Софийски университет "Св. Климент Охридски"
Факултет по математика и информатика
Катедра "Информационни технологии"

Дипломна работа

ТЕМА:

Автоматизация на тестването за вградени (embedded) системи в автомобилната промишленост

Дипломант: Антония Гюрова Лаврова

Специалност: Софтуерни технологии

Факултетен номер: М21510

Научен ръководител: доц. Силвия Илиева
(Катедра "Информационни технологии", ФМИ, СУ "Св. Климент Охридски")
Консултант: Илина Манова
(Рила Солюшънс)

София, Октомври 2006

Съдържание

Увод

През последните години значимостта на софтуера във всички сфери на индустрията се увеличава все повече. Съвременните софтуерни инженери са изправени пред предизвикателството да разработват софтуер, който трябва да е с изключително високо качество и да отговаря на изискванията на клиентите. Една от целите на осигуряването на качеството на софтуерния продукт, като неделима част от цялостния процес на разработка, е да провери до колко са покрити изискванията към системата.

Все повече технически нововъведения се реализират чрез използване на софтуерни методи. Софтуерът се използва в продукти, които традиционно са предмет на разработване на механиката и електроинженерството. В автомобилната индустрия например все по-голяма част от нововъведенията са базирани на електроника и софтуер, за да се повиши сигурността на автомобила, да се подобри комфорта на пътниците, да се намали консумацията на гориво и отделянето на вредни газове. Увеличаването на софтуерно базираните системи в автомобилите е съпроводено с увеличаване на тяхната сложност (комплексност). До началото на 90те години на миналия век системите в автомобилите е трябвало да изпълняват една определена задача и са били свързани чрез прости директни връзки (point-to-point connections). В последните години започват да се комбинират множество типове мрежи като CAN (Controller Area Network) и MOST (Media Oriented System Transport), които свързват различните системи.

Нарасналият брой на софтуерно-базирани функции значително е повлияло върху качеството на автомобила. В наши дни автомобилите трябва да отговарят на високите критерии към софтуера, за да задоволят клиента и за да изпълняват правните изисквания. Някои от най-важните критерии за качество са надеждност, сигурност и удобство за работа.

Автомобилната индустрия е изправена пред намирането на оптимален баланс между времето и разходите за производство. Компаниите трябва да реализират бързо нововъведенията си и да ги пуснат на пазара, за да могат да са достатъчно конкурентно способни. В същото време, тъй като нововъведенията се реализират софтуерно, разходите за разработване също нарастват. Статистиката показва, че средно 25% от разходите за производство на автомобил се отделят за електроника и софтуер. Тендецията е в близките години този процент да се увеличава. Използването на софтуер предлага много повече гъвкавост отколкото при използването единствено на традиционни автомобилни технологии като механика и електроинженерство.

Кратките срокове за разработване на софтуерни продукти за вградени системи за автомобилната промишленост водят до ограничаване на времето за тестване. Това налага тестовете да се изпълняват бързо и същевременно да покриват колкото е възможно по-голяма част от функционалността на софтуерния продукт поради високите критерии (изисквания) за качество, надеждност и сигурност в автомобилната индустрия. Автоматизацията на тестването подпомага значително тестовия процес. Тестовете се създават в началото на проекта и след това се изпълняват и модифицират при необходимост. Освен автоматизирането им, тестовете трябва и да се оптимизират. Пълното покритие на изискванията трябва да става с минимален брой тестови стъпки, което да намалява времето за изпълнение на тестовите сценарии. Използването на систематичен подход на тестване е задължително, за да може дефектите да се откриват колкото е възможно по-рано в жизнения цикъл на проекта. През последните години в автомобилната промишленост се полагат много усилия за подобряване на тестовия процес чрез използването на методи и инструменти (tools) за систематизирано и ефективно тестване. Отделянето на време за дизайн на тестовите сценарии оказва значително въздействие върху качеството на тестовете. Методът на класификационните дървета (МКД) и графичния редактор Classification-Tree Editor eXtended Logics (CTE XL) са разработени от компанията Daimler Chrysler AG. Тяхната цел е да гарантирант систематичен и разбираем подход за дизайн на тестови сценарии за функционално тестване и тестване с последователности от входни данни.

Целта на дипломната работа е да се направи сравнителен анализ между съществуваща технология за тестване във фирмата Johnson Controls Electronics, която разработва вградени автомобилни системи, и методът на класификационните дървета в съчетание с редактора CTE XL. На базата на анализа да се формулират препоръки за подобрение на тестовия процес във фирмата.

Задачите, които произтичат от целта са:

- Да се направи обзор на видовете тестване и съществуващите методи за оптимизация
- Да се мотивира изборът на Метода на класификационните дървета
- Да се опише съществуващият процес на тестване и да се дефинират проблемите, който възникват при използването му
- Да се избере функционалност от разработван във фирмата проект и да се реализират тестови сценарии за нея чрез използването на предлагания метод на класификационните дървета
- Да се дефинират критерии на базата, на които да се сравнят и оценят разгледаните методи. Да се формулират препоръки за оптимизация към текущия тестов процес.

Ограничаващи условия при разбработването на дипломната работа са:

- Вградената система, която се използва в експеримента, да е текущо разбработвана от фирмата.
- При дизайна на тестовите сценарии да се използва вече разработен инструмент CTE XL, който е базиран на метода на класификационните дървета.
- При реализирането на тестовите сценарии с конкретни данни да се използва Visual Basic приложение, разработено във фирмата и използвано в текущия процес на тестване.

Методът на класификационните дървета е възникнал от необходимостта за систематизиран подход при подбора на тестови данни за вградени системи, но може успешно да се прилага и при други типове софтуерни приложения. По тази причина дипломната работа е насочена както към тестовите инженери, които се занимават с вградени системи, така и към тестовите инженерите във всяка друга област от разработването на софтуер.

Изложението на дипломната работа се състои от пет глави. В първата глава съществуващите типове тестване са класифицирани спрямо три критерия – готовността на софтуерното решение, функционалните и технически изисквания към системата и дали изискват изпълнение на приложението или не. Описани са и някои от най-разпространените класически техники за подбор и оптимизация на тестовите сценарии, както и някои съвременни методи, взаимствани от Изкуствения интелект като еволюционни алгоритми и класификационни дървета.

Втората глава акцентира върху метода на класификационните дървета и по-специално върху неговото разширение, предназначено за вградени системи – МКД/ВС (Метод на класификационните дървета за вградени системи). В тази глава се прави описание на различните етапи на метода, които се илюстрират с пример, и се дефинират основните предимства при използването на метода.

В глава 3 е направено кратко описание на разработвана от фирмата система, както и на конкретната функционалност от нея, която е включена в експеримента. Описва се текущият тестов процес във фирмата, както и неговите основни недостатъци.

Предназначението на четвъртата глава е да опише накратко помощния редактор CTE XL, какви са основните му функции и възможности. Също така, в тази глава се описва как е осъществен дизайна на тестовите сценарии за избраната функционалност с помощта на метода на класификационните дървета и CTE XL, както и реализирането им с конкретни тестови данни.

Петата глава е обобщение на направения експеримент. В нея се дефинират основните критерии за сравнение на методи за дефиниране на тестови сценарии и се сравняват разгледаните методи – на класификационните дървета и съществуващия във фирмата метод. На базата на сравнението се формулират препоръки към текущия процес на тестване във фирмата.

В частта **Заключение** се прави обзор на дипломната работа – какво е свършено и какви решения на поставените проблеми са избрани. Споменават се потенциални възможности за разширение и подобрение на предлагания метод.

В **Приложение А** е направено подробно описание на изискванията към избраната функционалност, която участва в експеримента.

Приложение Б описва основните инструменти, който се използват в текущия процес на тестване – CANoe и Visual Basic AutoEMF.

В **Приложение В** са предоставени детайлните спецификации на тестовите сценарии, резултат от дизайна направен с помощта на МКД и CTE XL.

Приложение Г представлява тест-репорта, който е крайният резултат от използването на МКД и Visual Basic AutoEMF.

В частта **Речник** са предоставени използваните в дипломната работа термини и техните дефиниции.

Частите **Индекс на използваните таблици** и **Индекс на използваните фигури** представляват списък на използваните в дипломната работа таблици и фигури.

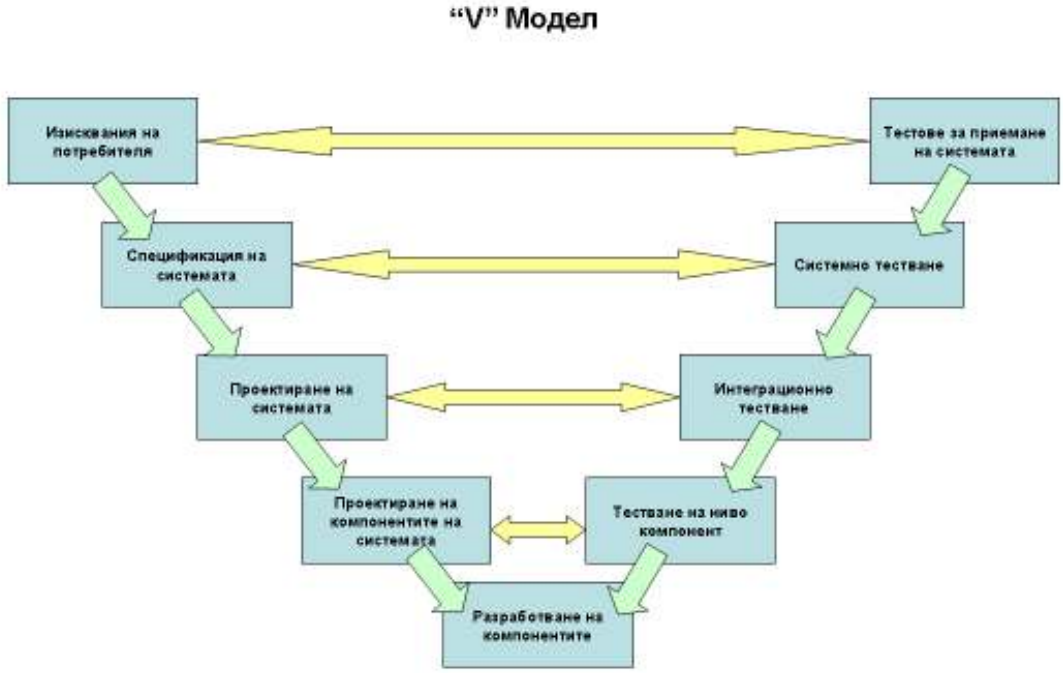
В частта **Използвана литература** се изброяват статиите, книгите и другите източни на информация, използвани по време на разработване на дипломната работа.

Глава 1. Обзор на видовете тестове и техники за оптимизацията им

1.1 Видове тестове

Има различни критерии, спрямо които може да се направи класификация на видовете тестове. При модификация на модела на водопада (*Waterfall*) на жизнения цикъл на процеса на разработване, нивата на тестване произлизат от начина, по които е проектирана и разработена системата. Тази модификация на Waterfall модела, която се нарича "V" модел, набляга на съпоставянето на различни нива на тестване с всеки един от етапите на проектиране. Така типовете тестове, които се определят, са тестване на ниво програмна единица, интеграционно тестване, системно тестване и тестове за приемане на системата. Те са разгледани подробно в част *1.1.1*.

Този модел е полезен тъй като по този начин се определят различните нива на тестване и се уточняват целите им. Схемата на "V" модела е показана на Фигура 1 [26].



Фигура 1: "V" модел

Друг критерий, по който могат да се групират тестовете, е в зависимост от функционалните и технически изисквания към системата. Системното тестване е необходимо да включва тестове за производителност, за възстановяване след авария, за сигурност, за инсталация/деинсталация и други. В част *1.1.2* ще се разгледат основните и най-често срещани типове тестове, групирани спрямо функционалните и технически изисквания към системата.

В зависимост от това как протича тестването – с или без изпълнение на тествания обект – могат да се определят два типа тестване – статично и динамично. Двата типа са разгледани подробно в част *1.1.3*.

1.1.1 Групиране на тестовете, в зависимост от готовността на софтуерното решение

Основните групи тестове спрямо готовността на софтуерното решение са тестване на ниво програмна единица (unit testing), интеграционно тестване (integration testing), системно тестване (system testing), тестове за приемане на системата (acceptance testing) и регресионно тестване (regression testing). Всеки един от типовете тестове има определени цели. Най-основният тип тестване е на ниво програмна единица. Той е разгледан в част *1.1.1.1*. Следващото ниво на тестване е интеграционното, чиято цел е да открие дефекти в интерфейсите на отделните компоненти. То е разгледано подробно в част *1.1.1.2*. Системното тестване, което е разгледано в част *1.1.1.3*, трябва да определи дали софтуерният продукт отговаря на всички изисквания на клиента. Друга група тестове са тези за приемане на системата. Те са подобни на системните, с тази разлика, че при тях клиентът участва активно. Тестовете за приемане на системата се разглеждат подробно в част *1.1.1.4*. Накрая в част *1.1.1.5* се разглеждат регресионните тестове, който изискват да се провери дали отстраняването на един дефект в софтуера не е довел до появата на други.

1.1.1.1 Тестване на ниво програмна единица (Unit testing)

Тестването на ниво програмна единица (Unit testing) е основният тип софтуерно тестване. При него се тества елементарна самостоятелна единица (компонент) от софтуера. Обикновено се извършва от самите програмисти и изисква детайлно познаване на кода и дизайна на системата. Тестването на ниво програмна единица се състои предимно от дебъгване, концентрирано върху премахването на грешки по време на писането на кода. Грешките, които се откриват по време на такъв тип тест, може да не са само в имплементацията, но също така и в дизайна и спецификацията на продукта. Въпреки че е част от ежедневната дейност по разработване на софтуер, препоръчително е тестването на отделните компоненти да се планира предварително – да се изясняват и документират тестовите данни и сценарии, както и да се опишат очакваните резултати. Като част от всяка инспекция на софтуерния продукт, трябва да се предвидят и прегледи на плановете за тестване на компоненти (unit test plans). [3; Chapter4-Types of testing]

При този тип тестване се използват няколко подхода:

- Отгоре-надолу (Top-down)
- Отдолу-нагоре (Bottom-up)
- Изолация (Isolation)
- Хибридно тестване (Hibrid Testing)

При подхода отгоре-надолу отделните единици се тестват като се използват от единиците, които ги извикват, т.е. които са по-високо в йерархията, но тестваните единици са изолирани от тези, които са след тях в йерархията – те се заместват със *стъб (stub)*. Така се осъществява едновременно и интеграционно тестване, но управлението на промените във вече тестваните единици става трудно.

При подхода отдолу-нагоре първо се тестват единиците, които са на най-ниските нива в йерархията. След това се преминава към тестване на единиците от по-горните нива, като се използват вече тестваните единици от по-ниските нива. Процесът продължава, докато се стигне до единицата, която е най-високо в йерархията. При този подход се използва *драйвер (test driver)* и се осъществява и интеграция между отделните единици. Той е подходящ, когато тестваните единици се използват многократно в различни софтуерни решения.

Подходът чрез изолация изследва всяка единица като я изолира от единиците, които я извикват, и единиците, които тя извиква. Така отделните единици могат да се тестват в произволна последователност. Промените във всяка единица се отразяват единствено на тестовете за нея. Използват се едновременно и драйвери, и стъбове, които са по-опростени в сравнение с тези, които се използват съответно при подходите отгоре-надолу и отдолу-нагоре. Този подход не включва елементи на интеграционното тестване.

В практиката най-често използвания подход е изолация модифициран с подхода отдолу-нагоре, като извикваните единици се заместват или с вече тествани единици или

със стъбове. Този тип тестване се нарича *хибридно (Hybrid Testing)* [2; Лекция3-Контрол на качеството].

1.1.1.2 Интеграционно тестване (Integration testing)

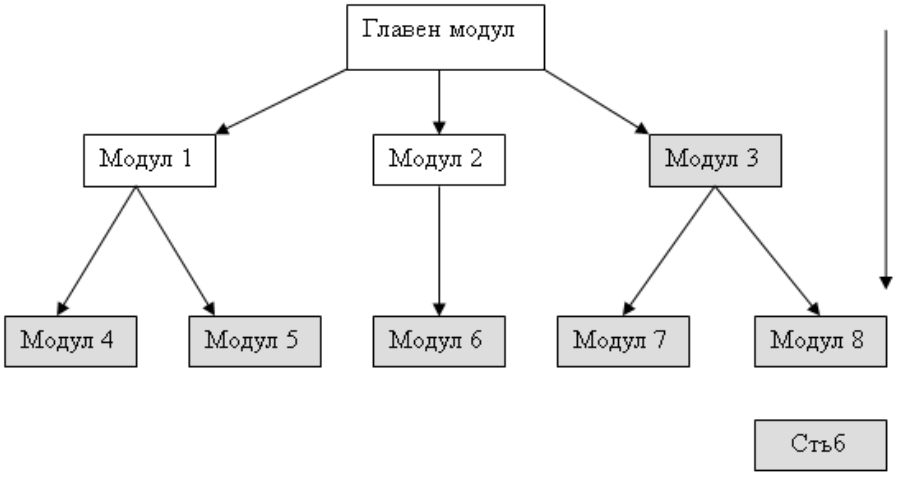
За начало на интеграционното тестване се счита моментът, в който отделните компоненти/единици започнат да се тестват свързано. Границите между тестването на ниво програмна единица и интеграционното тестване са до известна степен условни. Тестовете за интеграция могат да се разделят на две основни групи: с ниско ниво на интеграция (включват два или три модула) или такива, с високо ниво – с четири и повече модула. По време на първата група тестове вниманието е фокусирано по-скоро към качеството на кода. При втората група, когато нараства броя на модулите, се проследява по-скоро реализацията на функционалността посредством кода.

Този тип тестване е известен още като „gray-box" и не изисква детайлно познаване на кода, а на неговата структура и на интерфейсите между отделните програмни единици.

След като тестването на отделните компоненти на софтуера е завършило, отделните единици се интегрират във функционални групи и се тестват отново. Целта на този тип тестване е да се открият скрити дефекти и несъответствия в интерфейсите между отделните модули, използването на паметта, обмена на информация с базите данни. Могат да се открият и грешки в изчисленията (отклонения в изчисленията, които са били допустими за отделните модули, но които могат да се натрупат до неприемливо ниво в хода на експлоатация на системата). Конкуренцията за едни и същи ресурси в даден момент, както и проблеми с времената (за изпращане и получаване на съобщения, за реакция на системата) не могат да се открият по време на тестването на компонентите, но могат да се идентифицират в този сравнително ранен етап от разработката на софтуера.

Интеграционното тестване включва голям набор от дейности – като се започне от тестването на няколко модула и се стигне до тестване на цялата система.

Интегрирането на модулите инкрементално (стъпка по стъпка) е систематичен начин за интеграция, чрез който продуктът се „сглобява" и тества на малки парчета, за да може грешките да се откриват, изолират и редактират по-лесно. Инкременталната интеграция може да се извършва отгоре-надолу (top-down) или отдолу-нагоре (bottom-up). При метода отгоре-надолу интеграцията се осъществява като се започне с модула, който е най-високо в йерархията.



Фигура 2: Интеграция по метода „отгоре-надолу" (top-down)

Интеграцията по метода „отгоре-надолу", която е показана на Фигура 2, протича по следния начин:

1. Главният модул се използва като драйвер, а останалите модули, които са директно подчинени на главния, се заместват със стъбове.
2. В зависимост от избрания начин – първо в дълбочина (depth-first) или първо в

широчина (breadth-first) – подчинените стъбове се заместват с реалните модули един по един.
3. Тестовете се изпълняват всеки път след като някой модул е добавен.
4. След като тестовете преминат успешно, друг стъб се замества с реален модул.
5. Изпълняват се регресионни тестове, за да се осигури, че не са възникнали грешки в резултат от интеграцията на новия модул.
6. Стъпки от 2 до 5 се повтарят, докато се интегрират всички модули.

При използването на този метод възникват няколко проблема:
- много често изчисленията се извършват в модулите, които са най-ниско в йерархията
- стъбовете обикновено не пропускат данни към модулите, които са разположени по-високо в йерархията
- като резултат от забавянето на тестването, докато се интегрират модулите от най-долните нива, обикновено се получава така, че се интегрират много модули по едно и също време, а не един по един
- разработването на стъбове, които могат да пропускат данни към модули от по-горни нива, е почти толкова сложно, колкото и разработването на самите модули

Другият подход – „отдолу-нагоре" – се осъществява по следния начин:
1. Интеграцията започва с модулите, които са на най-ниското ниво в йерархията. Те са комбинират в клъстери (clusters), които изпълняват определена подфункция.
2. Създават се драйвери, които ще координират входните и изходните данни на тестовете.
3. Тестват се клъстерите.
4. Драйверите се премахват и клъстерите се комбинират.

Този подход също има недостатъци. Например цялостната програма не съществува, докато не се интегрира и последният модул. Проблемите с времената и конкуренцията за ресурси се откриват доста късно в процеса на интеграция.

В много организации разработчиците на софтуер са отговорни и за изпълняването на интеграционните тестове. Те използват т.нар. *Big Bang* подход. Всички модули се интегрират наведнъж и след това се изпълняват тестовете. Обикновено възникват доста проблеми и тъй като всички части от софтуера са интегрирани, много трудно се установяват проблемните модули [7].

1.1.1.3 Системно тестване (System testing)

Системното тестване започва след като всички функционални и технически изисквания са имплементирани. При този тип се тества цялата система – хардуер и софтуер, включително и потребителския интерфейс. Докато интеграционното тестване фокусира върху взаимодействията между отделните модули и между модулите и хардуера, системното тестване фокусира върху тестването на атрибутите на качеството на цялата система. То проверява дали системата отговаря на изискванията, дефинирани в софтуерната спецификация (Software Requirements Specification - SRS). Тестовете трябва да се изпълняват в реална среда – близка до тази, в която система е предназначена да работи, доколкото това е възможно. Тестването се извършва от гледна точка на потребителя на системата. Основните задачи са свързани с проверка на изискванията към системата, дизайна на системата и качеството на съпровождащата документация - ръководствата за потребителите, ръководство за администрация и поддръжка и т.н. [13; с.31-32]

Целта на системното тестване е да покаже, че системата изпълнява желаните бизнес процеси (функционални и технически изисквания) и да провери дали състоянията на системата, описани в нейната спецификация, могат да бъдат достигнати както е описано в изискванията. Трябва да се проверят следните характеристики:
- заеманата памет и поведението по време на работа на системата (употреба на системни ресурси и бързодействие)
- поведението на системата, когато се обработват големи количества данни
- поведение на системата съгласно изискванията за сигурност

- взаимодействието на системата с периферни устройства – например принтери и твърди дискове
- обмен на данни с паралелно работещи информационни системи
- възможности за възстановяване на системата след авария

Системното тестване включва голям набор от типове тестове в зависимост от спецификата на проекта [2; Лекция3-Контрол на качеството].

1.1.1.4 Тестове за приемане на системата (Acceptance testing)

Целта на тестовете за приемане на системата е получаване на одобрението на клиента за системата и е първа стъпка към реалната експлоатация на софтуера. Тестовете се изпълняват от гледна точка на клиента и с неговото участие – включват се не само потребителите на системата, но и екипът, който ще я обслужва. Предназначението им е да се установи дали системата отговаря на изискванията и дали е готова да бъде внедрена. Технически погледнато тестовете за приемане на системата са подобни на системните тестове, но при първите клиентът взима участие в провеждането на тестовете и тестовете се провеждат в реалните за приложението условия. Изходът от тези тестове дава възможност на клиента да реши дали да приеме или не системата. Това решение се взема въз основа на предварително дефинирани критерии за приемане и ясни правила и метрики, които дефинират условията за покритието им. Най-често стъпките за приемане се описват в Процедура за приемане на системата.

Основната цел на тестовете за приемане е да се определи дали системата изпълнява изискванията към нея. [13; с. 32-33]

1.1.1.5 Регресионно тестване (Regression testing)

Регресионното тестване означава да се тества отново вече тествана част от системата след промяна по някоя нейна функционалност. Целта е да се установи дали след отстраняване на регистрирана грешка, не е възникнала нова грешка в работещ клон на софтуера.

Тестовете трябва да проверят дали:

- непроменените части от системата имат непроменено поведение
- модифицираните части работят както се очаква
- системата като цяло изпълнява изискванията

Регресионното тестване се извършва например след отстраняване на дефекти, при внедряване на системата в нова среда, при замяна на някой компонент или при добавяне на нови функционалности към системата.

Важна предпоставка за този тип тестове е анализът на влиянието на промените – т.е. да се определи кои точно части на системата ще бъдат засегнати от промените [13; с. 33].

1.1.2 Групиране на тестовете, спрямо функционалните и технически изисквания към системата

В зависимост от приложението си всяка софтуерна система трябва да отговаря на определени изисквания – функционални или нефункционални. При планирането на тестването е много важно да се предвиди разработването и изпълнението на различни типове тестове. Това увеличава възможността от откриване на дефекти. [7]

1.1.2.1 Функционално тестване (Functional testing)

Целта на тези тестове е да провери дали определени функционалности и характеристики на системата работят според спецификацията. При този тип тестове се използва метода „black-box”, при който не е необходимо тестерът да познава архитектурата и кода на системата. Тестовете се изпълняват през графичния интерфейс на системата и на базата на подадените входни данни се прави анализ за изходните

резултати. Вниманието на тестера е насочено към коректността на реализираните бизнес процеси и тяхното съответствие с функционалните изисквания към системата. Необходимо е да се използват валидни и невалидни входни данни. Целта е да се установи дали:

- при валидни входни данни се генерират очакваните резултати
- при невалидни входни данни системата генерира адекватни съобщения за грешка и не допуска потребителя до следващите нива на системата
- навигацията между различните прозорци и полета е правилна
- определени индустриални стандарти са спазени

[2; Лекция3-Контрол на качеството]

1.1.2.2 Тестове за производителност (Performance testing)

Този тип тестове се използват за проверка и оценка на времето за отговор и реакция на систематата и използването на паметта при обичайното натоварване, както и при извънредни ситуации (свръхнатоварване). Фокусът е върху извънредните ситуации – като например обработването на големи количества данни, наличие на голям брой потребители, които работят едновременно в системата и времето за реакция на системата в този случай. [2; Лекция3-Контрол на качеството]

За осъществяването на тези тестове могат да се използват сценариите от функционалните тестове. Тестовете за производителност се разделят на два подтипа – за натоварване (load) и стрес тестове (stress).

При *тестовете за натоварване (load testing)* се тества производителността на системата при различни натоварвания както и възможността й да продължи да работи нормално при тези условия.

Стрес тестовете (Stress testing) се използват за намиране на грешки, дължащи се на ограничени ресурси или конкуренция за тях. Недостатъчно памет или дисково пространство могат да помогнат за откриване на дефекти в софтуерния продукт, които при нормални обстоятелства не се забелязват. Стрес тестовете също могат да се използват и за установяване на максималното натоварване, с което може да се справи обекта на тестването при конкретната конфигурация.

[Software Validation Plan BM1005 X7T7]

1.1.2.3 Тестове за възстановяване след авария (Recovery testing)

Този тип тестове се използват за проверка на възможността на системата да възстанови нормалната си работа след редица хардуерни, софтуерни, мрежови и други проблеми без да има загуба на данни [6]. Тестването протича като системата се подлага на извънредни ситуации (хардуерна повреда, входно-изходни грешки, грешки в данните), за да се предизвика повреда. След това се проверява реакцията на системата и времето, за което тя се възстановява след аварията. Проверява се адекватността на процедурата за възстановяване, ако има такава. За целта могат да се използват функционалните тестове, като по време на серия от транзакции се симулира появата на повреда – в захранване, в комуникациите, в цялостта на данните.

[SVP BM1005 X7T7]

1.1.2.4 Тестове за сигурност (Security testing)

Чрез тестовете за сигурност се правят опити да се нарушават проверките за сигурност на системата. Например могат да се създадат тестови сценарии, които нарушават механизмите за сигурност на система за управление на бази данни. Познати са следните нива на сигурност: системно, мрежово и приложно. Предмет на разглеждане тук са тестовете на приложно ниво. Един от начините за създаване на такива тестове е да се проучат проблемите със сигурността, възникнали при подобни системи. След това да се създадат тестови сценарии, които симулират тези проблеми върху тестваната система.

Този тип тестове са особено важни за Интернет приложенията. [6]

1.1.2.5 Тестове за конфигурация (Configuration testing)

Програмни среди като операционни системи и системи за управление на бази данни поддържат множество от хардуерни конфигурации, които включват множество типове и брой на входно-изходните устройства и комуникационните канали (communication lines). Често броят на различните конфигурации е много голям и е почти невъзможно да се тестват всички комбинации, но продуктът трябва да се тества на всеки тип хардуерно устройство и с минимална и максимална конфигурация. [6]

Обикновено много от функционалностите на дадена система могат да се конфигурират дали да присъстват или не. В такъв случай трябва да се изследват всички възможни конфигурации. Целта е да се провери, че системата работи правилно при всички хардуерни и софтуерни конфигурации. За изпълняването на този тип тестове могат да се използват скриптовете, създадени за функционалните тестове.

[SVP BM1005 X7T7]

1.1.2.6 Тестване на потребителския интерфейс (User interface testing)

Този тип тестове проверяват взаимодействието на потребителя със системата. Целта им е да осигурят, че интерфейсът предоставя на потребителя удобен достъп и навигация между функциите на системата. Също така тестването на потребителския интерфейс осигурява, че информацията, която се предоставя, е правилна и отговаря на изискванията. [6]

1.1.2.7 Тестване на удобството за работа със системата (Usability testing)

Този тип тестове са много субективни и зависят от профила на крайния потребител на системата. Няколко съображения трябва да се имат предвид, когато се изпълняват тестовете:

- Приспособен ли е потребителският интерфейс към профила (образование, социална среда) на крайния потребител?
- Предоставя ли програмата съдържателни и смислени изходни данни?
- При възникнали грешки, системата генерира ли ясни и разбираеми съобщения?
- Системата генерира ли бързо потвърждение на действията на потребителя? Ако някое действие изисква повече време, за да бъде обработено (което е често срещано, когато се достъпва отдалечена система), системата генерира ли съобщение, информиращо потребителя за извършваните операции?
- Лесно ли се използва системата? Например ако програмата изисква движение през поредица от менюта или опции, връщането в главното меню трябва да става лесно, както и преминаването през различните нива на менюто. [6]

1.1.2.8 Тестове за инсталация и деинсталация на системата (Installability testing)

Някои софтуерни системи имат сложни инсталационни процедури. Тестването на тези процедури е изключително важна част от тестовия процес. Грешка по време на инсталирането на системата може да попречи на потребителя да я използва. Много е важно как точно протича процесът на инсталиране, тъй като един сложен процес на инсталация може да откаже потребителя от понататъшна работа със системата. Съществен момент, който често се пропуска, е изследване на процедурата за деинсталация на софтуера и качеството, с което се възстановяват системните ресурси след прилагането й. [6]

1.1.2.9 Тестове за съвместимост (Compatibility testing)

Много от програмите, които се разработват, заменят стари програми. Целта на тестовете за съвместимост е да се осигури, че съществуващите данните от старата система, ще могат да се използват от новата система. [6]

1.1.2.10 Тестване на документацията (Documentation testing)

Ръководството за потребителя на системата трябва да бъде достатъчно пълно, точно и ясно написано. По тази причина за всички примери, описани в документацията, трябва да се създадат тестови сценарии, с които да се тества системата. [6]

1.1.2.11 Тестване на надеждността на системата (Reliability testing)

Надеждността на системата може да се дефинира като възможността на системата да извършва изискваните от нея функции при определени условия за определен период от време. Тестването на надеждността осигурява, че системата е стабилна. То е опит да се покрият всички функционалности. Целта обаче е то да не включва тестване на сложни функционалности, тъй като това е предмет на функционалното тестване. Тестването на надеждността трябва да потвърди, че системата няма да спре да функционира. [21]

1.1.3 Анализ на софтуерните системи

Използват се два диаметрално противоположни подхода за анализ на софтуерните системи – динамичен и статичен.

1.1.3.1 Динамичен анализ (Dynamic analysis)

Динамичният анализ осигурява информация, събрана при изпълнение на софтуерната програма. Най-общо при тези тестове се следи коректността на програмата съгласно функционалните изисквания, а също и „консумацията" на системните ресурси. Изследва се външното качество на кода. Предимството на този метод е, че системата може да се анализира при реални условия – например в реално време и в зависимост от конкретната хардуерна платформа. На ниво интеграционно и системно тестване могат да се изследват изискванията към:

- функционалността (functionality) – правилността и пълнотата на софтуерния продукт спрямо спецификацията
- производителността (performance) – времето, необходимо за извършване на определени действия
- сигурността (security) – защитата срещу неоторизирано използване на данни
- използваемост (usability) – дали системата отговаря на нуждите на потребителя
- икономичност (economy) – дали системата използва разумно ресурси
- удобство за работа (user-friendliness, usability) – дали системата е лесна за употреба за крайния потребител
- устойчивост (continuity) – сигурност, че системата е работоспособна и не реагира на смущения.

1.1.3.2 Статичен анализ (Static analysis)

Статичният анализ осигурява информация, базирана на преглед на текста на кода. При него не се извършва изпълнение на софтуерната програма. Целта е да се провери използвания синтаксис, да се прегледа дали съществува код, който не се използва, дали има променливи, които не са инициализирани, дали се използват правилно формалните и фактическите параметри на функциите, дали има неправилна работа с указатели. Тук се изследва вътрешното качество на кода. [2; Лекция3-Контрол на качеството]

Ефективна форма на статичния анализ, която има отношение към функционалността на тествания обект, се предоставя от прегледите (reviews). Има два

типа ревюта – неформален преглед на алгоритмите и инспекция.

