11. Формални модели и динамични софтуерни архитектури Резюме

I. Формални модели на софтуерната архитектура

Формални методи - Формалното моделиране представлява част от по-общ набор от технологии, познати като “формални методи”. Базират се на математическо представяне и анализ на софтуера. Формалните методи в софтуера включват:

- Формално моделиране (спецификация);
- Анализ и доказателство на спецификацията;
- Разработка, базирана на трансформации
- Програмна верификация

Необходимост от формални методи в софтуерното инженерство:

- Формален анализ на софтуерни системи
- Динамични архитектури
- Генерация на код

Всички те обаче са неприложими без съответните инструменти

Минуси на формалните методи:

- Ефективността на неформалните техн. в СЕ в разл. проблемни области, намалява необходимостта от формални методи
- Формалните методи не намаляват времето за разработка
- Не са подходящи за описание на потребителски интерфейс и взаимодействие
- Проблеми с приложението в големи и развиващи се системи

Формални модели на софтуерната архитектура: CSP (communicating sequential processes); π – calculus; Z schema; Мрежи на Петри …

Видове методи за описание:

- **Алгебрично описание**- С се описва като набор от операции и отношенията между тях
- **Описание базирано на състоянието**- С се описва чрез модел на състоянието, а операциите -като промени в състоян

Z-нотация (notation) - Разработена в края на 80-те . Първоначално не е била предназначена за описание на софт. С. Комбинира формално и неформално описание и има възможност за графичен модел

Езици за описание на архитектурата: Darwin, Wright, Rapide, ACME, Koala, xADL, AADL, π-ADL, PADL, UML, ComponentJ, ArchJava

Модели на качествени характеристики

Изправност (Dependability) - Една от важните характеристики на софтуерните системи. Характеризира се с няколко съставящи атрибута: **Наличност** (Availability); **Надеждност** (Reliability); **Безопасност** (Safety) и др. За разлика от други КХ има добре разработени показатели за измерване. Интерес представляват формалните методи за изследване на изправността и атрибутите й

Софтуерна надеждност- Леко се различава от традиционната математическа теория на надеждността. Отказите са детерминистични събития, поведението на потребителя не е. Дефинира се като вероятност за отказ в рамките на даден времеви интервал. На практика се използва като индикатор за това колко тестване на системата е достатъчно.

Как измерваме надеждността на софтуер: Разл. явни метрики:

Вероятност за отказ (успех) – показва колко е вероятно софтуерът да се провали или да работи успешно в даден период от време.

Средно време до отказ (Mean Time to Failure - μ) –средното време, в което софтур. работи без грешки, преди да се случи първият отказ.

Честота на откази (Failure Rate - λ) – колко често се случват откази в опр. период от време. Формулата за честотата на откази е:

$\lambda=1/\mu$ **Честота на откази = 1 / Mean Time to Failure**

Това означава, че **честотата на откази е обратнопропорционална на средното време до отказ – колкото по-дълго работи софтуерът без грешки, толкова по-ниска е честотата на откази**.

Неявни метрики: Брой редове код, Test coverage и др.

Измерването на надеждността на софтуер е трудно, поради сложната природа на софтуерните продукти, която все още остава неразбрана Метрики, които са очевидни, като размер на софтуера, все още нямат общоприета дефиниция. Отказите на софтуера, за разлика от тези на хардуера са детерминистично явление, поведението на потребителя не е детерминистично. Трудно е да се оцени надеждността на компонентно-базирана С, поради сложността на възможните взаимодействия между съставящите я компоненти

Методи за оценка на надеждността:

Модели от тип черна кутия - Основани са на статистическа обработка на данни, най-често от тестване на системите: Software Reliability Growth Models (SRGMs)

Модели от тип бяла кутия - Базират се на данни за вътрешната организация на С, напр. – архитектурата. Формализират процеса на отказите на компонентите в С. Интегрира се поведението на отказите със модел на архитектуата. Методите на бялата кутия не изключват тези на черната. Съществува значителен проблем с гарантирането на много висока надеждност: $\sim (1- \mid 0^{-9})$ само чрез тестване 99,999%

Изчисляване чрез данни от тестването

Доказано е, че е неподходящо надеждност на критични системи да се изчислява само чрез тестване [Butler & Finelli 1993]. Само малки

надеждности (99,999%) може да се изчислят на базата на данни, придобити от разумно дълъг период на тестване

За да сме сигурни че дадена С има надеждност от порядъка на (1–| 0^{-7}), трябва да тестваме хиляди години без прекъсване –т.е С да е изключително надеждна (никакъв шанс за отказ- 0.0000001), ще трябва да я тестваме непрекъснато в продължение на хиляди години, за да сме сигурни, че няма да се повреди.

Модели на черната кутия - Разработени са десетки такива модели. Те се основават на данни от тестването на софтуер: време между отказите, брой откази и т.н. Правят се редица опростяващи допускания, които влошават качеството на оценката за надеждност. Отказите се подчиняват на някакво статистическо разпределение: Експоненциално – модел на Jelinski-Moranda; Поасново (Nonhomogeneous Poisson process – NHPP) – модели на Musa

Базов модел за надеждност на Муса обяснява как се променя честотата на отказите на една система по време на тестване. Формулата за честотата на отказите е**:** $\lambda(\mu)=\lambda_0(1-\mu/\nu_0)$

$\lambda(\mu)$ – текущата честота на отказите в системата.

λ_0 – началната честота на отказите в самото начало на тестването

μ – средният брой открити и отстранени грешки до даден момент

ν_0 – общият брой грешки, които първоначално съществуват в С.

