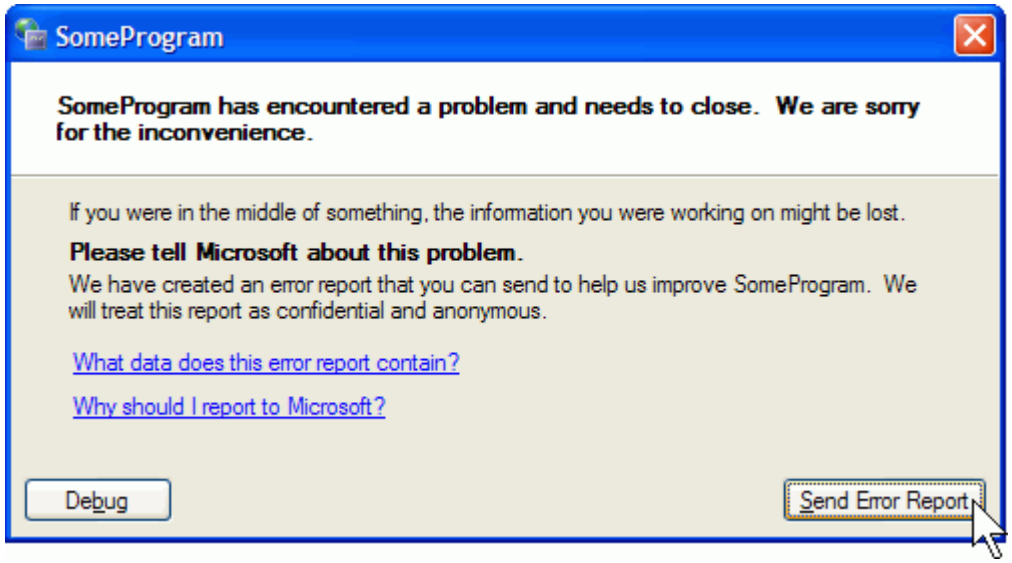
Niezawodność i odporność na błędy systemów informatycznych

Witold Paluszyński
Katedra Cybernetyki i Robotyki
Wydział Elektroniki, Politechnika Wrocławska
http://www.kcir.pwr.edu.pl/~witold/

2011–2020

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.

Awarie i katastrofy — perspektywa historyczna

Awarie systemów budowanych przez człowieka, i wynikających z nich zagrożeń i/lub katastrof są prawdopodobnie tak stare jak ludzkość. Przykładem mogą być katastrofy budowlane, takie jak zawalenie się budynków, mostów, itp.

W kontekście gwałtownego przyspieszenia rozwoju przemysłu i technologii w XIX-tym wieku, pojawiło się zainteresowanie zagadnieniami niezawodności. W braku dobrych modeli niezawodności i teorii pozwalających precyzyjnie obliczać wytrzymałość konstrukcji, były one budowane z dużym zapasem wytrzymałości — dwukrotnym, czterokrotnym, a nawet sześcio- lub więcej.

Takie praktyki są rzadko (lub nigdy) stosowane w inżynierii systemów komputerowych.

Therac-25

Therac-25 to produkowana w latach 1980-tych przez Atomic Energy of Canada Limited seria akceleratorów cząstek do leczenia nowotworów.

W latach 1985-1987 doszło do serii wypadków w czasie leczenia pacjentów, po których co najmniej pięcioro pacjentów zmarło na skutek napromieniowania.

Przyczyną awarii były wyścigi przy inicjalizacji parametrów maszyny, które w pewnych, trudnych do odtworzenia warunkach, nie zostawały poprawnie ustawiane. Wcześniejsze serie tych maszyn Therac-6 i Therac-20 posiadały zabezpieczenia mechaniczne, które jednak wyeliminowano w celu redukcji kosztów.

Ariane 5

4 czerwca 1996 bezzałogowa rakieta Ariane 5 Europejskiej Agencji Kosmicznej ESA wystartowała z poligonu w Gujanie Francuskiej. Po 37 sekundach lotu wykonała obrót o 90 stopni w niewłaściwym kierunku, następnie wskutek powstałych przeciążeń uległa destrukcji, po czym wybuchowi uległo całe paliwo z ciekłego wodoru.

Koszt tej katastrofy: $\gg$ 500 milionów USD.

Jak do tego doszło?