При *неформален преглед на алгоритмите (walkthrough)* се симулира изпълнението на софтуерния продукт чрез преминаване през кода. Предварително са избрани тестови данни, които се използват за преминаване стъпка по стъпка през програмата. Това позволява да се провери как работи тестваният обект. Прегледът на алгоритмите (*walkthrough)* е подходящ за откриване на тромави алгоритмични решения, както и грешки в алгоритмите и е подходящо за изпълнение по време на тестването на ниво програмна единица.

При *инспекциите* (*inspections*) се проверява тестваният обект чрез четене на кода и дискутиране на всеки от редовете. Пример за инспекция са *техническите прегледи* (*technical reviews*).

Чрез статичния анализ се откриват липсващи, излишни или нежелани функционалности в обекта на анализ. Могат да се изследват изисквания към гъвкавостта (flexibility), поддръжката (maintainability), леснота на управлението (manageability), повторната използваемост (reusability), възможността за тестване (testability). Предимството на статичния анализ е възможността да се използва в ранните фази на разработката на продукта. Може да се използва и за проверка на изискванията и дизайна на системата. [13; с.18-19]

Двата вида анализ – статичен и динамичен – се допълват взаимно. Така се получава задълбочен и пълен анализ на софтуерния продукт. В зависимост от вида на системата и нейната критичност се използват различни комбинации от техники. [5; Part 2 – Overview of Testing Techniques Static vs. Dynamic Testing]

1.2 Оптимизации на тестовете. Подходи и техники за тестване

Обикновено тестването протича на следните стъпки. Първо се прави дизайн на тестовите сценарии. След това се генерират реалните тестови данни, които ще се използват в сценариите за изпълнение на тествания обект, и се определят очакваните резултати. По време на изпълнението на тестовете резултатите от тях се следят и се сравняват с очакваните. Сем Канер дефинира понятието *техника за тестване* като „рецепта за изпълняване на тези задачи (стъпки) с цел откриването на нещо заслужаващо внимание". [23]

Най-важната част от тестването на софтуера е определянето и създаването на ефективни тестови сценарии. Дизайнът на тестовете е важен, защото пълно и изчерпателно тестване не е възможно. Поради тази причина ключовият момент в тестването е намирането на такова подмножество от всички възможни случаи, което да открива голяма част от дефектите.

В част *1.2* се разглеждат някои от най-разпространените методи за дизайн.

1.2.1 Класически техники

В тази глава се разглеждат различните класически типове техники, които се прилагат по време на тестването на ниво програмна единица - структурно тестване (*1.2.1.1*) и тестването с мутации (*1.2.1.2*). Разглеждат се и „black-box" техники като тестване с гранични стойности (*1.2.1.5*), с класове на еквивалентност (*1.2.1.6*), с последователности от данни (*1.2.1.9*) и други.

1.2.1.1 Структурно тестване (Structure testing)

Структурното тестване се придържа към структурата на тествания обект, а не толкова към неговата семантика. Тези тестове са предназначени за изпълнение на специфични структурни елементи - изисквания (requirements), компоненти (units), функции, пътища, оператори (statements). Целта е да се постигне възможно най-голямо покритие на кода.

Структурното тестване може да се използва за създаване на тестови сценарии за

покритие. Те се определят на базата на вътрешната структура на тествания обект. Изпълняват се специфични структурни елементи и се проверява дали обектът се държи според очакваното или има отклонения в поведението му.

Предимството на структурното тестване е, че при него ясно се вижда кои части от системата не се изпълняват (редове код, които никога не се достигат). Недостатък е, че връзката с функционалните изисквания на този тип тестване е слаба – например липсващи разклонения в програмата се откриват доста трудно, ако тестващият не е запознат с функционалните изисквания.

За разлика от структурното тестване функционалното тестване е тясно свързано със спецификацията. При него тестовите сценарии се определят на базата на функционалните изисквания. Двата типа тестване не са взаимнозаменяеми, а се допълват. За целите на тестване от гледна точка на клиента и потребителя, структурното тестване може да се използва като допълващо, а не като определящо тестовите сценарии.

Ефективна стратегия е да се комбинират двата метода. На ниво програмна единица на практика се използва структурно и функционално тестване. По време на тестовете за интеграция предимно се изпълняват функционалните тестове. На базата на функционалните изисквания се определят тестовите сценарии, генерират се тестовите данни и се определят очакваните резултати. Следващата стъпка е да се изпълни тествания обект с избраните данни, да се установят получените резултати и да се оцени теста. След края на функционалните тестове, се извършват отново структурни тестове, но само за онези части от софтуера, които не са били покрити вече или при които е идентифициран проблем, свързан с вътрешното качество на кода. Например бързодействие при обработка на дадена заявка към базата данни.

За да може да се използва тази стратегия, трябва да се следи тествания обект – в коя част на тестовете кои вътрешни структури се изпълняват.

На базата на *граф на изпълнението (control flow graph)* могат да бъдат дефинирани няколко типа метрики за покритие:

Покритие на операторите (statement coverage)

При този тип всеки един от операторите трябва да бъде изпълнен поне веднъж. Това условие е минимален критерий и не е ефективно при по-цялостно тестване.

Покритие на разклоненията (branch testing, decision coverage)

При него всяко от разклоненията на програмата трябва да бъде изпълнено поне веднъж. Празните разклонения, т.е. тези които нямат оператори, трябва също да се изпълняват. Това тестване включва в себе си покритието на операторите, но набляга повече на структурата на тествания обект.

Покритие на условията (condition coverage)

Има три типа тестване на условията – покритие на прости условия, минимално покритие на съставни условия и покритие на съставни условия.

При *покритието на прости условия (simple condition coverage)* се предполага, че условието за преход (decision) е съставено от атомарно условие т.е. такова условие, което не съдържа в себе си други условия. Всяко атомарно условие трябва да е поне веднъж *вярно* и поне веднъж *невярно* по време на тестването. Например в условието за преход $((A = 0)AND((B < 0)OR(C > 0)))$ има три атомарни условия - $(A = 0)$, $(B < 0)$ и $(C > 0)$.

При *минималното покритие на съставни условия (minimal multiple condition coverage)* всяко отделно условие – атомарно или съставно – трябва поне веднъж да е *вярно* и поне веднъж *невярно* по време на изпълнението. Например като допълнение на покритието на прости условия описано по-горе, съставните условия $((B < 0)OR(C > 0))$ и $((A = 0)AND((B < 0))OR(C > 0)))$ трябва да са поне веднъж *верни* и поне веднъж *неверни*.

При *покритието на съставни условия (multiple condition coverage)* всички възможни комбинации от булевите стойности (Декартовото произведение) на всички условия трябва да се тестват.

Покритие на пътищата (path coverage)

При този тип метрика всеки възможен път от графа на изпълнение на програмата

от началния до крайния връх трябва да се изпълни поне веднъж. Ако в графа има цикли, броят на пътищата, които трябва да се преминат, става неопределен. Следователно, трябва да се дефинират определен брой итерации от цикъла, които да се тестват. Покритието на пътищата съдържа в себе си покритие на разклоненията.

Друг тип критерий за оценка на покритието е покритието на *граф на извикванията на функциите (call graph)*, който представя връзките между функциите в системата. На базата на граф на извикванията на функции могат да се разгледат няколко типа метрики за покритие:

- всяка функция трябва да се извика поне веднъж (**all-functions**)
- всяко извикване между две функции трябва да се изпълни поне веднъж (**all-relations**)
- всяко възможно извикване между две функции трябва да се изпълни поне по веднъж. Многократни изпълнения се извършват тогава и само тогава когато в извикващата функция има няколко обръщения към извикваната функция (**all-multiple-relation**)
- всяка възможна последователност от извиквания на функции трябва да се изпълни поне веднъж (**all-call-sequences**)

Ако върховете на графа на извикванията не са функции, а са компоненти, всеки връх (компонент) предоставя няколко функции. В този случай могат да се дефинират следните метрики за покритие подобни на горните:

- всички предоставени (експортирани) от даден компонент функции трябва да се извикат поне по веднъж (**all-exports**)
- всички външни (импортирани) за даден компонент функции трябва да се извикат поне по веднъж (**all-imports**)
- всички точки на извикване на външните функции трябва да се изпълнят поне по веднъж (**all-multiple-imports**)
- нека да считаме, че за всяка външна функция се взима само една точка на извикване и всички останали нейни извиквания се пренебрегват. На базата на това предположение, всички възможни последователности от различни импортирани функции трябва да се изпълнят поне по веднъж (**all-import-call-sequences**)
- всички точки на извикване на външните функции се взимат предвид и всички възможни последователности трябва да се изпълнят поне веднъж (**all-multiple-import-sequences**)

[13; с. 45-50]

1.2.1.2 Тестване с мутации (Mutation testing)

Тестването с мутации е техника, която се използва по време на тестване на ниво програмна единица. Идеята му е, че се създават много копия на оригиналната програма, като във всяко копие се въвежда по една умишлена грешка (мутация). Тези копия се наричат *мутанти*. *Мутиращ оператор (mutation operator)* е синтактично или семантично правило за преобразуване като например замяната на < с >. Нека P е програмата :

```
1. if (x > 0)
2.     doThis();
3. if (x > 10)
4.     doThat();
```

Мутация на P е :

```
1. if (x < 0)       //мутация
2.     doThis();
3. if (x > 10)
4.     doThat();
```

Друга мутация на P е :

```
1. if (x > 0)
2.     doThis();
3. if (x < 10)          //мутация
4.     doThat();
```

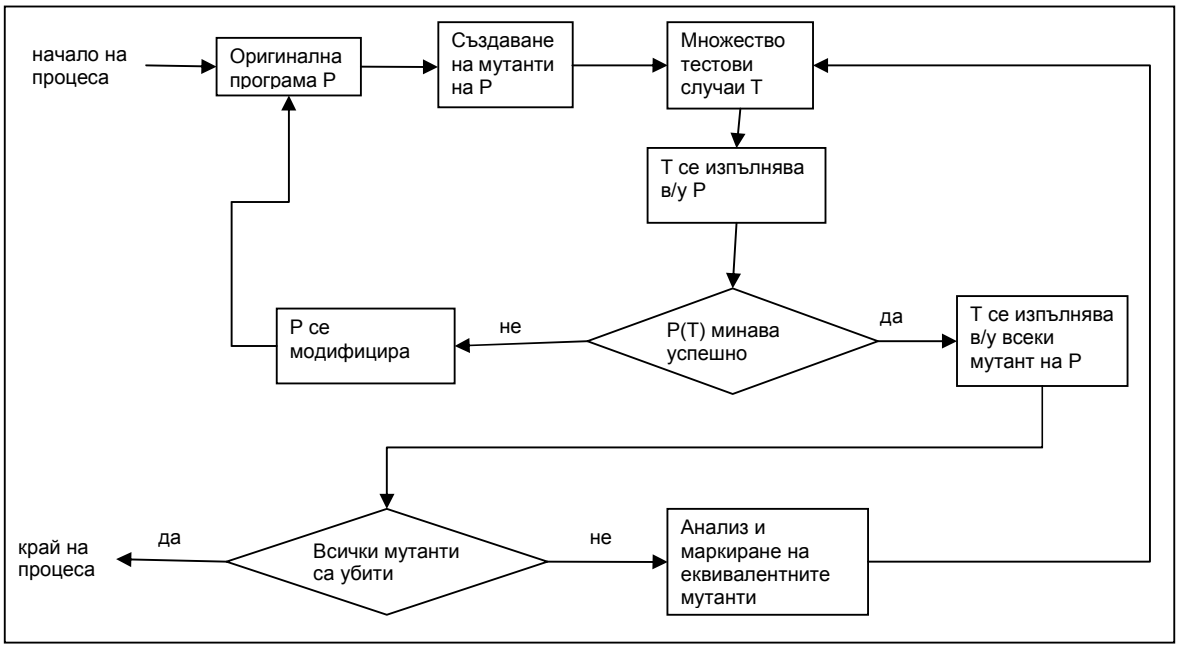
Нека с D отбележим множеството от всички възможни входни данни за P. Нека сме избрали тестово множество от данни T ($T \subset D$) такова, че P минава успешно всички тестове от T. Всеки мутант на програмата се изпълнява с тестовете от T и след това се отчита броя на мутантите, за които тестовете не са преминали успешно. Така се измерва възможността на тестовото множество да се справя с откриването на мутации. Ако T не минава успешно за конкретен мутант казваме, че мутанта е *„убит" (killed)*. Идеята е, че ако T открива конкретната умишлена грешка, то T ще открие и много други неизвестни грешки. Мутанти, които не са „убити" се наричат *живи*. Мутант, който винаги връща същия резултат като оригиналната програма P, наричаме *еквивалентен*. Възможността на дадено тестово множество да се справя с откриването на мутанти се дефинира като:

(броя на всички „убити" мутанти)/(общия брой на нееквивалентните мутанти)*100

Традиционният подход при тестването е да се намери подмножество T на множеството от входни данни D, такова че ако за всеки набор данни от T програмата P връща очаквания резултат, то и за всички останали набори данни от D програмата ще се държи според очакваното.

При тестването с мутации се счита, че ако програмата P премине успешно всички тестове от T и всички нееквивалентни мутации на P са „убити" от някои тест от T, то P удовлетворява множеството от тестове T. Ако успеем да намерим такова T, което удовлетворява горния критерий, казваме че P е преминала теста с мутации. [9]

Процесът на тестване с мутации може да се представи чрез диаграмата, показана на Фигура 3.



Фигура 3: Процес на тестване с мутации

Сложността на тестването с мутации е в избора на мутиращите оператори, който зависи от използваните технологии за разработка. [24]

1.2.1.3 Интуитивно тестване (Error guessing)

Това е един от най-често срещаните методи, тъй като използването му не изисква предварителни инвестиции и специализирана подготовка. Той се основава на опита и интуицията на тестера. Тестерът интуитивно дефинира тестовите сценарии чрез опита, който е придобил в проблемната област, и така открива грешките. Например тестерът подозира, че може да възникне проблем при стойност нула за някоя променлива и да включи този случай в тестовете. Той е приложим при сравнително прости софтуерни системи. [13; с.50]

1.2.1.4 Тестване с произволни данни (Random testing)

При тази техника тестовите данни се генерират случайно. Вероятността да се генерират специфични комбинации от стойности е много малка. Този тип тестване често е много обширно, когато се определят очакваните резултати, тъй като контекстът, в който се генерират данните, все още не е идентифициран. [18] [19]

1.2.1.5 Тестване с гранични стойности (Boundary-value testing)

Това е най-разпространената техника за функционално тестване. Тя е фокусирана върху изследване на областта на входните параметри. Идеята на тази техника е за всяка променлива да се разгледат пет валидни стойности – минимална (*min*), близка до минималната, но в интервала от допустими стойности (*min+*), номинална (*nom*), близка до максималната, но в интервала от допустими стойности (*max-*) и максимална (*max*). Основният принцип, който се използва, е Принципът на единичната грешка от Теория на надеждността (single fault assumption of reliability theory) – причината за дефект в някакъв продукт много рядко се дължи на едновременното допускане на две различни грешки. В резултат на този принцип наборите от тестови сценарии се построяват по следния начин: във всеки тестов сценарий точно една от променливите се взима с някоя от граничните стойности, тъй като възможността за грешки при тях е най-голяма, и се комбинира с номиналните стойности на останалите променливи. Техниката на граничните стойности е подходяща за числови променливи, чиито допустими стойности са зададени с интервал, когато входните променливи са независими една от друга и когато променливите имат характер на физически величини – температура, скорост, ъгъл.

Тестването за устойчивост (robustness testing) е разширение на тестването на гранични стойности. При него се добавят още две възможни стойност – близка до минималната, но извън интервала от допустими стойности *(min-)*, и близка до максималната, но извън интервала от допустими стойности *(max+)*. Целта на тези тестови сценарии е да се провери дали системата е стабилна, когато с нея се работи неправилно.

Друга разновидност на тестването с гранични стойности е *тестването в най-лошия случай (worst case testing)*. При него не се спазва Принципа за единичната грешка и се изследва поведението на системата, когато повече от една променлива има стойност, различна от номиналната. [1; Лекция4 - Гранични стойности]

1.2.1.6 Тестване с класове на еквивалентност (Equivalence partitioning)

Идеята на тази техника за тестване е да се разбие на класове на еквивалентност множеството от допустими стойности на всяка променлива и от всеки клас на еквивалентност да се използва по един елемент – този елемент покрива целия клас. Ефектът от тази техника е, че се получава пълнота на тестовете, както и се избягват повторения на тестовите сценарии. [2; Лекция4 - Техники за тестване]

Да предположим, че имаме функция на три променливи – a, b и c, чиито области от допустими стойности (домейни) са съответно A, B и C. Да предположим и че сме избрали подходяща *релация на еквивалентност*, която разделя домейните на следните части:

$$A = A_1 \cup A_2 \cup A_3$$
$$B = B_1 \cup B_2 \cup B_3 \cup B_4$$
$$C = C_1 \cup C_2$$

Прави се разлика между *слабо (weak)* и *силно (strong)* тестване с класове на еквивалентност.

Слабото тестване се осъществява като се използва по една променлива от всеки клас на еквивалентност за всеки тестов сценарий (Таблица 1). Т.е.:

- *Стъпка 1*: Създаваме толкова тестови стъпки, колкото е броят на класовете в домейна с най-много подмножества (в примера това е B) и във всеки тест поставяме по един елемент от всеки клас на еквивалентност на този домейн.
- *Стъпка 2*: От останалите домейни редуваме елементите от всеки клас на еквивалентност, докато се запълнят всички тестови сценарии.

Таблица 1: Пример за слабо тестване с класове на еквивалентност

Тестов сценарий	a	b	c
TC1	a_1	b_1	c_1
TC2	a_2	b_2	c_2
TC3	a_3	b_3	c_2
TC4	a_1	b_4	c_1

При силното тестване се прави пълно комбиниране на класовете на еквивалентност - Декартово произведение (Таблица 2). Т.е.:

- *Стъпка 1*: За всеки домейн се избира по един елемент от всеки негов клас на еквивалентност. Нека да означим множеството от представители на домейна A с P_A, на домейна B с P_B и на домейна C с P_C (за примера)
- *Стъпка 2*: Тестовият план всъщност е Декартовото произведение от всички такива множества – $P_A \times P_B \times P_C$, а броят на тестовите сценарии е $\left|P_A \times P_B \times P\right|$.

Таблица 2: Пример за силно тестване с класове на еквивалентност

Тестов сценарий	a	b	c
TC1	a_1	b_1	c_1
TC2	a_1	b_1	c_2
TC3	a_1	b_2	c_1
…	…	…	…
TCN	a_3	b_4	c_2

[1; Лекция5-Еквивалентност]

1.2.1.7 Тестване с разделяне на категории (Category-partition method)

Тестването с разделяне на категории (TPK) е систематичен, базиран на спецификацията метод, който използва разделяне (partitioning) на областта от входни данни на дадена функция, за създаване на функционални (black-box) тестове за комплексни софтуерни системи. Този метод включва използването на формална спецификация на тестовете и инструмент за генериране на описания на тестови сценарии на базата на направения дизайн на тестовете. При него се преминава през серия от декомпозиции, като се започне с оригиналната функционална спецификация на продукта и се продължи с детайлите на всяка подфункционалност, която се тества. Основните

характеристики на тестването с разделяне на категории са следните:

- Спецификацията на тестовете е кратко и унифицирано представяне на информацията за тестване на дадена функционалност.
- Спецификацията на тестовете може да се променя лесно, ако това се налага от промяна във функционалната спецификация, открити грешки в дизайна на тестовете или ако са необходими по-голямо или по-малко количество тестови сценарии.
- Спецификацията на тестовете предоставя възможност на тестера да следи броя и размера на тестовете.
- Инструментът за генерация предоставя автоматизиран начин за създаване на пълен тест за всяка функционалност и не позволява невъзможните или нежелателни комбинации от параметри.
- Методът набляга както на пълното покриване на функционалната спецификацията, така и на откриването на грешки в софтуерния продукт.

Първата стъпка от тестването с разделяне на категории е *Анализ на функционалната спецификация на продукта*. Тестерът трябва да идентифицира различните функционални единици, които могат да се тестват независимо една от друга. Функционална единица може да е потребителска команда или описана в спецификацията функция, която се извиква от други функции на системата. За по-сложни системи такава декомпозиция трябва да се направи на няколко нива. Процесът на „разбиване" на спецификацията е подобен на декомпозицията, която се извършва от софтуерните дизайнери. Декомпозицията, направена по време на дизайна, може да се използва като основа за определянето на функционалните единици. Препоръчва се използването на независима декомпозиция по време на тестването, тъй като така могат да се открият грешки във функционалната спецификация и в дизайна на системата. За всяка функционална единица трябва да се определят параметрите и техните характеристики, както и условията на средата по време на изпълнението й. Тестерът класифицира тези показатели в *категории (categories)*, които влияят на поведението на функционалната единица.

В следващата стъпка всяка категория се разделят на *възможности (choices)*, които включват всички различни типове стойности, допустими за категорията. Тестерът определя различните значими случаи, които могат да възникнат за отделните категории. Всяка възможност е множество от стойности с подобни свойства. За създаването на тестовите сценарии от възможностите ще се избират различни представители. Докато категориите се определят изцяло от функционалната спецификация, възможностите могат да са базирани както на нея, така и на опита и интуицията на тестера и на вътрешната структура на програмата.

Тъй като възможностите в различните категории често си взаимодействат по различни начини и влияят на резултатите от тестовите сценарии, в спецификацията на тестовете могат да се въведат *ограничаващи условия (constraints)*, които да описват тези връзки. Тестерът решава как си взаимодействат възможностите, как появата на една възможност влияе върху наличието на друга и какви специални ограничения могат да повлияят на дадена възможност.

След установяването на категориите, възможностите и ограничаващите условия, те се записват във формална спецификация на тестовете. Тя се състои от списък на категориите и списъци с възможностите за всяка категория. Спецификацията се използва за създаването на множество от *тестови рамки (test frames)*, които са основата за действителните тестове. За тази цел се използва инструмент за генерация.

Тестерът трябва да провери тестовите рамки, създадени от инструмента и да реши дали са нужни промени в спецификацията на тестовете. Причина за такива промени може да е липсата на определени тестови сценарии, появата на невъзможни комбинации или наличието на твърде много тестови сценарии.

Определянето на ограничения върху възможностите, генерирането на спецификацията и оценката й могат да се повторят няколко пъти.

Последната стъпка на тестването с разделяне на категории е от генерираните тестови рамки да се получат тестови сценарии с реални данни и след това да се

комбинират в набор тестови сценарии. Информацията от категориите, определени от параметрите на функционалната единица, се използват за определяне на конкретни входни данни. Категориите, базирани на условията на средата, водят до установяване на състоянието на системата, когато даден тестов сценарий е стартиран.

Тестов сценарий е последователност от елементарни тестови стъпки за една или няколко функционални единици. Тестерът трябва да реши дали стъпките ще са зависими един от друг или не. Обикновено всяка тестова стъпка изисква средата, в която ще се изпълнява, да се установи в определено положение преди стартирането й. Добър начин за групиране на тестовете е стъпките, изискващи еднакви настройки на средата, да се комбинират в един сценарий. Тази техника е подходяща за тестове, които не променят средата по време на изпълнението си. Друг начин е да се използват резултатите от една тестова стъпка, за да се установи системата в определено състояние, което се изисква за изпълнението на друга стъпка. Най-сигурният начин е преди изпълнението на всеки тест, средата да се настройва спрямо изискванията му. [16]

1.2.1.8 Тестване с таблици на решенията (Decision table)

Таблиците на решенията са много подходящи за описване на ситуации, при които комбинации от действия се изпълняват при определени условия. Същността на много софтуерни приложения може да се опише по следния начин: определя се кое от условията е постъпило на входа и след това се изпълняват съответните за това условия действия. За това таблица от следния вид (Таблица 3) ще наричаме таблица на решенията:

Таблица 3: Таблица на решенията

	действие1	***действие2***	***действие3***	***...***	***действиеM***
условие1	+		+		
условие2		+			+
условие3	+	+			
условие4		+	+		
условие5			+		
...					
условиеN	+		+		+

Таблиците на решенията се използват в много области като модел за алгоритмично решение на практически задачи. Основите на тестването с таблици на решенията са следните:

- *Първи етап:* На базата на спецификацията и изискванията към софтуерното приложение се формират списъците от условия и действия. Много е важна пълнотата на двата списъка.
- *Втори етап:* Отново на базата на спецификацията и изискванията, се изясняват връзките между условията и действията – т.е. наличието на кое входно условие изпълнението на кои от действията би предизвикало. Идеята е от всяко условие да се извлече тестов сценарий.

При тази техника също могат да се разгледат два случая – силен и слаб. Ако избраният подход е слабият, се преминава към следващия (трети) етап. Ако избрания подход е силният този етап се пропуска.

- *Трети етап:* Водещият критерий в този е етап е да се постигне оптимален брой по отношение на тестовите сценарии. Както и при други техники е възможно да се появят многобройни тестови сценарии и изпълнението на всичките да е практически невъзможно. Отново са възможни два случая. Първият е, когато с няколко условия $c_{i1}, c_{i2}, \ldots, c_{ik}$ се свързват едни и същи действия $a_{j1}, a_{j2}, \ldots, a_{jl}$. При формирането на списъка от условия, условието $c_{i1} \vee c_{i2} \vee \ldots \vee c_{ik}$ е разбито ненужно на k отделни случая. Така можем да обединим тези случаи в един. Възможна е и обратната ситуация – когато няколко действия $a_{j1}, a_{j2}, \ldots, a_{jl}$ се

свързват едновременно с всяко от условията $c_{i1}, c_{i2}, \ldots, c_{ik}$. В този случай е раздробено на l отделни действия по-естественото условие $a_{j1} \wedge a_{j2} \wedge \ldots \wedge a_{jl}$. Този подход може да се използва за построяване на тестовите процедури откъм изхода.

- *Четвърти етап:* Построяването на тестовите процедури може да се осъществи откъм входа. Тогава всяко условие ще породи една тестовастъпка. Но е възможно, ако се налага, да построим процедурата откъм изхода – тогава за всяко действие ще има една тестова стъпка. [1; Лекция6-Таблици на решенията]

Както и другите техники за тестване, използването на таблици на решенията е подходящо за някои приложения – например когато се използва *If-Then-Else* логика, когато има логическа връзка между променливите, когато има причинно-следствена връзка между входните променливи и изходните резултати. [4; Chapter 7]

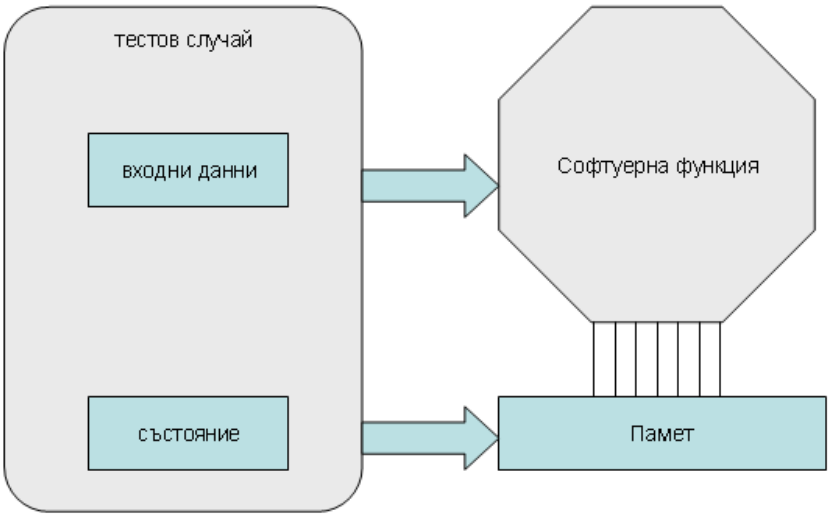
1.2.1.9 Тестване с последователности от данни (Sequence testing)

Тази техника се базира на допускането, че поведението на функциите се определя от вътрешното им състояние (internal settings) и входните въздействия, прилагани към тях. В зависимост от комплексността на тестовата система се разглеждат два основни подхода за реализация. [17]

Генериране на двойки входни данни-състояние (Single input-state-pairs)

При този метод се генерират двойки от вътрешни състояния и входни ситуации - т.е. една тестова стъпка представлява конкретен момент от времето в последователност от тестове (Фигура 4). Този метод работи добре за сравнително прости софтуерни модели, но няколко проблема възникват при работа с него.

Директното установяване на вътрешните състояния обикновено не е възможно във всички случаи и изисква да се правят промени в тествания обект. Когато се генерират вътрешни състояния, трябва да се провери дали те са валидни спрямо спецификацията. Тестерът трябва да проучи как се генерират данните, тъй като някои от тях могат да са източник на грешки. Той трябва да установи кое е първоначалното състояние, което е предизвикало проблема. [2; Лекция4-Техники за тестване]

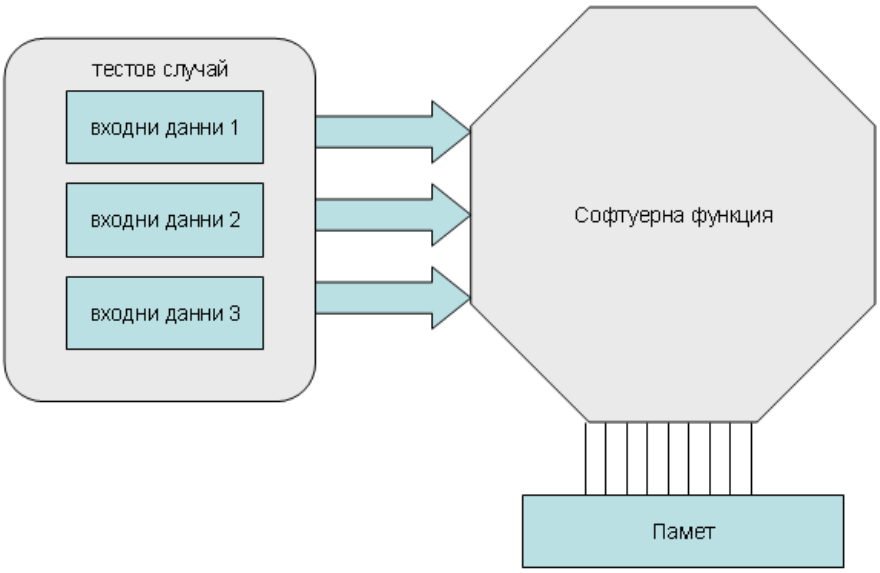


Фигура 4: Тестване с двойки входни данни-състояние

Генериране на последователности от входни данни (Input sequence)

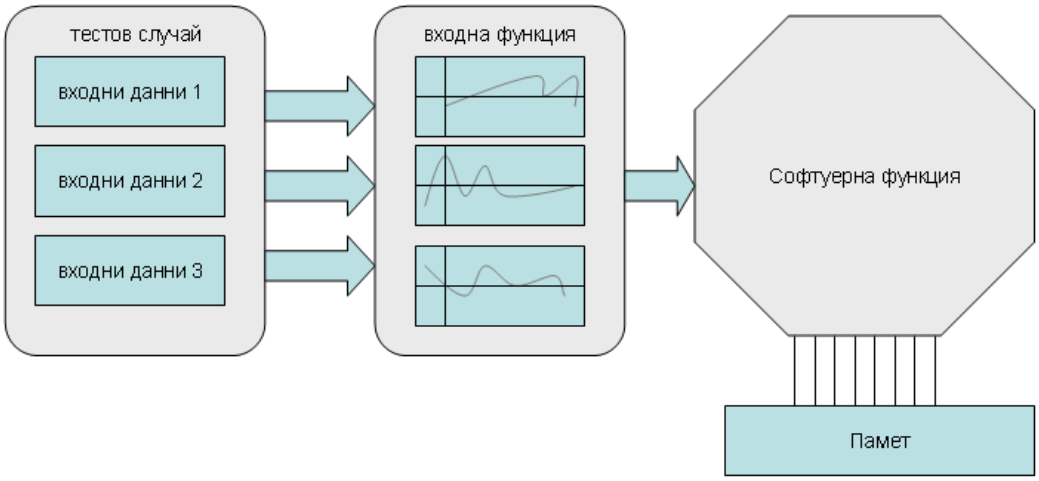
За по-сложни системи е необходимо да се търсят реални последователности от входни данни. Един от начините да се направи това е да се създаде списък от входни

данни, които последователно да се подават към тествания обект (Фигура 5). Това означава, че последователност от тестове се формира, като се използва по един елемент от списъка за всеки момент от теста. Този начин е подходящ за софтуерни системи базирани на модели на състоянията (state models).



Фигура 5: Тестване с генериране на списък от входни данни

За контролни системи (control systems), при които се изискват продължителни последователности от входни данни, се използват кодирани входни функции (encoded input functions). Последователностите от тестове се описват с техните основни характеристики (Фигура 6). Това става като се опишат само избрани моменти от тестовата последователност и се определят функциите на преходите между тези стойности.



Фигура 6: Тестване с генериране на входни последователности от кодирани входни функции

От гледна точка на тестера, методът с последователност от входни данни е по-подходящ, тъй като той гарантира, че софтуерът се тества по начина, по който по-късно ще се използва.

Чрез използването на тестването с последователности от данни, входните данни и реакциите на системата не се смятат за статични величини, а по-скоро за зависими от времето сигнали (signal waveforms). [8]

1.2.2 Съвременни техники

Тази част се разглеждат две сравнително нови техники за дизайн на тестови сценарии, взаимствани от Изкуствения интелект. В част *1.2.2.1* се разглежда тестването с използване на еволюционни алгоритми както при функционално, така и при структурно тестване. Част *1.2.2.2* се разглежда прилагането на класификационните дървета за систематично определяне на тестове.

