Univerzitet Union

Testiranje sigurnosno-kritičnog koda korišćenjem metode ubrizgavanja greške u ugrađenim (embedded) sistemima u automobilskoj industriji

- Diplomski rad -

Mentor:
Petar Bojović

Student:
Nikola Šimšić RM10/11

Beograd, Oktobar 2020.

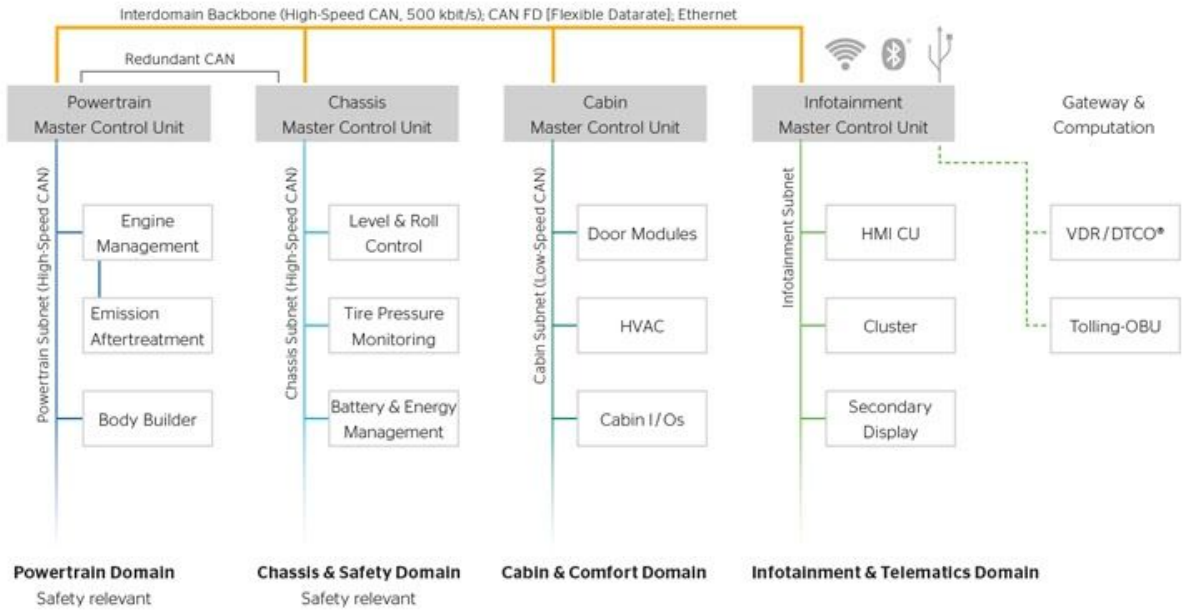
Sadržaj

1. Uvod

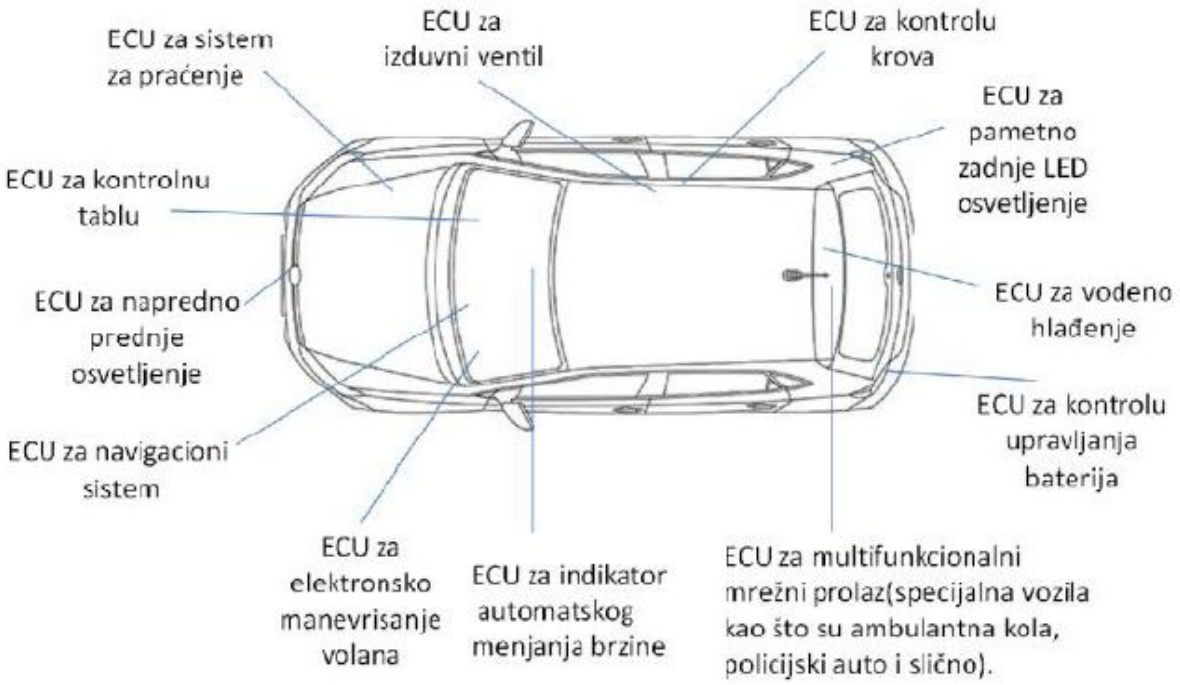
U prethodnih desetak godina, dešava se eksponencijalni rast računarskih funkcija i komponenti ugrađenih u vozila. Razvojni procesi, tehnološke tehnike i alati, razvijaju se da isprate tu evoluciju. Celokupan set elektronskih funkcija, kao što su navigacija, kontrola prilagođavanja, informacije iz saobraćaja, kontrola trakcije, stabilizacija vozila, kao i razni aktivni sigurnosni sistemi, su implementirani u današnja vozila. Mnoge od tih funkcija nisu samostalne funkcije, već su zasnovane na konstantnoj sinhronizaciji i komunikaciji sa ostalim delovima sistema. Kompleksnost **embedded (ugrađene)** arhitekture konstantno raste. Sa dolaskom procesorskih platformi u vozila, dolazi do velikog rasta performansi, u pogledu memorije, računarske snage i povezivanaja, što ujedno povećava i složenost samih softvera[1].

Čitava automobilska industrija trenutno se sučava sa dramatičnim tehnološkim promenama. Ova promena nastaje dolaskom poptuno novih električnih i elektronskih arhitektura (E/E arhitekture, koja je konvergencija domena: elektronički hardver, mrežna komunikacija, softverske aplikacije i ožičenje, koji se zajedno kombinuju u arhitekturi vozila. Upravljanje motorom, kočenje, upravljanje informacijama, zabavnim sadržajem, i ostale karakteristike komfora i praktičnosti unutar automobila, oslanjaju na električne i elektroničke sisteme, grafički prikazano na slici 1.) i sa njima, potpuno nove softverske arhitekture. Ključni pokretači ovakvog razvoja su sledeći:

- Prelaz sa klasičnih ugrađenih kontrolnih jedinica zasnovanih na mikrokontroleru, na mikroprocesorska rešenja ili rešenja zasnovana na cloud-u, upravo zbog količine i raznovrsnosti podataka različitih aplikacija koji se obrađuju, za razliku od mikrokontrolera koji su dizajnirani za predefinisane zadatke.
- Prelazak sa algoritama zasnovanih na kontroli na algoritme vođene podacima (veštačka inteligencija, obrada slike, fuzija podataka, povezanost). Kako se sve se manje oslanja na čovekovu ulogu u vožnji, od raznoraznih senzora, informacija iz okruženja koje su na raspolaganju vozaču, sve do potpuno autonomnih vozila, ka čemu se i teži, obrada velike količine podataka je od esencijalnog značaja za razvoj.
- Odvajanje softvera u vozilu od osnovnog hardvera. Celokupan enterijer modernih vozila je digitalan, mnoge svetske vodeće kompanije kao što su Google, Apple, Baidu, Alibaba i ostale, razvijaju raznorazne aplikacije koje su na raspolaganju u Infotainment sistemima, koje razvijaju dobavljači (Tier). Kako bi OEM-ovi imali fleksibilnost da lakše menjaju svoje dobavljače i ažuriraju vozila novih funkcionalnostima nakon prodaje, čime se i produktivnost inženjera daleko povećava, odvajanje od hardvera je jako bitno. Uz to, sam razvoj softvera je daleko sigurniji i lakši na poznatim, standardizovanim operativnim sistemima i hardverskim platformama. Samim tim, dokazani i provereni aplikativni softver, se lako može preuzeti i ponovo koristiti.



Slika 1: Prikaz E/E Arhitekture



Slika2 : Prikaz elektronskih kontrolnih uređaja unutar modernog automobila

Od uvođenja ABS-a (anti-lock breaking system, deo celokupnog sistema za stabilnost, utiče na kočioni sistem, obezbeđuje nam da se točkovi automobila ne zaključaju i prestanu okretati pri kočenju, što bi dovelo do trenja, tako što u kontinuitetu neprekidno vrše optimalan pritisak na točkove sa povremenim trenutnim otpuštanjem i stiskanjem, dovodeći do kočenja i postepenog usporavanja) za automobile 1978. veliki broj elektronskih kontrolnih jedinica (ECU) je sukcesivno uveden u vozila poslednjih decenija. U današnja premium vozila se ugrađuje na stotine kontrolnih uređaja različitih Tier 1 dobavljača (slika 2). Tier 1 dobavljači su kompanije koje su direkni proizvođači delova i komponenti za glavne OEM proizvođače (Audi, Opel, Škoda, BMV…) Tokom decenija, ove komponente su iterativno razvijane u kontekstu OEM-specifičnih E/E arhitektura u vozilu. Nivo 1 se kreće u sličnom kontekstu s nekoliko OEM-ova za svaki segment proizvoda i samim tim se optimizuje putem odgovarajućih hardverskih platformskih rešenja. U ovoj situaciji, hardverske komponente određuju glavne pokretačke programe za softver. Ovaj obrazac saradnje može se naći u različitim oblastima primene. Međutim, postoje značajne razlike u načinu na koji se komponente (postupci, metode, alati) razvijaju u pojedinim oblastima. Softverske komponente moraju biti nezavisne od hardvera, za širu upotrebu. Tehnologija mikrokotrolera određuje restriktivne zahteve za dostupnu memoriju i računarsko vreme. Moderni softver za upravljanje motornim jedinicama se mora zadovoljiti sa 8MB fleš memorije na mikrokontroleru. Za sam softver, ovo rezultuje arhitekturom koja je prilagođena memoriji.