Oprogramowanie sterujące lotem, napisane w języku Ada, przeniesione z wcześniejszej generacji rakiet Ariane 4, było traktowane jako dobrze przetestowane i niezawodne. Jednak szereg czynników uległo zmianie, i między innymi 64-bitowa zmienna Horizontal Bias zawierająca kąt pochylenia typu float przy konwersji na 16-bitowy signed int przekroczyła 65535, i wykazała bezsensowną wartość.

Awarie systemów komputerowych

Wymienione awarie zdarzyły się dawno, ale dotyczą spektakularnych porażek systemów komputerowych, które zostały dobrze zbadane i opisane. Wnioski z tych awarii wpłynęły na metodologię tworzenia oprogramowania przez wiele lat.

Pytanie: czy w dzisiejszych czasach nie zdarzają się już takie awarie?

Oczywiście, że się zdarzają, a nawet jest ich tak wiele, że większość z nich nie jest powszechnie znanych, katastrofy na krótko stają się sensacją medialną, ale przyczyny awarii i dokładna historia do nich prowadząca są opisywane jedynie w prasie fachowej.

Np. podsystem MCAS systemu sterowania samolotów Boeing 737 MAX ...

Niezawodność i odporność na błędy

Systemy czasu rzeczywistego i systemy wbudowane mają 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.

Niezawodność a bezpieczeństwo systemów

Niezawodność jest czasami utożsamiana z **bezpieczeństwem** systemów komputerowych, zwłaszcza w odniesieniu do oprogramowania. Jednak o ile niezawodność jest zwykle definiowana w kategoriach realizacji przewidzianych funkcji systemu, to bezpieczeństwo definiuje się w kategoriach unikania zagrożeń i wypadków, niezależnie od realizacji funkcji systemu i kosztów.

Technologie pozwalające osiągać jedną i drugą jakość są często różne, a wręcz często wymagania niezawodności pozostają w konflikcie z wymaganiami bezpieczeństwa. Zauważmy, że najprostszym sposobem sprawienia aby samolot był 100%-owo bezpieczny jest by nie wzbijał się on w ogóle w powietrze, nie miał silników ani zbiorników paliwa ...

Ogólnie postępowanie w razie awarii systemu często prowadzi do następującego kompromisu:

- przejście do trybu bezpiecznego
- podtrzymanie pełnej funkcjonalności i wydajności

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.

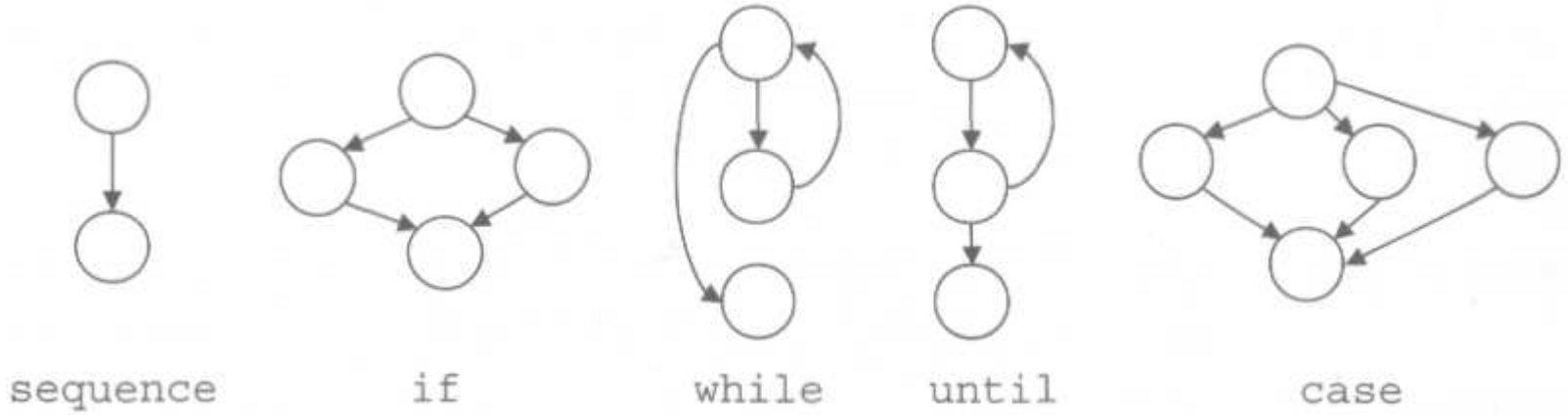
Stosowane metryki złożoności oprogramowania:

- 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 (*defect, fault*) nazywamy wadę programu, błędny fragment kodu, np. brak sprawdzenia wielkości bufora przed wczytaniem do niego danych nieznanej 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 jakiekolwiek negatywne konsekwencje.