1.2.2.1 Еволюционно тестване (Evolutionary testing)

Еволюционните алгоритми се използват за определяне на входни данни. Те са итеративна процедура за търсене, която използва оператори за изменение и подбор, за да копира поведението на биологичните системи и тяхната еволюция. За целта е необходимо да се определи пространство на търсенето (search space). Това е пространството от възможните входни комбинации, които удовлетворяват целите на тестването.

Началото на еволюционния алгоритъм е *подбора (selection)* на подходящи индивиди. Това става в зависимост от тяхната *годност (fitness value)*. Следващата стъпка е да се определят кои от индивидите ще се *комбинират (recombination)*, за да се получат нови индивиди. Също така, индивидите могат да се подложат на *мутации (mutation)*. Новите индивиди се оценяват и в зависимост от тяхната годност се определя кои и колко от новите и от старите индивиди ще *преминат в следващото поколение (reinsertion)*.

За оптимизиране на алгоритъма могат да се използват разширени еволюционни алгоритми (extended evolutionary algorithms). Те включват използването на няколко популации от индивиди, при всяка от които стратегиите за търсене са различни. Могат да се използват и стратегии за конкуренция между отделните популации. [2, Лекция4-Техники за тестване]

Еволюционно структурно тестване (Evolutionary structural testing)

Целта на структурното тестване е да се намери множество от тестови сценарии (избрани входни данни), чрез които да се направи пълно покритие на структурата на софтуера. Всеки тест се разделя на множество от цели и за търсене на данни, които да удовлетворяват тези цели, се използват еволюционни алгоритми. Всяка цел е свързана с програмна структура, която трябва да се изпълни, за да се постигне пълно покритие спрямо избрания критерий. Например, когато се използва критерий за покритие на операторите, всеки оператор представлява цел на търсенето. Предварително изискване за успешното прилагане на еволюционните алгоритми е дефинирането на подходяща функция за определяне на годността (fitness function), която да представя целите точно.

Еволюционно функционално тестване (Evolutionary functional testing)

По-сложните динамични системи трябва да се оценяват за сравнително дълъг период. Това означава, че върху системата трябва да се въздейства чрез множество от последователни входни данни. Много е важно описанието на входните последователности да е компактно. Те трябва да съдържат минимално количество елементи, но същевременно да предоставят достатъчно разнообразие от данни, за да се симулира реалната работа на системата. Това означава, че може да се говори за качество на входната последователност, като елемент от качеството на целия тестов процес. [17]

Еволюционни алгоритми за структурно тестване с последователности (Evolutionary algorithms for structural sequence testing)

Еволюционното структурно тестване първоначално е било предназначено за определяне на множество от входни данни за дадена функция. Като резултат от това се получава голяма степен на покритие на тествания софтуер. Основната идея е генерирането на тестови данни да се сведе до задача за търсене, която се решава с помощта на еволюционни алгоритми.

При структурното тестване с последователности целта отново е да се получи голяма степен на покритие на тествания обект, както при еволюционното тестване (ЕТ). При тестването с последователности пространството на търсене и функцията за определяне на

годност трябва да се предефинират, тъй като тестовите сценарии са състоят от последователности от входни данни, всяка от които покрива определен път в програмата.

При класическия метод за еволюционно тестване пространство на търсене се формира от входните параметри на функция, която се тества. При еволюционното тестване на последователности една тестова данна (индивид) е списък от входове. Резултатът от изследването на тестовия сценарий ще е списък от изпълнените разклонения от програмата – за всеки вход по едно разклонение. Дължината на списъка от входове не е ограничена.

При еволюционните алгоритми е много важно какво представяне на индивидите ще се избере. Лесен начин за това е дължината на входните последователности да се определи ръчно. Всеки индивид, генериран от еволюционния алгоритъм, представлява данните в една последователност.

Еволюционни алгоритми за функционално тестване с последователности (Evolutionary algorithms for functional sequence testing)

Генерирането на тестови последователности за динамични системи поставя някои предизвикателства.

- Входната последователност трябва да е достатъчно дълга, за да симулира работата на системата в реални условия.
- Входната последователност трябва да притежава определени качества, за да въздейства адекватно на системата.
- Изходната последователност трябва да се оцени спрямо различни условия, които често си противоречат и само част от тях могат да се активни едновременно.

Следните задачи трябва да се решат, за да се разработи тестова среда за тестване на динамични системи:

- Да се създаде компактно описание на входните последователности
- Да се оценят изходните последователности спрямо дефинираните изисквания
- Генерирането на входни последователности и оценката на системните изисквания трябва да се включат в процеса на оптимизация. [8]

1.2.2.2 Метод на класификационните дървета (Classification-Tree method)

Методът на класификационните дървета (МКД) е подход за функционално (black-box) тестване, който използва и усъвършенства метода с разделяне на категории. Основната идея на МКД е, че областта от допустими за тествания обект входни данни (input domain) се разделя на подмножества (*класове*) спрямо характеристиките (aspects) на обекта, които са свързани с целите на теста. След това се генерират тестови сценарии като се комбинират класовете от различните класификации.

Първата стъпка при използването МКД е да се определят характеристиките, отнасящи се към теста. Най-важният източник на информация за тестера е функционалната спецификация на тествания обект. Опитът на тестера също е много важен при определянето на характеристиките. За всяка една от тях областта от входни данни се разделя напълно на взаимноизключващи се подмножества – *класификации (classifications)*. За всяка класификация се прави разделяне *на класове (classes)*. В много случаи е полезно да се използват *подкласификации (sub-classifications)*, които не включват цялата област от входни данни, а само един клас от вече съществуваща класификация. Рекурсивно използване на класификации може да се направи на много нива, докато се постигне точно разделяне на възможните входни данни. Резултатът е дърво от класификации и класове – класификационно дърво. Коренът му представя областта от входни данни на тествания обект.

Втората стъпка при използването на МКД е построяването на тестовите сценарии на базата на класификационното дърво. Тестова стъпка се определя чрез комбинирането на класове от различни класификации. За всяка тестова стъпка се използва точно един клас от всяка класификация. За тази цел дървото се разполага над *таблицата на комбинациите (combination table)*, където се маркират класовете, които ще се

комбинират. Всеки ред в таблицата представлява тестова стъпка, а всяка колона представлява клас от дървото, за който няма да бъдат определяни още класификации (т.е. листо в дървото). Броят на тестовите стъпки зависи от избора на тестера – водещи са критериите за минималност и максималност на тестовите сценарии. *Критерият за минималност* описва броя на тестовите стъпки необходими, за да се покрие всеки клас от класификационното дърво поне веднъж – т.е. в този случай броят на тестовете ще е равен на броя на класовете в класификацията, разделена на най-много подмножества. *Критерият за максималност* се използва, когато трябва да се получи пълно покритие на всички класове – т.е. броят на тестовете е Декартовото произведение от броя на класовете във всички класификации.

Методът на класификационните дървета е подходящ за автоматизиране тъй като чрез него процеса на проектиране на тестовете се разделя на стъпки, които могат самостоятелно да се автоматизират. [14]

1.2.2.3 Метод на класификационните дървета за вградени системи – МКД/ВС (Classification-tree method for embedded systems – CTM/ES)

Разновидност на метода на класификационните дървета е методът на класификационните дървета за вградени системи. Той допълва разработването на софтуерни продукти на базата на модели (model-based development) като използва нов начин за систематичен подбор и описание на тестови сценарии за софтуер, вграден в контролни системи. Този метод позволява графичното описание на зависещи от времето тестови сценарии като използва изменящи се сигнали (signal waveforms), които се дефинират стъпка по стъпка за всеки набор от входни данни.

Класификациите при този метод са входните променливи за даден функционален модел (функция). Класовете се определят като дефиниционната област на всяка променлива се разделя на множество от пълни и незастъпващи се интервали. На базата на това разделяне тестовите сценарии описват изменението на входните променливи спрямо времето. Постепенна промяна на стойностите на даден входен параметър във времето се осъществява чрез избирането на различни класове от класификацията, която отговаря на съответния входен параметър. Преходите между отделните класове в дадена класификация могат да се осъществят чрез различни функции – сплайн, синусови, косинусови и други. [12]

Методът ще бъде разгледан подробно във следващата глава.

Глава 2. Систематично тестване на вградени системи за автомобилната промишленост чрез метода на класификационните дървета

Тестването е важна част от процеса на разбработване на софтуерни продукти. В тази глава ще се разгледат различни проблеми, които възникват в процеса на тестване на софтуер за вградени системи и ще се предложи използването на един метод от Изкуствения интелект за решаването им. В част *2.2* се разглеждат различните етапи, през които минава прилагането на метода на класификационните дървета, при определяне на необходимите тестови сценарии за изчерпателно и пълно тестване на дадена вградена система. Накрая, в част *2.3* се описват основните предимства на предлагания метод.

2.1 Дефиниране на проблемите при тестване на софтуер за автомобилната промишленост

Бързото нарастване на сложността на софтуера за вградени системи прави тестването важна част от разработването им. Динамичното тестване (т.е. пускането за изпълнение на тествания обект с избрани последователности от данни с цел анализ на поведението му при тези условия) е най-важният и разпространен метод за оценка на качеството на даден продукт.

Изчерпателното тестване на всички състояния на софтуера е неизпълнимо. За това трябва да се създаде множество от тестови сценарии, което да е възможно най-малко, но същевременно да е в състояние да открива възможно най-много грешки в тествания обект. Изборът на подходящи тестови сценарии е определящ за размера и качеството на тестовете.

Заради изключителното значение на избора на тестови сценарии, възниква нуждата от систематични техники, които да подпомагат тестера при определянето на подходящи сценарии. Използването на техники за определяне (дизайн) на тестовите стъпки прави избора на тестови сценарии изчерпателен и лесен за възпроизвеждане.

Освен подходящи техники за определяне на тестовете, възниква и нуждата от подходяща нотация за документиране на избраните тестови сценарии. Дизайнът на сценариите е дейност, която се осъществява предимно ръчно и за това изисква нотация, която да улеснява тестера.

Обикновено при тестването на вградени системи за автомобилната промишленост се използва произволно определяне на тестовите сценарии (*ad-hoc* техника) – т.е. сценариите се избират без предварително установени техники. Така дизайнът зависи изключително от опита и познанията на тестера и възпроизвеждането му е много трудно. Случайното определяне на тестовите сценарии може да доведе, от една страна, до излишество в тестовете и, от друга, страна до пропуски в тях. Поради тази причина е невъзможно да се определи качеството и пълнотата на тестовете. Също така, по отношение на броя на тестовите сценарии – тяхната способност за откриване на грешки често е незадоволителна.

Нотациите за описание на тестовете при разработката на софтуер за автомобилната промишленост са най-често на директни описания на тестовите сценарии във формата на зависимо от времето развитие на поведението на тествания обект. Това води до описание на тестовете на много ниско ниво на абстракция, което прави поддръжката и повторното им използване трудно.

Методът на класификационните дървета за вградени системи (МКД/ВС) е техника за дизайн на тестови сценарии за функционално тестване, чиято цел е да се справи с дефинираните проблеми. Той позволява систематичното определяне и описание на тестовите сценарии и на необходимите входни данни за тях. Този метод позволява също така да се направи графично описание на зависими от времето тестови сценарии. Това се осъществява чрез използването на сигнали за стимулиране на тествания обект, които се

дефинират постъпково за всяка входна променлива. Основната идея на МКД/ВС е да се раздели областта от входните данни на тествания обект спрямо неговите характеристики, които съответстват на различните източници на входните данни. Различните деления (всяко едно отговаря на една характеристика на обекта) се наричат *класификации*. Те от своя страна се разбиват на *класове на еквивалентност*. Накрая се формират различни комбинации от класовете на всяка от класификациите и така се определят тестовите последователности.

2.2 Описание на етапите на метода на класификационните дървета за определяне на тестови сценарии

В тази глава се дефинират отделните етапи на предлагания метод МКД/ВС и се обяснява всеки един от тях чрез използването му за дизайн на тестови сценарии за един примерен компонент. Обектът, който е цел на дизайна на тестовете, е функционалност на бордови компютър. Неговата задача е преобразуването на единиците за покритото разстояние *Cov_Distance*, средната скорост *Avg_Speed* и средния разход *Avg_Consumption* по избран маршрут. Тези единици зависят от избора на единиците за разстояние *Dist_units* – километри (*Dist_units* = 0) или мили (*Dist_units* = 1), за консумация *Cons_units* - единици за обем спрямо 100 единици за разстояние (*Cons_units* = 0) или единици за разстояние спрямо единици за обем (*Cons_units* = 1), както и от единиците за обем *Vol_units* – литри (*Vol_units* = 0) или галони (*Vol_units* = 1). Бордовият компютър приема стойностите на *Avg_Speed, Cov_Distance и Avg_Consumption* в мерни единици съответно км/ч, км и л/100км, след което ги преобразува в желаните от шофьора чрез функцията си за преобразуване на единици *Units_Converter* като спазва следните изисквания (Таблица 4):

Таблица 4: Систематизиране на изискванията към функцията за преобразуване на единици

Номер на изискване	Описание на изискване
SR-OBC-001	Преобразуването на единиците се извършва по правилата: - 1 галон = 4.54 литра - 1 миля = 1.609 километра
SR-OBC-002	Ако *Dist_units* = километри, тогава единиците на *Avg_Speed* са км/ч. Иначе, *Avg_Speed* се показва в единици мили/ч.
SR-OBC-003	Минималната стойност, която може да приема *Cov_Distance* е 0, а максималната е 50000. Невалидна стойност е 65535.
SR-OBC-004	Минималната стойност, която може да приема *Avg_Speed* е 0, а максималната е 250. Невалидна стойност е 255.
SR-OBC-005	Минималната стойност, която може да приема *Avg_Consumption* е 0, а максималната е 30. Невалидна стойност е 65535.
SR-OBC-006	Ако *Cons_units* = л/100км и *Vol_units* = литри и *Dist_units* = километри, тогава *Disp_Dist = Cov_Distance* *Disp_Speed = Avg_Speed* *Disp_Cons = Avg_Consumption* (с резолюция 0.1)

SR-OBC-007	Ако *Cons_units* = км/л и *Vol_units* = литри и *Dist_units* = километри, тогава *Disp_Dist* = *Cov_Distance* *Disp_Speed* = *Avg_Speed* Ако *Avg_Consumption* <> 0, тогава *Disp_Cons* = *100/Avg_Consumption* Иначе *Disp_Cons* = 999 (с резолюция 0.1, ако *Disp_Cons* <100 и резолюция 1, ако *Disp_Cons* >= 100).
SR-OBC-008	Ако *Cons_units* = мили/галон и *Vol_units* = галони и *Dist_units* = мили, тогава: *Disp_Speed* = *Avg_Speed* / 1.609 *Disp_Dist* = *Cov_Distance* / 1.609 Ако *Avg_Consumption* <> 0, тогава *Disp_Cons* = (100 / *Avg_Consumption*) * (4.54 / 1.609) Иначе *Disp_Cons* = 999 (с резолюция 0.1, ако *Disp_Cons* <100 и резолюция 1, ако *Disp_Cons* >= 100).
SR-OBC-009	Всички останали комбинации на *Cons_units, Vol_units* и *Dist_units* са невалидни и тогава се взимат единиците по подразбиране: *Cons_units* = л/100км *Vol_units* = литри *Dist_units* = километри
SR-OBC-0010	Ако *Dist_units* > *9999,* тогава стойността на *Disp_Dist* показва като първата цифра се отрязва *[N]NNNN* – например, ако *Dist_units* = 23456, *Disp_Dist* = 3456.
SR-OBC-0011	За *Cov_Distance* стойността 65535 е невалидна и в този случай *Disp_Dist* показва "_ _ _ _" За *Avg_Speed* стойността 65535 е невалидна и в този случай *Disp_Speed* показва "_ _._" За *Avg_Consumption* стойността 255 е невалидна и в този случай *Disp_Cons* показва "_ _ _" Всички стойности за трите променливи, които не са дефинирани и попадат извън интервала от възможни стойности се считат за невалидни.

2.2.1 Описание на интерфейса на тествания обект

Първата стъпка на метода на класификационните дървета е *описание на интерфейса на тествания обект* - т.е. да се определи интерфейса, който е от значение за тестовете. Предполага се, че $\underline{n}$ входни променливи стимулират тествания обект и той извежда $\underline{m}$ изходни променливи. Множеството I от входни променливи се представя чрез променливите i_k $(k = 1..n)$, а множеството от изходни променливи O - чрез променливите o_j $(j = 1..m)$. Променливите, които реално се използват в тестовете, се наричат *ефективен (тестов) интерфейс* – $I \cup O$. За целите на тестването всяка променлива от ефективния интерфейс трябва да се стимулира със сигнали, които са функция на времето. По този начин ефективният интерфейс определя *сигнатурата* на тестовите сценарии – т.е.

техните характерни особености. Нека с I_k и O_j означим областите от допустими стойности за всяка входна променлива i_k и всяка изходна променлива o_j. При тези предположения сигнатурата на тестовия сценарий ще е:

$$\boldsymbol{I} : i_{1,1} : Време \rightarrow I_1 \quad \ldots \quad i_{n,1} : Време \rightarrow In \qquad \boldsymbol{O} : o_{1,1} : Време \rightarrow O_1 \quad \ldots \quad o_{m,1} : Време \rightarrow O_m$$

Така за функцията *Units_Converter* ефективният интерфейс ще е показаният в Таблица 5.

Таблица 5: Описание на интерфейса на функцията Units_Converter

променлива	входна/изходна	единици	домейн	тип
Cons_units	входна	-----	{0, 1}	булева
Dist_units	входна	-----	{0, 1}	булева
Vol_units	входна	-----	{0, 1}	булева
Cov_Distance	входна	километри	[0, 50000]	цяло число
Avg_Speed	входна	км/ч	[0, 250]	реално число
Avg_Consumption	входна	л/100 км	[0, 30]	реално число
Disp_Dist	изходна	-----	[0, 9999]	цяло число
Disp_Speed	изходна	-----	[0, 250]	реално число
Disp_Cons	изходна	-----	[0, 999]	реално число

Сигнатурата на тестовите сценарии за функцията *Units_Converter* ще е:

$$\boldsymbol{I} : Cons\_units, Dist\_units, Vol\_units : Време \rightarrow \{0, 1\}$$
$$Cov\_Distance : Време \rightarrow N_{[0, 50000]}$$
$$Avg\_Speed : Време \rightarrow R_{[0, 250]}$$
$$Avg\_Consumption : Време \rightarrow R_{[0, 30]}$$

$$\boldsymbol{O} : Disp\_Dist : Време \rightarrow N_{[0, 9999]}$$
$$Disp\_Speed: Време \rightarrow R_{[0, 250]}$$
$$Disp\_Cons : Време \rightarrow R_{[0, 30]}$$

2.2.2 Разбиване на входните данни

Вторият етап е *разбиване на входните данни*, които ще се използват в тестовите сценарии. Това е етапът, по време който областите от допустими стойности на променливите, образуващи ефективния входен интерфейс, трябва да се разделят на напълно изключващи се (непресичащи се) части – класове на еквивалентност. Тези класове са подходяща абстракция на конкретните стойности на входните променливи, които ще се използват за целите на тестването. Разделянето се представя графично под формата на класификационно дърво. Целта на това разбиване е класовете да се изберат по такъв начин, че тестваният обект да се държи по един и същи начин за всеки от елементите на даден клас. Т.е., за всяка стойност от един клас на еквивалентност поведението на тестваният обект е еднакво – или се държи според спецификацията или има отклонения от желаното поведение. Това предположение може да се направи на базата следствието от *хипотезата за еднаквост (uniformity hypothesis)*.

За постигане на това разделяне се използва следната *евристична процедура*.

Типът на входните данни и техните области от допустими стойности служат за основа на разбиването на класове на еквивалентност. Ако възможните стойности на дадена променлива са зададени с интервал, за всяка от крайните стойности на интервала трябва да се създаде по един клас на еквивалентност и един клас за вътрешността на интервала. Тези класове са специфични за типа на данните и не са достатъчни за създаване на изчерпателни и систематични тестови сценарии. Класовете, създадени на базата на типа данни, трябва да се преработят допълнително, като се вземе предвид и спецификата и значението на входната променлива (контекста, в който се използва входната променлива).

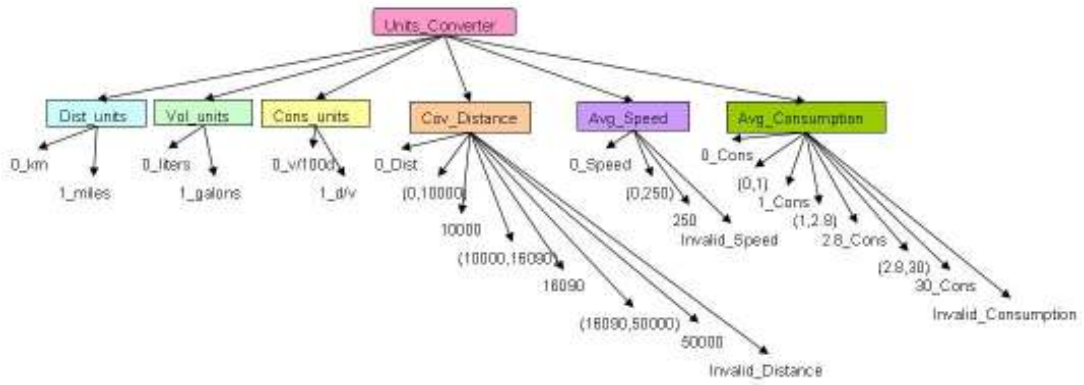
В разглеждания пример за функцията *Units_Converter*, на базата на стандартното разделяне на входните данни, описано по-горе, за променливата *Cov_Distance* може да се определят три класа – *0, (0, 50000)* и *50000*. За променливата *Avg_Speed* класовете са *0, (0, 250)* и *250*. За променливата *Avg_Consumption* могат да се дефинират класовете *0, (0, 30)* и *30*. Останалите променливи - *Cons_units, Dist_units u Vol_units* могат да приемат стойности 0 или 1 и по тази причина за тях се определят по два класа на еквивалентност за всяка от двете стойности.

На тези три променливи не може да се направи по-детайлно разбиване. От друга страна, за променливата *Cov_Distance* е нужно да се провери поведението на системата в случай, че въведената стойност е 10000 км и избраните единици за разстояние са километри. Стойността 16090 км в случай, че избраните единици са мили, също е специфична. Нужно е да се създаде и един клас, който да включва всички недефинирани (невалидни) стойности. Така за *Cov_Distance* се получават следните класове: *0, (0, 10000), 10000, (10000, 16090), 16090, (16090, 50000), 50000* и *Invalid_distance*.

Променливата *Avg_Speed* няма специфични стойности, освен тези в интервала [251, 255], които са невалидни и за който трябва да се създаде един клас на еквивалентност *Invalid_speed*. Получените класове за *Avg_Speed* са *0, (0, 250), 250* и *Invalid_speed*.

За да се определят по-детайлните класове за променливата *Avg_Consumption* трябва да се разгледат трите случая, който са възможни за единиците за консумация – л/100 км, км/л и мили/галон. Ако единиците са л/100км – специфични стойности няма и не е нужно допълнително разбиване. При единици км/л трябва да се проверят по отделно стойностите на *Avg_Consumption*, за които 100 / *Avg_Consumption* < 100 и 100 / *Avg_Consumption* >= 100 – т.е. когато *Avg_Consumption* е равно на 1. При избрани единици мили/галон трябва да се проверят стойностите, за който (100 / *Avg_Consumption*) * (4.54 / 1.609) < 100 и (100 / *Avg_Consumption*) * (4.54 / 1.609) >= 100 – т.е. когато *Avg_Consumption* е равно на 2.8. Още един клас е необходим, който да представя множеството от невалидни стойности – *Invalid_consumption*. Така се получават следните класове на еквивалентност за *Avg_Consumption* : *0, (0, 1), 1, (1, 2), 2, (2, 30), 30* и *Invalid_consumption*.

Разбиването на класове може да се представи чрез класификационното дърво показано на Фигура 7.



Фигура 7: Класификационното дърво за примера Units_Converter

2.2.3 Определяне на тестовите сценарии

Третият етап на метода на класификационните дървета е *определяне на тестовите сценарии* на базата на разбиването на входните данни, направено в предишния етап. Тестовите сценарии описват по абстрактен начин промените във входните данни в зависимост от времето. Всеки сценарий се състои от абстракции на входните данни на тествания обект (класовете в класификационното дърво) и по този начин описва какво ще се тества. Това описание е независимо от конкретните входни данни.

Класификационното дърво служи за определяне на колоните на таблицата на комбинациите. Всеки тестов сценарий се състои от стъпки, които формират редовете в таблицата на комбинациите, като са подредени в зависимост от последователността си във времето. Ситуациите, описващи всяка стъпка, се дефинират като се комбинират класове от различните класификации, които се маркират в таблицата на комбинациите. По този начин се получават поредици от входни ситуации. Продължителността на всяка стъпка също може да се зададе в таблицата на комбинациите. Комбинациите от маркирани класове за всяка стъпка определят входните данни на тествания обект за дадения момент от време.

Поведението, зависещо от времето - т.е. промяната на входните стойности спрямо времето - може да се моделира като за всяка стъпка се маркират различни класове от една класификация. Изменението на сигнала, отговарящ на дадена входна променлива, между две последователни стъпки се дефинира чрез преходи между маркираните класове от една и съща класификация. Различните типове преходи представят различните форми на сигналите – линейна функция, синусова функция, косинусова функция, сплайн функция. По този начин последователностите от входни стойности, които се използват в тестовете, се описват абстрактно с помощта на постъпково дефинирани функции.

След като е завършило определянето на тестовите последователности е нужно да се провери дали те осигуряват необходимото покритие. Методът на класификационните дървета осигурява проверка на покритието на тестовете на много ранен етап от създаването на тестовете. Анализ на покритието на изискванията може да провери дали всички изисквания от спецификацията са включени в подходящи и изчерпателни тестови последователности. Най-общо, между изискванията и тестовите сценарии съществува връзка *n:m*. В хода на анализа трябва да се докаже, че всяко изискване се проверява най-малко от един сценарий и всеки съществуващ сценарий тества адекватно и ефективно съответното изискване.

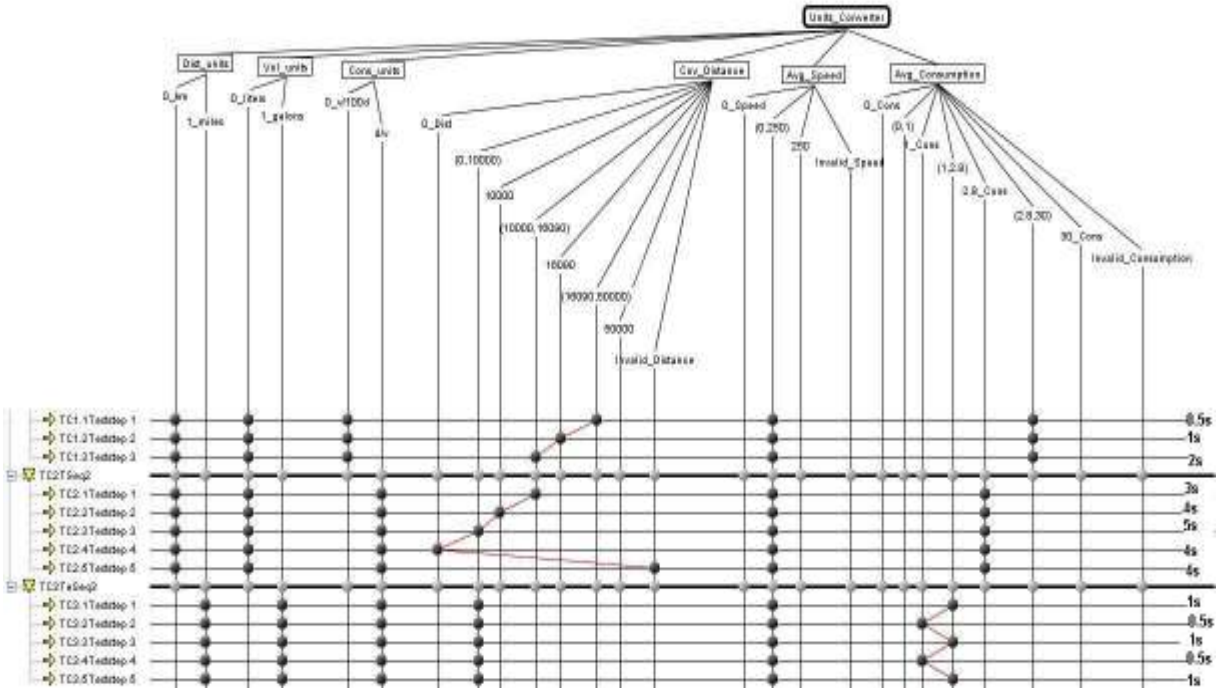
Освен това, МКД/ВС поддържа и възможност за анализ на обхвата на покритието. Този анализ проверява дали всички класове на еквивалентност дефинирани в класификационното дърво са взети подходящо предвид в тестовите последователности. Тази проверка може да бъде направена като се използват различните критерии за покритие на класификационното дърво:

- Критерият за минималност изисква всеки клас от дървото да бъде избран в поне една тестова стъпка. Този критерий обикновено се постига с малко на брой тестови последователности. Степента на откриване на грешки, обаче, е доста ниска.
- Критерият за максималност изисква всяка възможна комбинация от класове да се съдържа в поне една стъпка. Удовлетворяването на този критерий води до много висока степен на откриване на грешки. Поради рязкото увеличаване на пълния брой на възможните комбинации, този критерий е приложим само, когато броят на класовете на еквивалентност е малък.
- Критерият за оптимално комбиниране (n-wise combination criteria) е компромис между двата споменати. При него е необходимо да се осигури, че всяка възможна комбинация между n дадени класа е избрана в поне една стъпка.

Изборът на подходящ критерий за покритие трябва да се вземе предвид по време на планирането на тестването, като водеща е спецификата на тествания обект. Ако предварително дефинираният критерий не е достатъчно удовлетворен, трябва да се добавят допълнителни тестови стъпки или последователности, докато се удовлетвори критерият. [11]

За примера с функцията Units_Converter класификационното дърво и таблицата

на комбинациите с дефинирани тестови сценарии ще изглежда по начина показан на Фигура 8:



Фигура 8: Класификационното дърво и таблицата на комбинациите с дефинирани тестови сценарии за функцията Units_Converter

Целта на първия тестов сценарии (Фигура 8, TSeq1) е да провери поведението на функцията при промяна на оставащото разстояние, при положение, че избраните единици за разстояние за километри, за обем – литри и за консумация – единици за обем спрямо 100 единици за разстояние. Скоростта на автомобила, както и консумацията, са постоянни.

Във втория тестов сценарии (Фигура 8, TSeq2) се разглежда поведението на функцията в случай че на входа ѝ се получи невалидна стойност. Избраните единици за разстояние отново са километри, за обем – литри, а за консумация са единици за разстояние спрямо единици за обем.

Третият тестов сценарий, описан на Фигура 8, TSeq3, разглежда случая, когато консумацията на гориво се променя на интервали 0.5 и 1 секунда, като избраните единици са мили, галони и единици за разстояние спрямо единици за обем.

2.2.4 Определяне на конкретните тестови данни

По време на четвъртия последен етап абстрактните данни от тестовите сценарии, определени в третия етап, трябва да се заменят с реални числови стойности – т.е. това е етапът, на който става *определянето на реалните (конкретните) тестови данни*. Определеното абстрактно развитие на всяка променлива във времето служи като основа за определяне на конкретните входни сигнали за всяка променлива. Интервалите, дефинирани с класовете на еквивалентност, формират ограниченията върху стойността на дадена променлива за дадена стъпка. Така на базата на хипотезата за еднаквост може да се избере всяка една стойност, попадаща в обхвата на маркирания клас. [10]

2.3 Предимства на метода на класификационните дървета при определянето на тестови сценарии

Изборът на подходящи тестови сценарии – т.е. такива, които откриват възможно

най-много дефекти – е от особено важно значение за тестването на софтуер за вградени системи. Той определя обхвата и качеството на тестването. Методът на класификационните дървета за вградени системи предоставя нужните средства за систематично и изчерпателно тестване на вградени системи. [10] Чрез него може да се направи графично описание на тестовите сценарии, като по този начин те стават лесни за проверка както от автора им, така и от други тестери. Това прави дизайна на сценариите лесен за възприемане, поддръжка и многократно използване.

Особено важна задача при прилагането на МКД/ВС е анализирането на спецификацията на тествания обект. Така спецификацията се проверява задълбочено за възможни пропуски, неточности и противоречия. Например ако очакваното поведение на тестовия обект не може да се определи за дадена ситуация, очевидно има пропуск в спецификацията.

Когато се създават тестове за дадена функционалност, възникват много идеи за това, което трябва да се тества. Тестерът трябва да структурира идеите си, за да се избегнат повтарящи се и излишни сценарии. Ако е възможно, трябва да се минимизира броя на тестовете с цел избягване на стъпки, тестващи само една, а не множество функционалности. МКД е много полезен в това отношение. При разделянето на обхвата на всяка променлива на класове става ясно колко точно стойности на всяка характеристика на тествания обект ще се тестват. Например няма нужда да се създават различни стъпки за стойности от един и същ клас на еквивалентност. Ако такава необходимост възникне, следователно този клас трябва да се раздели на подкласове.