Takođe, dinamičko alociranje memorije se ne koristi, kako bi se sačuvali resursi a i iz bezbednosnih razloga, o čemu će biti reči kasnije. Kako bi se postigli vremenski zahtevi izvršavanja, koji su pritom u skladu sa funkcionalnim sigurnostima, koriste se operativni sistemi u realnom vremenu (RTOS - real time operating system), koji raspodeljuju računarsku snagu na različite zadatke na određenim vremenskim odeljcima. Ovakvi sistemi se koriste ukoliko postoje rigidni vremenski zahtevi za rad procesora ili protok podataka, vremenska ograničenja moraju biti dobro definisana. Definišu ih krajnji rokovi, odnosno vreme odgovora u kojem izvršavanje mora biti gotovo. RTOS bi trebalo, da u što većoj meri “sakrije” hardversku složenost od programera (detaljno poznavanje prekida, A/D konvertera, tajmera..), tj da za pristup hardveru obezbedi standardizovane softverske procedure. Pored toga, koriste se statičke komunikacione strukture (CAN komunikacione matrice, definisane na nivou vozila) koje sa jedne strane čine sisteme bezbednim, otpornijim na upade hakera, a sa druge strane ne podržavaju fleksibilne uslužno orijentisane aplikacije[2]. Kako se CAN (Controller Area Network) komunikacija koristi kao standard za komunikaciju izmedju mikrokontrolera i uređaja unutar vozila, pomenute matrice se prave, da bi opisale tu komunikaciju[3][4]. To je tabela struktura, koja definiše:

- Koja ECU šalje koju poruku pod kojim uslovima i u kom vremenskom intervalu
- Koja ECU prima određenu poruku/signal
- Koji signali se nalaze u poruci, kao i njihova interpretacija, tj kako se heksadecimalne veličine pretvaraju u fizičke ili logičke veličine.
- Koji identifikator i koji prioritet imaju poruke

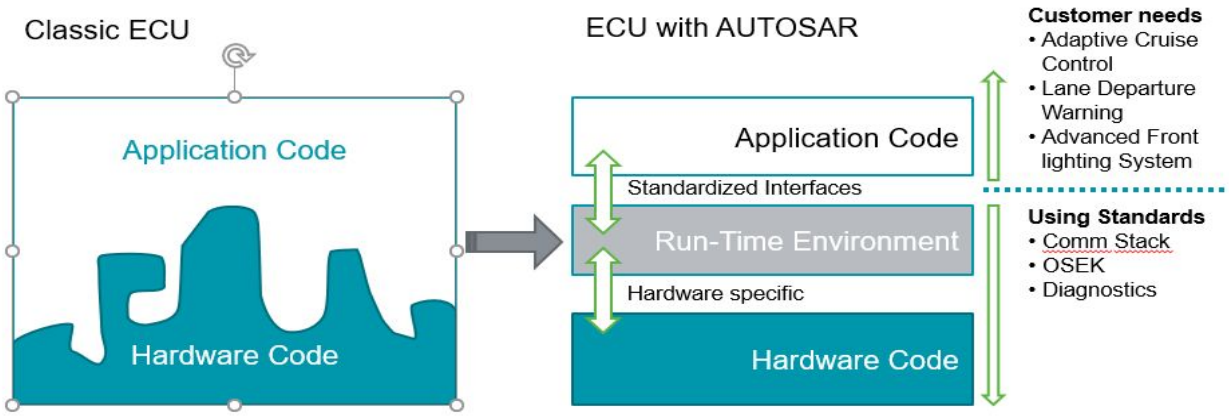
Struktura rada je sledeća - Nakon uvoda i opšte slike u prvom poglavlju, u narednom je opisana softverska arhitektura ugrađenog sistema, AUTOSAR kao projekat usko vezan za automobilsku industriju. Poglavlje 3 se sastoji od pregleda sistemske arhitekture sigurnosno-kritičnog sistema, kako dolazi do izrade takve arhitekture, kao i ukratko koje su odlike najučestalijih u primeni. U poglavlju 4 su opisane osnovne odlike jednog sigurnosno-kritičnog koda, pravila i smernice u izradi, kao i dodir sa standardima u industriji (npr. ISO26262-Funkcionalna sigurnost) koji definišu pravila. U poglavlju 5 je opisano zašto se koristi metoda ubrizgavanja greške u ovakvom jednom sistemu, kao i jednostavni primeri ovog vida testiranja, ručnim unošenjem direktno u kod a i automatskim procesom kroz jedan od alata za tu namenu.

2. Softverska arhitektura embedded sistema

2.1 AUTOSAR

Jedan od najvećih izazova automobilske industrije danas, kao što je spomenuto, jeste da postavi metode i alatke, koji mogu da zadovolje integraciju različitih elektronskih podsistema, koji dolaze od različitih dobavljača u globalnu elektronsku arhitekturu vozila. Nekoliko velikih projekata je nastalo u te svrhe, i do danas, različiti standardi doprinose toj sinhronizaciji. Izrada arhitekture otvorenog softvera, uz povezanost sa razvojnim procesima i alatima, što bi dovelo do lake integracije različitih funkcija elektronskih kontrolnih jedinica, od različitih automobilskih proizvođača kao i dobavljača, dovodi nas do Autosar organizacije[3]. Ova organizacija nastala 2003. godine, koju čine partnerska grupa proizvođača automobila i automobilske opreme, se bavi standardizacijom platforme za razvoj softvera za automobile[5].

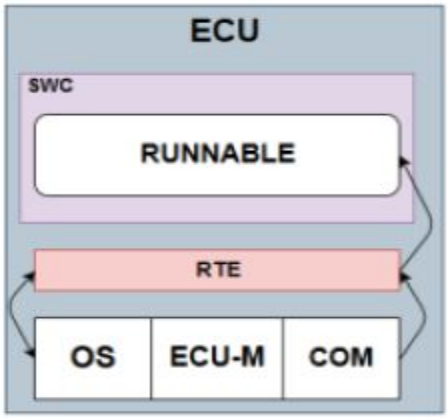
Autosar je osmišljen tako da njegova arhitektura, na najvišem stepenu apstrakcije razdvaja između tri softverska sloja koji rade na mikrokontroleru (kao što je prikazano na slici 3): aplikaciju, RTE (runtime environment), kao i osnovni softver (basic software BSW)[3].



Slika 3 . Prednosti autosara u odnosu na stari model, unar BSW se nalazi "Hardware specific", u sloju ECU apstrakcije, koji se implementira za specifičnu ECU, što znači da je zavistan od hardvera

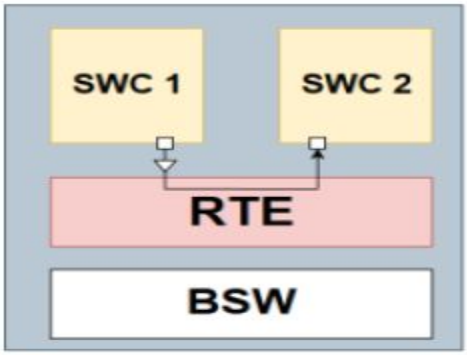
2.1.1 AUTOSAR ARHITEKTURA

Glavni koncept standardizovane ECU softver arhitekture jeste razdvajanje hardver-nezavnisnog aplikacionog softvera, i osnovnog softvera (BSW) koji je orijentisan ka hardveru, uz pomoć softverskog sloja apstrakcije RTE (runtime environment). Sa gornje strane RTE-a, ovaj sloj apstrakcije omogućava razvoj OEM – specifičnih i konkurentnih softverskih aplikacija, a sa donje strane RTE-a, omogućava se standardizacija i OEM nezavisnost od osnovnog softvera (slika 6)[1]. Uloga RTE-a je da realizuje komunikaciju između softverskih komponenti i osnovnog softvera. Softverske komponente komuniciraju sa drugim komponentama i/ili modulima osnovnog softvera isključivo preko RTE sprege, što je upravo ono što omogućava da softverske komponente budu međusobno nezavisne, ali i nezavisne od pojedinačnih ECU jedinica. RTE sprega predstavlja jednu ECU jedinicu i generiše se posebno za svaku konfiguraciju ECU jedinice. RTE predstavlja apstrakciju operativnog sistema, servisa za komunikaciju, hardverske sprege i planiranje rada softverskih komponenti, što je i prikazano na slikama 4,5 [5].

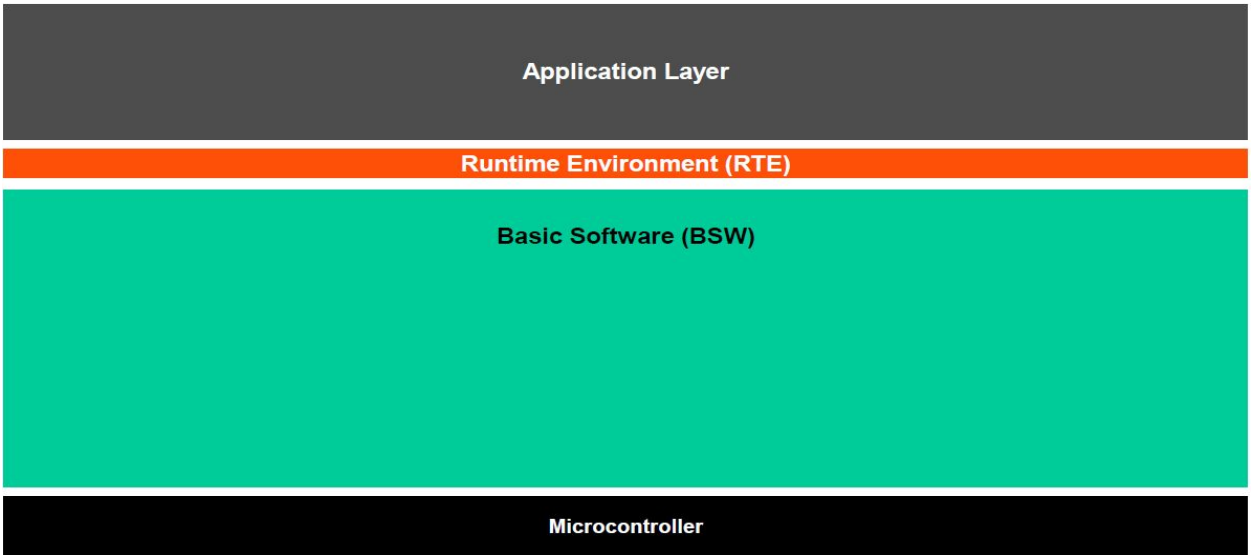


Slika 4: RTE za runnable komponentu

Runnable komponenta je niz sekvenci instrukcija (koje obezbeđuje komponenta) koje RTE okruženje tokom izvođenja može pokrenuti. Izvodi se u kontekstu zadatka. Aktivni je deo softverskih komponenti, mogu se izvršavati istovremeno, mapiranjem u različite zadatke.