Błędem (*error*) nazywamy sytuację, gdy program znajdzie się w stanie różnym niż stan pożądany i poprawny. Np. przypadek odwołania 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 dopuszcza 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 błędów i poprawianie defektów nie jest jedynym celem testowania. Ogólnie, testowanie nie jest w stanie ani wykryć wszystkich błędów, i tym bardziej defektów, 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 zestawy 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 systemów czasu rzeczywistego

Testowanie systemów czasu rzeczywistego jest specjalnym przypadkiem. Te systemy muszą reagować w przewidzianym czasie na różne możliwe zdarzenia, których dokładnej sekwencji czasowej nie sposób przewidzieć.

Z tego powodu w zakresie tworzenia systemów czasu rzeczywistego i systemów wbudowanych, testowanie ma ograniczone znaczenie. Prace w zakresie tworzenia takich systemów i narzędzi do ich budowy koncentrują się na metodach formalnej weryfikacji oprogramowania.

Testowanie sprzętu

Elektronika podlega procesom starzenia, efektom przepięć, promieni kosmicznych, korozji, wibracji, itp., i może degradować się i ulegać uszkodzeniom. W systemach pracujących 24x7 przez długi okres czasu ma sens regularne testowanie sprzętu. Testy powinny być uruchamiane okresowo, w czasie mniejszego obciążenia systemu.

- testowanie CPU — starannie przygotowany zestaw testowy powinien sprawdzać poprawność pracy procesora we wszystkich trybach adresowania

- testowanie pamięci RAM — w przypadku pamięci ze sprzętową kontrolą parzystości lub korekcyjnymi kodami Hamminga (ECC/EDC) ewentualne błędy są korygowane w czasie odczytu; jednak nie powoduje to poprawienia zawartości pamięci, dlatego stosuje się **szorowanie pamięci** (*memory scrubbing*), polegające na cyklicznym odczytywaniu i powrotnym zapisywaniu wszystkich komórek pamięci, wymuszające zapis poprawionej wartości; oczywiście nie powoduje to naprawienia permanentnie uszkodzonych komórek pamięci