МКД/ВС позволява пресмятането на някои метрики, свързани с тестовете – например минималния и максималния брой от тестови стъпки, нужни за покриването на дадено класификационно дърво. По този начин на много ранен етап от разработването на продукта може да се оцени необходимото време за изпълняване на тестовете. [22]

Методът на класификационните дървета може да служи като обобщение и средство за документиране на резултатите от използването на други техники за определяне на тестови сценарии – например гранични стойности. По този начин МКД може да се използва като универсално средство за описание на почти всички сценарии, които възникват като част от тестването на софтуер за вградени автомобилни системи. [10]

При използването на МКД/ВС тестовите сценарии се описват по абстрактен начин, отделно от реалните данни, които ще се използват по-нататък в процеса на тестване. Това прави сценариите лесни за поддръжка и модификация при промяна на изискванията в спецификацията.

Тъй като вградените автомобилни системи трябва да работят в реално време и с дълги последователности от входни данни, едно от основните изисквания при тестването им е проверката на продължителната работа на системата при определени ситуации, описани последователно във времето. Методът на класификационните дървета предоставя средства за описание на тестови сценарии, които за зависими от времето – за всяка стъпка от даден тестов сценарий се определя нейната продължителност (абсолютно време) или отместването ѝ във времето спрямо предхождащата я стъпка (относително време).

Вградените системи работят в реално време и за това е изключително важно техните реакции на дадени входни данни да отговарят на изисквания към времето за отговор - отговорът не трябва да пристига нито твърде рано, нито твърде късно. Функционалното тестване на вградени системи с помощта на МКД/ВС може лесно да се комбинира с тестване на времевите ограничения с използването на еволюционни алгоритми. Първоначално се проверява дали системата работи логически правилно. След това тестовете, систематично определени при това тестване, се използват като основа (начална популация) за търсене с използване на еволюционни алгоритми на последователности от входни ситуации с много дълго или много кратко време за изпълнение и се проверява за появата на грешки при тях. По този начин се прави задълбочен и детайлен анализ на поведението на тестваната система във времето. [20]

Подходящо построено класификационно дърво и адекватно определени спецификации на тестови сценарии осигуряват, че избраното множество от тестове е

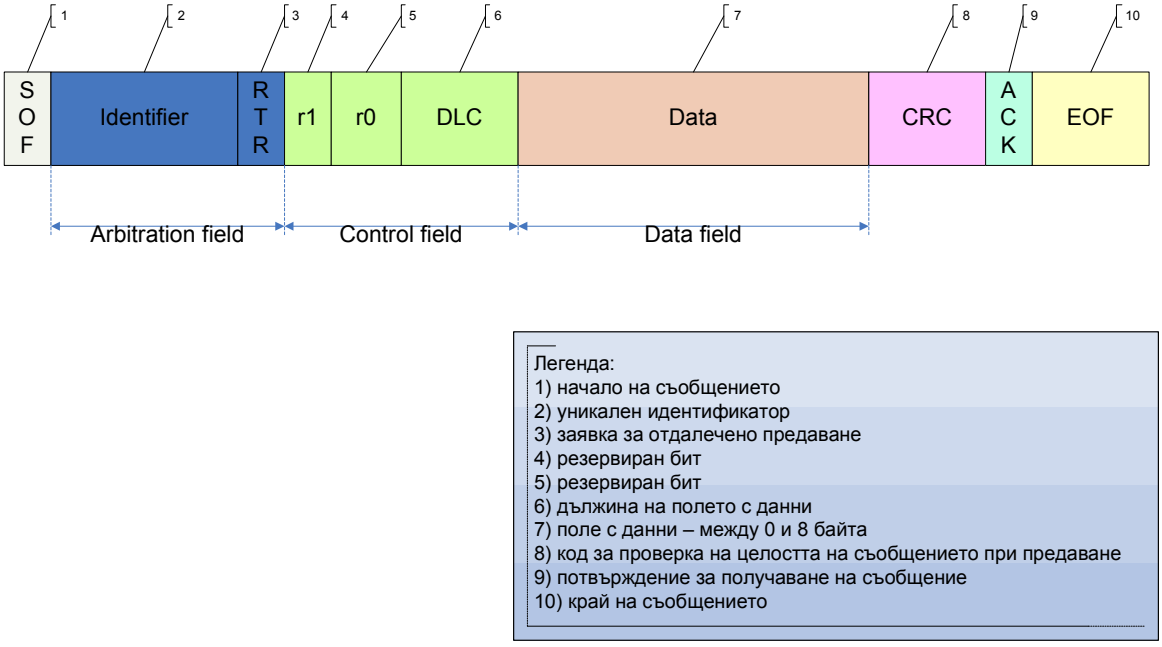
подходящо и ефективно – т.е. тези тестове ще открият (почти) „всички" дефекти в софтуера. Ако дизайнът на тестовите сценарии е преминал през един или няколко прегледа (reviews), това твърдение може да се каже с по-голяма увереност. В този случай е лесно да се определи кога е надеждно да се прекрати тестването – това е моментът, в който приключва изпълнението на планираните тестове. Ако не съществува такова надеждно множество от тестове, е трудно да се определи дали трябва да се изпълняват допълнителни тестове. [22]

Глава 3. Описание на изследваната функционалност и текущо състояние на тестовия процес

За целите на дипломната работа в Глава 3 е разгледана вградена система за автомобилната промишленост, разработвана от фирмата Johnson Controls Electronics. Тя е описана подробно в част *3.1*. Обръща се внимание на една конкретна функционалност на (част *3.2*). Накрая, в част *3.3*, се разглежда текущото състояние на тестовия процес във фирмата.

3.1 Общо описание на системата

В съвременните автомобили различните устройства са свързани помежду си най-често в CAN (*Controller Area Network*) мрежа, която се състои от подмрежи и подсистеми. Връзката между подмрежите и предаването на съобщения между тях се осъществява посредством маршутизатор (*router*). Целта на подмрежата е да предава съобщения между устройствата в нея. Всяко съобщение (message), идващо от някое от устройствата има идентификатор, който е уникален в цялата мрежа и определя както съдържанието (структурата на данните в съобщението), така и приоритета на съобщението. Съобщенията могат да съдържат от нула до осем байта данни. Формата им е показан на Фигура 9 [25].



Фигура 9: Структура на съобщенията, предавани в CAN мрежата

Системата, която ще участва в експеримента, представлява мултифункционален екран (Ecran MultiFonctions), предназначен за автомобили от различни продуктови линии за голямата френска автомобилна компания PSA Peugeot Citroën. Системата има различни модификации в зависимост от продуктовата линия, за която е предназначена. Състои се от хардуерна и софтуерна част.

Мултифункционалният екран е част от описаната CAN мрежа в автомобила и по-точно от CAN Low Speed (125kbps). Връзката между Low Speed и High Speed подмрежите се осъществява с използването на маршрутизатор - т.нар. *Intelligent Service Box*. Той получава информация от устройствата в High Speed мрежата и ги предава към Low Speed мрежата и обратно. Екранът получава информация от останалите устройства в Low Speed

мрежата и я визуализира в зависимост от различните приоритети. Той комуникира с тях като получава и изпраща съобщения. Те могат да се изпращат периодично (на определен интервал от време) и при настъпването на определено събитие.

На Фигура 10 е показана схема на подмрежата, в която се намира мултифункционалния екран.

Фигура 10: Връзка на мултифункционалния екран с останалите устройства в CAN мрежата

Хардуерната част на мултифункционалния дисплей се състои от три основни елемента:

- процесор
- EEPROM (Electronic Erasable Programmable Read Only Memory) – памет за съхранение на информация, по начин независим от прекъсване на захранването
- TFT дисплей – изходно устройство за визуализиране на информация
- CAN контролер – осигурява получаването и изпращането на съобщения по CAN мрежата

Софтуерната част на екрана се състои от функционален и интерфейсен елемент. Функционалният елемент се грижи за получаване и обработване на съобщенията от другите устройства в мрежата, след което препредава информацията към интерфейсния елемент. Той от своя страна се грижи да визуализира получените данни на цветен TFT дисплей. Негова е задачата да покаже необходимите икони и надписи и да отвори менюта и прозорци на екрана в зависимост от съдържанието на получените съобщения.

Основните функционалности на системата са:

- показване на текущия час и дата – устройството само се грижи за управлението на часа и датата
- показване на външната температура
- помощ при паркиране
- показване на различни предупредителни и информационни съобщения, идващи от други устройства в колата. Например, предупреждения спукана гума, за отворена врата или разкопчан колан, за повреда в скоростната кутия, ръчната спирачка, за активация/деактивация на дадена функция и други.
- възможност за записване на тези съобщения в списък, който позволява достъп до всички налични в момента предупредителни и информационни съобщения, сортирани в реда на появяването си.
- възможност за промяна на състоянието (активиране/деактивиране) на различните функции на колата като въздушна възглавница, помощ при паркиране,

автоматично заключване на вратите, електронна стабилизираща програма (ESP) и други.

- показване на информацията, идваща от (авто)радиото и CD Changer. Екранът показва текущия избран аудио сорс – радио, CD, CD Changer – и свързаната с него информация като честота и име (ако е налично) на слушаната станция, списъци със запаметени станции, песни и албуми, информация за слушаната в момента песен. Чрез различните елементи на менюто на екрана могат да се активират и деактивират опции като пускане на песните от CD в произволен ред, пускане в режим IntroScan, радио-текст, RDS информация и други. Информация за състоянието на силата и други настройки на звука също се визуализират на екрана.
- показване на информацията от бордовия компютър – целта на тази функционалност е да информира шофьора за оставащото гориво, консумация, оставащо разстояние и други. Тази информация е налична за два различни маршрута или за текущото състояние. Потребителят може да избере единиците, в които ще се визуализира информацията.
- показване на информацията от климатика – тази функция визуализира състоянието на климатика във всеки момент – изключен/включен, автоматичен/ръчен режим, текущата температура в колата, посоката на вентилацията, филтриране на въздуха. Чрез опциите в менюто е възможно активирането/деактивирането на компресора, активирането/деактивирането на едновременното управление на двете зони на климатика, както и програмирането на климатика да стартира работата в определено време.
- показване на информацията от handsfree устройство, свързващо се с мобилен телефон посредством Bluetooth – тази функционланост позволява да се избере телефона, с който ще се осъществи връзката, да се отговори или откаже разговор, да се проследят етапите на набиране на даден номер от телефонния указател, да се преглеждат списъците с телефонни номера, с обаждания (набирани номера, приети и пропуснати повиквания).
- конфигуриране на автомобила – тази функционалност позволява на потребителя да конфигурира колата според предпочитанията си – например активиране/деактивиране на опционални устройства в колата като автоматично включване на фаровете, адаптация на фаровете в завой и други. Също така позволява и запазване на настройките в различни профили и зареждането им на по-късен етап.
- избор на език, на единици за консумация и температура – потребителят може да избира измежду осем езика. Функционалната част на екрана се грижи за преобразуването на единиците до тези избрани от потребителя – километри за литри/мили за галон/литри за 100 км и Целзий/Фаренхайт.

3.2 Описание на функционалността, включена в експерименталното изследване

Функционалността, която ще е обект на експерименталното изследване, е визуализиране на информацията от Помощ при паркиране. Нейната цел е информиране на шофьора за разстоянията между автомобила и заобикалящата го среда – намиращи се в близост други автомобили или различни видове препятствия. В тази част ще бъдат разгледани основните характеристики на тази функционалност, а конкретните изисквания към нея ще бъдат описани в *Приложение А*.

Мултифункционалният екран получава информация за намиращи се близо до автомобила предмети чрез обмяната на съобщения със системата Помощ при паркиране. Тя от своя страна се състои от шест сензора разположени върху автомобила – три в предната и три в задната му част. Те следят за наличието на препятствия на определени разстояния от автомобила.

Визуализирането на информацията от Помощ при паркиране се осъществява по желание на шофьора. Разстоянието между автомобила и заобикалящите го предмети е

разделено на множество зони. При появата на препятствие в определена зона, тази зона се визуализира на дисплея. В случай, че зоната е много близо до автомобила освен маркирането й, се визуализира и знак, предупреждаващ за опасност. Прозорецът, върху които се показва информацията от Помощ при паркиране, има изключително висок приоритет и припокрива информацията от повечето останали системи в автомобила. По-висок приоритет от него имат единствено прозорците за приемане или отказване на разговор по телефона и прозореца, информиращ за стартиране на икономичния режим на автомобила – например при отслабване на акумулатора.

3.3 Текущо състояние на тестовия процес

Системното тестване (валидация) на мултифункционалния екран е базирано на функционалния (black-box) подход като целта му е да провери дали софтуера изпълнява дефинираните изисквани. Процесът на валидация се състои от два основни етапа – *подготовка и изпълнение*. По време на етапа на подготовка се дефинира стратегията за тестване (описана в тест-плана) и се създават тестовите сценарии, след което се преминава към изпълнението им. Тези етапи ще бъдат разгледани подробно в част *3.3.1, 3.3.2* и *3.3.3*. Инструментите, използвани по време на тестването, ще бъдат детайлно разгледани в *Приложение Б*.

В началото на всяка итерация от тестовия процес лидерът на тестовия екип подготвя тест-плана (Software Validation Plan), като за негова основа се използват клиентските изисквания (requirements) и плана за интеграция (Integration Plan). Тест-планът е документ, който описва стратегията за тестване. В него се съдържа следната информация:

- описание на средата за тестване
- списък от функционалности, които ще се тестват
- задълженията на всеки един член от тестовия екип
- ресурсите, които ще се използват по време на фазите на тестването
- типовете тестове, които ще се изпълняват
- приложение на стратегия при регресия (*Regression Strategy)*
- хронология на изпълнението на тестовете
- критерии за успешно завършване на тестването

След като е дефинирана стратегията за тестване, се преминава към създаване на тестовите процедури. Всяка процедура се състои от множество тестови сценарии, което от своя страна се състои от предпоставки (preconditions) и тестови стъпки. Всяка тестова стъпка съдържа действия, очаквани резултати, тествано изискване и идентификатор на дефект в случай, че за тази стъпка е имало дефект в предишна софтуерна версия. Предпоставките определят средата, в която ще се изпълняват стъпките от даден тестов сценарии.

Дефинирането на отделните стъпки в тестовите сценарии се реализира чрез *ad-hoc* подход. Използват се комбинация на методите тестване с гранични стойности, тестване с произволни данни и интуитивно тестване. Няма точно определена процедура за избор на тестови стъпки и последователности от тях. Разчита се предимно на опита и интуицията на тестера, пишещ стъпките.

По време на тази фаза всички изисквания, които трябва да се реализират в текущата итерация, както и съществуващи дефекти, се взимат предвид и за тях се създават тестови сценарии или се добавят стъпки във вече съществуващи. Резултатът от този етап е документ, който се нарича *Software Validation Procedure*.

По време на изпълнението на тестове, за всяка дефинирана стъпка се определя резултат – *„OK*" или *„NOK*", който се определя при сравняването на очаквания и реалния резултат. Тестването се извършва ръчно или полу-автоматично с използването на инструмента AutoEMF, описан в *Приложение Б*. Ако определена стъпка не може да се изпълни, като резултат се поставя *„NT" (Not tested)*. Ако резултатът от дадена стъпка е *„NOK*", за нея трябва да се добави коментар, който да описва точно наблюдаваната ситуация, както и да се добави дефект в системата за управление на дефекти. Тестов сценарии е *„Passed*", когато всички стъпки в него са *„OK*". Във всички други случаи

тестовият сценарии се счита за *„Failed*". Резултатът от изпълнението на тестовете е тест-доклада (*Software Validation Report*). Той съдържа резултатите от изпълнените стъпки и резултатите от всички тестови сценарии. За всяка *„NOK"* стъпка като коментар трябва да е добавен идентификатора на дефекта от системата за управление на дефекти.

3.3.1 Стратегия при регресия (Regression strategy)

Целта на стратегията при регресия е да осигури, че качеството на софтуерния продукт не се е влошило след добавянето на нови функционалности или отстраняване на дефекти. За всяка софтуерна версия, която ще се тества, се изпълняват тестовите сценарии в следния ред:

- тестовите сценарии, които покриват промените – добавяне на имплементацията на нови и/или променени софтуерни изисквания (резултат от премахване на функционалности или оправяне на дефекти)
- тестовите сценарии, които не са покрити по време на Интеграционното тестване
- тестови сценарии, предложени от програмистите

3.3.2 Инкрементална валидация

Инкременталната валидация започва след като е завършила подготвителната фаза. По време инкременталната валидация на всяка софтуерна версия трябва да се изпълнят всички тестови сценарии дефинирани от стратегията при регресия. Резултатите от изпълнените тестове по време на инкременталната валидация трябва да се пазят в системата за управление на конфигурациите. Инкременталната валидация се изпълнява върху реално устройство, но понякога се допуска и изпълнение върху софтуерна симулация на устройството.

3.3.3 Пълна валидация

Пълната валидация започва след приключването на инкременталната валидация и след края на промените по кода (*code freeze*). Целта на пълната валидация е да се провери, че отстраняването на дефектите, намерени по време на инкрементралната валидация, не е довело до появата на нови. Също така трябва да се провери, че системата като цяло работи според клиентските изисквания и очаквания. По време на пълната валидация се изпълняват тестовете в следния ред:

- тестовите сценарии, определени от стратегия при регресия
- изпълнение на всички останали тестови сценарии определени в документа *Software Validation Procedure*.

Резултатите от тестовете се съхраняват в системата за управление на конфигурациите. Пълната валидация се изпълнява само на реално устройство. Резултатът от нея е тест-репорта (*Software Validation Report*). Критерий за успешно завършване на пълната валидация е всички клиентски изисквания да са тествани. Причините за нетествано изискване трябва ясно да се обосноват в тест-репорта.

3.4 Проблеми при съществуващия подход на тестване

Проблемите, които възникват при описания метод на тестване, са основно във фазата на създаване на тестовите сценарии. В процеса не е отделено специално време за дизайн на тестовете и по тази причина няма дефинирана процедура за избор на тестови стъпки. Какви ще са отделните стъпки и последователността им в тестовите сценарии се определя по време на самата реализация на сценариите. Разчита се на преценката и опита на автора, както и на произволен избор на тестови данни. Този подход за дефиниране на тестови стъпки често води до пропускане на важни ситуации, при които има голяма вероятност за проявяване на дефект. В други случаи едни и същи тестови стъпки се повтарят многократно в сценариите, при което се стига до излишество и ненужно удължаване на времето за изпълнението им. По този начин трудно може да се определи

кои изисквания за покрити и дали наистина се тестват пълно.

При този подход за дизайн на сценариите много рядко се получават последователности от стъпки, при които да се изследва поведението на системата в реални ситуации. При тестването с последователности от данни преди началото на тестовия сценарии се определя набор от предпоставки, които поставят системата в определено състояние. Тези предпоставки трябва да влияят единствено на първата стъпка. Всяка следваща стъпка трябва да зависи единствено от състоянието на системата, резултат от действията в предходната стъпка. При текущия подход често преди всяка стъпка системата се установява в определено състояние насилствено, след което се извършват необходимите действия. По този начин стъпките са изолирани една от друга и поведението на системата не може да се изследва при продължително и последователно подаване на тестови данни – т.е. при симулиране на реални условия на работа.

Тъй като не се използва систематичен избор на тестови стъпки, а се разчита на интуицията на тестера, проследяването на стъпките в сценариите е трудно. Това води до проблеми при промяна на изискванията към системата. Модифицирането на тестовите сценарии спрямо новите изисквания отнема много време, тъй като липсва документация за това какво и как се тества чрез съответните стъпки. При провеждането на прегледи на тестовите процедури също възникват проблеми, защото логиката, която следват тестовите сценарии, се проследява трудно поради липсата на ясно дефинирани цели на отделните стъпки.

Глава 4. Реализация на тестовите сценарии за изследваната функционалност с помощта на МКД

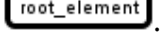
В тази глава се прави описание на реализацията на тестовите сценарии за функционалността Помощ при паркиране с помощта на метода на класификационните дървета и помощния редактор Classification Tree Editor eXtended Logics. В първата част *4.1* е направено описание на CTE XL и неговите основни функции. Обяснено е как се създава класификационно дърво, тестови стъпки и сценарии. Втората част (*4.2*) описва съображенията, които се имат предвид по време на дизайна на тестовите сценарии по метода на класификационните дървета – как става разбиването на множествата от допустими стойности на входните променливи, какви връзки между променливите се задават и други. В третата част (*4.3*) се описва подборът на конкретни стойности от избраните в предишната стъпка (част) класове.

4.1 Описание на Classification Tree Editor eXtended Logics

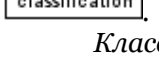
Методът на класификационните дървета е техника за функционално тестване, при която областта от входни данни на тествания обект се разделя на класове на еквивалентност в зависимост от различни характеристики. Classification Tree Editor eXtended Logics (CTE XL) е графичен редактор, създаден да подпомага и улеснява този метод. Той е разработен от автомобилния производител Daimler Chrysler AG с помощта на Java технологии. Двата основни етапа на МКД – разделянето на класове на еквивалентност и определянето на тестовите сценарии – се поддържат от CTE XL.

На Фигура 11 са показани частите на основния прозорец на приложението CTE XL. С номер 1 е отбелязано полето за създаване и редактиране на класификационно дърво. Това става чрез добавяне и изтриване на елементи (основен елемент, класификации, класове и композиции) и връзките между тях. С номер 2 е отбелязано полето за редакция на характеристиките на различните елементи на дървото – като име, описание, спецификация и други. За всеки елемент могат да се дефинират допълнителни „тагове" (tags), които съдържат определена информация за обекта. С номер 3 е означено полето, в което се намира списъкът с тестови сценарии. В него могат да се добавят тестови сценарии, тестови последователности и тестови стъпки. С номер 4 е отбелязано полето, в което е разположена таблицата на комбинациите. В нея класовете от класификационното дърво се комбинират помежду си и се свързват в тестови стъпки и сценарии.

Елементи на класификационното дърво са основен елемент (root_element), класификации (classifications), класове (classes) и композиции (compositions).

Основният елемент е всъщност корена на класификационното дърво и представя обекта, които ще се тества. Този елемент е единствен за цялото дърво и не може да се премахва. Негови наследници могат да са класификации и композиции. В нотацията на CTE XL, основният елемент се отбелязва като правоъгълник със заоблени ъгли - 

.

Класификациите представляват различните характеристики или входни променливи на тествания обект – като цвят, форма, размери и т.н.. Наследници на тези елементи могат да са класовете. Класификациите се отбелязват като правоъгълници - 

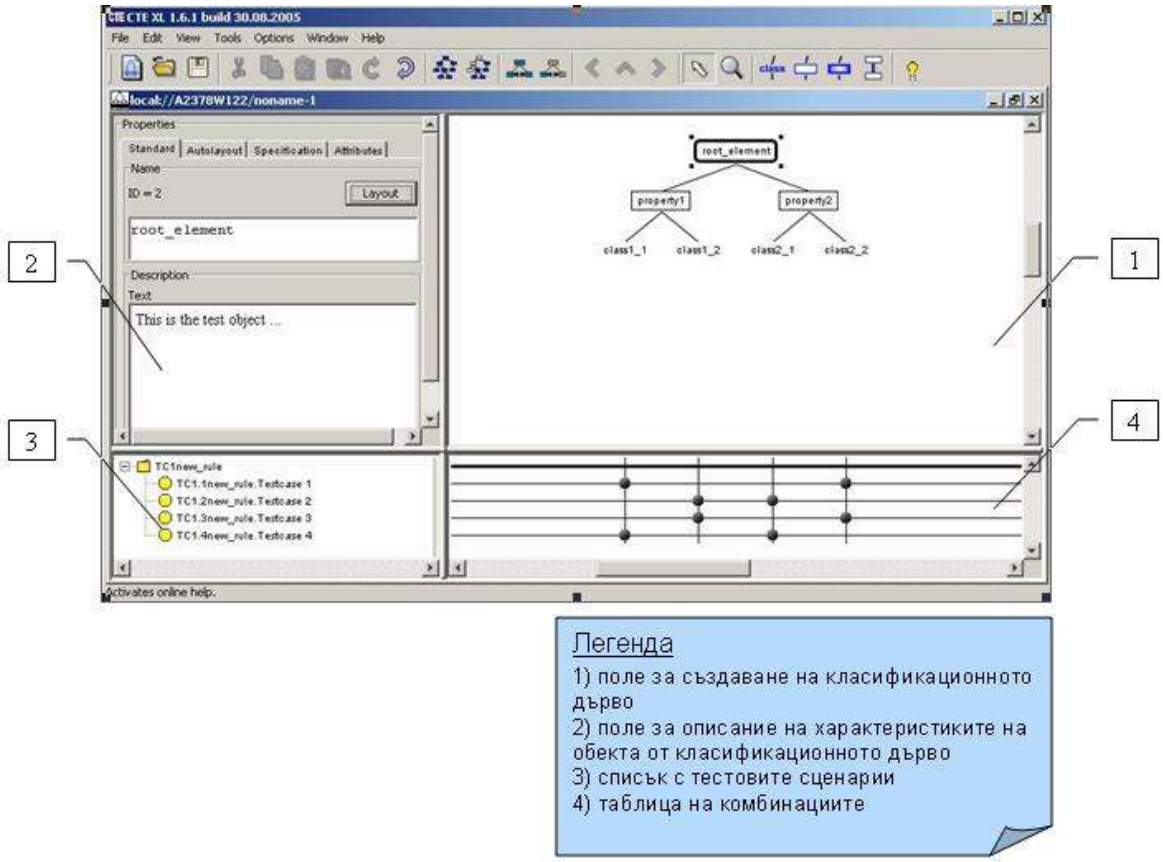
.

Класовете представляват областта от допустими стойности за всяка класификация, разделена на непресичащи се множества – т.е. класове на еквивалентност. Наследници на класовете в класификационното дърво могат да са класификации и композиции. Класовете, които са листа в дървото – т.е. нямат други наследници могат да се свързват с тестови стъпки в таблицата на комбинациите. Отбелязват се само с името си,

без рамка отстрани - class .

Композициите могат да обобщават няколко класификации. Техни наследници могат да са класификации и други композиции. Отбелязват се с правоъгълник като класификациите, но с по-плътна рамка - composition.

Някои части на класификационното дърво могат да се скриват – т.е. да се не се виждат в главното дърво. Това е полезно когато класификационното дърво стане много голямо и сложно. По този начин се създават *поддървета*. Такива „скрити части" се разпознават по стрелката в дясната част на елемента - classification , noname_23 или composition . [29]



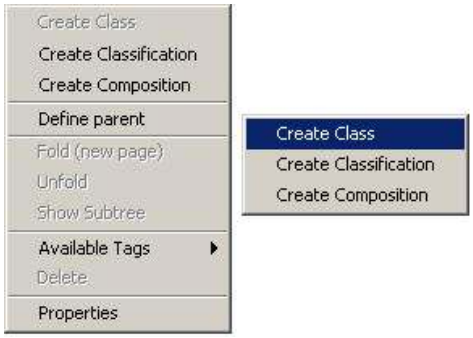
Фигура 11: Елементите на основния прозорец на CTE XL

4.1.1 Създаване на класификационно дърво

Има два начина за създаване на класификационно дърво: построяване като се тръгне от корена – дизайн отгоре-надолу (top-down design) или елементите се създават независимо и след това се свързват помежду си – дизайн отдолу-нагоре (bottom-up design).

За създаване на дърво е нужно да се създаде нов проект – *File → New*. CTE XL автоматично отваря нов прозорец, съдържащ полетата показани на Фигура 11. Основният елемент също автоматично се създава. Добавянето на елементи в дървото става или чрез избирането им от лентата с инструменти (toolbar) или чрез използване на едно от контекстните менюта показани на Фигура 12. В ляво е показано менюто, което се използва при метода отгоре-надолу, а в дясно – менюто, което се използва при метода отдолу-нагоре. Изтриването на елемент от дървото става отново или чрез лентата с инструменти или чрез контекстното меню.

Връзките между елементите могат да се осъществят автоматично при добавянето на елементи под елемента „родител” или да се добавят ръчно чрез лентата с инструменти или чрез контекстното меню като се оказва кой ще е „родител” на избрания елемент.



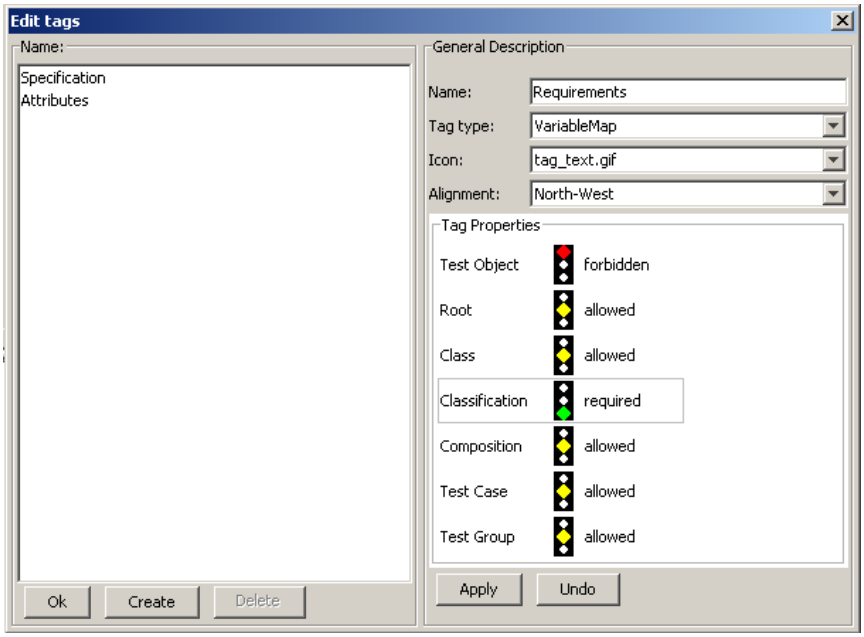
Фигура 12: Добавяне на елемент в класификационното дърво чрез контекстно меню

Специфичните за различните проекти атрибути могат да се документират в репорта, съдържащ дизайна на тестовите сценарии. Тези атрибути могат да се свържат с елементите на класификационното дърво чрез така наречените *тагове* (tags). Те се дефинират от *Tools → Tags* опцията в главното меню (Фигура 13). Има три възможности за даден таг :

- *забранен (forbidden)* – ако за някои елемент този таг е забранен, то той никога не се показва в полето за описани на характеристиките (Фигура 11) и съответно не може да се запълни с информация
- *позволен (allowed)* – ако за някой елемент този таг е позволен, то той може да се добави ръчно към елемента чрез неговото контекстно меню
- *задължителен (required)* – ако за някои елемент този таг е задължителен, то той е винаги показан в полето за описани на характеристиките (Фигура 11).

Трите основни типа тагове са:

- *TextualTag* – съдържа текстова информация
- *TreeLinkTag* – съдържа връзка към друг CTE XL файл
- *VariableMap* – съдържа списък от двойки променливи от типа атрибут-стойност.



Фигура 13: Добавяне на нови атрибути, специфични за отделните проекти

Чрез използването на тага *TreeLinkTag* е възможно да се импортират тестовите сценарии за друг тестов обект в текущото класификационно дърво. По този начин могат да се свържат тестовите сценарии за един тестов обект с различни свойства и класификационни дървета и да се тества спрямо различни гледни точки.

В много случай в практиката тестовите обекти са доста сложни и следователно класификационното дърво става много голямо и сложно. По тази причина е възможно части от него (поддървета) да се „сгънат" (fold) – т.е. да се сведат до един негов елемент. С тези поддървета може да се работи след това в техен отделен прозорец. За корен на поддърво може да се дефинира класификация, клас или композиция като се избере опцията *Fold (New page)* от контекстното меню.

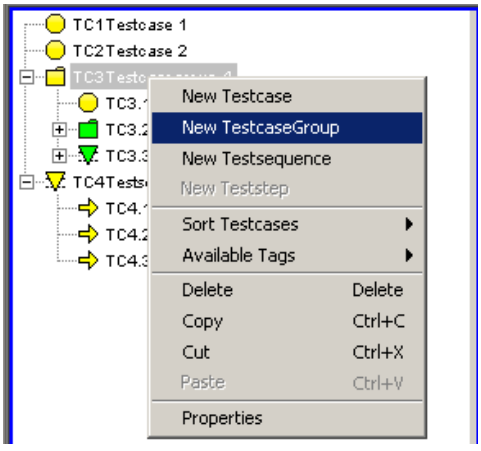
4.1.2 Създаване на тестови сценарии

Тестовите стъпки се създават чрез контекстното меню, което се появява в полето, съдържащо списъка с тестови сценарии, показано на Фигура 14. След като дадена стъпка е вече създадена, тя може да се свърже с определени класове от класификационното дърво чрез поставяне на маркери в таблицата на комбинациите. Маркерите са четири типа и техните значения са показани в следната таблица (Таблица 6):

Таблица 6: Типовете маркери използвани в таблицата на комбинациите

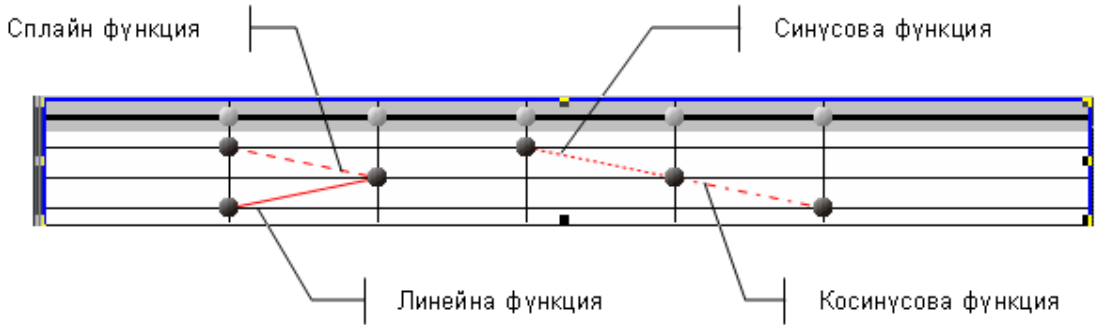
Тип маркер	Описание
	този маркер означава, че конкретния клас е свързан с тестовия сценарий
	този маркер - *Don't care marker* – се използва, когато е без значение кой конкретен клас от дадена класификация ще се използва в тестовия сценарий
	този маркер означава, че от дадена класификация все още не е избран конкретен клас, който ще се използва в тестовия сценарий
	когато няма някои от изброените по-горе маркери, означава, че за тестовия сценарий вече е избран друг клас от дадената класификация

С цел внасяне на по-голяма яснота в списъка с тестови сценарии, е възможно обединяването им в групи – *TestcaseGroup*. Създаването на такава група от стъпки става по същия начин като създаването на тестова стъпка, описан по-горе. След като групата е създадена в нея могат да се добавят нови тестови стъпки. За тази цел групата, в която трябва да се добави стъпката, трябва да е избрана. В група могат да се добавят други групи рекурсивно и тестови последователности.



