które powinny następnie być korygowane, wykrywanie i poprawianie błędów nie jest jedynym celem testowania. Ogólnie, testowanie nie jest w stanie ani wykryć wszystkich błędów, ani potwierdzić ich braku. Na odwrót, za pomocą testowania można jedynie wykrywać istniejące błędy. Natomiast dodatkową rolą testowania jest wytworzenie zaufania do programu, jeśli zachowuje się on poprawnie w dobrze zaprojektowanych, wszechstronnych testach.

# Testowanie oprogramowania

Testowanie może przeanalizować jedynie małą część całej przestrzeni możliwych danych wejściowych. Powinno ono być zatem tak przeprowadzone, aby jego wyniki w przekonujący sposób potwierdziły hipotezę, że system będzie działał poprawnie dla wszystkich danych. Metody wyboru danych wejściowych:

- wybór losowy,
- pokrycie wymagań — dla każdego z wymagań zestawu danych potwierdzające spełnienie danego wymagania,
- testowanie *white-box* — realizowane jest pokrycie według jakiegoś kryterium wynikającego z analizy programu, np. przejście wszystkich rozgałęzień logicznych w programie,
- wybór oparty na modelu — dane są generowane z modelu systemu pracującego w połączeniu z modelem obiektu fizycznego,
- profil operacyjny — bazą do wyboru danych testowych jest profil operacyjny,
- szczytowe obciążenie — generowane jest ekstremalne obciążenie systemu, i w takich warunkach sprawdzane spełnienie wymagań czasowych,
- przypadek najgorszego czasu wykonania (WCET) — dane generowane na podstawie analizy kodu w kierunku WCET,
- mechanizmy tolerancji defektów — testowanie z zastosowaniem „wstrzykiwania defektów”,
- systemy cykliczne — testowanie w zakresie jednego pełnego cyklu.