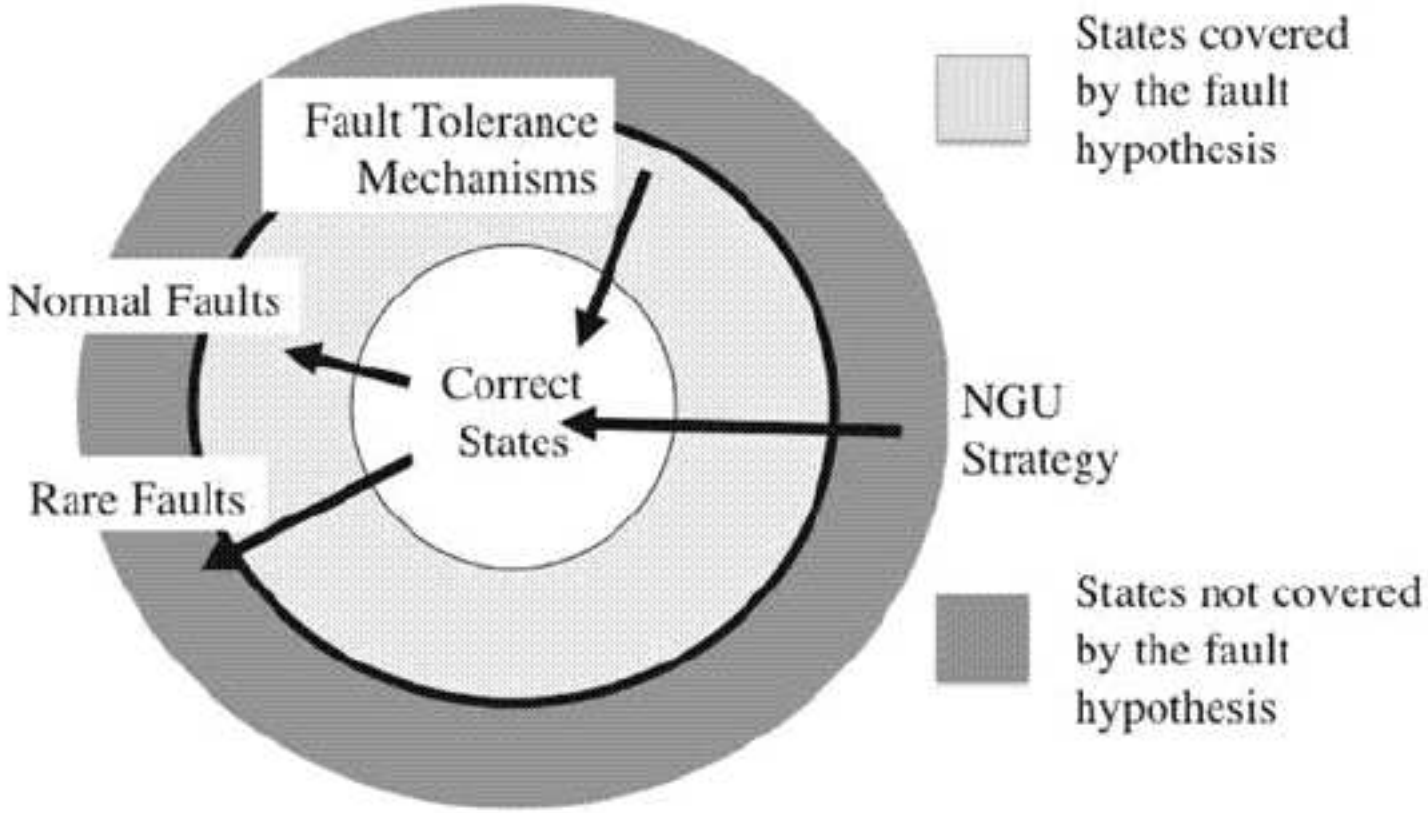
  w przypadku użycia pamięci bez sprzętowego wykrywania i korygowania błędów stosuje się cykliczne testowanie pamięci metodą zapisu i kontrolnego odczytu starannie wybranych wzorców bitowych, co pozwala na wykrycie zarówno przepalonych bitów jak i przesłuchów między ścieżkami sygnałowymi

- testowanie pamięci ROM — zawartość pamięci ROM można testować za pomocą obliczonych w czasie instalacji i konfiguracji systemu sum kontrolnych, albo — lepiej — kodów CRC, pozwalających wykryć wszystkie błędy 1-bitowe i praktycznie wszystkie błędy wielobitowe

- testowanie innych urządzeń — urządzenia takie jak przetworniki A/D, D/A, multipleksery i kanały wejścia/wyjścia mogą mieć wbudowane moduły testowe, sprawdzające i zapisujące stan urządzenia do pamięci za pomocą DMA

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 większościowo. TMR ma głównie zastosowanie do uodpornianie na błędy sprzętu.

Redundancja dynamiczna polega na wyposażeniu systemu w element oceniający 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 chcemy uodpornić system, zatem podejście TMR ma ograniczone zastosowanie do systemów oprogramowania.