Фигура 14: Създаване на тестови сценарии, тестови групи и тестови последователности

Друг елемент на списъка с тестови сценарии е тестовата последователност, която представлява списък от стъпки, които са зависими от времето. Към всяка от стъпките може да се добави стойност. Това става като на дадената тестова последователност се дефинира метрика и съответната ѝ единица. Метриката може да е разстояние, ъгъл или време. Например, тази метрика може да показва колко е продължителността на всяка от стъпките в абсолютно или относително време. Всяка стъпка задава стойността на входния сигнал, дефиниран с конкретната класификация, в даден момент. Изменението на сигнала между отделните тестови стъпки може да се задава с връзки или преходи (*transitions*). В зависимост от формата на сигнала, тези преходи представляват различни функции – линейна, тригонометрична и други. Типовете преходи и техните графични представяния в CTE XL са показани на Фигура 15:

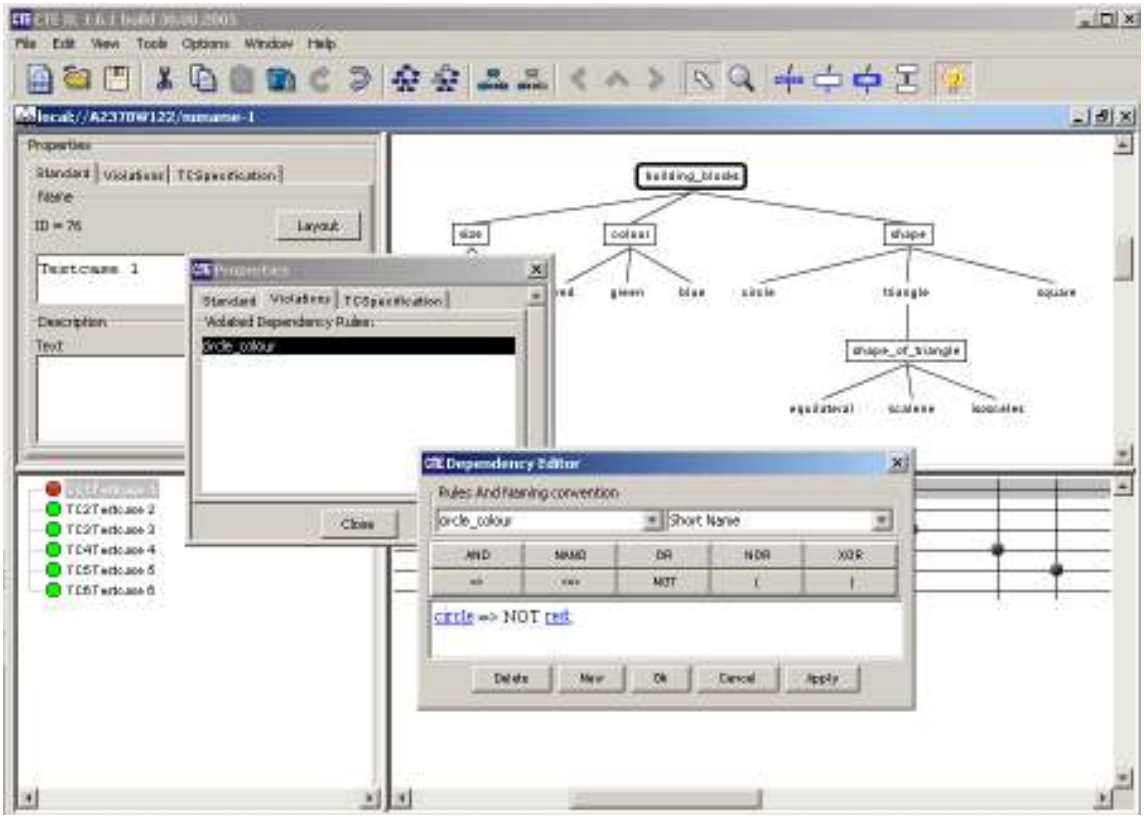


Фигура 15: Типовете функции на преход между стъпките в тестова последователност

4.1.3 Задаване на правила за логическа връзка между елементите на класификационното дърво

По време на определянето на тестовите сценарии чрез комбиниране на различни класове в таблицата на комбинациите, тестерът трябва да се грижи за логическата съвместимост на класовете. Логически зависимости в класификационното дърво възникват, когато определени комбинации от класове са невалидни или изключително необходими. CTE XL използва формули от *съждителна (пропозиционална) логика* (*propositional logic*) за описание на логическите зависимости, тъй като моделирането на зависимости с нея е достатъчно и не е необходимо да се използва логика от по-висок ред (например логика от първи ред). Всички класове в дървото могат да се използват като променливи.

В CTE XL всяко логическо правило може да се свърже с класификационно дърво, като за яснота всяко едно правило е именувано. За специфицирането на правилата, CTE XL предоставя редактор на зависимостите *Dependecy editor* (показан на Фигура 16), в които се добавят класовете от класификационното дърво и необходимите логически операции между тях. Съществуващите логически зависимости между класовете могат да се съобразяват при ръчно определяне на тестовите случаи. От една страна използването на логически правила предотвратява избора на противоречащи си класове, и от друга страна тестовите сценарии могат автоматично да се допълват като се взимат предвид вече избраните класове и дефинираните правила. В случай, че тестерът промени класификационното дърво или определено логическо правило, CTE XL осигурява автоматична проверка на всички тестови сценарии. На Фигура 16 правилото circle_colour няма да позволи избирането на тестова стъпка „large red circle”. Ако процедурата за проверка открие нарушаване на някое логическо правило, съответната тестова стъпка се маркира (TC1Testcase 1 показан на фигурата долу). Проверката за нарушение на логическите зависимости в тестовите стъпки става чрез активирането на *Dependency manager*, който се намира върху лентата с инструменти. Иконката  означава, че проверката за нарушения на правилата е активирана, а иконката  - проверката е деактивирана.



Фигура 16: Създаване на логически правила за връзка между класовете на класификационното дърво

Използването на логически зависимости предоставя възможност на тестерът да дефинира тестови сценарии по-лесно и дори полу-автоматизира процеса на комбиниране на класовете в таблицата. В много от случаите броят на теоретично възможните тестови стъпки, произлизащи от дървото, може да се редуцира с използването на правила. По този начин голям брой невалидни стъпки се избягват.

4.1.4 Дефиниране на правила за комбиниране и автоматично генериране на тестови сценарии

Отделните класификации и класове в класификационното дърво имат различна важност в зависимост от целите на тестването. По тази причина, за да са изчерпателни тестовите сценарии, е необходимо да се комбинират всички класове от значимите класификации, докато не е нужно останалите класове да бъдат включени в пълното комбиниране.

Друга важна характеристика на CTE XL е възможността да се дефинират правила за комбиниране на класификации или класове на класификационното дърво. Чрез тези правила може да се наблегне на значението за тестването на определена класификация или клас и на комбинацията на няколко класа и класификации. Правилата за комбиниране дават възможност да се определи например, че класовете на две важни класификации трябва да се комбинират напълно в тестовите сценарии и че класовете на другите класификации не е необходимо да се комбинират пълно, а само да се добавят подходящо към съществуващите комбинации. Също така е възможно един клас от дадена класификация да се комбинира със всички класове на друга класификация.

С цел улесняване на дефинирането на правилата за комбиниране, те се представят чрез изрази, които могат да се формират с използването на редактор на комбинациите (*combination editor*). Всички класове и класификации от класификационното дърво могат да се използват като променливи в изразите. Операциите, които се използват, са:

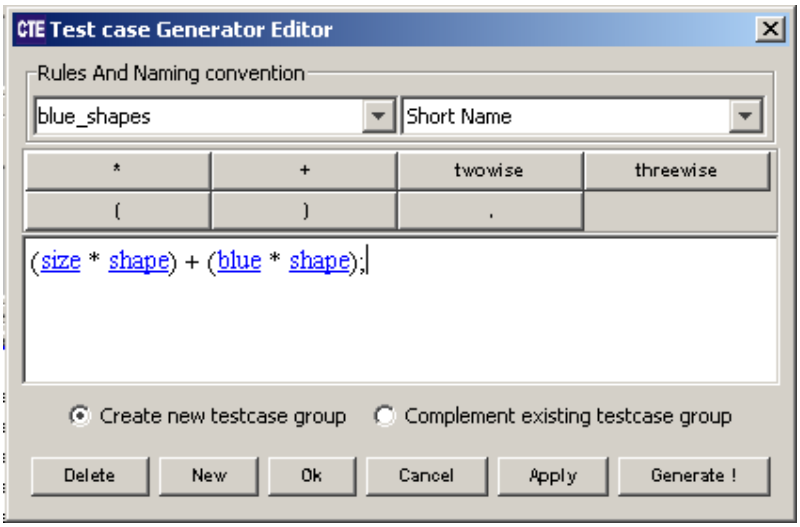
- пълно комбиниране (*complete combination*) A**B: Всеки клас от A се комбинира с всеки клас от B
- минимално комбиниране (*minimal combination*) A++B: Всеки клас от A и B се взима поне по веднъж
- оптимално комбиниране (*n-wise combinations*) n-wise(A_1, A_2, ... , A_m): Всяка възможна комбинация на *n* класа от A_1, A_2, ... , A_m ($1 \le n \le m$) се взима поне веднъж

За едно класификационно дърво могат да се дефинират няколко правила за комбиниране. По този начин различните акценти на тестването се представят чрез различни правила. Всяко правило за комбиниране има име за по-голяма яснота и представянето им чрез изрази ги прави много гъвкави и лесни за употреба.

Определянето на тестови сценарии ръчно отнема много време особено когато става въпрос за сложни тестови обекти. Класификационното дърво трябва да се създаде ръчно и също да се попълни таблицата на комбинации. При обширен и изчерпателен тест с голямо количество тестови сценарии това води до неясноти и нарушена структура на таблицата на комбинациите. Възможността за автоматично генериране на тестови сценарии улеснява доста работата по дизайна на тестовете. То гарантира систематична проверка на тестовия обект чрез всички важни комбинации от класове. Целта на описанието на логическите зависимости и правилата за комбиниране е тази информация да се използва за автоматично генериране на тестови сценарии (на Фигура 17 е показан редакторът за създаване на правила за комбиниране и за генериране на тестови сценарии на базата им). Следните стратегии могат да се използват при попълването на таблицата на комбинациите:

- генериране на минимален брой тестови сценарии, което гарантира, че всеки клас е взет предвид в поне един сценарий, като се имат предвид логическите зависимости.
- генериране на максимален брой тестови сценарии, което е резултат от вземането предвид на логическите зависимости, определени за класификационното дърво.
- генериране на оптимален брой тестови сценарии, което изпълнява правила за комбиниране и логическите зависимости, избрани от тестера

При третата стратегия за определяне на тестови сценарии алгоритъмът, който се използва, изпълнява евристично комбинаторно търсене. Използването на детерминистичен алгоритъм също е възможно, но е неизпълнимо тъй като той е с експоненциална сложност.



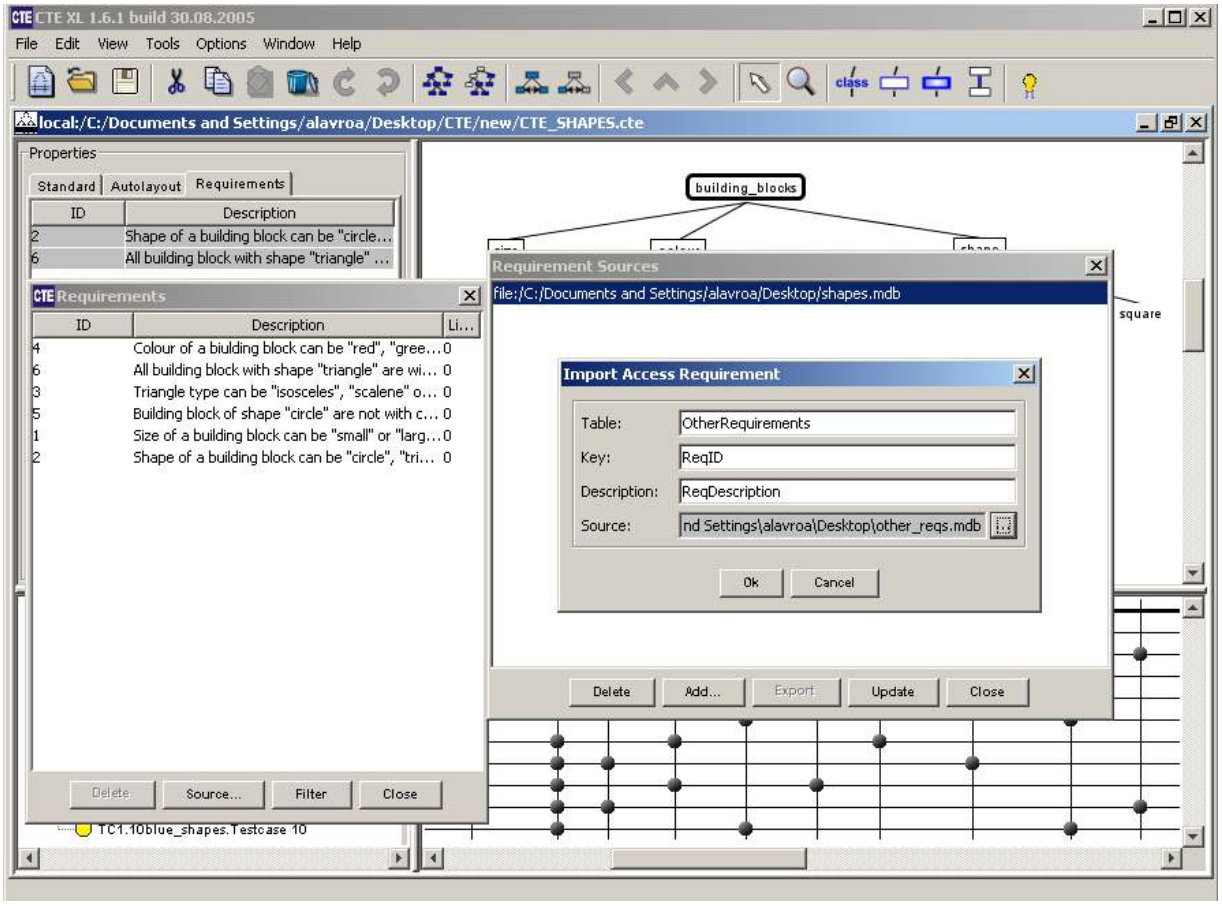
Фигура 17: Генериране на тестови сценарии чрез създаване на правила за комбиниране

Автоматичното попълване таблицата на комбинациите позволява тестерът да се концентрира върху избора на характеристики на обекта, значими за тестването, и съществуващите връзки между тези характеристики. [10]

4.1.5 Връзка на CTE XL с други програми

При използването на CTE XL за реални приложения е важна интеграция му с останалите инструменти, използвани по време на разработването. Например, системи за управление на изисквания и конфигурации, програми за спецификация и дизайн, и най-вече програми за изпълнение на тестови сценарии. За това е необходимо информацията, получена по време на дизайна на тестовите сценарии, да се предоставя на други програми, както и да се импортират данни от други програми към CTE XL.

CTE XL предоставя възможност за импортиране на изисквания от системи за управление на изисквания и свързването им с определени елементи от класификационното дърво и тестови сценарии. По този начин дизайна на тестовите сценарии се интегрира напълно в цикъла на разработване. Също така, по този начин може да се прави проверка за степента на покритие на изискванията към продукта, когато се правят прегледи на дизайна. Изисквания могат да се импортират от DOORS, Access бази данни или HTML документи. За всяко изискване се записва двойка идентификатор-описание (ID-Description). С тагът *Requirements* може да се работи след като е добавена връзка с програма за управление на изискванията (Фигура 18). Изискванията, свързани с дадени елементи на дърво, автоматично се свързват и с тестовите стъпки, съдържащи тези елементи.



Фигура 18: Добавяне на бази с изисквания и свързването им с елементи на класификационното дърво

Тестовите сценарии могат да се експортират в различни формати – HTML, RTF, JPG, MS Excel, Tessy и др. Това позволява дефинираните сценарии, направени чрез CTE XL, да се използват в други програми.

4.2 Дизайн на тестовите сценарии с помощта на МКД и CTE XL

Според дефинираните в част *2.2* етапи на Метода на класификационните дървета, началото на дизайна на тестовите сценарии включва определяне на входния и изходния интерфейс на тествания обект. В случая тестваният обект е функционалността Помощ при паркиране. Променливите, които му се подават като вход и които той обработва, както и получените изходни променливи са показани в Таблица 7. Тези променливи са класификациите в класификационното дърво с главен елемент `Parking_Assistance`.

Таблица 7: Входните и съответните им изходни променливи за функционалността Помощ при паркиране

Входна променлива	Домейн	Тип	Изходна променлива
`State_of_Display`	{активен, неактивен}	булева	*<< няма >>*
`Display_Requested_PA`	{0, 1}	булева	`Display_Requested_PA_Displ`
`Distance_BL`	[0, 7]	цяло число	`Distance_BL_Displayed`
`Distance_BM`	[0, 7]	цяло число	`Distance_BM_Displayed`
`Distance_BR`	[0, 7]	цяло число	`Distance_BR_Displayed`
`Distance_FR`	[0, 7]	цяло число	`Distance_FR_Displayed`
`Distance_FM`	[0, 7]	цяло число	`Distance_FM_Displayed`
`Distance_FL`	[0, 7]	цяло число	`Distance_FL_Displayed`

След като входните и изходните данни са уточнени, следващият етап е разделянето на дефиниционните им области на класове на еквивалентност.

От описанието на `State_of_Display` и `Display_Requested_PA` се вижда, че те са булеви и единственото възможно разбиване за техните домейни е на по два класа на еквивалентност. За `State_of_Display` класовете са `Active` и `Inactive`, а за `Display_Requested_PA` – `True` и `False`.

За останалите променливи, които отразяват разстоянието от колата до близкостоящ обект, дефиниционните области могат да се разделят като се вземе предвид Таблица 11 от *Приложение А*.

За параметрите `Distance_BL` и `Distance_BR` дефиниционните им области могат да се разделят по следния начин:

- при стойност 0 – тогава се показва иконата за предупреждение и иконата за изключително близък обект.
- стойност 1 също е особена стойност за тези две променливи. При нея има промяна на иконата за разстояние и иконата за предупреждение не се показва.
- стойност 2 е особена стойност също –променя се иконата за разстояние и иконата за предупреждение не се показва.
- при стойности 3, 4, 5, 6 и 7 състоянието на съответната част от екрана е едно и също – не се показва никаква икона и за това можем да ги обединим в един клас на еквивалентност.

Така за тези два параметъра се получават по четири класа на еквивалентност. За `Distance_BL` класовете са `BL_0`, `BL_1`, `BL_2` и `BL_Others`. За `Distance_BL` класовете са

BR_0, BR_1, BR_2 и BR_Others.

За параметрите Distance_FR и Distance_FL дефиниционните им области могат да се разделят на следните класове на еквивалентност:

- при стойност 0 – отново се показва иконата за предупреждение и иконата за изключително близък обект.
- при стойности 1 и 2 иконата за разстояние се променя и иконата за предупреждение не се показва.
- при стойности 3 и 4 отново има промяна на иконата за разстояние и не се показва иконата за предупреждение.
- при стойности 5, 6 и 7 всички зависещи от тези две променливи икони не се показват.

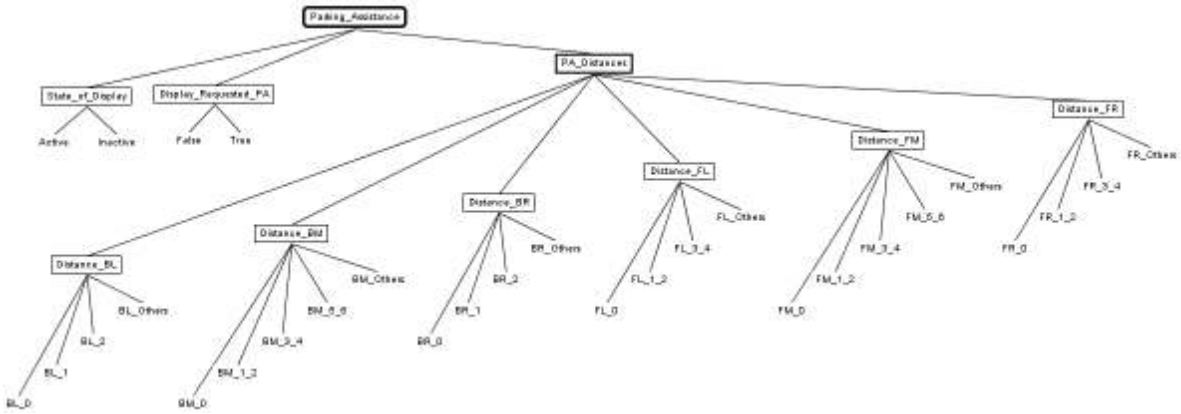
Получените класове на еквивалентност за двата параметър са отново по четири. Съответно за Distance_FR - FR_0, FR_1_2, FR_3_4 и FR_Others и за Distance_FL - FL_0, FL_1_2, FL_3_4 и FL_Others.

Дефиниционните области на параметрите Distance_FM и Distance_BM могат да се разделят на класове на еквивалентност като се вземат предвид следните стойности:

- при стойност 0 иконата за опасност се показва и се сформира иконата за максимална близост на обект до автомобила.
- при стойности 1 и 2 иконата за опасност не се показва, както и сегментът от иконата, който показва, че има обект много близо до автомобила, също изчезва
- при стойности 3 и 4 иконата за опасност не се показва и също изчезва още един сегмент от съответстващата на параметъра икона
- при стойности 5 и 6 иконата за опасност не се показва и изчезва третият сегмент от съответстващата на параметъра икона
- при стойност 7 иконата за опасност не се показва и изчезва и последния четвърти сегмент от съответстващата на параметъра икона

За тези два параметъра получените класове на еквивалентност са по пет за всеки. За Distance_FM класовете са FM_0, FM_1_2, FM_3_4, FM_5_6 и FM_Others. За Distance_BM класовете са BM_0, BM_1_2, BM_3_4, BM_5_6 и BM_Others.

Полученото класификационно дърво с главен елемент Parking_Assistance е показано на Фигура 19.



Фигура 19: Класификационното дърво за Помощ при паркиране

Следващият етап от дизайна на тестовите сценарии е избора на отделните стъпки на базата на класификационното дърво. Подробните спецификации на тестовите сценарии, генерирани от CTE XL, са описани в *Приложение B*, а в тази част се обосновава избора на конкретните стъпки – защо дадена стъпка е включена в тестовите сценарии.

Според изисквания $-EMF-PA-FCT-001.1 и $-EMF-PA-IHM-003 прозорецът S_PA_POP_RUNNING може да се отвори само когато дисплеят е в активно състояние и променливата Display_Requested_PA премине от състояние *„не е поискана визуализация*" към състояние *„поискана е визуализация*" или когато Display_Requested_PA е в състояние *„поискана е визуализация*" и дисплеят премине от

неактивно в активно състояние. Следователно, за покриване на тези изисквания, може да се дефинира следната последователност от стъпки, наречена `ShowParkingAssistance`:

1. Дисплеят е в неактивно състояние и не е поискана визуализация на информацията от Помощ при паркиране.
2. Дисплеят все още е в неактивно състояние, но `Display_Requested_PA` преминава в състояние *„поискана е визуализация*"
3. Състоянието на дисплея се променя в активно, а `Display_Requested_PA` остава в *„поискана е визуализация*"
4. Дисплеят остава в активно състояние, а `Display_Requested_PA` преминава в *„не е поискана визуализация*".
5. Докато дисплеят е в активно състояние, стойността на `Display_Requested_PA` се променя в *„поискана е визуализация*".
6. Дисплеят преминава в неактивно състояние, а `Display_Requested_PA` остава в старото си състояние *„поискана е визуализация*".

Останалите входни променливи, които съдържат информация за разстоянието от автомобила до близък обект, не са от значение в този случай, за това съответните им класове на еквивалентност се маркират с *Don't care* маркер. Тяхното коректно поведение се изследва от групата сценарии `MeasureDistancesFromObstacles`. Целта на тези тестове е от една страна да проверят, че иконата за опасност (ICO_WARNING) се визуализира коректно във всички ситуации, при които са налице условията за показването ѝ. От друга страна тестовете проверяват, че останалите икони също се визуализират коректно при получените входни данни и за това е необходимо да са налице условията дисплеят да е активен и да е поискана визуализация на Помощ при паркиране. Осигуряването на това условие става като се дефинира правило посредством функцията на CTE XL за задаване на правила за логическа връзка между елементите на класификационното дърво. Правилото за визуализация на прозореца S_PA_POP_RUNNING (`ShowingParkingAssistance`) изглежда по следния начин:

```
Active AND True;
```

Тази група от своя страна се състои от два сценария – `SingleObstacles` и `MultipleObstacles`. Първият сценарий изследва поведението в случай, че има само един обект в непосредствена близост до автомобила. За генерирането на тези тестове се използва функцията на CTE XL Generate Test cases. При проверката на всеки параметър стойността му се променя като преминава последователно през всички дефинирани за него класове на еквивалентност. Останалите променливи остават в такова състояние, което не регистрира обект в непосредствена близост и съответно не визуализира иконите за разстояние. Условията за генериране на тестови стъпки за всеки един от параметрите са описани в Таблица 8.

Таблица 8: Правила за комбиниране на входните параметри за автоматично генериране на тестовите сценарии `SingleObstacles`

Име на условие	Условие
`RightBackObstacles`	`Distance_BR * (BL_Others + BM_Others + FL_Others + FM_Others + FR_Others);`
`LeftBackObstacles`	`Distance_BL * (BR_Others + BM_Others + FL_Others + FM_Others + FR_Others);`
`MiddleBackObstacles`	`Distance_BM * (BR_Others + BL_Others + FL_Others + FM_Others + FR_Others);`
`RightFrontObstacles`	`Distance_FR * (BR_Others + BL_Others + BM_Others + FL_Others + FM_Others);`
`LeftFrontObstacles`	`Distance_FL * (BR_Others + BL_Others + BM_Others + FM_Others + FR_Others);`

MiddleFrontObstacles	Distance_FM * (BR_Others + BL_Others + BM_Others + FL_Others + FR_Others);

Вторият сценарий изследва поведението на дисплея в случай, че близките обекти са повече от един. Тъй като проверката на всички възможни комбинации между стойностите на шестте параметъра е невъзможно (възможните комбинации са 4*4*4*4*5*5 = 6400 стъпки), може да се използва критерия за минималност и всеки един от параметрите да вземе с различна стойност в поне една тестова стъпка. В този случай необходимия брой стъпки е пет колкото е броят на класовете в класификацията, разделена на най-много подмножества (в случая това са класификациите `Distance_FM` и `Distance_BM`). Освен произволни комбинации между параметрите трябва да се провери и поведението в още два случая:

- когато около автомобила няма близки обекти
- когато и от шестте сензора се засича препятствие, което е в опасна близост.

Условията използвани при генерирането на тестовите сценарии, са показани в Таблица 9.

Таблица 9: Правила за комбиниране на входните параметри за автоматично генериране на тестовите сценарии `MultipleObstacles`

Име на условие	Условие
MultipleObstacles	Distance_BL + Distance_BM + Distance_BR + Distance_FL + Distance_FM + Distance_FR;
AllObstacleTooClose	BL_0 * BR_0 * BM_0 * FL_0 * FM_0 * FR_0;
NoCloseObstacles	BL_Others * BR_Others * BM_Others * FL_Others * FM_Others * FR_Others;

Общият брой на дефинираните тестови стъпки за функционалността Помощ при паркиране е 34.

4.3 Реализация на тестовите сценарии с конкретни данни с помощта на VB AutoEMF

След като е завършил дизайнът на тестовите сценарии, е необходимо да се премине към реализацията на тестовете с конкретни данни. Това е четвъртия (последен) етап при използването на Метода на класификационните дървета за вградени системи. От определените за всяка стъпка класове на еквивалентност трябва да се избере конкретна стойност на променлива.

По време на този етап Методът на класификационните дървета може да се съчетае с използването на други техники за тестване като тестване с гранични стойности, с произволни стойност и т.н. За дефинираните в част *4.2* тестове за функционалността Помощ при паркиране е удобно да се използва тестването с произволни данни. Също така, това е етапът, на който трябва да се определят очакваните резултати за всяка тестова стъпка на базата на дефинираните за нея действия.

В тази глава ще се демонстрира как се определят очакваните резултати за няколко примерни тестови стъпки на базата на действията, който се извършват в тях. Пълният тест репорт с дефинираните сценарии, който е резултат от валидацията на функционалността Помощ при паркиране, е предоставен в *Приложение Г*.

Първата стъпка от етапа на реализация с конкретни данни е да се определят предпоставките за всеки от тестовите сценарии. Предпоставките трябва да поставят системата в определено състояние, така че очакваните резултати само за първата тестова стъпка да зависят от тях. Очакваните резултати за всяка следваща стъпка се определят от резултата от предходната и действията, извършени в текущата стъпка – т.е. всяка тестова стъпка е предпоставка за следващата.

За дефинираните три групи тестове за Помощ при паркиране предпоставките

трябва да поставят системата в нормално състояние – екранът да е активен, да няма отворени прозорци (главни и контекстни менюта, съобщения за грешки, настройки на звука и други) освен базовите. За първия сценарий (ShowParkingAssistance) не е необходимо параметрите за разстояние да се установяват в определени стойности, тъй като те не са съществени за целите на тези тестове и могат да са с произволни стойности.

При втория сценарий - MeasureDistancesFromObstacles.SingleObstacles - обаче тези параметри са обекта на проверка и е важно какви ще са техните начални стойности. Освен другите предпоставки трябва и всички параметри за разстояние да са със стойности, при които не се визуализират икони за близък обект.

Целта на третия сценарий е да провери поведението на екрана при произволни комбинации от стойностите на всеки от параметрите. Поради това за тази група стъпки също не е необходимо тези параметри да се установяват предварително в някаква определена стойност – в първата стъпка се определят техните начални стойности.

След като предпоставките за даден сценарий са определени, следващата стъпка е изборът на конкретни данни, които да се използват в тестовите действия. Например за тестовия сценарий ShowParkingAssistance важни са променливите State_of_Display и Display_Requested_PA. Стойностите на параметрите за разстояние не са от първостепенна важност и тях ги избираме по произволен начин като за всяка от стъпките тези променливи се взимат с различни стойности с цел покриване на повече тестови ситуации.

При втория сценарий - MeasureDistancesFromObstacles.SingleObstacle - в стъпките последователно се променят стойностите на всеки параметър за разстояние така, че в даден момент само един от сензорите регистрира близък обект. В третия сценарий – MeasureDistancesFromObstacles.MultipleObstacle – стойностите на всички параметри се променят във всяка стъпка. В предишния етап на МКД за всяка тестова стъпка е определено от кои класове на еквивалентност трябва да се вземат стойностите на променливите. За реализацията на тестовете трябва да се избере по един представител на този клас. Тук отново може да се използва метода на тестване с произволни стойности.

Следващата стъпка от този е етап е определянето на очаквания резултат, който е следствие от дефинираните за всяка тестова стъпки действия. За стъпките в първия тестов сценарий очакваният резултат се определя според дефинираните изисквания ($-EMF-PA-IHM-001, $-EMF-PA-IHM-003, $-EMF-PA-FCT-001.1 и $-EMF-PA-FCT-002.1) – т.е. прозорецът S_PA_POP_RUNNING трябва да се визуализира само когато са изпълнени и двете условия:

State_of_Display = Active и Display_Requested_PA = True

Ако поне едно от тях не е изпълнено, прозорецът не трябва да се показва. Например за дефинираната в *Приложение В* тестова стъпка 2, при която State_of_Display е Inactive, а Display_Requested_PA се променя от False на True, очакваният резултат е екранът да остане неактивен и да не се визуализира никаква информация от Помощ при паркиране. Очакваният резултат за стъпка 3, който се определя от действията Display_Requested_PA = True и промяна на State_of_Display в Active, е екранът да стане активен и това да предизвика визуализацията на прозорецът S_PA_POP_RUNNING.

Тези две стъпки, реализирани на Visual Basic AutoEMF, изглеждат така:

```
'Step2 - Display is in Inactive state and display of parking assistance is
requested. The six obstacles are at random distance
    For i = 0 To UBound(PADistances) - 1
        dist = rand(0, 7)
        Call SetData(PADistances(i), dist)
    Next i
    WAIT (0.5)
    Call SetData("Env_DMDE_AFFICHAGE", 1)
```

```
    Call UserResult("Display is in inactive state. Parking assistance window
is not displayed.")


    'Step3 - Display is in Active state and display of parking assistance is
requested. The six obstacles are at random distance
    For i = 0 To UBound(PADistances) - 1
        dist = rand(0, 7)
        Call SetData(PADistances(i), dist)
    Next i
    WAIT (0.5)
    Call SetData("Env_ETAT_PRINCIP_SEV", 1)
    Call UserResult("Display goes in active state. Parking assistance window
is displayed.")
```

За двата сценария - `MeasureDistancesFromObstacles.SingleOsbtacle` и `MeasureDistancesFromObstacles.MultipleOsbtacle` – очакваният резултат описва какво е текущото състояние на прозореца S_PA_POP_RUNNING, т.е. какво е съдържанието на всяка една от неговите части (описание на показаната икона или празно поле).