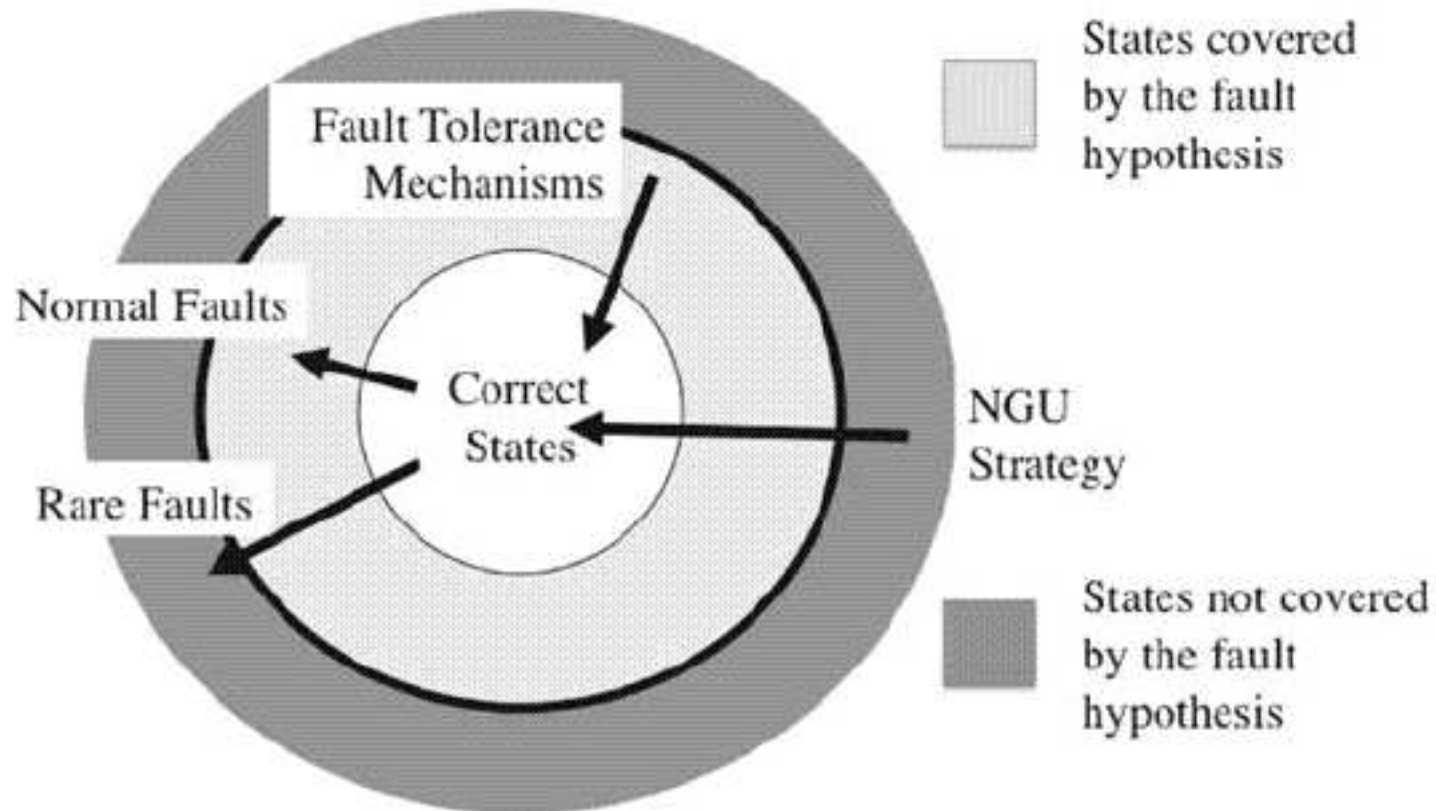
# Testowanie sprzętu

- testowanie CPU
- testowanie pamięci ROM
- testowanie pamięci RAM
- testowanie innych urządzeń



# Odporność na błędy

Podstawą zbudowania odporności systemu na błędy (*software fault tolerance*) jest sformułowanie **hipotezy defektów** określającej jakie rodzaje defektów mają być tolerowane przez system. Hipoteza dzieli przestrzeń stanów systemu na trzy regiony:



Dodatkowo: strategia NGU (**Never Give Up** — Nigdy Nie Rezygnuj).

# Redundancja

**Redundancja** (nadmiarowość) jest jedną z głównych technik budowania odporności na błędy, zarówno w sprzęcie jak i oprogramowaniu. W oczywisty sposób, ponieważ prowadzi ona do budowy bardziej złożonych systemów, może sama w sobie wprowadzać ryzyko dalszych defektów i związanych z nimi awarii.

**Redundancja statyczna** (maskująca) jest wbudowana wewnątrz systemu, który usiłuje dzięki niej utrzymać poprawne działanie maskując występujące błędy. Jedną z podstawowych technik jest TMR (*Triple Modular Redundancy*), polegająca na zastosowaniu trzech identycznych elementów, i układu **głosowania**, który porównuje sygnały na wyjściach wszystkich elementów, i jeśli jeden różni się od dwóch pozostałych to na wyjście układu kierowany jest sygnał wyjściowy wybrany większością. TMR ma głównie zastosowanie do uodpornianie na błędy sprzętu.

**Redundancja dynamiczna** polega na wbudowaniu w system elementu oceniającego, czy nie występują błędy. W przypadku wykrycia błędu, układ sygnalizuje to, pozostawiając jednak elementom zewnętrznym podjęcie odpowiednich działań. Redundancja dynamiczna jest zatem metodą wykrywania błędów. Przykładami mogą być bity parzystości pamięci, albo sumy kontrolne w pakietach komunikacji.

# Programowanie N-wersji

Zastosowanie redundancji typu TMR opiera się na założeniu, że błąd powstanie wewnątrz jednego z układów, i będzie to błąd przypadkowy, albo wynikający ze starzenia się sprzętu. Ponieważ systemy programowe nie starzeją się, a błędy przypadkowe nie są najważniejszymi błędami, na które system powinien być odporny, zatem podejście typu TMR ma ograniczone zastosowanie do systemów oprogramowania.