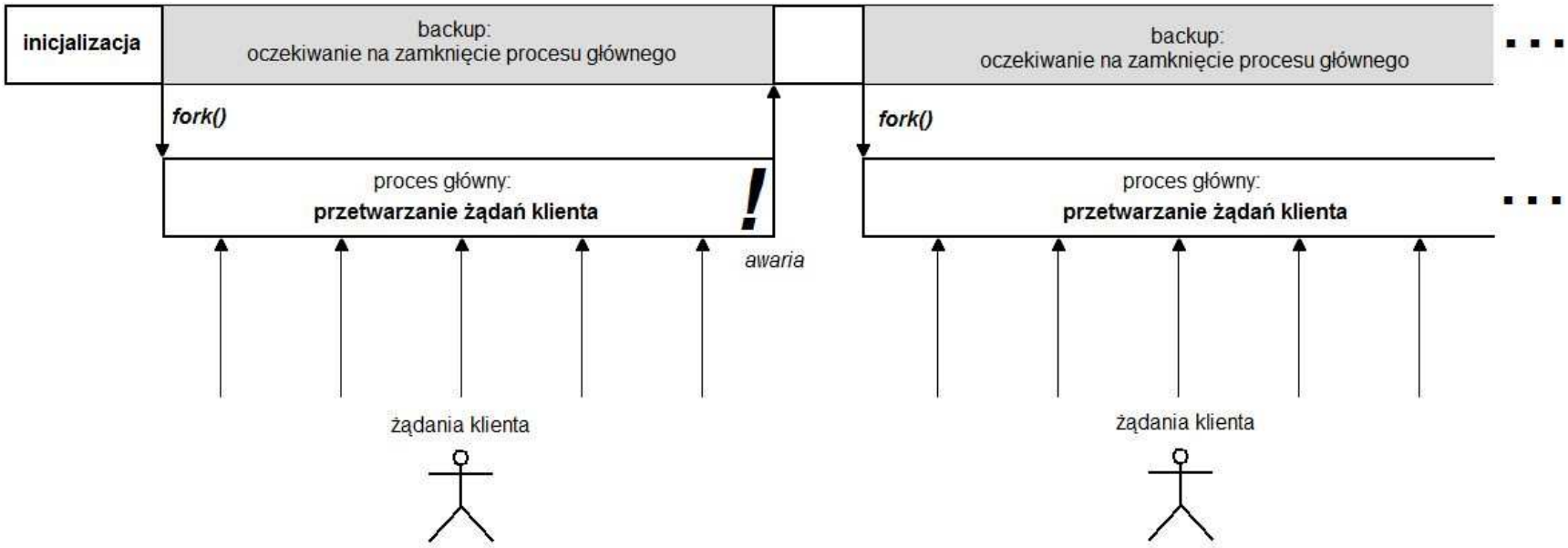
Zamiast tego, budowanie odporności koncentruje się na możliwych błędach programowych. Metoda zwana **programowaniem N-wersji** (*N-version programming*) polega na stworzeniu N funkcjonalnie równoważnych programów odpowiadających jednej specyfikacji. Programy powinny być budowane przez N różnych programistów (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 i/lub z tymi samymi bibliotekami.

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 wznawiania (recovery blocks)

Metoda **bloków wznawiania** wykorzystuje wiele (kilka) wersji zwykłych bloków programowych uzupełnionych o **punkt wznawiania** umieszczony na początku bloku, i **test akceptowalności** na końcu. Po wykonaniu bloku wykonywany jest test dla stwierdzenia czy system znajduje się w akceptowalnym stanie. Jeśli nie, to wykonanie programu wraca do punktu wznawiania na wejściu do bloku.

Po wznowieniu obliczeń, program uruchamia **alternatywny moduł** obliczeniowy. (Wznowienie obliczeń z użyciem modułu podstawowego pozwoliłoby na zabezpieczeniu programu jedynie przed błędami ulotnymi.) Gdyby zawiodły obliczenia wszystkich modułów alternatywnych, to sterowanie wraca do modułu nadrzędnego, który też może mieć swój blok wznawiania.

Metoda bloków wznawiania jest popularną techniką, jednak jej zastosowanie w systemach czasu rzeczywistego jest ograniczone do przypadków, w których ograniczenia czasowe pozwalają na powtarzanie 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 wznawiania 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 wykryciu zbyt długiego czasu wykonywania się programu.

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 sam nie zostanie zresetowany zaprogramowanym kodem wpisanym na wejście. Restart systemu następuje również w przypadku zaniku zasilania watchdoga przez system.

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.

Na przykład, wiedząc, że zmienna powinna zawierać wartość temperatury zmierzonej przez czujnik, i że powinna ona zawierać się w przedziale 0-70° moduł poprawiania stanu mógłby, po wykryciu wartości 95 napisać ja wartością 70, albo po wykryciu wartości -13 nadpisać ją wartością 0. Inaczej mówiąc, zauważywszy, że wartość jest niepoprawna, ustawiamy ją na poprawną, wybraną tak, by z pewnym prawdopodobieństwem była zbliżona do prawidłowego stanu systemu.

Jest to więc rodzaj objawowego leczenia choroby. Zamiast podawać antybiotyk, 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.

Bibliografia

Algirdas Avižienis, A., et al., Basic concepts and taxonomy of dependable and secure computing, IEEE Trans. on Dependable and Secure Computing, Vol.1(1), pp.11-33, 2004