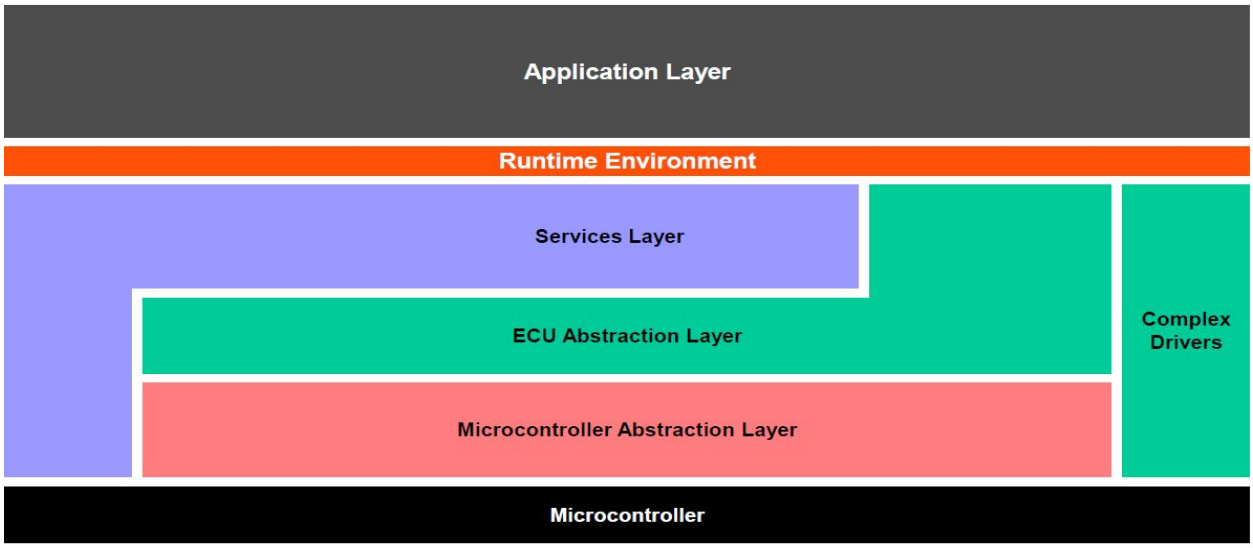
Slika 5: RTE kao komunikaciona sprega



Slika 6 : Autosar osnovna podela

Osnovni softver (BSW), u arhitekturi AUTOSAR, je dalje podeljen na sledeće slojeve (slika 7):

- Servisi, ECU apstrakcija i apstrakcija mikrokontrolera



Slika 7 : Arhitektura osnovnog softvera

Sloj apstrakcije mikrokontrolera (*Microcontroller Abstraction layer - MCAL*) - Najniži sloj osnovnog softvera (slika 8). On je zavistan od mikrokontrolera i sadrži drajvere koji omogućavaju pristup perifernim uređajima na čipu mikrokontrolera i perifernim uređajima preslikanim memorijom izvan čipa pomoću definisanog API-ja. Svrha ovog sloja je u suštini da viši softverski slojevi budu nezavisni od hardvera. Unutar ovog sloja, nalaze se:

- Komunikacioni drajveri (CAN, Eternet, FlexRay, Lin, TTCAN, SPI Handle Driver)
- I/O drajveri (ADC, DIO, ICU, PWM, Port drajveri)
- Memorijski drajveri (EEPROM, Flash, RAM)
- Drajveri mikrokontrolera (Core Test, GPT, MCU, Watchdog)

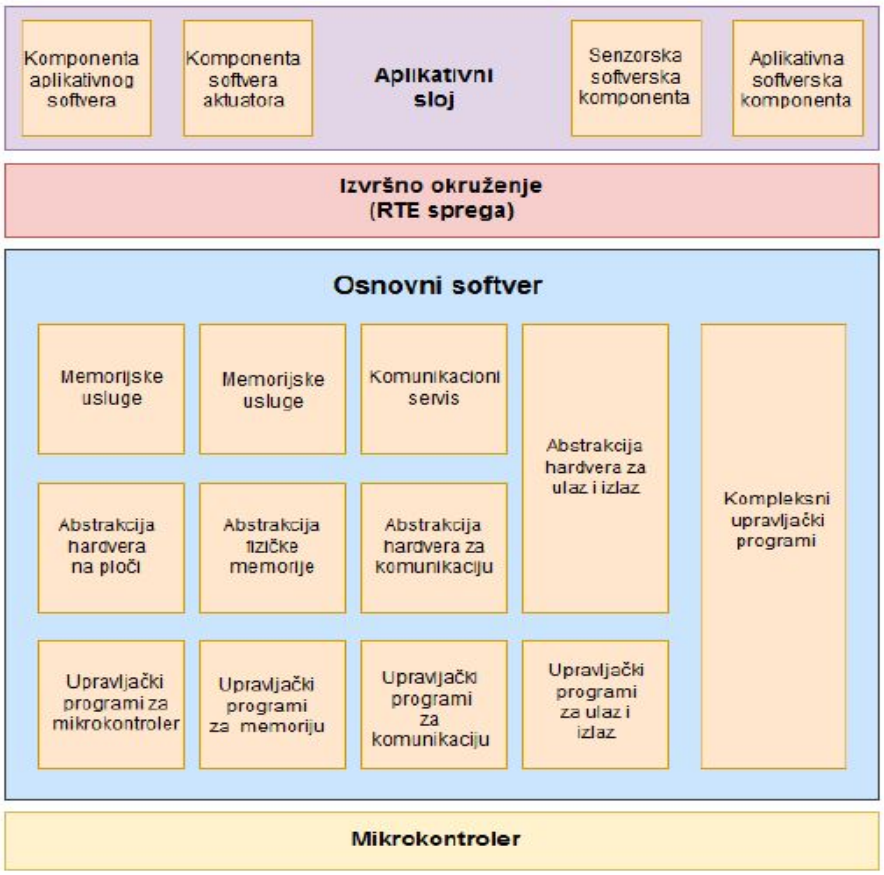
Sloj apstrakcije ECU nalazi se iznad sloja apstrakcije mikrokontrolera MCAL, primenjuje se za određenu ECU (zavistan je od hardvera), i nudi API za pristup perifernim uređajima bez obzira na njihovu lokaciju (na čipu, van čipa) i njihovu vezu sa mikrokontrolerom (pinovi porta, tip interfejsa), kako bi se napravili viši slojevi softvera, nezavisno od hardverskog rasporeda ECU-a. Ovaj sloj se sastoji od:

- Komunikacija hardverske apstrakcije (CanIf, CanTrcv, ErhIf, EthTrcv, FrIf, FrTrcv, LinIf, LinTrcv, TTCAN Interface)
- Memorijska hardverska apstrakcija (MemIf, Fee, EA)

- Watchdog interfejs
- IO hardverska apstrakcija

Najviši sloj je Servisni sloj, koji pruža osnovne servise za aplikativne i module osnovnog softvera (računajući i magistralnu komunikaciju za aplikativni sloj), dok je pristup I/O signalima pokriven slojem apstrakcije mikrokontrolera, ovaj sloj nudi:

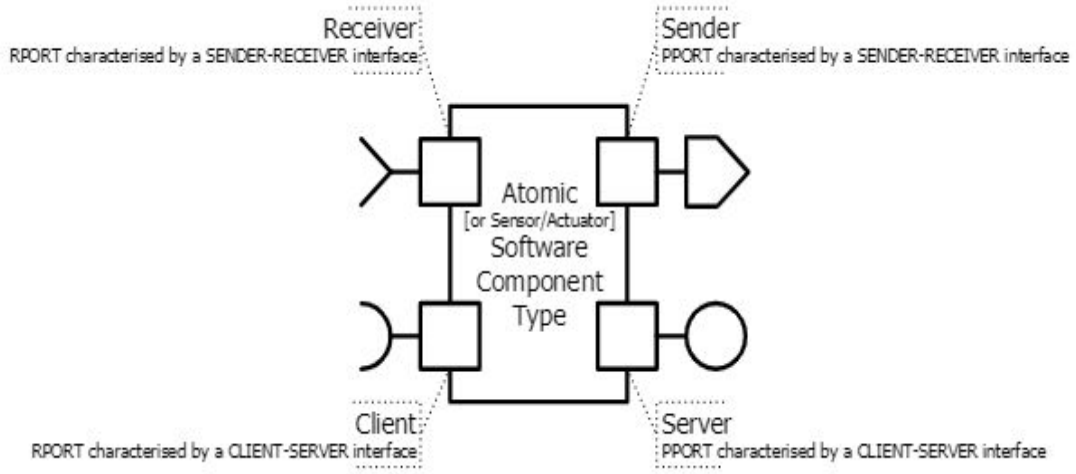
- Usluge operativnog sistema
- Usluge komunikacije i upravljanja mrežnom komunikacijom u vozilu
- Upravljanje memorijom - NVRAM
- Dijagnostičke usluge
- Upravljanje stanjima ECU-a



Slika 8: Podela slojeva osnovnog softvera (Basic Software) na funkcjonalne grupe

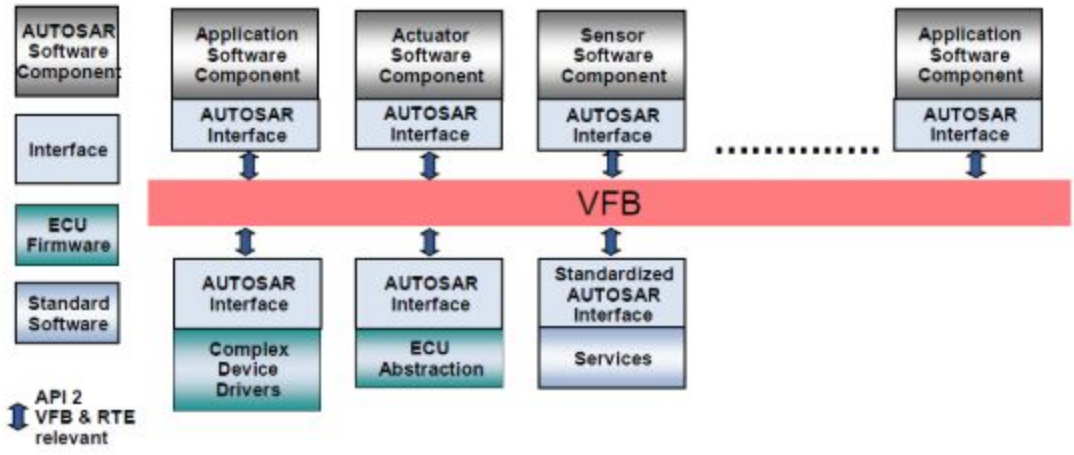
Razdvajanje aplikacionog sloja, od osnovnog softvera, realizovanog od strane RTE-a, uključuje i kontrolu razmene podataka između ovih slojeva.

Jako bitan koncept je virtualna funkcionalna magistrala (Virtual Functional Bus - VFB) koju je odredio AUTOSAR. Ovaj virtualni vod odvaja aplikacije od infrastrukture. Komunikacija se odvija putem namenskih logičkih portova (prikaz jedne softverske komponente sa portovima je na slici 9), što podrazumeva da komunikacioni interfejsi aplikacije moraju da budu pravilno mapirani ka odgovarajućim portovima. VFB upravlja komunikacijom unutar pojedinog ECU-a (slika 10). Bitno je napomenuti i da je VFB implementiran za određeno vozilo isporukom posebno konfigurisanog RTE-a u kombinaciji sa odgovarajuće konfigurisanim osnovnim softverom za svaku pojedinačnu ECU. Iz ugla aplikacije, nije potrebno nikakvo poznavanje tehnologija sa tih nižih nivoa niti zavisi od istih, što je upravo ono što omogućava razvoj bez upotrebe hardvera.



Slika 9: Prikaz softverske komponente sa portovima

Sa jedne strane imamo Portove dobavljanja -PPort-ove (Provided ports), koje koristi softverska komponenta za pružanje podataka ili usluga drugim softverskim komponentama, dok prijemni portovi (Reciever ports) služi da zatraži podatke ili usluge od druge sf. komponente. Ovi portovi se koriste za transfer podataka između komponenti.