Например стъпка 1 от сценария `MeasureDistancesFromObstacles.SingleOsbtacle` ще изглежда по следния начин в Visual Basic AutoEMF :

```
'Step1 - there are very close obstacles to the car at each side
    For i = 0 To UBound(PADistances)
        resultDistances(i) = SetDistances(PADistances(i), 0)
        s = s & resultDistances(i)
    Next i
    result = result & icoWarningFull & s
    WAIT (0.5)
    Call SetData("Env_DMDE_AFFICHAGE", 1)
    Call UserResult(result)
```

Частта от тест-репорта, отнасящ се за тази стъпка, е :

No	Action	Expected Results	OK NOK NT	Comments	Covered Req.
3.1.3.1	- PA_DATA/ DIST_FL = 0 - Obstacle in Front left side of the car is in Zone 1 - PA_DATA/ DIST_FR = 0 - Obstacle in Front right side of the car is in Zone 1 - PA_DATA/ DIST_BL = 0 - Obstacle in Back left side of the car is in Zone 1 - PA_DATA/ DIST_BR = 0 - Obstacle in Back right side of the car is in Zone 1 - PA_DATA/ DIST_FM = 0 - Obstacle in Front middle side of the car is in Zone 1 - PA_DATA/ DIST_BM = 0 - Obstacle in Back middle side of the car is in Zone 1 - PA_DATA/ DISP_RQST = 1 - Parking assistance information is requested to be displayed	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : ICO_WARNING - In OT_IMAGE_5 : ICO_PA_FL_3 (Front left obstacle is in Zone1) - In OT_IMAGE_3 : ICO_PA_FR_3 (Front right obstacle is in Zone1) - In OT_IMAGE_6 : ICO_PA_BL_3 (Back left obstacle is in Zone1) - In OT_IMAGE_4 : ICO_PA_BR_3 (Back right obstacle is in Zone1) - In OT_IMAGE_7 : ICO_PA_F_B_2, ICO_PA_F_B_1, ICO_PA_F_B_1 and	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1

		ICO_PA_F_B_3 (Front middle obstacle is in Zone1) - In OT_IMAGE_8 : ICO_PA_F_B_3, ICO_PA_F_B_1, ICO_PA_F_B_1 and ICO_PA_F_B_2 (Back middle obstacle is in Zone1)			

Глава 5. Сравнение на разгледаните подходи

В тази глава се прави обобщение на проведения експеримент и резултата от него. За целта трябва да се дефинират критерии, спрямо които да се сравнят и оценят двата метода – текущо използвания във фирмата и метода на класификационните дървета за вградени системи. В книгата си Software Testing: A Craftsman's Approach Пол Йоргенсен формулира три критерия за оценка на подходи при тестване – ефективността (effectiveness) и ефикасността (efficiency) на тестовите сценарии и усилията, които са положени за техния подбор. В част *5.1* критериите са изяснени и са допълнени с четвърти – документиране на сценариите. Резултатите от сравнението на методите са описани в част *5.2*. В последната част *5.3* от тази глава са дефинирани препоръки за подобрение на текущия процес на тестване във фирмата.

5.1 Дефиниране на критерии за сравнение

При тестването на софтуерни приложения и системи е особено важно поведението им да се изследва във възможно най-много ситуации и състояния, в които те могат да попаднат по време на работата си. Поради тази причина тестовете трябва да покриват колкото може по-голяма част от изискванията към продукта и ако е възможно да се тестват ситуациите описани в софтуерната спецификация. От особена важност е това да става с минимален брой стъпки (да няма излишни), тъй като при разработването на даден софтуерен проект в повечето случаи времето дадено за разработка е доста кратко. Оптимизирането на броя стъпки води до съкращаване на времето за изпълнение на сценариите. Следователно най-важният критерий при сравнението на различни методи за тестване е *ефикасността (efficiency) на тестовите сценарии* – способността им да покриват описаните изисквания с оптимален брой стъпки.

Възможността за откриване на дефекти се увеличава, когато определените тестови сценарии обхващат по-голям брой действителни (реални) ситуации. – т.е. увеличава се *ефективността (effectiveness)* им, която може да се дефинира като втори критерий за сравнение.

Определянето на тестовите сценарии чрез използване на методи като тестване с гранични стойности става почти механично, тъй като между входните данни няма логическа връзка. Разграничаването на особените стойности не изисква особени усилия от страна на тестера. При използването на метода с класове на еквивалентност например се обръща внимание на връзките между входните данни, както и на особеностите на самия тестов обект. Този метод изисква по-детайлен анализ на областта от входни данни. Така като следващ (трети) критерий може да се дефинира *усилията (effort)*, които се полагат, за идентифициране на необходимите тестови сценарии.

Много често се налага да се правят промени в софтуерния проект, което налага и промени в тестовите процедури. Поради тази причина тестовете трябва да се лесни за поддръжка – да могат лесно да се добавят, изтриват или променят стъпките в тях. Също така провеждането на прегледи (ревюта, reviews) е по-лесно, ако има по-абстрактно описание на тестовите стъпки, от което да става ясна целта им – защо точно тези данни са избрани, защо ситуацията се тества на точно определено място. Това улеснява преглеждащия тестовете в откриването на евентуални пропуски или несъответствия. Четвъртият критерий за сравнение – възможността за *документиране на дизайна на тестовите сценарии* – е обобщение на двете споменати изисквания.

5.2 Резултати от сравнението на разгледаните методи

В тази част двата метода – текущо използваният във фирмата и метода на класификационните дървета – ще бъдат сравнени спрямо всеки от описаните в предната част *5.1* критерии и на базата на това сравнение ще бъдат оценени.

При използването на метода на класификационните дървета има систематичен подход за дефиниране на тестови стъпки, които биха били изпуснати например при интуитивно тестване или тестване с произволни данни и при които евентуално може да се прояви дефект в софтуерния продукт. Тъй като при МКД има дефинирана процедура за подбор на тестовите стъпки може да се определи дали сценариите покриват напълно описаните изисквания. Гарантираното покритие на софтуерните изисквания (спецификация) прави тестовите сценари по-ефективни – т.е. дава по-голяма сигурност по отношение (на) откриването на дефекти. Колкото повече реални ситуации обхващат тестовете, толкова е по-голяма вероятността при изпълнението им да се открият грешки в тествания обект.

Текущо използваният метод във фирмата – комбинация от интуитивно тестване, тестване с произволни данни и тестване с гранични стойност – сам по себе си не е достатъчно надежден, когато става дума за определяне покритието на изисквания от тестови стъпки. При избора на произволни стойности от домейна на една или няколко променлива не може да се твърди със сигурност, че всички възможни ситуации са проверени с тестовите сценарии. Използването на интуитивното тестване изолирано, когато не е комбинирано с друг метод за тестване, също не дава достатъчно информация за това дали изискванията са пълно покрити или не. Тестването с гранични стойности донякъде е най-систематичният подход, който се използва сега в тестовия процес на фирмата, но той е подходящ само за независими една от друга променливи. Ако има променливи, които са зависими, и при точно определени комбинации от техни стойности настъпва дадено събитие, този случай може да бъде пропуснат от метода.

От друга страна, използването на тези методи понякога води до излишни тестови стъпки – множество еднакви или подобни ситуации се тестват на няколко места в тестовете. От тук и времето за изпълнение на тестовите процедури нараства. При текущият метод на тестване броят на дефинираните стъпки за функционалността Помощ при паркиране, която е обект на експеримента, е 154, докато броят на дефинираните стъпки с използването на метода на класификационните дървета е 34. Малкият брой стъпки в случая, когато се използва МКД, води до значително намаляване на времето за изпълнението им. Например за човек, който е сравнително наясно със спецификацията на функционалността Помощ при паркиране и който има опит с изпълнението на тестовете, времето за изпълнението им е между 5 и 6 минути, докато за изпълнението на тестовите сценарии, дефинирани по текущия метод във фирмата, са необходими повече от 20 минути. Сравнението на времето за изпълнение на тестовете, дефинирани чрез двата метода, и покритието на софтуерните изисквания, което те правят, показва, че МКД е много по-ефикасен. При него броят на дефинираните стъпки е около 4,5 пъти по-малък, а изпълнението им е 4 пъти по-бързо. Това сравнение показва, че при тестване на по-обемна функционалност с много повече изисквания към нея и при наличието на голям брой функционалности, времето за изпълнение на тестовете значително ще намалее.

Използването на подход с разделяне на класове на еквивалентност и базираните на него методи изискват повече ресурси като време и труд за разлика от тестването с произволно и интуитивно избрани данни, и с гранични стойности. Усилията, които се полагат при подбора на тестовите сценарии, са по-големи при използването на МКД отколкото при текущия подход на тестване.

При съществуващият процес на тестване не се отделя време за дизайн на тестовите сценарии. Решението за това какво точно ще се тества се взима в момента на създаване на тестовите стъпки и не се документира по никакъв начин. Това прави много трудно редактирането на съществуващите сценарии дори от техния автор. Например, ако трябва да се добави стъпка, която тества даден дефект и е била пропусната до сега, изборът на подходящо място за добавянето й ще отнеме доста време. Най-често стъпката се добавя просто в края на съществуващ сценарий като допълнително се определя средата, в която тя трябва да се изпълни – т.е. предхождащата я стъпка не е предпоставка за новата стъпка. По време на прегледите на тестовите сценарии е трудно да се прецени дали дадено изискване е покрито напълно или не, тъй като понякога изискванията не се тестват в един сценарий, а стъпките за тях са разпръснати в няколко сценария. Също така, преглеждащият тестовете няма критерий, по който да прецени дали всички необходими

ситуации са включени като стъпки или не.

Използването на метода на класификационните дървета позволява спецификацията на софтуерния продукт да се погледне на по-абстрактно ниво. При построяването на класификационното дърво се откриват стойности или групи стойности, които при използването на други методи могат да бъдат пропуснати. Това спомага и за по-коректното и пълно определяне на тестовите стъпки. Освен това, при използването на МКД в съчетание с Classification Tree Editor предоставя възможност и за документиране на целите на всяка една от стъпките. По време на етапа, когато се определят конкретните стойности на входните променливи, тестерът може да приложи различни техники за тестване (с гранични стойности, с произволни данни и т.н.), но неговите идеи за това какво и как трябва да се тества ще останат документирани. При планираните прегледи е достатъчно да се проверява абстрактния модел на тестовите сценарии (т.е. техния дизайн), а не техните реализации с конкретни данни, което облекчава работата на проверяващия. Изолирането от детайлите му помага да придобие глобален поглед както върху обекта на тестване, така и върху вече дефинираните сценарии. Това помага за по-лесното определяне на пълнотата на тестовите сценарии спрямо изискванията към софтуерния продукт.

В сравнение със съществуващия в момента подход на тестване методът на класификационните дървета изисква повече усилия за дефиниране на тестови сценарии, но за сметка на това предоставя по-голяма гаранция за пълнота на покритието им (процента на покритите софтуерни изисквания чрез дефинираните стъпки). Той успява да осигури максимално покритие чрез дефиниране на минимален брой стъпки в тестовите сценарии (което съответно намалява и времето за изпълнението им) и улеснява значително поддръжката им, тъй като предоставя възможност за документиране на техния дизайн.

5.3 Препоръки за подобрение на тестовия процес във фирмата

Сравнението направено в част 5.2 показа, че методът, който се използва текущо във фирмата, не е достатъчно ефективен и не дава задоволителни резултати спрямо дефинираните в част 5.1 три критерия за сравнение. Могат да се дефинират следните препоръки за подобрение към (на) съществуващия тестов процес във фирмата:

1. Във времето отделено за създаване и модифициране на тестовите сценарии да се планира време за техния дизайн.
2. При подбора на тестовите стъпки да се използва систематичен подход, който може да се съчетава с различни методи за тестване.
3. Дизайнът на тестовите сценарии да се документира и пази в системата за управление на конфигурациите.
4. Да се планират прегледи да направения дизайн на тестовите сценарии и да се планира време за оптимизация на дефинираните сценарии, с цел премахване на излишни или повтарящи се стъпки.
5. Да се използва инструмент (tool), който на базата на направения дизайн, генерира основната част („скелета”) на необходимия за изпълнение на стъпките Visual Basic код. Така повече време ще се отделя на подбора на тестови стъпки, а не на тяхната конкретна реализация.

Заключение

Изборът на подходящи тестови сценарии е от решаваща важност при тестването на софтуер. Той определя обхвата и качеството на дефинираните тестове. Дипломната работа разглежда систематичен подход за оптимизиране на тестовия процес при разбработване на софтуерни продукти и по-специално на подбора на тестови сценарии. Набляга се на използването на метода на класификационните дървета при тестване на вградени системи за автомобилната промишленост, но той може да се прилага и в други области на софтуерното инженерство. Разгледани са предимствата на метода, които спомагат за повишаване на качеството на тестовия процес. Методът е базиран на разбиване на областта от входни данни на класове на еквивалентност и предоставя възможност тестовете да се дефинират на по-високо ниво на абстракция, което ги прави лесни за модифициране и повторно използване. Също така използването на метода на класификационните дървета спомага за оптимизиране на отношението покритие на изисквания към брой дефинирани тестови стъпки. По време на разработването на дипломната работа е проведен експеримент, чиято цел е да сравни предлагания метод на класификационните дървета и текущо използван във фирмата Johnson Controls Electronics подход за тестване. По време на експеримента функционалност от съществуващ и разработван в момента от фирмата проект е описана и тестовете за нея са реализирани с помощта на МКД и базирания на него графичен редактор CTE XL. Според набора от дефинирани критерии МКД показва значително по-добри резултати спрямо текущо използвания подход.

Прилагането на метода на класификационните дървета би могло по естествен начин да се разшири с използване на еволюционни алгоритми. Чрез еволюционните алгортими се модифицират дефинираните с помощта на МКД сценарии, с цел покриване на повече ситуации и увеличаване на възможността за откриване на по-голям брой потенциални дефекти.

Друга възможна насока за развитие на разглеждания метод е интеграцията на CTE XL с различни инструменти за автоматично тестване и/или езици за разработка. Това би съкратило времето за реализиране на тестовете и би пренасочило повече ресурси към подбора на тестовите сценарии. Възможно решение на този проблем е инструмент за автоматично генериране на основната част от тестовите скриптове на базата на дефинираните с CTE XL сценарии.

Приложения

Приложение A: Описание на изискванията към функционалността Помощ при паркиране

Таблица 10: Списък на функционалните и интерфейсните изисквания

Номер на изискване	Описание на изискване
$-EMF-PA-IHM-001	*Входни и изходни точки на Помощ при паркиране*
$-EMF-PA-IHM-003	*Управление на интерфейсната функция F_PA_RUNNING*
$-EMF-PA-IHM-004	*Описание на прозореца S_PA_POP_RUNNING*
$-EMF-PA-IHM-002	*Дефиниция на състоянията на полетата на прозореца S_PA_POP_RUNNING*
$-EMF-PA-IHM-007	*Дефиниция на използваните икони*
$-EMF-PA-FCT-001.1	*Управление на Помощ при паркиране*
$-EMF-PA-FCT-002.1	*Управление на разстоянията в Помощ при паркиране*

$-EMF-PA-IHM-001	***Входни и изходни точки на Помощ при паркиране***

Интерфейсната част на функционалността Помощ при паркиране се състои от една функция – F_PA_RUNNING, която позволява да се отчитат разстоянията до заобикалящите автомобила препятствия.

При вход във функционалността Помощ при паркиране функцията F_PA_RUNNING се стартира.

При изход от функционалността Помощ при паркиране на мултифункционалния дисплей се визуализира информацията от функционалността, показана преди изискването за визуализация на помощ при паркиране.

$-EMF-PA-IHM-003	***Управление на интерфейсната функция F_PA_RUNNING***

Функцията F_PA_RUNNING има един прозорец – S_PA_POP_RUNNING.

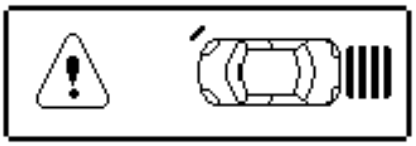
Когато Display_Requested_PA = 1 (т.е. поискана е визуализация на информацията от Помощ при паркиране), прозорецът S_PA_POP_RUNNING се отваря.

При преминаване на Display_Requested_PA от състояние „*не е поискана визуализация*" в състояние „*поискана визуализация*" на информацията от Помощ при паркиране (Display_Requested_PA : 0→1), прозорецът S_PA_POP_RUNNING се отваря.

$-EMF-PA-IHM-004	***Описание на прозореца S_PA_POP_RUNNING***

Прозорецът S_PA_POP_RUNNING се визуализира в зона Z3 на мултифункционалния екран при отварянето си. Има приоритет AUTO_PRIORITY (чието поведение е дефинирано в документа *Управление на приоритетите*, които не е обект на изследването).

Графично прозорецът S_PA_POP_RUNNING също е разделен на части, всяка от които визуализира определена информация :



Дефинициите на условията за показване определени икони във всяка част са описани в изискване $-EMF-PA-IHM-002.

При преминаване на Display_Requested_PA към състояние *„не е поискана визуализация”* на информацията от Помощ при паркиране (Display_Requested_PA : 1→0), прозорецът S_PA_POP_RUNNING се затваря.

$-EMF-PA-IHM-002	***Дефиниция на състоянията на полета на прозореца S_PA_POP_RUNNING***

Таблица 11: Дефиниция на състоянията на полетата в зависимост от различни условия

Състояние	**Условие**	**Икона**
ST_ICO_PA_WARNING	Distance_FR = Zone1 **OR** Distance_FM = Zone1 **OR** Distance_FL = Zone1 **OR** Distance_BR = Zone1 **OR** Distance_BM = Zone1 **OR** Distance_BL = Zone1	ICO_WARNING
	Иначе	« *празно* »
ST_ICO_PA_FR	***Distance_FR***	
	= Zone1	ICO_PA_FR_3
	= Zone2 **OR** Zone3	ICO_PA_FR_2
	= Zone4 **OR** Zone5	ICO_PA_FR_1
	= Zone6 **OR** Zone7 **OR** Zone8	« *празно* »
ST_ICO_PA_BR	***Distance_BR***	
	= Zone1	ICO_PA_BR_3
	= Zone2	ICO_PA_BR_2
	= Zone3	ICO_PA_BR_1
	= Zone4 **OR** Zone5 **OR** Zone6 **OR** Zone7 **OR** Zone8	« *празно* »
ST_ICO_PA_FL	***Distance_FL***	
	= Zone1	ICO_PA_FL_3
	= Zone2 **OR** Zone3	ICO_PA_FL_2
	= Zone4 **OR** Zone5	ICO_PA_FL_1
	= Zone6 **OR** Zone7 **OR** Zone8	« *празно* »
ST_ICO_PA_BL	***Distance_BL***	
	= Zone1	ICO_PA_BL_3
	= Zone2	ICO_PA_BL_2
	= Zone3	ICO_PA_BL_1
	= Zone4 **OR** Zone5 **OR** Zone6 **OR** Zone7 **OR** Zone8	« *празно* »
ST_ICO_PA_FM_1	***Distance_FM***	
	= Zone1	ICO_PA_F_B_2
	Otherwise	« *празно* »
ST_ICO_PA_FM_2	***Distance_FM***	
	= Zone1 **OR** Zone2 **OR** Zone3	ICO_PA_F_B_1
	Иначе	« *празно* »
ST_ICO_PA_FM_3	***Distance_FM***	
	= Zone1 **OR** Zone2 **OR** Zone3 **OR** Zone4 **OR** Zone5	ICO_PA_F_B_1
	Иначе	« *празно* »
ST_ICO_PA_FM_4	***Distance_FM***	
	= Zone1 **OR** Zone2 **OR** Zone3 **OR** Zone4 **OR**	ICO_PA_F_B_3

	Zone5 **OR** Zone6 **OR** Zone7	
	Иначе	« *празно*»
ST_ICO_PA_BM_1	***Distance_BM***	
	= Zone1	ICO_PA_F_B_3
	Иначе	« *празно* »
ST_ICO_PA_BM_2	***Distance_BM***	
	= Zone1 **OR** Zone2 **OR** Zone3	ICO_PA_F_B_1
	Иначе	« *празно* »
ST_ICO_PA_BM_3	***Distance_BM***	
	= Zone1 **OR** Zone2 **OR** Zone3 **OR** Zone4 **OR** Zone5	ICO_PA_F_B_1
	Иначе	« *празно* »
ST_ICO_PA_BM_4	***Distance_BM***	
	= Zone1 **OR** Zone2 **OR** Zone3 **OR** Zone4 **OR** Zone5 **OR** Zone6 **OR** Zone7	ICO_PA_F_B_2
	Иначе	« *празно* »

$-EMF-PA-IHM-007	***Дефиниция на използваните икони***

Таблица 12: Дефиниция на използваните икони

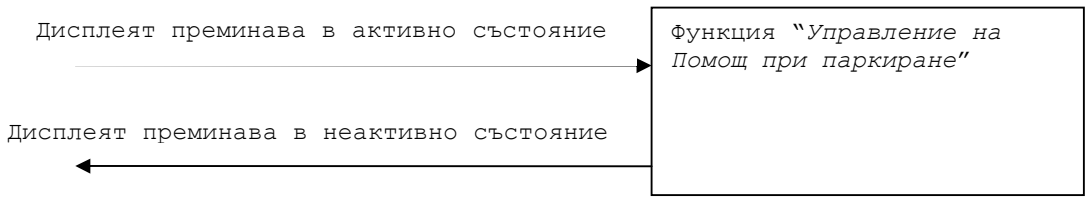
Име	Икона	Име	Икона	Име	Икона
ICO_PA_BR_1		ICO_PA_F_B_1		ICO_PA_FL_1	
ICO_PA_BR_2		ICO_PA_F_B_2		ICO_PA_FL_2	
ICO_PA_BR_3		ICO_PA_F_B_3		ICO_PA_FL_3	
ICO_PA_BL_1		ICO_PA_FR_1		ICO_PA_CAR	
ICO_PA_BL_2		ICO_PA_FR_2		ICO_WARNING	
ICO_PA_BL_3		ICO_PA_FR_3			

$-EMF-PA-FCT-001.1	***Управление на Помощ при паркиране***

Функционалността Помощ при паркиране се управлява от мултифункционалния екран само когато той е в активен режим.

При излизането му от активен режим управлението на функционалността Помощ при паркиране се преустановява.

Това поведение може да се представи чрез следната диаграма:



$-EMF-PA-FCT-002.1	***Управление на разстоянията в Помощ при паркиране***

Тази функция оценява разстоянията от автомобила до заобикалящите го предмети и изпраща към дисплея тези разстояния, както и от сензор е постъпила информацията.

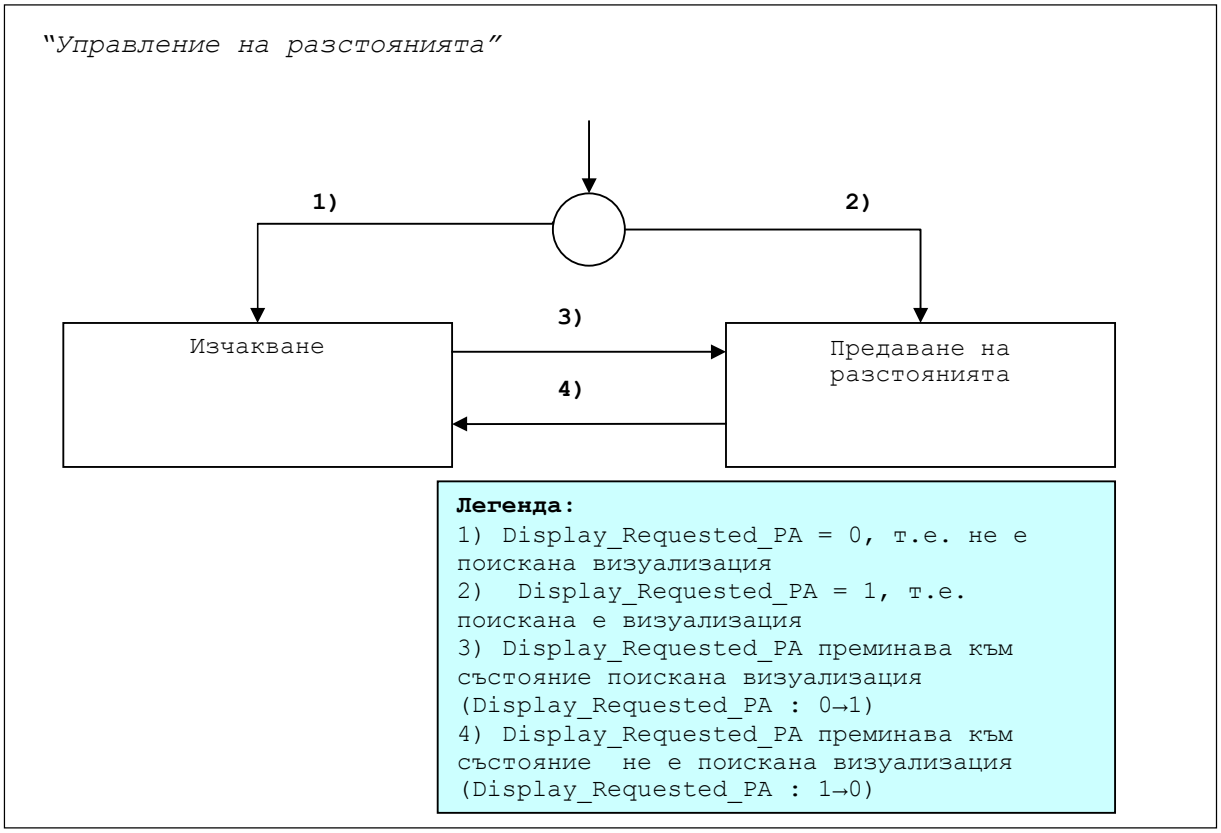
Входните променливи са показани в Таблица 13.

Таблица 13: Описание на входните променливи и техните дефиниционни области

Име на променлива	Описание на променлива	Стойности на променлива
State_Of_Display	Съдържа информация за текущото състояние на дисплея – активно или неактивно.	0 = активно 1 = неактивно
Display_Requested_PA	Съдържа информация за това дали е поискана визуализация на Помощ при паркиране.	0 = *не е поискана визуализация* 1 = *поискана е визуализация*
Distance_BL	Съдържа информацията за разстоянието от задната лява част на автомобила до заобикалящите го препятствия.	0b000 = Зона 1 0b001 = Зона 2 0b010 = Зона 3 0b011 = Зона 4 0b100 = Зона 5 0b101 = Зона 6 0b110 = Зона 7 0b111 = Зона 8
Distance_BM	Съдържа информацията за разстоянието от задната средна част на автомобила до заобикалящите го препятствия.	0b000 = Зона 1 0b001 = Зона 2 0b010 = Зона 3 0b011 = Зона 4 0b100 = Зона 5 0b101 = Зона 6 0b110 = Зона 7 0b111 = Зона 8
Distance_BR	Съдържа информацията за разстоянието от задната дясна част на автомобила до заобикалящите го препятствия.	0b000 = Зона 1 0b001 = Зона 2 0b010 = Зона 3 0b011 = Зона 4 0b100 = Зона 5 0b101 = Зона 6 0b110 = Зона 7 0b111 = Зона 8
Distance_FR	Съдържа информацията за разстоянието от предната дясна част на автомобила до заобикалящите го препятствия.	0b000 = Зона 1 0b001 = Зона 2 0b010 = Зона 3 0b011 = Зона 4 0b100 = Зона 5 0b101 = Зона 6 0b110 = Зона 7 0b111 = Зона 8
Distance_FM	Съдържа информацията за разстоянието от предната средна част на автомобила до заобикалящите го препятствия.	0b000 = Зона 1 0b001 = Зона 2 0b010 = Зона 3 0b011 = Зона 4 0b100 = Зона 5 0b101 = Зона 6 0b110 = Зона 7 0b111 = Зона 8
Distance_FL	Съдържа информацията за	0b000 = Зона 1 0b001 = Зона 2

	разстоянието от предната лява част на автомобила до заобикалящите го препятствия.	0b010 = Зона 3 0b011 = Зона 4 0b100 = Зона 5 0b101 = Зона 6 0b110 = Зона 7 0b111 = Зона 8

Функцията „Управление на разстоянията в Помощ при паркиране" има две състояния – *изчакване* и *предаване на разстоянията*. Преходите от едно състояние към друго се осъществяват според показаното на следната диаграма:



В състояние `изчакване` дисплея не визуализира информацията от сензорите, а само следи стойността на Display_Requested_PA (която е 0 в това състояние). Ако стойността на Display_Requested_PA се промени в 1 – функцията „Управление на разстоянията" преминава в състояние `предаване на разстоянията`.

Приложение Б: Описание на основните инструменти, използвани в текущия процес на тестване

Основният инструмент, които се използва по време на тестването, е CANoe, които е продукт на немската фирма Vector Informatik. CANoe е универсална среда за разработване, тестване и анализ на системи, част от CAN мрежа.

С цел автоматизация на тестовете, в проекта се използва и разработен от екипа инструмент на Visual Basic – AutoEMF.

Vector CANoe

Целта на CANoe е да симулира работата на различните устройства в CAN мрежата на автомобила, като по този начин създава на тестваната система реални условия за работа. CANoe симулация за даден проект се състои от следните части :

- CAPL (CAN Access Programming Language) скриптове, които описват кои данни как ще се променят при определени промени по CAN мрежата
- бази данни с получаваните и изпращаните от устройството CAN съобщения
- панели, които осигуряват интерфейс на потребителя, чрез който той може да управлява CAPL скриптовете.
- конфигурационни файлове, които съдържат информация за това кои панели, бази данни и скриптове се използват от симулацията, както и настройки на CAN мрежата.

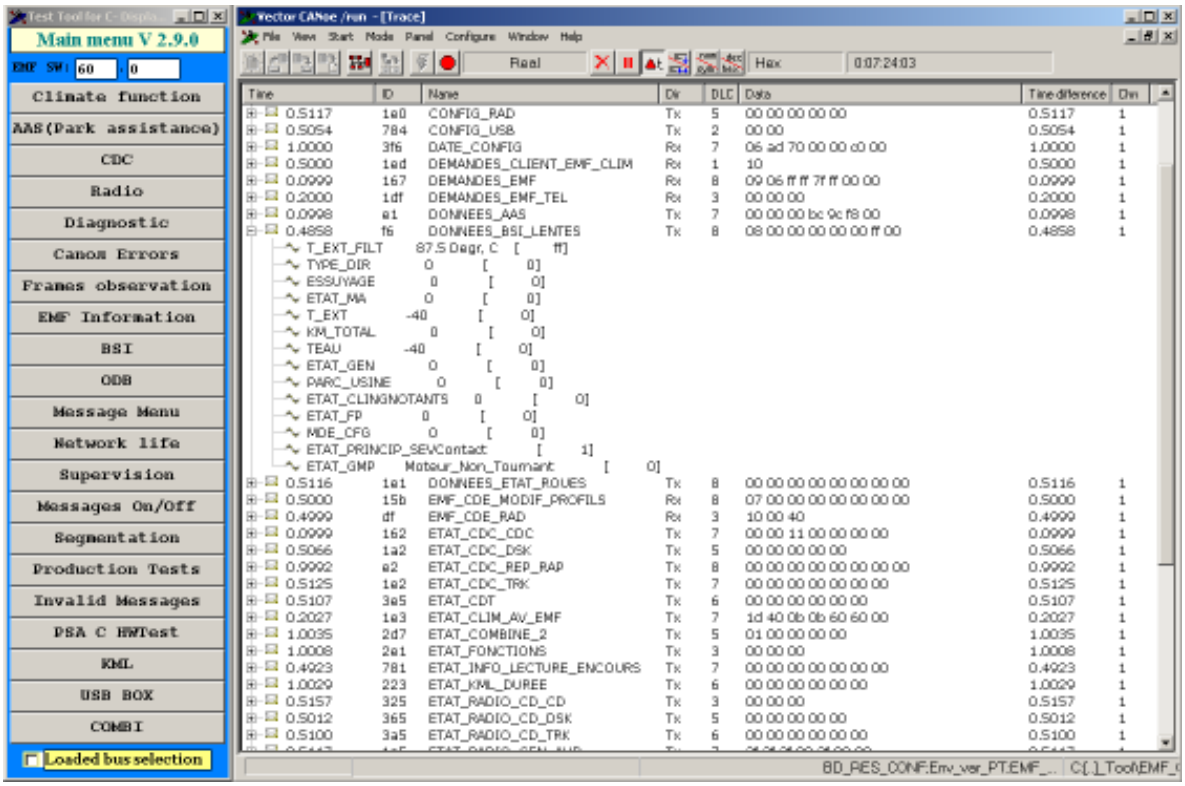
Базите данни съдържат информация за всички използвани CAN сигнали – име, дължина, отместване и фактор. В тях се пази и структурата на CAN сигналите – име, дължина, кои сигнали са част от съобщението. Данните за променливите на средата – тип, минимална и максимална стойност, стойност по подразбиране – също се съхраняват в базите данни.

Променливите на средата се създават автоматично при зареждане на конфигурационен файл и са достъпни за всички CAPL файлове и панели. Тези променливи са достъпни и за програми, външни за CANoe, като Visual Basic или C#.