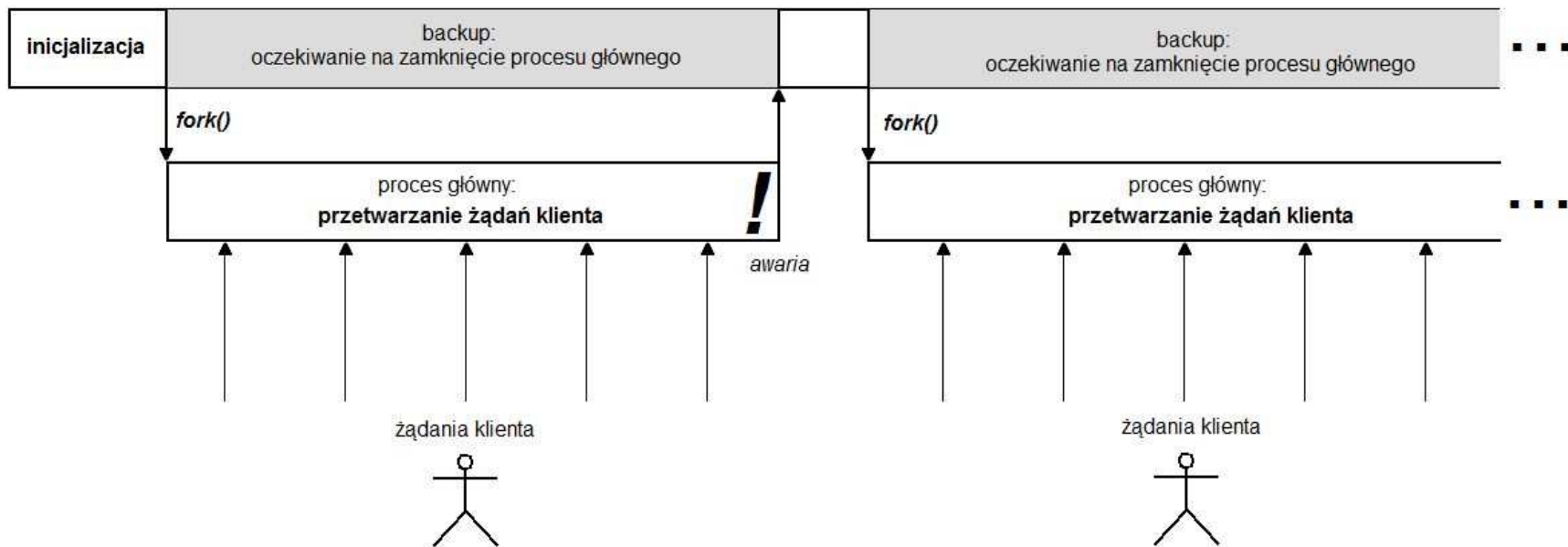
Zamiast tego, budowanie odporności koncentruje się na możliwych błędach programowych. Metoda zwana **programowaniem N-wersji** (*N-version programming*) polega na stworzeniu N funkcjonalnie równoważnych programów odpowiadających jednej specyfikacji. Programy powinny być budowane przez N różnych programistów (lub grup), bez komunikowania się między sobą. Programy następnie wykonywane są jednocześnie w systemie, i dodatkowy proces **drivera** porównuje uzyskane wyniki i wybiera jeden metodą głosowania.

Skuteczność tej metody opiera się na założeniu, że programy stworzone niezależnie, mają różne defekty, i będą powodowały błędy niezależnie od siebie. To założenie może być niesłuszne, jeśli np. programy zostały napisane w tym samym języku programowania, i skompilowane tym samym kompilatorem.

# Dublowanie procesów

Techniką znacznie prostszą niż programowanie N-wersji jest **dublowanie procesów**. Ma ona zastosowanie do uodporniania systemu na błędy ulotne.

Metoda polega na tworzeniu nadmiarowego podprocesu do wykonywania obliczeń, a w przypadku gdyby napotkał on na jakiś błąd, zamyka się on zwracając odpowiedni kod błędu. Proces nadzorujący stwierdza wystąpienie błędu, i tworzy identyczny proces do ponownego wykonania tych samych obliczeń.



Dublowanie procesów może być stosowane wielokrotnie na różnych poziomach. Na przykład procedura może najpierw inicjalizować środowisko obliczeń, pozyskiwać dane, itp., a dopiero potem inicjować zasadnicze obliczenia. Zarówno pierwsza faza jak i druga mogą podlegać oddzielnemu dublowaniu. W oczywisty sposób, dublowanie drugiej fazy jest mniej uciążliwe i nie powoduje konsekwencji wykraczającej poza program.

Zastosowanie tego podejścia jest szczególnie łatwe i atrakcyjne w systemach Unikso-podobnych, wykorzystujących model tworzenia procesu przez klonowanie funkcją fork.

# Punkty kontrolne

Metoda polega ona na tworzeniu w programie punktów bezpiecznego wycofania się. Jeżeli program w trakcie pracy sam wykryje błąd, to znaczy jakiś niepoprawny stan, to najlepiej byłoby cofnąć się o kilka kroków, kiedy stan był jeszcze poprawny, i powtórzyć małą porcję obliczeń.

Aby to było możliwe, należy w trakcie pracy okresowo, po zweryfikowaniu, że stan programu jest poprawny, zapamiętać go w sposób umożliwiający cofnięcie programu do tego stanu. W ten sposób program tworzy **punkt kontrolny**. Po wykryciu błędu, program cofa się do ostatniego takiego punktu, i wznawia obliczenia tak jakby nic się nie stało. Założeniem tej metody jest, że ponowne wykonanie pewnej fazy obliczeń da tym razem inne wyniki. Założenie jest poprawne jeśli błąd był wywołany czynnikami zewnętrznymi, albo jakąś kombinacją mikrostanów programu, która nie zostanie powtórzona.