Slika 10: Uloga virtualne funkcionalne magistrale

3. Sistemska arhitektura sigurnosno-kritičnih sistema

Razumne, pravilno definisane i konstruisane sistemske arhitekture sigurnosno-kritičnih sistema su od neprocenjive važnosti – služe da zaštite ljudske živote, jer upravo ovakvi, kritični sistemi su embedded (ugrađeni) sistemi, u kojima greške, kvarovi i problemi dovode do ozbiljnih povreda, pa i smrti. Koriste se u raznim industrijama, npr u sistemima kontrole leta, u drive-by-wire sistemu u automobilima, kontrolerima nuklearnih reaktora ili čak u mašinama za podršku rada srca i pluća u operacionim salama.

Kvarovi su greške ili problemi koji dovode do neuspeha. Kvarovi koji se javjaju su ponekad jako mali, npr blokirani memorijski bit, neinicijalizovana softverska promenljiva, a nekad mogu da rezultiraju error statusom iz koga nastaje fault, i javlja se neočekivano ponašanje u sistemu. To se manifestuje npr. u pogrešnoj kalkulaciji dobijenih rezultata ili nevažećoj vrednosti za status promenljive. Ovi sistemi nisu uvek dizajnirani tako da pružaju maksimalan vremenski rad. Ukoliko je potrebno, imaju sigurnosne mehanizme koji gase taj sistem ili odgovarajući podsistem (Opisano detaljnije u poglavlju 3.2), u situacijama kad se ljudski život može naći u pitanju. Sistem se dovodi u takozvano “sigurno stanje (safe state)” kako bi se prekinuo niz od fault-a do hazard-a pre nego što uopšte dođe do situacije opasne po život. U poglavlju 3.3 je opisano na koji način se rad može održavati unutar sistema koji ne sadrži “sigurno stanje”, implementacijom dvokanalne arhitekture. U mnogim industrijama gde se koriste ovi sistemi, ovakvo stanje podrazumeva i potpuno zaustavljanje rada sistema i kompletno gašenje. Takodje, postoje i sistemi za koje postoje sigurna stanja rada, a da bi se sistem doveo u ovakvo stanje, mora se sprovesti dug i složen niz aktivnosti. Izbor arhitekture za sigurnosno-kritični sistem ne može da se zasniva na uobičajnoj definiciji zahteva, već na

temeljnoj analizi rizika. U narednom poglavlju (3.1) je opisano kako dolazi do izbora arhitekture i kojim se popularnim tehnikama dubinske analize dolazi do toga (3.1.1, 3.1.2).

3.1 Analize opasnosti i rizika sigurnosno-kritičnih sistema

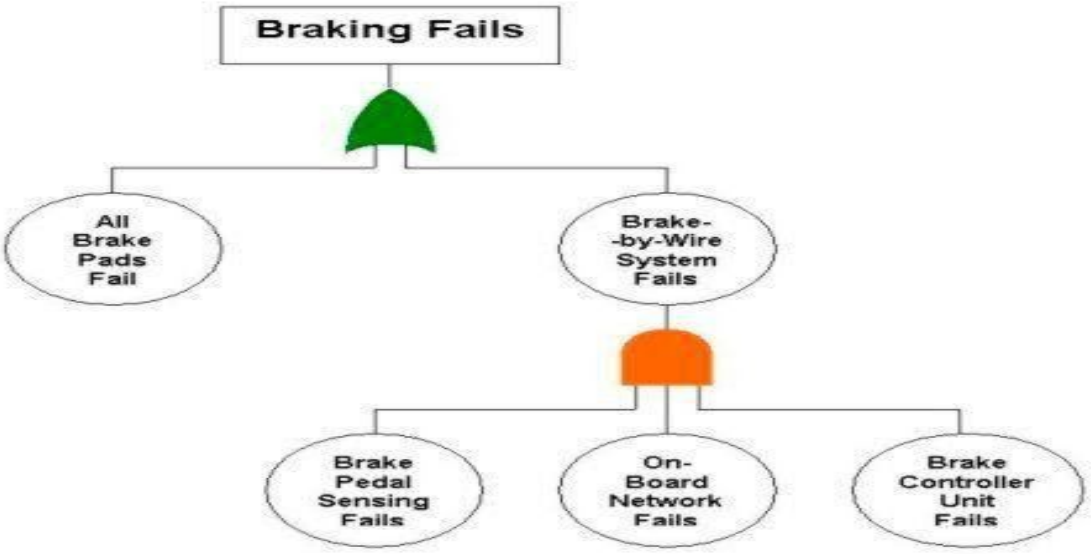
Kod svih embedded (ugrađenih) sistema, izradi dizajna prethodi definicija sistemskih zahteva, sa fizičkom i funkcionalnom specifikacijom. Sigurnosno-kritični sistemi takođe zahtevaju sveobuhvatnu analizu opasnosti i rizika, nakon koje sledi dizajn arhitekture. Prema ISO 26262, hardver-softver interfejs je potrebno specifirati tokom faze razvoja proizvoda na sistemskom nivou. Kao preduslov za spcifični hardversko-softverski interfejs, mora se uspostaviti sistemski dizajn [6].

Kako je sistem skupa komponenti u definisanoj arhitekturi, koja ima jedinu svrhu obavljanja funkcija tog sistema, verovatnoća kvara tih funkcija određuje se integritetom sastavnih komponenti, kao i logikom arhitekture sistema. Što je sistem složeniji, to je veća potreba za tehnikama dubinske analize kako bi se identifikovale sve moguće kombinacije otkaza, koji bi mogli rezultirati gubitkom integriteta sistema [7]. Fault tree analysis (analiza stabla kvarova) i event tree analysis (analiza stable događaja) su samo dva primera tehnika dubinske analize. Glavne funkcije ovakvih analiza su [5]:

- Pomoć da se razjasni logika i redosled događaja koji dovode do opasnosti
- Zabeležiti i pratiti frekventnost neželjenih događaja i posledica
- Koriste se u proceni izrade zaštitnih slojeva

3.1.1 Analiza stabla kvarova (Fault tree analysis):

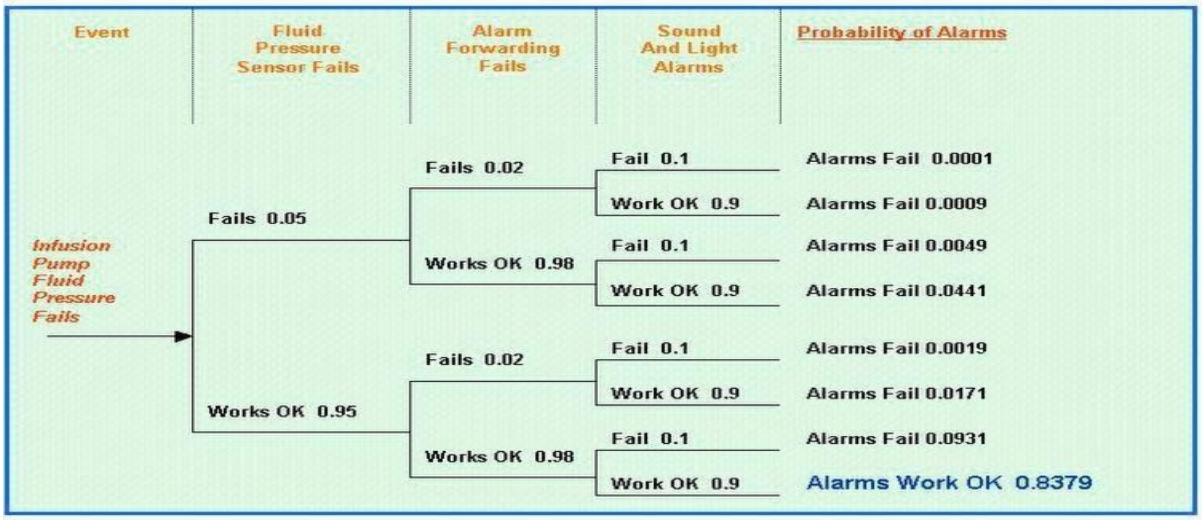
Analiza stabla kvarova (slika 11) je uobičajna i jako korišćenja metoda analize rizika. Zasniva se na hijerarhijskoj dekompoziciji odozgo na dole. Kako probleme izazivaju nepoželjni sistemski događaji, to nam omogućava identifikaciju kombinacija hardvera, softvera, operativnih i drugih greški koje dovode do bezbednosnih rizika. Analiza stabla greške se započinje pitanjima, ”Koje su stvari kojima moj sistem može da ugrozi ljudski život?” Svaki sigurnosni rizik koji se pojavi, unese se u zasebno stablo grešaka. Zatim se postavlja pitanje, „Šta uzrokuje ovakve stvari?” Odgovori na ova pitanja se pojavljuju na samom vrhu stabla iz kojih kreće grananje. Konstantnim postavljanjem pitanja i odgovora, dolazi se do sledećeg nivoa grešaka. Kako bi se prikazale logičke veze u dijagramima, koriste se AND/OR operatori.



Slika 11: Analiza stabla kvarova

3.1.2 Analiza stabla događaja (Event_tree_analysis)

Analiza stabla događaja – pristup odozdo prema gore, koji podrazumeva ispitivanje rezultata rada ili kvara komponenata i podsistema jednog sistema. Često se prikazuje horizontalno (slika 12). Na slici je prikazan primer ovakvog sistema na kome je izvršena analiza sledeće situacije: ”Ako pritisak tečnosti u infuzijskoj pumpi ne uspe, da li sistem izdaje potrebno upozorenje?”. Komponente i podsistemi su prikazani horizontalno na vrhu. Verovatnoće uspešnog rada ili kvara se unose za sve uključene komponente i podsisteme.



Slika 12: Analiza stabla događaja

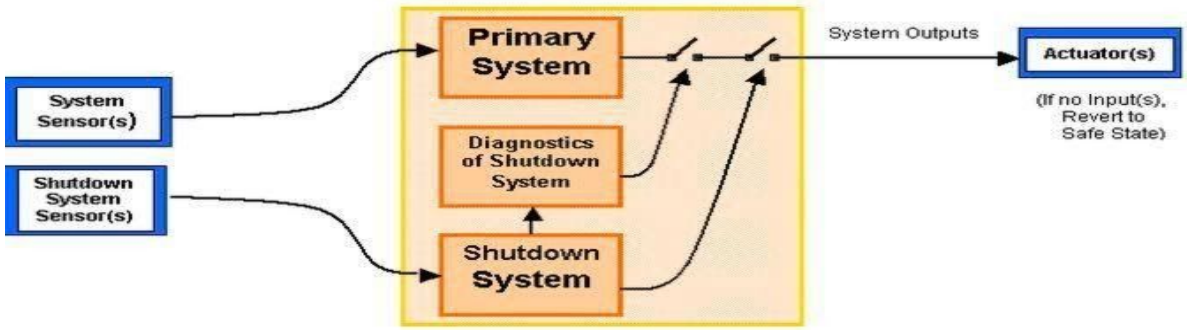
Rizik se posmatra kao kombinacija verovatnoće i očekivanih posledica negativnog događaja. To se može kvantifikovati npr. kao "smrt u 100 godina rada sistema". Nakon sto su identifikovani najveći rizici koje sistem može predstavljati, važno ih je uzeti u obzir prilikom dizajniranja sistema, osnovne opasnosti treba izbeći ili otkloniti u najvećoj mogućoj meri. To se često može sprovesti merama:

- Hardver namenski izbegava određene upitne softverske komponente (Ukoliko dobijeni softver nije pouzdan i postoje delovi koji nisu testirani pod određenim okolnostima, izvršavanje takvih komponenti bi se trebalo izbeći ukoliko je to moguće)
- Blokade za sprečavanje sistema da uđe u rizično stanje i sprečavanje određenih događaja kako bi se izbegle opasnosti (Predefinisane blokade se postavljaju kako se određeni splet okolnosti ne bi desio, što bi dovelo npr. da sistem radi pod velikim opterećenjem)

Ako se opasnosti ne mogu u potpunosti izbeći ili otkloniti, rizik od nesreća mora biti smanjen. Ako ipak dođe do nesreće, rizik od ugrožavanja života mora biti sveden na minimum. Zajedno sa sistemskim zahtevima, rezultati analize opasnosti i rizika pružaju smernice za dizajn arhitekture sistema.