CAPL е скриптов език, които се използва от CANoe за създаване на модел на реални устройства. Синтаксисът му е близък до този на езика за програмиране C, но е по-опростен. На всеки сигнал (които е част от дадено съобщение, предавано по мрежата) съответства променлива на средата (environment variable). В панелите на симулацията се използват тези променливи на средата, като техните стойности се променят от CAPL скриптовете чрез функциите get и set.

Панелите на симулацията служат за комуникация между потребителя и CAPL скриптовете. Чрез тях се променят или визуализират данните. Панелите представляват прозорци, върху които се разполагат различни елементи – като текстови полета, радио-бутони, чек-боксове (check-boxes), картинки и други. Всеки елемент от панела е свързан променлива на средата. При промяна на променливата в един панел, нейната стойност се променя автоматично и в останалите панели, в които тя се среща.

Програмата CANoe се състои от един основен прозорец и множество панели, които са характерни за конкретния проект. В основния прозорец се визуализира информация за съобщенията, които се обменят в CAN мрежата – изпращани от или към симулацията. Освен информация за съобщения, от него могат да се правят настройки на симулацията като записване на последователности от съобщения във файл, възпроизвеждане на тези последователности, настройване на филтри за визуализиране само на съобщения, избрани от потребителя и други. [30]



Фигура 20: Главният прозорец на приложението CANOe и основния прозорец на симулацията, използвана за тестване на мултифункционалния дисплей

Visual Basic AutoEMF

AutoEMF се използва както по време на пълната валидация, така и по време на инкременталната. Той използва библиотеките, инсталирани заедно с CANoe, за да се свързва с него и да управлява променливите му. Условно AutoEMF може да се раздели на три нива.

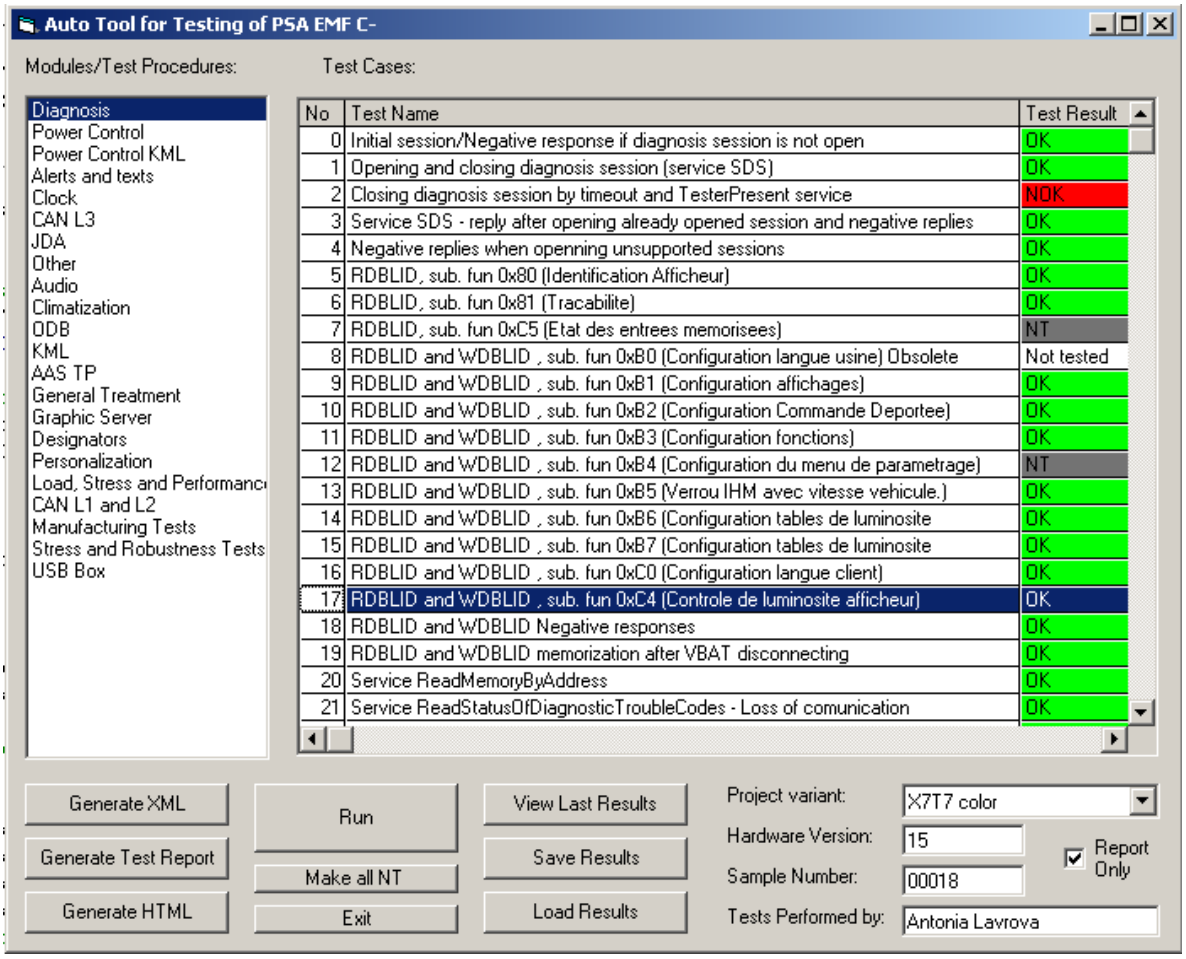
Най-ниското ниво осъществява връзката с CANoe. Неговите функции се грижат за стартирането на CANoe в началото на изпълнението на тестовете, установяването на променлива в определена стойност, прочитане стойността на променлива в определен момент и други.

Следващото ниво осигурява набор от често използвани в тестовете функции. Например изпълнение на тестови действия и добавянето им репорта, добавяне на очакван резултат, на коментари и идентификатори на дефект в репорта.

Третото (най-високо) ниво се състои от множество файлове, всеки един от които съдържа по една тестова процедура, отговаряща на функционалност на мултифункционалния дисплей.

При стартиране на приложението се зарежда главната (основната) форма (Фигура 21), която съдържа списък с тестови процедури. При избиране на дадена процедура във формата се визуализира списък с нейните тестови сценарии. Тестовите сценарии могат да се изпълняват в произволен ред. В началото, преди да е изпълнен сценарии, неговият резултат е „Not tested”. Преди началото на изпълнение на тестовете е необходимо в главната форма да се въведе определена информация. Например име на тестера, който ще ги изпълни, серийния номер на устройството, върху което ще се изпълняват, варианта на проекта, който се тества. Последната информация е особено важна, тъй като от нея зависи изпълнението на правилните стъпки за всеки вариант.

След изпълнението му съответния резултат – OK, NOK, NT – се отбелязва в списъка с тестови сценарии. Тестовите стъпки на всеки сценарии, които е изпълнен, могат да се прегледат като се избере желания сценарии.



Фигура 21: Главната форма на приложението AutoEMF

Резултатите от изпълнението на един или много тестови сценарии могат да се запазят във файл, с цел използването им по-нататък при ново стартиране на приложението.

По време на инкременталната или пълната валидация се изпълняват тестовете и след приключването им, на базата на резултатите от тях, се генерират XML файлове, които се използват за генерация на тест-репорта. Генерацията на репорта се извършва от друго приложение, чиито входни данни са XML файловете, а изходни – Word файл.

Част от тестовите сценарии са напълно автоматични. За всяка стъпка са дефинирани тестови действия – установяване на променливи в дадени стойности, отваряне на менюта и други прозорци. Очакваният резултат се състои в това да се прочете стойността на променлива след тези действия. Това става по автоматичен начин и не изисква вниманието и намесата на тестера по време на теста.

Останалата част от тестовите сценарии са полуавтоматични. При тях след изпълнението на тестовите действия се изисква потвърждение от тестерът дали очакваният резултат съвпада с реално наблюдавания. Такъв тип тестове се използват обикновено при тестването на потребителския интерфейс на системата, когато целта е да се провери правилното разположение на икони и текстове по екрана.

Приложение B: Спецификация на дефинираните с помощта на CTE XL тестови сценарии

Test Report

alavroa

Table of Contents

Table of Figures

1 Figures

1.1 Parking_Assistance

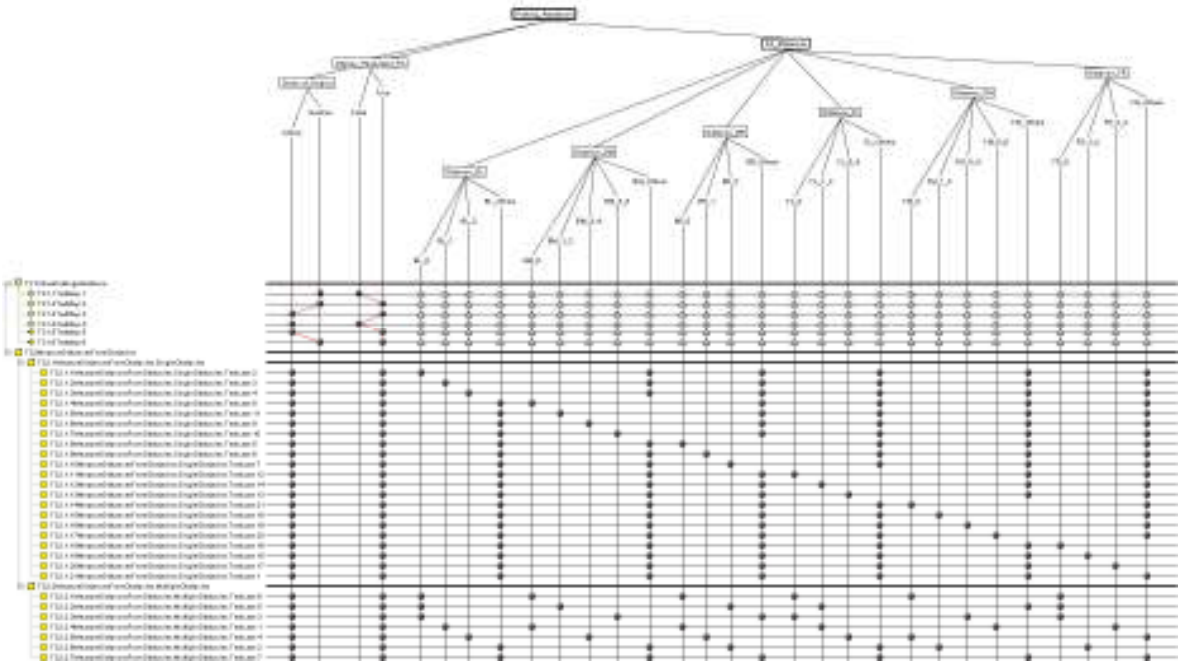


Figure 1:Parking_Assistance

2 Tree Node Descriptions

Parking_Assistance/State_of_Display/Inactive	
Description	Information indicating that the display is in inactive mode.

Parking_Assistance/Display_Requested_PA	
Description	Information indicating if displaying of Parking Assistance is requested.

Parking_Assistance/PA_Distances/Distance_FM/FM_Others	
Description	Information indicating that there is no object closely to front middle side of the car - object is in Zone 8.

Parking_Assistance/PA_Distances/Distance_FL	
Description	Information coming from the parking Assistance, indicating the distance to the obstacle compared with the vehicle front left.

Parking_Assistance/PA_Distances/Distance_FR/FR_0	
Description	Information indicating that there is an object very closely to front right side of the car - in Zone 1.

Parking_Assistance/PA_Distances/Distance_BL/BL_2	
Description	Information indicating that there is an object closely to back left side of the car - in Zone 3.

Parking_Assistance/PA_Distances/Distance_BR/BR_0	
Description	Information indicating that there is an object very closely to back right side of the car - in Zone 1.

Parking_Assistance/PA_Distances	

Parking_Assistance/PA_Distances/Distance_BM/BM_3_4	
Description	Information indicating that there is an object comparatively closely to back middle side of the car - in Zone 4 or Zone 5.

Parking_Assistance/State_of_Display	
Description	Information indicating the display state - active or inactive mode.

Parking_Assistance/PA_Distances/Distance_BM	
Description	Information coming from the Parking Assistance, indicating the distance from the obstacle compared with the vehicle back (middle).

Parking_Assistance/PA_Distances/Distance_FR	
Description	Information coming from the parking Assistance, indicating the distance to the obstacle compared with the vehicle front right.

Parking_Assistance/PA_Distances/Distance_BM/BM_Others	
Description	Information indicating that there is no object very closely to back middle side of the car - object is in Zone 8.

Parking_Assistance	
Description	Function Parking Assistance objective is to display the pieces of information on the screen to inform the user about the distance that separates his vehicle from the surrounding obstacles.

Parking_Assistance/Display_Requested_PA/True	
Description	Displaying of information from Parking Assistance is requested.

Parking_Assistance/PA_Distances/Distance_BL/BL_1	
Description	Information indicating that there is an object closely to back left side of the car - in Zone 2.

Parking_Assistance/PA_Distances/Distance_FM	
Description	Information coming from the Parking Assistance, indicating the distance from the obstacle compared with the vehicle front (middle).

Parking_Assistance/PA_Distances/Distance_FR/FR_3_4	
Description	Information indicating that there is an object closely to front right side of the car - in Zone 4 or Zone 5

Parking_Assistance/PA_Distances/Distance_FL/FL_1_2	
Description	Information indicating that there is an object closely to front left side of the car - in Zone 2 or Zone 3.

Parking_Assistance/PA_Distances/Distance_BL/BL_0	
Description	Information indicating that there is an object very closely to back left side of the car - in Zone 1.

Parking_Assistance/Display_Requested_PA/False	
Description	Displaying of information from Parking Assistance is not requested.

Parking_Assistance/PA_Distances/Distance_FR/FR_Others	
Description	Information indicating that there is no object closely to front right side of the car - object is in Zone 6, Zone 7 or Zone 8.

Parking_Assistance/PA_Distances/Distance_BM/BM_5_6	
Description	Information indicating that there is an object closely to back middle side of the car - object is in Zone 6 or Zone 7.

Parking_Assistance/PA_Distances/Distance_BR/BR_1	
Description	Information indicating that there is an object closely to back right side of the car - in Zone 2.

Parking_Assistance/State_of_Display/Active	
Description	Information indicating that the display is in active mode.

Parking_Assistance/PA_Distances/Distance_FM/FM_1_2	
Description	Information indicating that there is an object comparatively closely to front middle side of the car - in Zone 2 or Zone 3.

Parking_Assistance/PA_Distances/Distance_FL/FL_3_4	
Description	Information indicating that there is an object closely to front left side of the car - in Zone 4 or Zone 5

Parking_Assistance/PA_Distances/Distance_BL/BL_Others	
Description	Information indicating that there is no object closely to back left side of the car - the object is in Zone 4, Zone 5, Zone 6, Zone 7 or Zone 8.

Parking_Assistance/PA_Distances/Distance_FM/FM_3_4	

Description	Information indicating that there is an object comparatively closely to front middle side of the car - in Zone 4 or Zone 5.

Parking_Assistance/PA_Distances/Distance_BR	
Description	Information coming from the parking Assistance, indicating the distance to the obstacle compared with the vehicle back right.

Parking_Assistance/PA_Distances/Distance_FL/FL_0	
Description	Information indicating that there is an object very closely to front left side of the car - in Zone 1.

Parking_Assistance/PA_Distances/Distance_BL	
Description	Information coming from the parking Assistance, indicating the distance to the obstacle compared with the vehicle back left.

Parking_Assistance/PA_Distances/Distance_FM/FM_0	
Description	Information indicating that there is an object very closely to front middle side of the car - in Zone 1.

Parking_Assistance/PA_Distances/Distance_FL/FL_Others	
Description	Information indicating that there is no object closely to front left side of the car - object is in Zone 6, Zone 7 or Zone 8.

Parking_Assistance/PA_Distances/Distance_BM/BM_1_2	
Description	Information indicating that there is an object comparatively closely to back middle side of the car - in Zone 2 or Zone 3.

Parking_Assistance/PA_Distances/Distance_BR/BR_Others	
Description	Information indicating that there is no object closely to back right side of the car - the object is in Zone 4, Zone 5, Zone 6, Zone 7 or Zone 8.

Parking_Assistance/PA_Distances/Distance_FR/FR_1_2	
Description	Information indicating that there is an object closely to front right side of the car - in Zone 2 or Zone 3.

Parking_Assistance/PA_Distances/Distance_FM/FM_5_6	
Description	Information indicating that there is an object closely to front middle side of the car - in Zone 6 or Zone 7.

Parking_Assistance/PA_Distances/Distance_BR/BR_2	
Description	Information indicating that there is an object closely to back right side of the car - in Zone 3.

Parking_Assistance/PA_Distances/Distance_BM/BM_0	
Description	Information indicating that there is an object very closely to back middle side of the car - in Zone 1.

3 Test Case Descriptions

ShowParkingAssistance	

TCSpecification	- State_of_Display:Active - State_of_Display:Inactive - Display_Requested_PA:False - Display_Requested_PA:True - Distance_BL:BL_0 - Distance_BL:BL_1 - Distance_BL:BL_2 - Distance_BL:BL_Others - Distance_BM:BM_0 - Distance_BM:BM_1_2 - Distance_BM:BM_3_4 - Distance_BM:BM_5_6 - Distance_BM:BM_Others - Distance_BR:BR_0 - Distance_BR:BR_1 - Distance_BR:BR_2 - Distance_BR:BR_Others - Distance_FL:FL_0 - Distance_FL:FL_1_2 - Distance_FL:FL_3_4 - Distance_FL:FL_Others - Distance_FM:FM_0 - Distance_FM:FM_1_2 - Distance_FM:FM_3_4 - Distance_FM:FM_5_6 - Distance_FM:FM_Others - Distance_FR:FR_0 - Distance_FR:FR_1_2 - Distance_FR:FR_3_4 - Distance_FR:FR_Others

Teststep 1	
TCSpecification	- State_of_Display:Inactive - Display_Requested_PA:False - Distance_BL:$ don't care - Distance_BM:$ don't care - Distance_BR:$ don't care - Distance_FL:$ don't care - Distance_FM:$ don't care - Distance_FR:$ don't care

Teststep 2	
TCSpecification	- State_of_Display:Inactive - Display_Requested_PA:True - Distance_BL:$ don't care - Distance_BM:$ don't care - Distance_BR:$ don't care - Distance_FL:$ don't care - Distance_FM:$ don't care - Distance_FR:$ don't care

Teststep 3	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:$ don't care - Distance_BM:$ don't care - Distance_BR:$ don't care - Distance_FL:$ don't care - Distance_FM:$ don't care - Distance_FR:$ don't care

Teststep 4	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:False - Distance_BL:$ don't care - Distance_BM:$ don't care - Distance_BR:$ don't care - Distance_FL:$ don't care - Distance_FM:$ don't care - Distance_FR:$ don't care

Teststep 5	

TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:$ don't care - Distance_BM:$ don't care - Distance_BR:$ don't care - Distance_FL:$ don't care - Distance_FM:$ don't care - Distance_FR:$ don't care

Teststep 6	
TCSpecification	- State_of_Display:Inactive - Display_Requested_PA:True - Distance_BL:$ don't care - Distance_BM:$ don't care - Distance_BR:$ don't care - Distance_FL:$ don't care - Distance_FM:$ don't care - Distance_FR:$ don't care

MeasureDistancesFromObstacles	
TCSpecification	- State_of_Display:Active - State_of_Display:Inactive - Display_Requested_PA:False - Display_Requested_PA:True - Distance_BL:BL_0 - Distance_BL:BL_1 - Distance_BL:BL_2 - Distance_BL:BL_Others - Distance_BM:BM_0 - Distance_BM:BM_1_2 - Distance_BM:BM_3_4 - Distance_BM:BM_5_6 - Distance_BM:BM_Others - Distance_BR:BR_0 - Distance_BR:BR_1 - Distance_BR:BR_2 - Distance_BR:BR_Others - Distance_FL:FL_0 - Distance_FL:FL_1_2 - Distance_FL:FL_3_4 - Distance_FL:FL_Others - Distance_FM:FM_0 - Distance_FM:FM_1_2 - Distance_FM:FM_3_4 - Distance_FM:FM_5_6 - Distance_FM:FM_Others - Distance_FR:FR_0 - Distance_FR:FR_1_2 - Distance_FR:FR_3_4 - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles	

TCSpecification	- State_of_Display:Active - State_of_Display:Inactive - Display_Requested_PA:False - Display_Requested_PA:True - Distance_BL:BL_0 - Distance_BL:BL_1 - Distance_BL:BL_2 - Distance_BL:BL_Others - Distance_BM:BM_0 - Distance_BM:BM_1_2 - Distance_BM:BM_3_4 - Distance_BM:BM_5_6 - Distance_BM:BM_Others - Distance_BR:BR_0 - Distance_BR:BR_1 - Distance_BR:BR_2 - Distance_BR:BR_Others - Distance_FL:FL_0 - Distance_FL:FL_1_2 - Distance_FL:FL_3_4 - Distance_FL:FL_Others - Distance_FM:FM_0 - Distance_FM:FM_1_2 - Distance_FM:FM_3_4 - Distance_FM:FM_5_6 - Distance_FM:FM_Others - Distance_FR:FR_0 - Distance_FR:FR_1_2 - Distance_FR:FR_3_4 - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase_2	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_0 - Distance_BM:BM_Others - Distance_BR:BR_Others - Distance_FL:FL_Others - Distance_FM:FM_Others - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase_3	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_1 - Distance_BM:BM_Others - Distance_BR:BR_Others - Distance_FL:FL_Others - Distance_FM:FM_Others - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase_4	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_2 - Distance_BM:BM_Others - Distance_BR:BR_Others - Distance_FL:FL_Others - Distance_FM:FM_Others - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase_8	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_0 - Distance_BR:BR_Others - Distance_FL:FL_Others - Distance_FM:FM_Others - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase 11	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_1_2 - Distance_BR:BR_Others - Distance_FL:FL_Others - Distance_FM:FM_Others - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase 9	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_3_4 - Distance_BR:BR_Others - Distance_FL:FL_Others - Distance_FM:FM_Others - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase 10	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_5_6 - Distance_BR:BR_Others - Distance_FL:FL_Others - Distance_FM:FM_Others - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase 5	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_Others - Distance_BR:BR_0 - Distance_FL:FL_Others - Distance_FM:FM_Others - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase 6	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_Others - Distance_BR:BR_1 - Distance_FL:FL_Others - Distance_FM:FM_Others - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase 7	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_Others - Distance_BR:BR_2 - Distance_FL:FL_Others - Distance_FM:FM_Others - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase 12	

TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_Others - Distance_BR:BR_Others - Distance_FL:FL_0 - Distance_FM:FM_Others - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase 14	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_Others - Distance_BR:BR_Others - Distance_FL:FL_1_2 - Distance_FM:FM_Others - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase 13	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_Others - Distance_BR:BR_Others - Distance_FL:FL_3_4 - Distance_FM:FM_Others - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase 21	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_Others - Distance_BR:BR_Others - Distance_FL:FL_Others - Distance_FM:FM_0 - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase 18	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_Others - Distance_BR:BR_Others - Distance_FL:FL_Others - Distance_FM:FM_1_2 - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase 19	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_Others - Distance_BR:BR_Others - Distance_FL:FL_Others - Distance_FM:FM_3_4 - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase 20	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_Others - Distance_BR:BR_Others - Distance_FL:FL_Others - Distance_FM:FM_5_6 - Distance_FR:FR_Others

MeasureDistancesFromObstacles.SingleObstacles.Testcase 16	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_Others - Distance_BR:BR_Others - Distance_FL:FL_Others - Distance_FM:FM_Others - Distance_FR:FR_0

MeasureDistancesFromObstacles.SingleObstacles.Testcase 15	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_Others - Distance_BR:BR_Others - Distance_FL:FL_Others - Distance_FM:FM_Others - Distance_FR:FR_1_2

MeasureDistancesFromObstacles.SingleObstacles.Testcase 17	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_Others - Distance_BR:BR_Others - Distance_FL:FL_Others - Distance_FM:FM_Others - Distance_FR:FR_3_4

MeasureDistancesFromObstacles.SingleObstacles.Testcase 1	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_Others - Distance_BR:BR_Others - Distance_FL:FL_Others - Distance_FM:FM_Others - Distance_FR:FR_Others

MeasureDistancesFromObstacles.MultipleObstacles	

TCSpecification	- State_of_Display:Active - State_of_Display:Inactive - Display_Requested_PA:False - Display_Requested_PA:True - Distance_BL:BL_0 - Distance_BL:BL_1 - Distance_BL:BL_2 - Distance_BL:BL_Others - Distance_BM:BM_0 - Distance_BM:BM_1_2 - Distance_BM:BM_3_4 - Distance_BM:BM_5_6 - Distance_BM:BM_Others - Distance_BR:BR_0 - Distance_BR:BR_1 - Distance_BR:BR_2 - Distance_BR:BR_Others - Distance_FL:FL_0 - Distance_FL:FL_1_2 - Distance_FL:FL_3_4 - Distance_FL:FL_Others - Distance_FM:FM_0 - Distance_FM:FM_1_2 - Distance_FM:FM_3_4 - Distance_FM:FM_5_6 - Distance_FM:FM_Others - Distance_FR:FR_0 - Distance_FR:FR_1_2 - Distance_FR:FR_3_4 - Distance_FR:FR_Others

MeasureDistancesFromObstacles.MultipleObstacles.Testcase 6	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_0 - Distance_BM:BM_0 - Distance_BR:BR_0 - Distance_FL:FL_0 - Distance_FM:FM_0 - Distance_FR:FR_0

MeasureDistancesFromObstacles.MultipleObstacles.Testcase 5	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_0 - Distance_BM:BM_1_2 - Distance_BR:BR_2 - Distance_FL:FL_1_2 - Distance_FM:FM_Others - Distance_FR:FR_0

MeasureDistancesFromObstacles.MultipleObstacles.Testcase 3	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_0 - Distance_BM:BM_5_6 - Distance_BR:BR_Others - Distance_FL:FL_0 - Distance_FM:FM_3_4 - Distance_FR:FR_0

MeasureDistancesFromObstacles.MultipleObstacles.Testcase 1	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_1 - Distance_BM:BM_0 - Distance_BR:BR_0 - Distance_FL:FL_1_2 - Distance_FM:FM_5_6 - Distance_FR:FR_3_4

MeasureDistancesFromObstacles.MultipleObstacles. Testcase 4	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_2 - Distance_BM:BM_3_4 - Distance_BR:BR_1 - Distance_FL:FL_3_4 - Distance_FM:FM_0 - Distance_FR:FR_Others

MeasureDistancesFromObstacles.MultipleObstacles. Testcase 2	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_Others - Distance_BR:BR_2 - Distance_FL:FL_Others - Distance_FM:FM_1_2 - Distance_FR:FR_1_2

MeasureDistancesFromObstacles.MultipleObstacles. Testcase 7	
TCSpecification	- State_of_Display:Active - Display_Requested_PA:True - Distance_BL:BL_Others - Distance_BM:BM_Others - Distance_BR:BR_Others - Distance_FL:FL_Others - Distance_FM:FM_Others - Distance_FR:FR_Others

4 Dependency rules

ShowingPakringAssistance	Active AND True ;

Приложение Г: Тест-репорт за функционалността Помощ при паркиране

BM1005 (C-COLOR) SOFTWARE VALIDATION REPORT

SKILL	Error! Unknown	**PROJECT**	**BM1005 (C-Color)**

ABSTRACT / CONCLUSION

This report is the result of the validation done on the BM1005 (C-Color) for the 60.0 version

34 Tests steps defined
100 % Tests steps executed
100 % Tests steps OK

100 % of requirements are covered by the set of all test steps.
100 % of requirements are tested.
100 % of requirements are OK.

Validated Software Checksum : 0xE8F9

CIRCULATION LIST (**E** : E-network, **P** : partial, **C** : complete)

	Incomplete electronic document - see the paper version						
D	Name	Service	Location	D	Name	Service	Loc
E	T. Chasserieau	RDU14	PON	E	A. Zhelyazkov	Quality Dpt	SOF
E	E. Brodin	RDU14	PON				
E	E. Barbaux	RDU14	PON				
E	Sv. Andreev	RDU PSA	SOF				
E	J.P. Beguier	RDU14	PON				
E	C. Kerveillant	RDU14	PON				
E	I. Baghri	RDU14	PON				
E	O. Stoilov	RDU PSA	SOF				
E	V. Rashev	RDU VAL PSA	SOF				
E	A. Lavrova	RDU VAL PSA	SOF				

According to : SVR Template 001-0437

	Written by	*Checked by*	*Approved by*	*Diffusion*
Function	*Author*	*Sw Lead Engineer*	*Project Leader*	*PL EM*
Name	Antonia Lavrova			
Date	8/28/2006			

REVISION SUMMARY

Edition	Author	Modified Paragraphs and Type of the Modification
1	EMF C- Validation team	Generation

CONTENTS

1. INTRODUCTION

1.1 Purpose

The purpose of the document is to propose the result of the software validation of the project.

1.2 Scope

The scope of the validation is :

Test Name / Funct. Name	Funct.	Data Integrity	Perf.	Load	Stress	Failover/ Recovery	Config.	Long Term	User Interface	Instal.
Parking Assistance	**V**								**V**	

1.3 Target Audience

The audience of this document includes Lead Engineers, Project Leader, Software Coordinator and SW project's validators .

This document can also be disclosed to the customer.

1.4 Definitions, Acronyms and Abbreviations

Definition	Description
SW	Software
SVR	Software Validation Report
STL	Software Team Leader
VCT	Version Control Tool

1.5 Reference Documents

Ref.	Document	Reference
[R1]	Software validation process	PONAEHNT043382

2. SUMMARY OF VALIDATION RESULTS

Number of Steps Distributed by Functionality and Step's Result :

Functionality	Nb Steps			Coverage Rate	Failed Requirements	Not Tested Requirements	Comments
	OK	NOK	NTest				
Parking Assistance	34	0	0	100 %			
Total	34	0	0		NOK Requirements Count: 0	NT Requirements Count: 0	Defects Count: 0

3. VALIDATION TEST CASES

The detail of the test cases for each functionality is defined below.

3.1 Parking Assistance

Objective of the report: Purpose of this set of tests is to verify that Parking Assistance works properly.

According to the following documents:

Ref	Documents	Reference	Version
[0]	STD IHM AAS C-	96 478 044	99 A
[1]	STD Fonctionnelle - Gerer l'Aide Au Stationnement	96 478 043	99 OR

Test Environment

Software Version	Hardware Version	Sample Number	Validation Tools[1]	Other devices and Tools
60. 0	13	00018	CanOe compatible card (our configuration: Vector infomatik GmbH CanCardX) CanOe version 4.0.57 PRO with SP2 or above Test Tool for EMF C-display version 2.7.5 Test PC, CanOe compatible (our configuration: Pentium4 2.8 GHz, 512 RAM,40 GB HDD, Win XP) with PCMCIA adapter or CanOe compatible desktop computer	

3.1.1 Showing Parking Assistance

Goals:

- Objective of this test case is to verify that S_PA_POP_RUNNING is displayed correctly when needed conditions are present.

Case Author	Performed by	Date Tested
Antonia Lavrova	Antonia Lavrova	8/28/2006

- VBAT is connected.

[1] **indicate if calibration of the validation tool is up to date or not applicable.**

- Display is in Normal mode and ignition key is ON.
- Set 'Normal (COMMANDES_BSI/ PHASE_VIE=1)
- Turn ignition key in position contact (DONNEES_BSI_LENTES/ ETAT_PRINCIP_SEV=1)
- Turn radio off (ETAT_RADIO_GEN_GEN / POWER=0)
- Radio amplitude off (ETAT_RADIO_GEN_GEN / AMPLI_ON=0)

No	Action	Expected Results	OK NOK NT	Comments	Covered Req.
3.1.1.1	- PA_DATA/ DIST_FL = 5 - Obstacle in Front left side of the car is in Zone 6 - PA_DATA/ DIST_FR = 4 - Obstacle in Front right side of the car is in Zone 5 - PA_DATA/ DIST_BL = 1 - Obstacle in Back left side of the car is in Zone 2 - PA_DATA/ DIST_BR = 2 - Obstacle in Back right side of the car is in Zone 3 - PA_DATA/ DIST_FM = 0 - Obstacle in Front middle side of the car is in Zone 1 - PA_DATA/ DIST_FL = 0 - Obstacle in Front left side of the car is in Zone 1 - PA_DATA/ DISP_RQST = 0 - Parking assistance information is not requested to be displayed	- Display is in inactive state. Parking assistance window is not displayed.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-003 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.1.2	- PA_DATA/ DIST_FL = 6 - Obstacle in Front left side of the car is in Zone 7 - PA_DATA/ DIST_FR = 2 - Obstacle in Front right side of the car is in Zone 3 - PA_DATA/ DIST_BL = 7 - Obstacle in Back left side of the car is in Zone 8 - PA_DATA/ DIST_BR = 2 - Obstacle in Back right side of the car is in Zone 3 - PA_DATA/ DIST_FM = 1 - Obstacle in Front middle side of the car is in Zone 2 - PA_DATA/ DISP_RQST = 1 - Parking assistance information is requested to be displayed	- Display is in inactive state. Parking assistance window is not displayed.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-003 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.1.3	- PA_DATA/ DIST_FL = 4 - Obstacle in Front left side of the car is in Zone 5 - PA_DATA/ DIST_FR = 4 - Obstacle in Front right side of the car is in Zone 5 - PA_DATA/ DIST_BL = 1 - Obstacle in Back left side of the car is in Zone 2 - PA_DATA/ DIST_BR = 0 - Obstacle in Back right side of the car is in Zone 1 - PA_DATA/ DIST_FM = 2 - Obstacle in Front middle side of the car is in Zone 3 - PA_DATA/ DIST_FL = 1 - Obstacle in Front left side of the car is in Zone 2	- Display goes in active state. Parking assistance window is displayed.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-003 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.1.4	- PA_DATA/ DIST_FL = 3 - Obstacle in Front left side of the car is in Zone 4 - PA_DATA/ DIST_FR = 3 - Obstacle in Front right side of the car is in Zone 4 - PA_DATA/ DIST_BL = 7 - Obstacle in Back left side of the car is in Zone 8	- Display is in active state. Parking assistance window is not displayed.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-003 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1

	- PA_DATA/ DIST_BR = 0 - Obstacle in Back right side of the car is in Zone 1 - PA_DATA/ DIST_FM = 6 - Obstacle in Front middle side of the car is in Zone 7 - PA_DATA/ DISP_RQST = 0 - Parking assistance information is not requested to be displayed				
3.1.1.5	- PA_DATA/ DIST_FL = 0 - Obstacle in Front left side of the car is in Zone 1 - PA_DATA/ DIST_FR = 2 - Obstacle in Front right side of the car is in Zone 3 - PA_DATA/ DIST_BL = 0 - Obstacle in Back left side of the car is in Zone 1 - PA_DATA/ DIST_BR = 6 - Obstacle in Back right side of the car is in Zone 7 - PA_DATA/ DIST_FM = 7 - Obstacle in Front middle side of the car is in Zone 8 - PA_DATA/ DISP_RQST = 1 - Parking assistance information is requested to be displayed	- Display is in active state. Parking assistance window is displayed.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-003 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.1.6	- PA_DATA/ DIST_FL = 1 - Obstacle in Front left side of the car is in Zone 2 - PA_DATA/ DIST_FR = 4 - Obstacle in Front right side of the car is in Zone 5 - PA_DATA/ DIST_BL = 0 - Obstacle in Back left side of the car is in Zone 1 - PA_DATA/ DIST_BR = 6 - Obstacle in Back right side of the car is in Zone 7 - PA_DATA/ DIST_FM = 0 - Obstacle in Front middle side of the car is in Zone 1 - PA_DATA/ DIST_FL = 0 - Obstacle in Front left side of the car is in Zone 1	- Display goes in inactive state. Parking assistance window is not displayed.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-003 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1

Case Result (Passed/Failed) : PASSED

3.1.2 Measure distances from obstacles - single obstacle

Goals:

- Objective of this test case is to verify that icons which represent distances to obstacles are displayed correctly when single obstacle is present.