# Bloki wznowiania (recovery blocks)

Metoda **bloków wznowiania** wykorzystuje zwykle bloki programowe uzupełnione o **punkt wznowiania** umieszczony na początku bloku, i **test akceptowalności** na końcu. Test jest wykonywany dla stwierdzenia czy system znajduje się w akceptowalnym stanie po wykonaniu bloku. Jeśli nie, to wykonanie programu wraca do stanu na wejściu do bloku.

Po wznowieniu obliczeń, program powinien wykorzystać alternatywny moduł obliczeniowy. (Jeśli obliczenia zostaną wznowione z użyciem tego samego modułu, to tą metodą można się zabezpieczyć jedynie przed błędami ulotnymi.) Gdyby obliczenia wszystkich alternatywnych modułów zawiodły, to sterowanie wraca do modułu nadrzędnego, który też może mieć swój blok wznowiania.

Metoda bloków wznowiania jest popularną techniką, jednak jej zastosowanie w systemach czasu rzeczywistego jest ograniczone do przypadków, w których ograniczenia czasowe pozwalają na ponawianie obliczeń, i wynik uzyskany po dodatkowym nakładzie obliczeń jest nadal przydatny.

Zastosowanie tego podejścia w systemach Unikso-podobnych, wykorzystuje na ogół funkcje `setjmp` i `longjmp` do zachowywania stanu i wznowiania obliczeń.

# Odmładzanie

Metoda odmładzania procesów opiera się na założeniu, że stan początkowy po uruchomieniu jest zawsze najlepiej przetestowany, i przez jakiś czas po starcie system pracuje bezawaryjnie.



Metoda jest ograniczona przez tzw. stan twardy systemu, to jest stan po inicjalizacji, komunikacji z urządzeniami zewnętrznymi, itp. Ten stan zwykle nie powinien być utracony w procesie odmładzania. Jednak idea odmładzania polega na porzuceniu poprzedniego stanu, i zainicjowaniu go od nowa!!



# Mikro restarty

Restart systemu może być środkiem zapobiegania błędom, albo metodą przywrócenia systemu do stanu poprawnego. O ile jednak restart całego systemu często powoduje zaburzenie lub utrudnienie w pracy, to metodą może być podział systemu na szereg mniejszych elementów, które można restartować niezależnie od innych.

Zauważmy, że w niektórych systemach, jak Windows, po operacjach takich jak instalacja lub reinstalacja jakiegoś programu wymagany jest restart całego systemu. Wynika to z faktu, że bezpośrednio po starcie system konfiguruje sobie całe dostępne oprogramowanie. Gdyby tę warstwę konfiguracji oprogramowania wydzielić jako osobny podsystem, który mógłby być restartowany samodzielnie, nie byłoby konieczności restartu całego systemu. Metoda ta jest szeroko stosowana w innych systemach operacyjnych.

# Watchdog

W systemach czasu rzeczywistego procedura restartu po wystąpieniu i wykryciu awarii jest często zautomatyzowana i rutynowo implementowana. Jedną ze stosowanych metod jest tzw. *watchdog*, czyli system monitorujący pracę systemu, i wykonujący fizyczny restart po zauważeniu nieprawidłowości.



Watchdog może (powinien) być zrealizowany sprzętowo i niezależny od reszty systemu, co daje gwarancję jego poprawnej pracy nawet jeśli awaria systemu wyłącza z akcji inne jego mechanizmy odpornościowe.



Po uruchomieniu watchdog cyklicznie uruchamia timer na zaprogramowany odcinek czasu (np. 100 milisekund), po którym inicjuje restart systemu, o ile nie zostanie zresetowany zaprogramowanym kodem wpisanym na wejście. Restart systemu następuje również w przypadku zaniku zasilania watchdoga.

# Poprawianie stanu

Poprawianie stanu polega na podjęciu działań doraźnych, w przypadku wykrycia nieprawidłowości. Jednak zamiast poszukiwania jej źródeł, i podjęcia próby radykalnej naprawy sytuacji, likwidowane są tylko objawy.

Jest to więc podobne do objawowego leczenia choroby. Zamiast podawać antybiotyki, podajemy lekarstwo zbijające gorączkę. Jak wiemy, takie leczenie stosuje się, i jest to słuszna metoda, jeśli nic innego w danej chwili nie można zrobić, a poprawienie stanu może spowodować niedopuszczenie do natychmiastowej awarii, i dalsze działanie systemu, przynajmniej przez jakiś czas.