3.2 Sistemi gašenja (Shutdown systems)

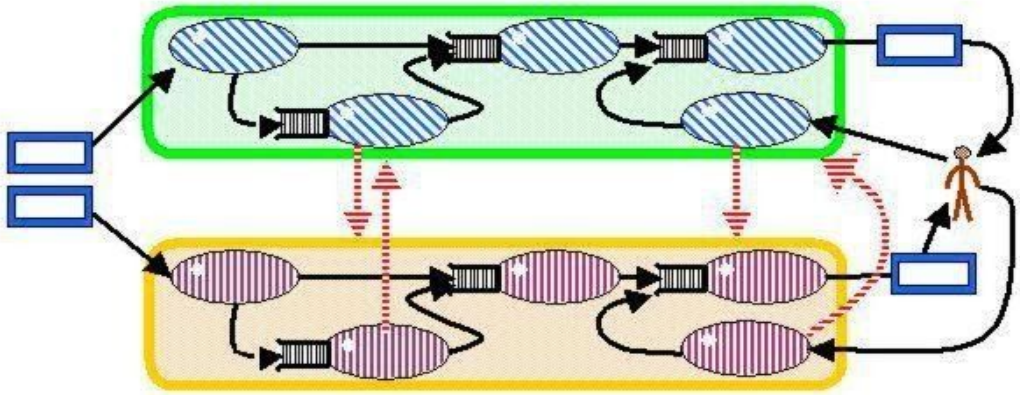
Sistemi gašenja (slika 13) su posebne jedinice za prepoznavanje opasnih situacija. Čim se otkrije opasnost, ceo sistem se dovodi u sigurno stanje (npr. ugašeno), u kome se ne ugrožava ljudski život. Sistem za gašenje je nezavisan od primarnog sistema, koji obično ima kontrolu i radi paralelno i poseduje sopstvene senzore. Integritet rada sistema za gašenje je osiguran preko dijagnostičkog podsistema. Ako dijagnostički podsistem otkrije da odluke sistema za gašenje mogu biti nepouzdane, on može odmah staviti ceo sistem u sigurno stanje (gašenje), koje sprečava primarni sistem da nastavi sa radom ukoliko ne postoji pouzdan nadzor paralenog sistema za gašenje. Mnogi veliki i kompleksni sistemi imaju veliku redundantnost, međutim ne postoji redundantnost za sistem za gašenje. Zbog ovoga, mnogi sigurnosno-kritični sistemi imaju dva sistema za gašenje koji rade paralelno (sa AND/OR logikom koja odlučuje kada isključiti primarni sistem). Postoje čak i sistemi sa tri sistema za isključivanje koji rade paralelno, i ukoliko je donešena odluka da se jedan sistem ugasi, ostali nastavljaju da rade paralelno.



Slika 13: Sistemi gašenja

3.3 Dvokanalna arhitektura

U sigurnosno-kritičnim sistemima bez sigurnog stanja (safe state), rad sistema se može održavati dvokanalnom arhitekturom, čak i ako se jedan kanal zaustavi zbog kvara. Slika 14 prikazuje izgled arhitekture, gde jedan kanal preuzima rad sistema ukoliko se pojavi greška u drugom kanalu.

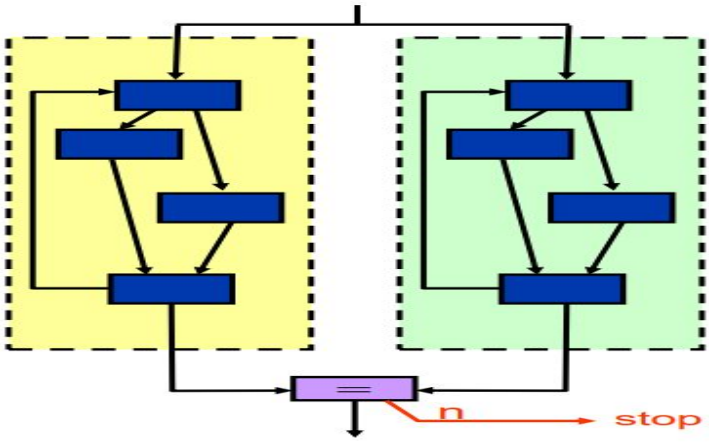


Slika 14: Dvokanalna arhitektura

Ovakva arhitektura uzrokuje daleko veće troškove sistema u odnosu na prethodne arhitekture, sadrži redudantne embedded procesne kanale sa redudantnim hardverom i senzorima. Glavna prednost ovakvog sistema jeste da rad sistema moze u potpunosti da bude neometan iako se javi kvar. U narednim poglavljima ćemo opisati glavne odlike dve jako primenljive dvokanalne arhitekture, sa komparacijom (poglavlje 3.3.1), kao i sa sistemom za nadgledanje (poglavlje 3.3.2).

3.3.1 Dvokanalna arhitektura sa komparacijom

Kod ovakve arhitekture, imamo deljeni ulaz u kanale, a na izlazu se nalazi kritična komponenta koja radi komparaciju izlaznih podataka. Ukoliko postoji razlika i ne podudaraju se, sistem se zaustavlja. Velike mane kod ovakvih sistema, su to što je vremenski interval jako dugačak, sto ne ide u prilog ovakvim sistemima, u kojima je vremenski interval izvršavanja radnje uglavnom jako kratak i uvek strogo definisan.



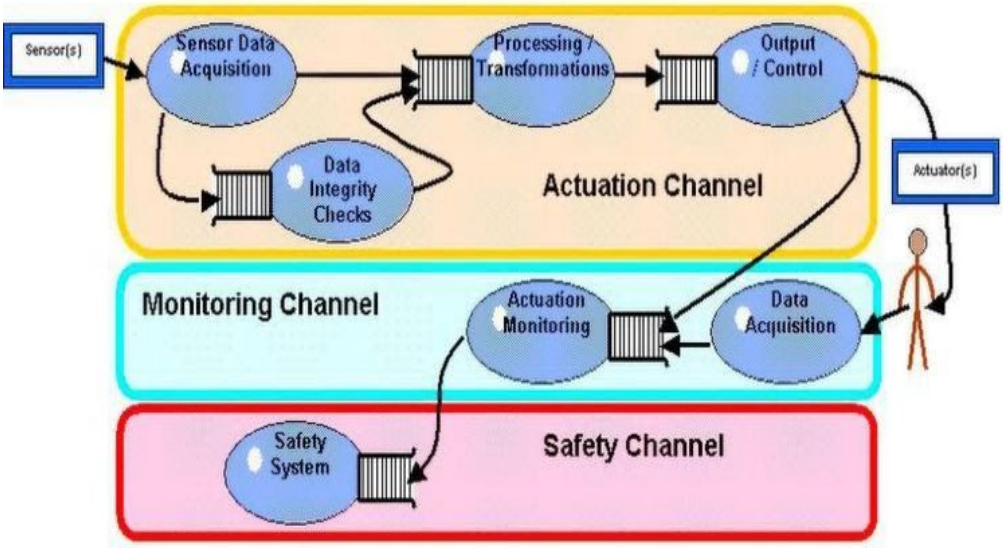
Slika 15 : Dvokanalna arhitektura sa komparacijom

3.3.2 Dvokanalna arhitektura sistemom za nadgledanje

Kod ovakve arhitekture, imamo 2 kanala od kojih je jedan nezavisan. Postoje primarni i sekundarni kanal, i stalno se vrši provera izlaznih podataka sa primarnog kanala.

Ukoliko 2 kanala koriste isti replicirani softver i hardver, arhitektura može dobro da podnese slučajne greške, ali ne i sistematske iz dizajna ili programiranja samog softvera. Oni bi se reprodukovali na oba kanala. Kako bi se izbegle ovakve situacije gde je to potrebno, razvija se heterogena dvokanalna arhitektura koja se sastoji od 2 kanala koja su izvedena potpuno drugačije. Primera radi, softver za dva kanala može poticati od dva potpuno odvojena razvojna tima i biti zasnovan na istim softvertskim specifikacijama. Druga varijanta je arhitektura donošenja odluka ili glasanja sa više kanala. U ovakvoj arhitekturi postoje tri ili više kanala koji paralelno rade. Takozvani “Birač”, upoređuje vrednosti sa ovih kanala, ukoliko se rezultati poklope, prosleđuju se dalje u sistem, a ukoliko nekoliko kanala nema iste vrednosti, zaustavljaju se. Primer ovakve arhitekture može se naći u vazduhoplovnoj industriji. Postoje četiri nezavisna kanala za obradu i raspoređena su kao dva para sa po dva kanala. Jedan par je aktivan istovremeno, a njegova 2 kanala kontinuirano upoređuju rezultate. Sve dok se kanali podudaraju, par ostaje aktivan, a u suprotnom odmah prenose kontrolu na drugi par koji postaje aktivan.

Obzirom da su rešenja dvokanalane arhitekture uglavnom jako skupa, bitno je navesti da postoji i arhitektura u kojoj ne postoje replicirani, identični kanali, već prilično heterogeni kanali koji se međusobno potpuno razlikuju. Postoji samo jedan primarni kanal koji upravlja sistemom (na slici 16 - Actutation kanal). Rad i rezultat ovog kanala su ispitani posebnim jednostavnijim kanalom za nadgledanje (Monitoring kanal na slici). Ako kanal za nadgledanje otkrije grešku u kanalu pokretača, on ne može nastaviti normalno da radi i kontrola sistema se prenosi na poseban bezbednosni kanal (Safety kanal na slici). Ovo mora osigurati da se sistem dovede u sigurno stanje.