Case Author	Performed by	Date Tested
Antonia Lavrova	Antonia Lavrova	8/28/2006

- VBAT is connected.
- Display is in Normal mode and ignition key is ON.
- Set 'Normal (COMMANDES_BSI/ PHASE_VIE=1)
- Turn ignition key in position contact (DONNEES_BSI_LENTES/ ETAT_PRINCIP_SEV=1)
- Turn radio off (ETAT_RADIO_GEN_GEN / POWER=0)
- Radio amplitude off (ETAT_RADIO_GEN_GEN / AMPLI_ON=0)
- PA_DATA/ DISP_RQST = 0 (Parking assistance information is not requested to be displayed)
- PA_DATA/ DIST_FL = 7 (Obstacle in Front left side of the car is in Zone 8)
- PA_DATA/ DIST_BL = 4 (Obstacle in Back left side of the car is in Zone 5)
- PA_DATA/ DIST_FM = 7 (Obstacle in Front middle side of the car is in Zone 8)
- PA_DATA/ DIST_FR = 6 (Obstacle in Front right side of the car is in Zone 7)
- PA_DATA/ DIST_BR = 4 (Obstacle in Back right side of the car is in Zone 5)

- PA_DATA/ DIST_BM = 7 (Obstacle in Back middle side of the car is in Zone 8)

No	Action	Expected Results	OK NOK NT	Comments	Covered Req.
3.1.2.1	- PA_DATA/ DIST_FL = 0 - Obstacle in Front left side of the car is in Zone 1 - PA_DATA/ DISP_RQST = 1 - Parking assistance information is requested to be displayed	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : ICO_WARNING - In OT_IMAGE_5 : ICO_PA_FL_3 (Front left obstacle is in Zone1) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.2	- PA_DATA/ DIST_FL = 1 - Obstacle in Front left side of the car is in Zone 2	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : <<empty>> - In OT_IMAGE_5 : ICO_PA_FL_2 (Front left obstacle is in Zone2) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.3	- PA_DATA/ DIST_FL = 3 - Obstacle in Front left side of the car is in Zone 4	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : <<empty>> - In OT_IMAGE_5 : ICO_PA_FL_1 (Front left obstacle is in Zone4) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.4	- PA_DATA/ DIST_FL = 5 - Obstacle in Front left side of the car is in Zone 6 - PA_DATA/ DIST_FR = 0 - Obstacle in Front right side of the car is in Zone 1	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : ICO_WARNING - In OT_IMAGE_5 : <<empty>> (Front left obstacle is in Zone6) - In OT_IMAGE_3 : ICO_PA_FR_3 (Front right obstacle is in Zone1) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.5	- PA_DATA/ DIST_FR = 2 - Obstacle in Front right side of the car is in Zone 3	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : <<empty>> - In OT_IMAGE_3 : ICO_PA_FR_2 (Front right obstacle is in Zone3) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.6	- PA_DATA/ DIST_FR = 4 - Obstacle in Front right side of the car is in Zone 5	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : <<empty>> - In OT_IMAGE_3 : ICO_PA_FR_1 (Front right obstacle is in Zone5) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.7	- PA_DATA/ DIST_FR = 6 - Obstacle in Front right side of the car is in Zone 7 - PA_DATA/ DIST_BL = 0 - Obstacle in Back left side of the car is in Zone 1	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : ICO_WARNING - In OT_IMAGE_3 : <<empty>>	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1

		(Front right obstacle is in Zone7) - In OT_IMAGE_6 : ICO_PA_BL_3 (Back left obstacle is in Zone1) - All other parts of the screen are empty.			
3.1.2.8	- PA_DATA/ DIST_BL = 1 - Obstacle in Back left side of the car is in Zone 2	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : <<empty>> - In OT_IMAGE_6 : ICO_PA_BL_2 (Back left obstacle is in Zone2) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.9	- PA_DATA/ DIST_BL = 2 - Obstacle in Back left side of the car is in Zone 3	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : <<empty>> - In OT_IMAGE_6 : ICO_PA_BL_1 (Back left obstacle is in Zone3) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.10	- PA_DATA/ DIST_BL = 3 - Obstacle in Back left side of the car is in Zone 4 - PA_DATA/ DIST_BR = 0 - Obstacle in Back right side of the car is in Zone 1	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : ICO_WARNING - In OT_IMAGE_6 : <<empty>> (Back left obstacle is in Zone4) - In OT_IMAGE_4 : ICO_PA_BR_3 (Back right obstacle is in Zone1) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.11	- PA_DATA/ DIST_BR = 1 - Obstacle in Back right side of the car is in Zone 2	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : <<empty>> - In OT_IMAGE_4 : ICO_PA_BR_2 (Back right obstacle is in Zone2) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.12	- PA_DATA/ DIST_BR = 2 - Obstacle in Back right side of the car is in Zone 3	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : <<empty>> - In OT_IMAGE_4 : ICO_PA_BR_1 (Back right obstacle is in Zone3) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.13	- PA_DATA/ DIST_BR = 6 - Obstacle in Back right side of the car is in Zone 7 - PA_DATA/ DIST_FM = 0 - Obstacle in Front middle side of the car is in Zone 1	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : ICO_WARNING - In OT_IMAGE_4 : <<empty>> (Back right obstacle is in Zone7) - In OT_IMAGE_7 : ICO_PA_F_B_2, ICO_PA_F_B_1, ICO_PA_F_B_1 and ICO_PA_F_B_3 (Front middle obstacle is in Zone1) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.14	- PA_DATA/ DIST_FM = 2 - Obstacle in Front middle side of the car is in Zone 3	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007

		- In OT_IMAGE_1 : <<empty>> - In OT_IMAGE_7 : ICO_PA_F_B_1, ICO_PA_F_B_1 and ICO_PA_F_B_3 (Front middle obstacle is in Zone3) - All other parts of the screen are empty.			$-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.15	- PA_DATA/ DIST_FM = 3 - Obstacle in Front middle side of the car is in Zone 4	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : <<empty>> - In OT_IMAGE_7 : ICO_PA_F_B_1 and ICO_PA_F_B_3 (Front middle obstacle is in Zone4) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.16	- PA_DATA/ DIST_FM = 5 - Obstacle in Front middle side of the car is in Zone 6	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : <<empty>> - In OT_IMAGE_7 : ICO_PA_F_B_3 (Front middle obstacle is in Zone6) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.17	- PA_DATA/ DIST_FM = 7 - Obstacle in Front middle side of the car is in Zone 8 - PA_DATA/ DIST_BM = 0 - Obstacle in Back middle side of the car is in Zone 1	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : ICO_WARNING - In OT_IMAGE_7 : <<empty>> (Front middle obstacle is in Zone8) - In OT_IMAGE_8 : ICO_PA_F_B_3, ICO_PA_F_B_1, ICO_PA_F_B_1 and ICO_PA_F_B_2 (Back middle obstacle is in Zone1) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.18	- PA_DATA/ DIST_BM = 1 - Obstacle in Back middle side of the car is in Zone 2	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : <<empty>> - In OT_IMAGE_8 : ICO_PA_F_B_1, ICO_PA_F_B_1 and ICO_PA_F_B_2 (Back middle obstacle is in Zone2) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.19	- PA_DATA/ DIST_BM = 4 - Obstacle in Back middle side of the car is in Zone 5	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : <<empty>> - In OT_IMAGE_8 : ICO_PA_F_B_1 and ICO_PA_F_B_2 (Back middle obstacle is in Zone5) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.20	- PA_DATA/ DIST_BM = 5 - Obstacle in Back middle side of the car is in Zone 6	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : <<empty>> - In OT_IMAGE_8 : ICO_PA_F_B_2 (Back middle obstacle is in Zone6) - All other parts of the screen are empty.	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.2.21	- PA_DATA/ DIST_BM = 7 - Obstacle in Back middle side of the car is in Zone 8	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 :	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004

		ICO_PA_CAR - In OT_IMAGE_1 : ICO_WARNING - In OT_IMAGE_8 : <<empty>> (Back middle obstacle is in Zone8) - All other parts of the screen are empty.			$-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1

Case Result (Passed/Failed) : PASSED

3.1.3 Measure distances from obstacles - multiple obstacles

Goals:

- Objective of this test case is to verify that icons which represent distances to obstacles are displayed correctly when multiple obstacles are present.

Case Author	Performed by	Date Tested
Antonia Lavrova	Antonia Lavrova	8/28/2006

- VBAT is connected.
- Display is in Normal mode and ignition key is ON.
- Set 'Normal (COMMANDES_BSI/ PHASE_VIE=1)
- Turn ignition key in position contact (DONNEES_BSI_LENTES/ ETAT_PRINCIP_SEV=1)
- Turn radio off (ETAT_RADIO_GEN_GEN / POWER=0)
- Radio amplitude off (ETAT_RADIO_GEN_GEN / AMPLI_ON=0)
- PA_DATA/ DISP_RQST = 0 (Parking assistance information is not requested to be displayed)

No	Action	Expected Results	OK NOK NT	Comments	Covered Req.
3.1.3.1	- PA_DATA/ DIST_FL = 0 - Obstacle in Front left side of the car is in Zone 1 - PA_DATA/ DIST_FR = 0 - Obstacle in Front right side of the car is in Zone 1 - PA_DATA/ DIST_BL = 0 - Obstacle in Back left side of the car is in Zone 1 - PA_DATA/ DIST_BR = 0 - Obstacle in Back right side of the car is in Zone 1 - PA_DATA/ DIST_FM = 0 - Obstacle in Front middle side of the car is in Zone 1 - PA_DATA/ DIST_BM = 0 - Obstacle in Back middle side of the car is in Zone 1 - PA_DATA/ DISP_RQST = 1 - Parking assistance information is requested to be displayed	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : ICO_WARNING - In OT_IMAGE_5 : ICO_PA_FL_3 (Front left obstacle is in Zone1) - In OT_IMAGE_3 : ICO_PA_FR_3 (Front right obstacle is in Zone1) - In OT_IMAGE_6 : ICO_PA_BL_3 (Back left obstacle is in Zone1) - In OT_IMAGE_4 : ICO_PA_BR_3 (Back right obstacle is in Zone1) - In OT_IMAGE_7 : ICO_PA_F_B_2, ICO_PA_F_B_1, ICO_PA_F_B_1 and ICO_PA_F_B_3 (Front middle obstacle is in Zone1) - In OT_IMAGE_8 : ICO_PA_F_B_3, ICO_PA_F_B_1, ICO_PA_F_B_1 and ICO_PA_F_B_2 (Back middle obstacle is in Zone1)	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.3.2	- PA_DATA/ DIST_FL = 2 - Obstacle in Front left side of the car is in Zone 3 - PA_DATA/ DIST_BR = 2 - Obstacle in Back right side of the car is in Zone 3 - PA_DATA/ DIST_FM = 7 - Obstacle in Front middle side of the car is in Zone 8 - PA_DATA/ DIST_BM = 2 - Obstacle in Back middle side of the car is in	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : ICO_WARNING - In OT_IMAGE_5 : ICO_PA_FL_2 (Front left obstacle is in Zone3) - In OT_IMAGE_3 : ICO_PA_FR_3 (Front right obstacle	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1

	Zone 3	is in Zone1) - In OT_IMAGE_6 : ICO_PA_BL_3 (Back left obstacle is in Zone1) - In OT_IMAGE_4 : ICO_PA_BR_1 (Back right obstacle is in Zone3) - In OT_IMAGE_7 : <<empty>> (Front middle obstacle is in Zone8) - In OT_IMAGE_8 : ICO_PA_F_B_1, ICO_PA_F_B_1 and ICO_PA_F_B_2 (Back middle obstacle is in Zone3)			
3.1.3.3	- PA_DATA/ DIST_FL = 0 - Obstacle in Front left side of the car is in Zone 1 - PA_DATA/ DIST_BR = 3 - Obstacle in Back right side of the car is in Zone 4 - PA_DATA/ DIST_FM = 3 - Obstacle in Front middle side of the car is in Zone 4 - PA_DATA/ DIST_BM = 5 - Obstacle in Back middle side of the car is in Zone 6	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : ICO_WARNING - In OT_IMAGE_5 : ICO_PA_FL_3 (Front left obstacle is in Zone1) - In OT_IMAGE_3 : ICO_PA_FR_3 (Front right obstacle is in Zone1) - In OT_IMAGE_6 : ICO_PA_BL_3 (Back left obstacle is in Zone1) - In OT_IMAGE_4 : <<empty>> (Back right obstacle is in Zone4) - In OT_IMAGE_7 : ICO_PA_F_B_1 and ICO_PA_F_B_3 (Front middle obstacle is in Zone4) - In OT_IMAGE_8 : ICO_PA_F_B_2 (Back middle obstacle is in Zone6)	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.3.4	- PA_DATA/ DIST_FL = 2 - Obstacle in Front left side of the car is in Zone 3 - PA_DATA/ DIST_FR = 4 - Obstacle in Front right side of the car is in Zone 5 - PA_DATA/ DIST_BL = 1 - Obstacle in Back left side of the car is in Zone 2 - PA_DATA/ DIST_BR = 0 - Obstacle in Back right side of the car is in Zone 1 - PA_DATA/ DIST_FM = 5 - Obstacle in Front middle side of the car is in Zone 6 - PA_DATA/ DIST_BM = 0 - Obstacle in Back middle side of the car is in Zone 1	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : ICO_WARNING - In OT_IMAGE_5 : ICO_PA_FL_2 (Front left obstacle is in Zone3) - In OT_IMAGE_3 : ICO_PA_FR_1 (Front right obstacle is in Zone5) - In OT_IMAGE_6 : ICO_PA_BL_2 (Back left obstacle is in Zone2) - In OT_IMAGE_4 : ICO_PA_BR_3 (Back right obstacle is in Zone1) - In OT_IMAGE_7 : ICO_PA_F_B_3 (Front middle obstacle is in Zone6) - In OT_IMAGE_8 : ICO_PA_F_B_3, ICO_PA_F_B_1, ICO_PA_F_B_1 and ICO_PA_F_B_2 (Back middle obstacle is in Zone1)	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.3.5	- PA_DATA/ DIST_FL = 4 - Obstacle in Front left side of the car is in Zone 5 - PA_DATA/ DIST_FR = 6 - Obstacle in Front right side of the car is in Zone 7 - PA_DATA/ DIST_BL = 2 - Obstacle in Back left side of the car is in Zone 3 - PA_DATA/ DIST_BR = 1 - Obstacle in Back right side of the car is in Zone 2 - PA_DATA/ DIST_FM = 0 - Obstacle in Front middle side of the car is in Zone 1	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : ICO_WARNING - In OT_IMAGE_5 : ICO_PA_FL_1 (Front left obstacle is in Zone5) - In OT_IMAGE_3 : <<empty>> (Front right obstacle is in Zone7) - In OT_IMAGE_6 : ICO_PA_BL_1 (Back left obstacle is in Zone3) - In OT_IMAGE_4 :	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1

	- PA_DATA/ DIST_BM = 3 - Obstacle in Back middle side of the car is in Zone 4	ICO_PA_BR_2 (Back right obstacle is in Zone2) - In OT_IMAGE_7 : ICO_PA_F_B_2, ICO_PA_F_B_1, ICO_PA_F_B_1 and ICO_PA_F_B_3 (Front middle obstacle is in Zone1) - In OT_IMAGE_8 : ICO_PA_F_B_1 and ICO_PA_F_B_2 (Back middle obstacle is in Zone4)			
3.1.3.6	- PA_DATA/ DIST_FL = 5 - Obstacle in Front left side of the car is in Zone 6 - PA_DATA/ DIST_FR = 2 - Obstacle in Front right side of the car is in Zone 3 - PA_DATA/ DIST_BL = 5 - Obstacle in Back left side of the car is in Zone 6 - PA_DATA/ DIST_BR = 2 - Obstacle in Back right side of the car is in Zone 3 - PA_DATA/ DIST_FM = 2 - Obstacle in Front middle side of the car is in Zone 3 - PA_DATA/ DIST_BM = 7 - Obstacle in Back middle side of the car is in Zone 8	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : <<empty>> - In OT_IMAGE_5 : <<empty>> (Front left obstacle is in Zone6) - In OT_IMAGE_3 : ICO_PA_FR_2 (Front right obstacle is in Zone3) - In OT_IMAGE_6 : <<empty>> (Back left obstacle is in Zone6) - In OT_IMAGE_4 : ICO_PA_BR_1 (Back right obstacle is in Zone3) - In OT_IMAGE_7 : ICO_PA_F_B_1, ICO_PA_F_B_1 and ICO_PA_F_B_3 (Front middle obstacle is in Zone3) - In OT_IMAGE_8 : <<empty>> (Back middle obstacle is in Zone8)	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1
3.1.3.7	- PA_DATA/ DIST_FL = 7 - Obstacle in Front left side of the car is in Zone 8 - PA_DATA/ DIST_BL = 5 - Obstacle in Back left side of the car is in Zone 6 - PA_DATA/ DIST_FM = 7 - Obstacle in Front middle side of the car is in Zone 8 - PA_DATA/ DIST_FR = 6 - Obstacle in Front right side of the car is in Zone 7 - PA_DATA/ DIST_BR = 4 - Obstacle in Back right side of the car is in Zone 5 - PA_DATA/ DIST_BM = 7 - Obstacle in Back middle side of the car is in Zone 8	- Screen S_PA_POP_RUNNING is displayed with : - In OT_IMAGE_2 : ICO_PA_CAR - In OT_IMAGE_1 : <<empty>> - In OT_IMAGE_5 : <<empty>> (Front left obstacle is in Zone8) - In OT_IMAGE_3 : <<empty>> (Front right obstacle is in Zone7) - In OT_IMAGE_6 : <<empty>> (Back left obstacle is in Zone6) - In OT_IMAGE_4 : <<empty>> (Back right obstacle is in Zone5) - In OT_IMAGE_7 : <<empty>> (Front middle obstacle is in Zone8) - In OT_IMAGE_8 : <<empty>> (Back middle obstacle is in Zone8)	OK		$-EMF-PA-IHM-001 $-EMF-PA-IHM-002 $-EMF-PA-IHM-003 $-EMF-PA-IHM-004 $-EMF-PA-IHM-007 $-EMF-PA-FCT-001.1 $-EMF-PA-FCT-002.1

Case Result (Passed/Failed) : PASSED

Речник

вградена система (embedded system) – комбинация от хардуер и софтуер, и евентуално механични части, която е проектирана да изпълнява специфични задачи. Може да включва и операционна система, но може да самостоятелна програма. В повечето случай от вградената система се изисква да отгоаря на заявки в реално време. Автомобилите, медицинското оборудване, камерите и фотоапаратите, мобилните телефони и PDA са част от множеството примери за вградени системи.

тестване на ниво програмна единица (unit testing) – това е основното ниво на тестване. При него всяка програмна единица (компонент) се изпълнява, за да се потвърди,че извършва функцията, за която е предназначена.

интеграционно тестване (integration testing) - извършва се след като е преминало тестването на ниво програмна единица. По време на интеграционното тестване, система се изгражда чрез постепенното прибавяне на един или повече компоненти към ядрото от вече интегрирани компоненти.

системно тестване (system testing) – системното тестване започва след приключване на интеграционните тестове. Целта му е да се провери да софтуерната система изпълнява изискванията на клиента и поведението ѝ в средата, в която ще работи.

тестовете за приемане на системата (acceptance testing) – тези тестове са подобни на системните тестове с тази разлика, че се осъществяват с участието на клиента. Предназначението им е да се установи дали системата отговаря на изискванията и дали е готова да бъде внедрена.

регресионно тестване (regression testing) – при този тип тестове се проверява отново вече тествана част от софтуера. Това се прави с цел да провеи дали отстраняването на дефект не е довело до появата на други.

функционално тестване (black-box testing) – тестовият обект се разглежда като черна кутия – подават му се входни данни и се следи какви изходни данни връща. Не е необходимо да се познава конкретната реализиция и структура на обекта.

динамичен анализ – анализ на софтуера, който се прави чрез изпълнение на програмата.

статичен анализ – анализ на софтуера, който не изисква изпълнение на програмата. Проверява се синтаксиса и се извършва ръчно преминаване през кода с цел откриване на грешки. Този тип анализ се изпълнява обикновено от самите разработчици и с използване на автоматични инструменти.

преглед (review) – процес, по време на който софтуерният продукт се проверява от участниците в проекта, ръководството на фирмата, клиента, потребителите или други заинтересувани от проекта лица. Под „софтуерен продукт” се разбира всеки технически документ или част от него, който е резултат от дейност по разработване на софтуера. Може да включва договори, планове на проекта, спецификации, дизайн, код, потребителска документация, документация за поддръжка, тест-планове, тест-спецификации и др.

инспекция (inspection) – е най-често срещания тип преглед. Тя представлява преглед на определен продукт и се извършва от група опитни участници, които проверяват за дефекти като използват дефинирана процедура за работа. Целта е продуктът да бъде удобрен от всички инспектори.

неформален преглед на алгоритмите (walkthrough) – процес на проверка на алгоритмите и кода чрез следване на разклоненията в алгоритмите или кода, които са определени от избрания набор входни данни. Целта на такава проверка е да се осигури, че алгоритмите и кода са подходящи за решаване на проблема.

тестова стъпка – състои се от тестови действия (установяване на входните променливи в определни стойности), очакван резултат и действителен резултат.

тестов сценарий – състои се от цел, предпоставки и поредица тестови стъпки. В целта се описва какво и защо ще се тества. Предпоставките определят настройките на средата, в която ще се изпълняват стъпките. Средата за изпълнение на всяка стъпка се определя от действията извършение в нея и в предхождащата я.

тестова последователност (test sequence) – поредица от тестови действия, чрез които се стимулира тестовия обект и се имитира работата му в реални условия и среда за продължителен период от време.

тестова група (процедура) – целта на тестовата група е да покрие напълно дадена функционалност на тествания софтуерен продукт. Тестовата група се състои от множество, независими един от друг сценарии.

ефикасност (efficiency) – способността дадена работа да се извърши качествено без загуба на време или разходи. В термините на тестването означава способността на тестовите сценарии да покриват необходимите софтуерни изисквания с минимален брой стъпки.

ефективност (effectiveness) – способността да се произведе желания резултат. В термините на тестването : способността на дефинираните тестове да се справят с откриването на дефекти в софтуерния продукт.

Декартово произведение (Cartesian product) – Декартовото произведение на две множества X и Y (отбелязва се $X \times Y$) е множеството от всички наредени двойки, чийто първи компонент е член на X, а вторият компонент е член на Y. Т.е. $\boldsymbol{X \times Y = \{(x, y) | x \in X, y \in Y\}}$. Тази дефиниция може да се обобщи за N множества.

интуитивно тестване – несистематичен подход за подбор на тестови стъпки, при който се разчита на опита и знанията на тестера.

тестване с произволни данни – несистематичен подход на тестване, при който тестовите данни се генерират на случаен принцип.

еволюционни алгоритми (evolutionary algorithms) – представляват метод за търсене чрез използване на вероятности. В него се използват механизми от биологичната еволюция като възпроизводство (reproduction), мутации (mutation), комбинации (recombination), естествен подбор (selection) и оцеляване на най-добрите (survival of the fittest). Еволюционните алгоритми разглеждат популация от потенциални решения на поставен проблем и прилагат принципа за оцеляване на най-добрия, за да се получи възможно най-добро приближение на решението.

класификационно дърво (classification tree, decision tree) – представлява дървовидна структура, при която листата представят класификации, а разклоненията представят множество от характеривитки, които водят до тези класификации. Класификационното дърво може да се построи чрез разделяне на входното множество на подмножества, базирани на един или повече признака. Класификационното дърво може дасе опише още като съчетание на математически и компютърни подходи, който подпомагат описанието, категоризирането и обобщението на дадено множество от данни.

ad hoc подход – буквално преведено от латински *ad hoc* означава „за тази цел”. В повечето случаи се използва за решение, което е предназначено за специфичен проблем и не е генерално. Използва се и за импровизирано, без предварителна подготовка решение или събитие.

маршрутизатор (router) – хардуерно мрежово устройство, което изпраща данни в мрежата към тяхното направление

релация на еквивалентност – релацията $R \subseteq AxA$ е релация на еквивалентност, ако е рефлексивна $(\forall a \in A, (a,a) \in R)$, симетрична $(a,b \in A, (a,b) \in R \Rightarrow (b,a) \in R)$ и транзитивна $(a,b,c \in A, (a,b) \in R, (b,c) \in R \Rightarrow (a,c) \in R)$

стъб (stub) – е имитация (imitation, emulation) на програмна единица, която се използва на мястото на истинската единица за улеснение на тестването.

драйвер (test driver) – софтуер, който изпълнява друг софтуер с цел да го тества и който предоставя работна среда (framework) за дефиниране на входни параметри, изпълнение на отделна единица и четене на изходните параметри.

граф на изпълнението (control flow graph) – граф с върхове, представящи операторите, и насочени ребра, представящи преходите между операторите. Може да се счита, че такъв граф има точно един начален и един краен връх.

граф на извикванията на функциите (call graph) – граф с върхове, представящи функциите, и ребра, представящи възможните извиквания между функциите. Например, ще има насочено ребро от функция func1 към функция func2, ако func1 извиква func2.

годност (fitness value) – числова стойност, която изразява представянето на индивида спрямо останалите индивиди в популацията.

хипотеза за еднаквост – хипотезата за еднаквост (uniformity hypothesis) е генерализация 1:n на поведението на дадена система. Тя се състои в предположението, че някои състояния и преходи между състояния са еквивалентни, от гледна точка на поведението на системата. Следствие от тази хипотеза е, че еквивалентните входни стойности генерират една и съща реакция от страна на системата. [15]

евристична процедура – техника, чиято цел е да реши даден проблем, и за която е без значение дали това решение е доказало правилността си, но често намира подходящи решения. [27]

съждителна (пропозиционална) логика – при нея твърдението може да е вярно или невярно. Съждителната логика използва верни твърдения, за да се формира или доказва друго вярно твърдение. Тя се състои от две части: представяне (синтактична част, която описва представянето на твърдение) и разсъждения (алгоритмична част, която описва как се създава или доказва ново твърдение). Операциите, които се използват в съждителната логика, са отрицание (‘!’ или ‘$\neg$’), дизюнкция (‘+’ или ‘$\vee$’), конюнкция (‘*’ или ‘$\wedge$’), импликация (‘$\Rightarrow$’) и логическа еквивалентност (‘$\Leftrightarrow$’) [28]

Индекс на използваните таблици

Индекс на използваните фигури

Използвана литература

Лекции

[1] Изборен курс Тестване на софтуер за магистърска програма Софтуерни технологии, летен семестър, 2005, лектор доц. Красимир Манев.

[2] Задължителен курс Управление на качеството за магистърска програма Софтуерни технологии, летен семестър, 2005, лектор Илина Манова.

Книги

[3] Horch, John (2003). *Practical Guide to Software Quality Management, 2nd edition*. Artech House.

[4] Jorgensen, Paul (2002). *Software Testing: A Craftsman's Approach*. CRC Press LLC

[5] Lewis, William (1998). *PDCA/TEST A Quality Framework for Software Testing*. CRC Press LLC

[6] Myers, Glenford (2004). *The Art of Software Testing, 2nd edition*. John Wiley & Sons, Inc.

[7] Rakitin, Steven (2001). *Software Verification and Validation for Practitioner and Managers, 2nd edition*. Artech House.

Статии

[8] Baresel, Andre, Pohlhelm, Hartmut, Sadeghipour, Sadegh (2003). *Structural and Functional Sequence Test of Dynamic and State-Based Software with Evolutionary Algorithms*. Lecture Notes in Computer Science, Volume 2724 Title: Genetic and Evolutionary Computation – GECCO 2003: Genetic and Evolutionary Computation Conference Chicago, IL, USA, July 12-16, 2003 Proceedings, Part II (pp. 2428-2441)

[9] Bybro, Mattias (2003). *A Mutation Testing Tool for Java Programs*. Master Thesis in Computer Science (August 15, 2003)

[10] Conrad, Mirko (2005). *Systematic Testing of Embedded Automotive Software: The Classification-Tree Method for Embedded Systems (CTM/ES)*. Seminar: Perspectives of Model-Based Testing. [Online] Available: http://drops.dagstuhl.de/opus/volltexte/2005/325/

[11] Conrad, Mirko, Fey, Ines, Lamberg, Klaus et al. (2004). *Model-Based Testing of Embedded Automotive Software using MTest*. SAE World Congress 2004, March 8-11, 2004, Detroit, Michigan

[12] Conrad, Mirko, Fey, Ines, Sadeghipour, Sadegh (2004). *Systematic Model-Based Testing of Embedded Control Software: The MB³T Approach*. ICSE 2004 workshop on Software Engineering for Automotive Systems, May 25th, 2004, Edinburgh, Scotland

[13] Herrmann, Jens (2001). *Guideline for Validation & Verification Real-Time Embedded Software Systems*. **I**nformation **T**echnology for **E**uropean **A**dvancement.

[14] Lehmann, Eckard, & Wegener, Joachim (no date). *Test Case Design by Means of the CTE XL*. DaimlerChrysler AG Research and Technology

[15] Martin, Hugues (1999). *Using Test Hypotheses to Build a UML Model of Object-Oriented Smart Card Applications*. 12th International Conference Software and Systems Engineering and their Applications (ICSSEA).Chapter 3.3.3. Uniformity

[16] Ostrand, Thomas, & Balcer, Mark (1988). The Cathegory-partition Method for Specifying and Generating Functional Tests. Communications of the ACM, Volume 31

[17] Polheim, Hartmut, Conrad, Mirko, Griep, Arne (2005). *Evolutionary Safety Testing of Embedded Control Software by Automatically Generating Compact Test Data Sequences*. SAE International.

[18] Wegener, Joachim (2001). *Overview of Evolutionary Testing* Presentation presented at

IEEE Seminar Workshop, 14 May 2001, Toronto

[19] Wegener, Joachim (2004). *Test Case Design by Means of CTM and CTE* Presentation presented at TACOS'04, March 2004, Barcelona

[20] Wegener, Joachim, Grimm, Klaus, Grochtmann, Matthias et al. (1996). *Systematic Testing of Real-Time Systems*. Proceedings of the 4th European Conference on Software Testing, Analysis & Review (EuroSTAR '1996), December 1996, Amsterdam, Netherlands.

Интернет ресурси

[21] Bryson, Brian (2004). *Understanding the Ugly Duckling: Reliability Testing and Rational TestFactory*. [Online] Available: http://www-128.ibm.com/developerworks/rational/library/3773.html

[22] Hitex Developments Tools (no date). Benefits of the Classification Tree Method. [Online] Available: http://www.hitex.com/products.html?con-benefits-of-the-classification-tree-method.html~content

[23] Kaner, Cem, Bach, James. (2005). Lectures on Black Box Software Testing. [Online] Available: http://www.testingeducation.org/BBST/

[24] Marathe, Manish (no date). *Basics of Mutation Testing*. [Online] Available: http://developer.spikesource.com/wiki/index.php/Basics_of_Mutation_Testing

[25] http://www.can-cia.org/can/protocol/index.html

[26] http://www.coleyconsulting.co.uk/testtype.htm

[27] http://en.wikipedia.org/wiki/Heuristic#Computer_science

[28] http://www.oursland.net/aima/propositionApplet.html

[29] CTE XL User Manual Guide. Classification Tree Editor eXtended Logics 1.6.1 build 30.08.2005 is online available for free download at: http://www.systematic-testing.com/functional_testing/cte_main.php?cte=1

[30] Vector CANoe Help v4.0.4