Slika 16. Arhitektura sa sistemom za nadgledanje

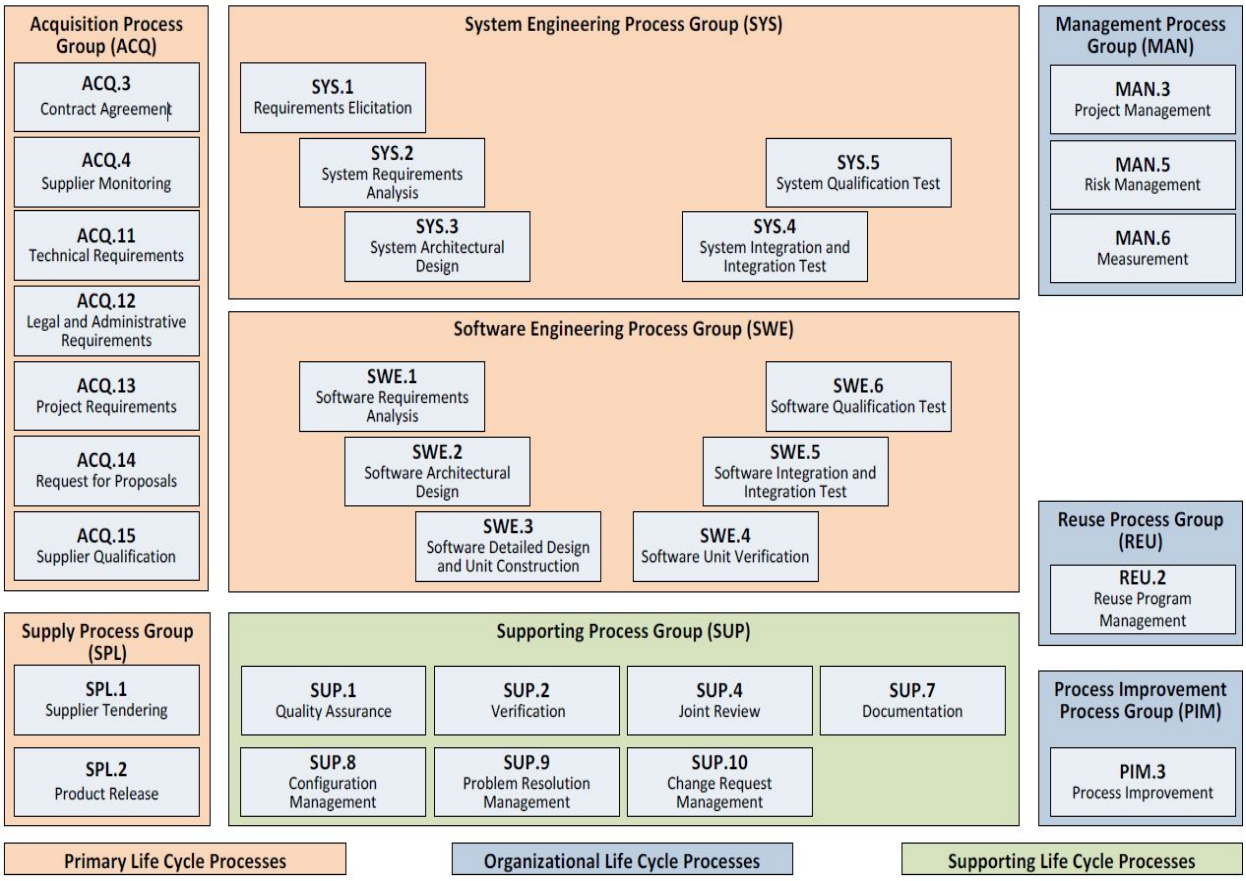
4. Sigurnosno-kritični kod

Kako hardverske komponente u ugrađenim sistemima mogu samo da transmituju, čuvaju i izvršavaju "mašinski kod" - jednostavni jezik koji se sastoji od 0 i 1, vremenom su se programski jezici (npr. C, C++) razvijali kao "high level" jezici, koji daleko više liče na ljudske jezike i nezavisni su od hardvera, za razliku od asemblerskih jezika koji su "low level" i oni su zavisni od hardvera, a to znači da postoji jedinsveni set instrukcija za procesore sa različitim arhitekturama [8].

U mnogim projektima i kompanijama, pri izradi koda, koriste se jasno definisane smernice za kodiranje (Coding guideline). Njihova namena jeste da definišu osnovna pravila o tome kako se softver piše, kako treba da bude struktuiran, koji ježičke karakteristike mogu, a koje ne, da se koriste. Provere ispravnosti koda, se uglavnom vrše preko alata (npr Astree), a u retkim slučajevima i ručno se prolazi kroz kod, ukoliko to obim dozvoljava.

Sigurnosno-kritični kod obično karakterišu raznolikost, samo-testabilno (self-test) svojstvo i konstantne provere ispravnosti (Code Review). Sve te mere i opreznosti su tu da bi sistem zadovoljio najveći izazov, da bude siguran, ali sve to donosi velike izazove za test inženjera, koji je od početka uključen u razvoj koda. Standardi za izradu softvera, kao sto su ISO 26262 (automotive industrija), EN/IEC 61508 (Funkcionalna sigurnost bezbednosno kritičnih sistema), EN 50128 (SF za električne sisteme u železnicama), IEC 62304 (medicina), zahtevaju veliki nivo sigurnosti, dokaze o sigurnosti kao sto je 100% pokrivenost koda (statement coverage), kroz dokumentovane rezultate testova.

Proces razvoja sigurnosno-kritičnih sistema optimizovan i za razvoj hardvera, gde odluke o dizajnu moraju rano da se donesu, kako bi se izbegle kasne promene u koje donose jako velike troškove. Da bi se ovakav razvoj ispratio uspešno od početka, koristi se V-model koji prati dizajn, implementaciju, testiranje i validaciju. U automobilskoj industriji koristi se standard - Automotive SPICE (A-SPICE) [11]. On predstavlja standard koji se koristi kao frejmvork za unapređenje i evaluaciju procesa. Primenjuje se u razvoju elektronskih sistema sa fokusom na softver i sistemske delove proizvoda. Bitno je napomenuti da ASPICE predstavlja samo model i malu kolekciju procesa koji mogu direktno da se izvrše. Samo prikazuje šta je potrebno da se uradi i pokrije, a ne i na koji način ti standardi treba da se izvedu. To je do svake kompanije zasebno da odluči. Na slici 17. je prikzan V-Model unutar Automotive Spice modela [10].



Slika 17: V-Model Automotive Spice

Izbor jezika za sigurnosno-kritični (safety-critical) kod je ključan za razmatranje. U mnogim organizacijama, kod se pise u jeziku C. Sa svojom dugom istorijom, postoji velika podrška alata za ovaj jezik, snažni analizatori izvornog koda, razni logički modeli, alati za metriku, alati za debug, podrška za test, kao i izbor stabilnih kompajlera. Iz ovih razloga, C je meta većine postojećih smernica kodiranja, koje su tu da optimizuju i detaljno provere pouzdanost kritičnih aplikacija pisanih u C-u[11].

U nastavku su navedeni neki primeri osnovnih standardizovanih smernica, koje dovode do izrade koda kog je lakše analizirati i testirati.

4.1 Primeri osnovnih smernica(pravila)

Osnovna pravila pri izradi sigurnosno-kritičnog koda
1. Izbegavati kompleksne konstrukcije toka, ne koriste se "GoTo" i rekurzije.
2. Sve petlje moraju da imaju gornju granicu
3. Ne koristi se dinamičko alociranje memorije
4. Dužina funkcije ne sme da bude veća od jedne stranice sa štampu
5. Deklarisati objekte podatka u što manjem obimu
6. Proveriti vraćenu vrednost non void funkcija
7. Predprocesori mogu da se koriste u zaglavljima i malim makro definicijama
8. Ograničena upotreba pokazivača
9. Kompajliranje je bitno od prvog dana razvoja

Slika 18: Grafički prikaz skupa smernica i pravila

1. Ograničiti kod na vrlo jednostavne konstrukcije kontrolnog toka, ne koriste se „goto”, „setjmp”, „longjmp”, kao ni direktne ili indirektne rekuzije. Jednostavniji kontrolni tok daje veće mogućnosti za analizu i rezultati su pobiljšanje jasnoće koda.
2. Svim petljama postaviti gornju granicu. Potrebno je da bude moguće za alatke provere, da statistički dokazu da petlja ne može da pređe postavljenu gornju granicu u broju iteracija. Ukoliko to nije moguće, pravilo se smatra nevažećim.
3. Ne koristi se dinamička alokacija memorije nakon inicializacije. Memorijski alokatori, kao što je “malloc”, često dovode do problema kao čto su: neoslobađanje memorije, nastavljanje da se koristi memorija koja je prethodno oslobođenja, dodeljivanje memorije više nego što je fizički dostupno, prekoračenje granica dodeljenje memorije i slično.
4. Funkcije ne smeju da budu dugačke, maksimalna dužina ne bi trebalo da bude više od onoga što može da se odštampa na jednom standardnom listu papira, obično ovo znači da kod ne bi trebalo da ima više od 60ak linija.
5. Svi objekti podatka treba da se deklarišu u najmanjem mogućem obimu.
6. Svaka funkcija koja se poziva, mora da proveri povratnu vrednost ne-void funkcije, i svaka pozvana f-ja mora da proveri validnost svih parametara prosleđenih od pozivača.
7. Upotreba predprocesora može biti ograničena na zaglavlja i jednostavne makro definicije. Deljenje tokena, argumenti promenljivih lista (elipse) i rekurzivni makro pozivi nisu dozvoljeni.

8. Upotreba pokazivača mora biti ograničena. Ne bi trebalo da se koristi više od jednog nivoa preusmerenja. Funkcijski pokazivači nisu dozvoljeni.
9. Kompletan kod mora da se kompajlira od prvog dana razvoja, sa svim omogućenim prikazima upozorenja kompajlera, jer bi kod trebalo da bude bez upozorenja. Kod bi trebalo proveravati svakodnevno, sa najmanje jednom statičkom kod analizom, koja bi trebalo da prođe bez upozorenja.

Sva ova pravila zajedno, donekle garantuju stvaranje jasne i transparentne kontrole strukture, koju je lakše graditi, testirati i analizirati.

5. Jednostavna automatizacija ubrizgavanja greške (fault injection)

HIL (Hardware in loop) testiranje, je metodologija ispitivanja koja se koristi za sistemsko testiranje softvera. Test gde se ugrađeni softver koji se testira izvršava na istoj upravljačkoj jedinici (ECU, mikokontroleru) na kojoj će pokretati u svom predviđenom okruženju, povezan na iste fizičke interfejse sa kojima će imati iteraciju. Signali na ovim interfejsima vode samo do i iz sistema za testiranje. Testni sistem oponaša upravo ostatak okruženja, i sve komponente koje bi uticale na signale, simulirane su softverom u ovakvom test sistemu, koji je često kreiran u jezicima modeliranja, npr. Simulink. U sistemskom, ili HIL testiranju, jako je lako generisati situaciju greške, jer se eksterni interfejsi mogu lako kontrolisati simuliranjem test sistema i situacije sa greškom se mogu reprodukovati. Iako ovo povećava pokrivenost instrukcija koda, i dalje ne zadovoljava traženih 100% pokrivenosti, jer se time ne može uticati na interne bezbednosne mere unutar softvera. Takođe, problem se javlja i kod tehnike koja se koristi za merenje pokrivenosti instrukcija. Merenje se radi instrumentacijom izvornog koda, sto može da utiče na dinamičko ponašanje sistema [12][13]. U ovom poglavlju biće opisano na koji način se koristi metoda ubrizgavanja greške i kako to utiče na finalnu pokrivenost instrukcija koda (poglavlje 5.1), kao i primer automatizacije ubrizgavanja greške uz pomoć alata Tesi (poglavlje 5.2).

5.1 Zasto se koristi metoda ubrizgavanja greške (Fault Injection) u testiranju?

U praksi, pokrivenost instrukcije se uglavnom određuje u fazi testiranja funkcija (Unit Testing), jer se tom prilikom mogu testirati interne mere bezbednosti u softveru. Korišćenjem standardizovanih metoda merenja, postiže se visok nivo kvaliteta testiranja i samim tim velika stopa pokrivenosti koda testirane funkcije. Međutim, kod izuzeteno kritično-sigurnosnog koda, uvek će postojati nekoliko procenata do potpune pokrivenosti. Sigurnosne funkcije poput primene odbrambenih programskih tehnika, mogu se testirati samo ukoliko je uspostavljena odgovarajuća situacija sa greškom. Ukoliko se testiranje vrši na mikrokontroleru, hardver se ne može staviti u bilo koje neophodno “neispravno” stanje, jer se registri periferijalnih jedinica mogu podesiti samo u skladu sa njihovom hardverskom implementacijom. Kao rezultat, u softveru nisu dostignuta sva stanja grešaka. Kada se koristi simulator, u zavisnosti od obima istog, periferne jedinice su simulirane i javljaju se iste poteškoće kao kod testa sa mikrokontrolerom [14].

Zbog ovoga se koristi tehnika „Ubrizgavanja greške” (Fault Injection). Prilikom ovakvog testiranja, u aplikaciju se ubacuju greške u izvornom kodu, npr. korišćenje makroa, kako bi se na taj način moglo testirati njihovo ponašanje. Ubrizgavanje greške se vrši u razvoju, sa pogledom na sistem i sistemske zahteve, ne uzimajući u obzir interne bezbednosne mere softvera, tako da

se neputpuna pokrivenost koda dešava samo kad se razvijaju testovi funkcija (Unit testovi). Test inženjer prepoznaje ovakvu situaciju i implementira potrebne greške da bi se postigla maksimalna pokrivenost [3][15].

Ručno ubrizgavanje greške putem makroa ili slično, ima svoje nedostatke jer u tom slučaju manualno se upravlja kodom i instrumentacija ostaje u produkcionom izvornom kodu. Ovakav slučaj uglavnom nije poželjan za kritične zahteve koji se tiču bezbednosti. Ovakvi slučajevi mogu da se prevaziđu i na druge načine, npr. upotrebom GIT-a (sistema za praćenje promena u kodu tokom razvoja softvera, to je tip sistema za verzionisanje (version-control system)) i posebnih brančeva koji se koriste za svrhe testiranja. Budući da u procesu razvoja kompatibilnog sa standardima, promena izvornog koda treba proći kroz sve instance ponovo, tj. promene se puštaju, prenose kroz kontrolni sistem i radi se ponovna analiza koda.

Upravo zbog ovakvih situacija, za profesionalne alate testiranja funkija, ubrizgavanje greške je implentirano da se vrši bez promena izvornog koda. Merenje pokrivenosti koda se takođe vrši instrumentacijom koda, međutim dinamično za izvršenje testa, i ne ostaje u izvornom kodu.

Isključivanje merenja pokrivenosti instrukcijama, takođe osigurava da instrumentacija nema uticaja na tok funkcije tokom novog testiranja, zbog istog rezultata prilikom izvršavanja oba testa.

U ovom jednostavnom primeru ćemo videti često korišćenu funkcionalnost testa memorije, sa instrumentacijom koda [1].

- Slučaj 1 (slika 19) – linija 6 upisuje podatak u memoriju i linija 7 proverava upisanu vrednost. Mikrokontroleri sa keš memorijom mogu imati dodatni kod izmedju tih linija kako bi se osiguralo da se memoriji može ponovo pristupiti. Budući da se može pretpostaviti da memorija ne sadrži greške, samim tim TRUE put IF ekspresije se ne može dostići.

```
1  extern void errorHandler(void);
2
3  int memTest (volatile int *pMemory, int pattern)
4  {
5      int result = 0;
6      *pMemory = pattern;
7      if (pattern != *pMemory)
8      {
9          errorHandler();
10     } else {
11         result = 1;
12     }
13     return result;
14 }
```

Slika 19 . Primer dela koda, u kome se TRUE put ne može ostvariti

Ubrizgavanje greške se vrši kontrolisano preprocesorskom definicijom FAULTINJECTION (slika 20). Ako postavimo definiciju #ifdef FAULTINJECTION, eror situacija može biti testirana i kod pokriven 100%. Dodatna test promenljiva tstFaulInjection kontroliše da li će situacija sa greškom biti izazvana ili ne. Ukoliko je TRUE put IF ekspresije u primeru ispod na liniji 8, u liniji 10 će se pokazivaču *pMemory, dodeliti inverzna bitska vrednost istog tog pokazivača (~ operator predstavlja bitsku inverziju, 0 će se zameniti u 1 i obrnutuo), tako da će u sledećoj proveri u liniji 13 biti ispunjen TRUE uslov. Kada se generiše binarni kod za finalni proizvod, naša definicija #ifdef FAULTINJECTION neće biti uzeta u obzir prilikom kompajliranja, produkcioni kod će biti čist.

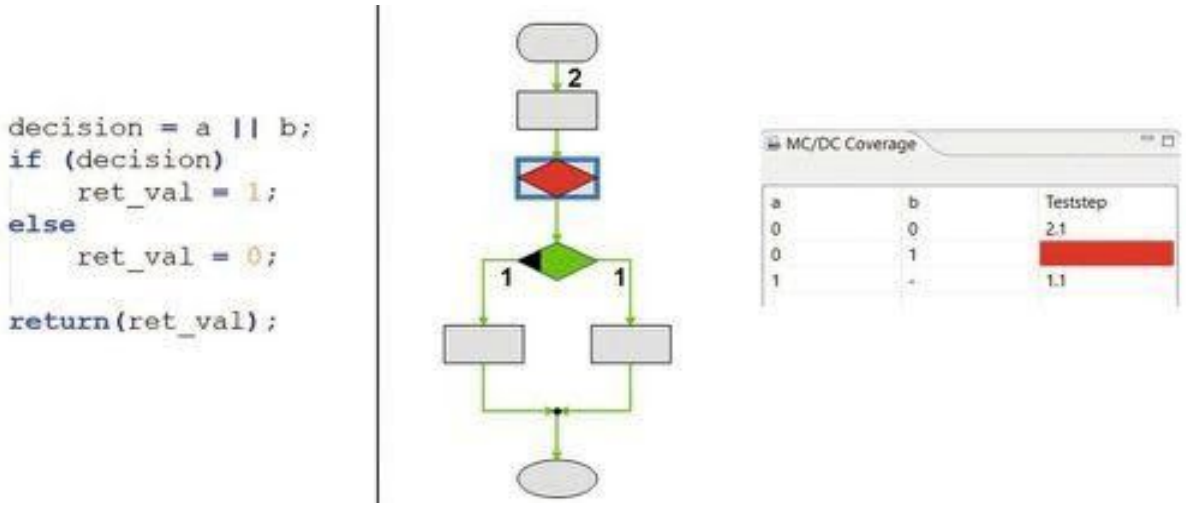
```
6      *pMemory = pattern;
7      #ifdef FAULTINJECTION
8          if (tstFaultInjection != 0)
9              {
10                 *pMemory = ~*pMemory;
11             }
12     #endif
13     if (pattern != *pMemory)
```

Slika 20. Implementirana instrumentacija koda

5.2 Ubrizgavanje greške u alatu Tesi

Ubrizgavanjem grešaka se upravlja test alatima. To automatski orkestrira izvorni kod i dopušta inženjeru da pojedinačno odredi koji se test slučajevi izvode sa ubrizganom greškom a koji ne, kako bi se postigla pokrivenost celog koda i svih uslova. Pošto se instrumentacija može isključiti, obezbeđuje se isti nivo bezbednosti kao i prilikom analize pokrivenosti koristeći rezultate ispitivanja. Svim ubrizganim greškama se upravlja pomoću alata za testiranje, a test slučajevi su posebno obeleženi. Automatizacijom izveštaja se dokumentuju svi rezultati i postavke testova.

Veoma poznat i široko korišćen alat za testiranje u embedded sistemima, jeste Tesi, od kompanije Razorcat. Verzija 4.1 ovog alata sadrži funkciju za upravljanje i primenu automatizovanih ubrizgavanja grešaka. Tesi uzima u obzir položaje ubrizgavanja grešaka prilikom promene izvornog koda, i automatski se postavlja dinamički na odgovarajuće položaje u izvornom kodu u analizi programske strukture. U Tesi zasebnom prozoru, ili perspektivi "Pregled pokrivenosti", prikazuje se dijagram toka funkcije, i putanje programa su označene crvenom i zelenom bojom. Delovi koda koji jos uvek nisu pokriveni su označeni crvenom i lako se identifikuju, takože i čvorovi u kojima su se izvršavale provere (IF) su označeni. Uporedo sa tim, odgovarajući izvorni kod je prikazan, tako da se veoma lako može analizirati gde je došlo do problema i koji deo koda nije pokriven testom.

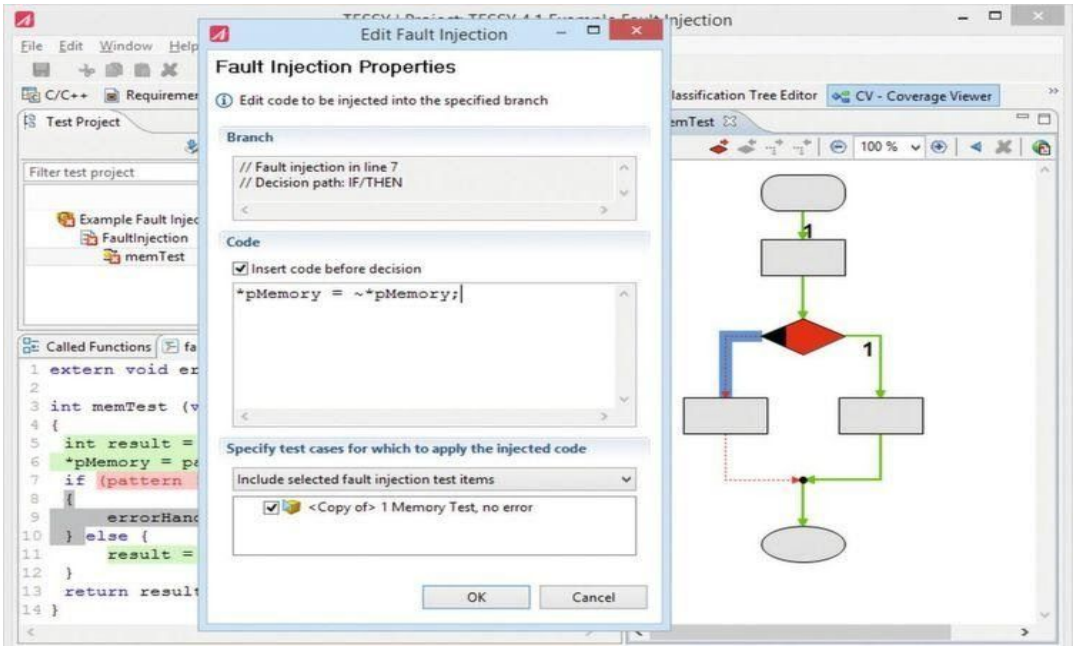


Slika 21. Slikovni prikaz IF instrukcije u alatu Tesi

Ukoliko programski put nije izvršen, test inženjer ima opciju da automatski instrumentizuje kod iz Tesija, prema odgovarajućoj grani dijagrama, kako bi se testirala situacija sa greškom.

Test slučaj za pristup memoriji bez greške je generisan i izveden. Analiza pokrivenosti instrukcijama vrši se u prozoru „CV – Coverage Viewer", i prema našem primeru, možemo da vidimo da se leva strana IF instrukcije nije izvršila (test je jednom prošao desnom stranom). Sa ovim testom, inženjer ima opciju da Tesi automatski uradi instrumentaciju dela koda, prema datim instrukcijama, kao što je prikazano na slici. Ubrizgavanje greške, je u nasem slučaju postavljanje inverzne vrednosti za memoriju, kako bi provera u liniji 7, prepoznala tu situaciju i linija 9 pozvala errorHandler() funkciju.

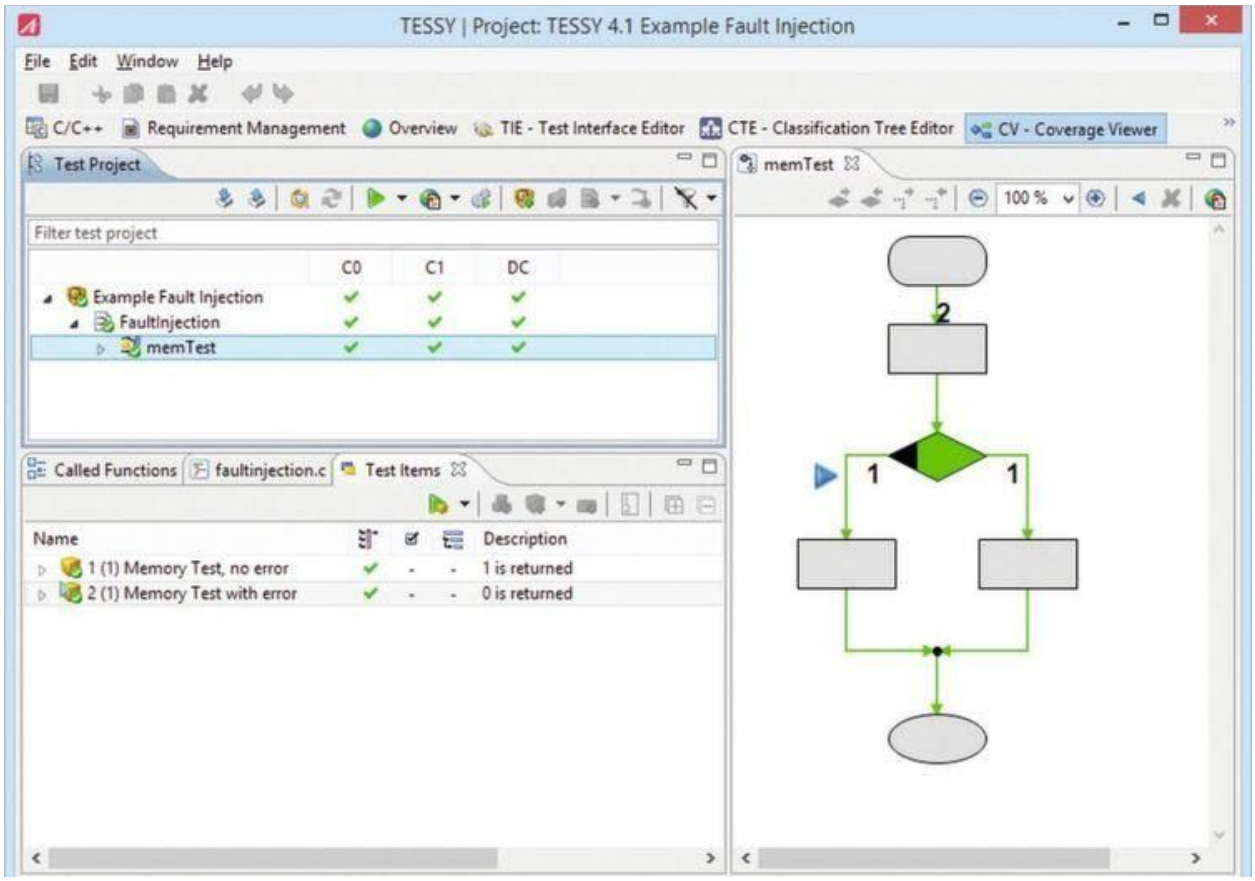
Ovo se izvršava u prozoru „Edit Fault Injection", kao sto je prikazano na slici 22. Tesi analizira koji postojeći i izvršeni test slučajevi pokrivaju problematičnu IF instrukciju, i predlaže da se ti slučajevi koriste za ubrizgavanje greške. Kad unesemo izazivanje greške, Tesi kreira kopiju test slučaja 1, i kreira test slučaj 2. Novi slučaj dobija „Fault Injection" obeležje i prikazan je na poseban način. Nakon što se ime i očekivani rezultat test slučaja 2 podese, test se izvršava, i pokrivenost instrukcijama je 100%, što je i traženo.



Slika 22. Prozor unutar alata predviđen za instrumentaciju

Nakon što je test izveden ponovo, sve putanje su prikazane zelenom bojom na dijagramu unutar „CV – Coverage Viewer" perspective, a putanja sa dodatnom greškom ima plavi trougao, kao i unutar "Test Item" prozora.

Prema prikazanom primeru, razne dodatne situacije sa greškom, kao što su ulazak u „default" slučaj, neke switch naredbe, ili izlazak iz beskonačnog kruga, mogu da se lako reše na ovaj način instrumentacijom. Ukoliko postoji nova verzija izvornog koda, Tesi analizira tu verziju, i automatski postavlja "Fault Injection" na odgovarajuće mesto. Instrumentacija ne ostaje u izvornom kodu i može se podesiti opciono tokom testova. Izveštaji o rezultatima testiranja, pružaju sve dodatne informacje o testovima. Daljnja iteracija kroz razvojni proces nije potrebna, i testovi za funkcije se mogu u potpunosti završiti u jednom radnom procesu kao što je ovaj, i dobijamo 100% pokrivenost instrukcijama u kodu.



Slika 23. Finalni izgled nakon ponovnog pokretanja koda sa instrumentacijom

6. Zaključak

Ovim radom je dotaknut način na koji se automobilska industrija suočava sa eksponencijalnim rastom kompleksnosti ugrađenih (embedded) sistema. Pokazano je zašto, i na koji način je nastala organizacija AUTOSAR, koja je trenutno ključni faktor u celokupnom razvoju. Pored orijentisanosti ka razdvajanju softvera od hardvera, jako bitan akcenat se stavlja i na pravilnu sistemsku arhitekturu, od krucijalne važnosti da sistem bude stabilan jer se u većini slučajeva ova tehnologija može odraziti na ljudske živote.

Kako bi se obezbedio stabilan celokupni rad sistema, potrebno je posebnu pažnju posvetiti podrobnom testiranju na svim nivoima, a ovim radom je napravljen osvrt na jedan od aspekata kojim se ova sigurnost obezbeđuje - metod ubrizgavanje greške. Iako dati primerak predstavlja samo delić ove sveobuhvatne teme, prikazan je postupak izvođenja, kao i problematika koja je pokrivena rešenjem.

Svi ovi procesi, metodologije i pravila kao npr. navedena u poglavlju 4, postoje kako bi se i u budućnosti industrija uspešno suočavala sa preprekama i razvojom tehnologije obezbedila da ljudski životi budu još zaštićeniji a putovanje od tačke A do tačke B što udobnije, brže i sigurnije.

Literatura

[1] Robert Oshana and Mark Kraeling, "*Software engineering for embedded systems*", Newnes 2019, https://books.google.at/books?hl=en&lr=&id=g6aeDwAAQBAJ&oi=fnd&pg=PP1&dq=Software+Engineering+for+Embedded+Systems&ots=8GNw3nQuQm&sig=oZPiUUrsZjznMnvB2ocJQKRtxTg&redir_esc=y#v=onepage&q=Software%20Engineering%20for%20Embedded%20Systems&f=false

[2] Michel D. Ingham, Robert D. Rasmussen, Matthew B. Benner, Alex C. Moncada, "*Engineering Complex Embedded Systems with State Analysis and the Mission Data System*", December 2005

[3] https://www.embedded-software-engineering.de/

[4] Nicolas Navet, Francoise Simonot-Lion, "*Automotive Embedded Systems Handbook*", CRC Press, 2009

[5] Aleksandar Lukić, Dragan Kukolj, Fakultet tehničkih nauka, Univerzitet u Novom Sadu, Velibor Ilić, Istraživačkorazvojni Institut RT-RK, Novi Sad, Srbija, Milena Milošević, Fakultet tehničkih nauka, Univerzitet u Novom Sadu, "*Automatsko generisanje testova za automotive sisteme zasnovane na AUTOSAR model*u", https://www.researchgate.net/publication/333602311_Automatsko_generisanje_testova_za_automotive_sisteme_zasnovane_na_AUTOSAR_modelu

[6] Harald Sporer, Geogr Macher, Christian Kreiner and Eugen Brenner, "*Resilient Interface Design for Safety-Cricital Embedded Automotive Software*", Institute of Technical Informatics, Graz, Austria, https://pdfs.semanticscholar.org/3e5a/607a7b2b97959a634bce69c23b964a3e544a.pdf

[7] Miroslaw Staron, "*Automotive Software Architectures*", Springer International Publishing AG, 2017

[8] Tammy Noergaard, "*Embedded systems architecture, A Comprehensive Guide for Engineers and Programmers*", Second Edition, 2013

[9] Helen Sharp, Tracy Hill, "*Agile Processes in Software Engineering and Extreme Programming*", 17th International Conference, XP 2016, Edinburgh, UK, May 24-27 2016.

[10] VDA QMC Working Group 13 / Automotive SIG, "*Automotive SPICE Process Assessment/ Reference Model, version 3.0l*", VDA QMC Working Group, 2015

[11] Brian W. Kernighan. Dennis M. Ritchie, "*The C Programming Language*", AT&T Laboratories, Murray Hill, New Yersey

[12] Paul D. Marinescu and Geogre Candea, "*Efficient Testing of Recovery Code Using Fault Injection*". Ecole Polytechnique Federale de Lausanne, Switzerland, https://dslab.epfl.ch/pubs/faultInjection.pdf

[13] Feinbube, L., Pirl, L. and Polze, A., *"Software fault injection: A practical perspective"*. In Dependability Engineering. IntechOpen. 2007, https://www.intechopen.com/books/dependability-engineering/software-fault-injection-a-practical-perspective

[14] J. Christmansson, M. Hiller, M. Rimen "*An Experimental Comparison of Fault and Error Injection*", Department of Computer Engineering Chalmers University of Technology, Sweden, http://deeds05.deeds.informatik.tu-darmstadt.de/publications/conf/christmansson_issre98.pdf

[15] Israel Koren, C. Mani Krishna, "*Fault-Tolerant Systems*", 2007, https://www.sciencedirect.com/topics/computer-science/fault-injection