Текуща честота откази = нач.честота откази * (1 - среден брой открити грешки/ общ грешки същ. в с-мата)

В началото на тестването С има много бъгове, затова честотата на отказите е висока. Колкото повече бъгове откриваме и отстраняваме (увеличава се μ), толкова по-ниска става честотата на отказите. Ако сме отстранили всички бъгове ($\mu = \nu_0$), честотата на отказите е 0.

Формула за броя на откритите грешки във времето

$\mu(\tau)=\nu_0(1-e^{-\lambda_0\tau/\nu 0})$

$\mu(\tau)$ – средният брой открити грешки до момента τ.

ν_0 – общият брой бъгове, които съществуват в системата в началото.

λ_0 – началната честота на отказите, когато започнем тестването.

τ – времето на тестване.

e – основата на естествения логаритъм (около 2.718), често използвана в експоненциални формули.

В **началото на тестването ($\tau = 0$)**, експоненциалната част $e^{-\lambda 0\tau/\nu}$ е равна на 1, така че $\mu(0)=0$ – т.е. не сме открили още никакви грешки.

С **увеличаване на времето (τ)**, експонентата намалява към 0, така че $\mu(\tau)$ се приближава до $\nu 0$ – в крайна сметка откриваме почти всички първоначални грешки в системата.

Колкото повече време тестваш системата, толкова повече грешки ще откриеш. В началото намираш грешките бързо, но с времето остават само трудните за откриване бъгове и п-сът се забавя.

Модели от тип „бяла кутия:

- Модели на състоянието (state-based models)
- Модели на пътя на изпълнение (path-based models)

Модели на състоянието- Основни стъпки при моделирането:

-Идентификация на модулите в системата

-Построяване на модел на архитектурата на системата-Характерно тук е че трябва да се определи т.нар. профил на употреба – т.е. кои компоненти в С с каква вероятност/честота се използват

-Определяне на поведението на отказите на всеки модул – намиране на конкретна стойност за надеждността

-Комбиниране на модела на архитектурата с поведението на отказите

Някои общи за всички модели допускания

-Данните за надеждността на компонент. са предварително известни

-Профилът на употреба е предварително известен

- Отказите на компонентите са независими събития

- Преходите от един компонент към друг са незавсими събития

- Вероятността за отказ на даден компонент е константа и не се променя във времето

Модели на състоянието (State Models) - използват **верига на Марков**, за да опишат как една С преминава от едно състояние в друго по време на работа.

Състоянията представляват различните компоненти на С или състоянията, в които тя може да се намира (работещ, отказал..)

Веригата на Марков- карта, която показва всички възм. състояния и как се преминава от едно към друго с определена вероятност.

Основни елементи:

$\mathbf{R_i}$ – *Надеждност на компонента i*: показва колко вероятно е компонентът да работи без грешки за определен период от време.

$\mathbf{P_{ij}}$ – *Вероятност за преход от компонент i към компонент j*: показва колко е вероятно С да премине от едно състояние към др. по време на работа. Зависи от начина, по който С се използва

Представи си, че C е като пътуване между градове (състояния). Всеки град е компонент от C. $\mathbf{R_i}$ казва колко е стабилен всеки град (компонент) – дали ще останеш там без проблеми. $\mathbf{P_{ij}}$ е шансът да отидеш от един град в друг по време на пътуването.

Така моделът показва как системата „се движи“ през различни състояния и как това влияе на нейната надеждност.

Модел на Рьоснер/Ченг (Reussner/Cheng Model) - използва **матрица S**, за да опише как C преминава от едно състояние в друго.

- **Матрица S -** таблица, в която се записват вероятностите за преходи между различни състояния на системата.
- В **матрицата S** се добавят специален ред и колона за:
  - **Начално състояние (I)** – откъде започва изпълнението.
  - **Крайно състояние (J)** – където приключва процесът.

Формула за надеждността на системата: $\mathbf{R_{system}=1-S_{IJ}}$

Означава, че надеждността на C зависи от вероятността за преминаване от начално към крайно състояние, като се вземат предвид всички възможни грешки по пътя.

Модели на пътя на изпълнение (Execution Path Models) - също се използва **верига на Марков**, но се фокусира върху **конкретните пътища**, които C преминава по време на работа. **Какво се прави?**

-Провеждат се **N теста**, всеки по определен път на изпълнение.

-За всеки път се изчислява неговата надеждност.

Формула за надеждността на системата: $R_c=\sum_{t\in T}\prod_{m\in M(t)}Rm$

Rc — това е общата надеждност на системата.

T – всички тестове, които сме направили.

M(t) – компонентите, през които минава системата за всеки тест.

$\mathbf{R_m}$ – надеждността на всеки компонент.

Произведение **(∏)**: За всеки тест t умножаваме надеждността на всички компоненти, които участват в него. Това ни дава общата надеждност на теста t. **Сума (∑)**: събираме надеждностите на всички възможни тестове t, за да получим общата надеждност на C.

Циклично изпълнение на компонент (loop) намалява значително надеждността, защото колкото повече пъти се използва един и същ компонент, толкова по-голям е шансът за грешка. **Профилът на употреба** (как се използва системата в реалността) оказва по-голямо влияние върху надеждността, отколкото самата архитектура.

Ако един компонент се използва многократно (например в цикъл), рискът от грешка се увеличава. Начинът, по който реално използваме C, е по-важен за надеждността ѝ, отколкото как е проектирана „на хартия“.

Архитектурна адаптация и динамични софтуерни архитектури

Софтуерна архитектура -Съвкупност от структури, които показват абстрактни елементи и връзките между тях.

Модулни структури - Да си представим, че модулите са елементи не само по време на дизайна на архитектурата, но и по време на изпълнение. Ако можем формално да описваме някакви правила за конфигурация на модулите, то може да дефинираме правила, по които тя да се променя; Ако архитектуриата на дадена C се променя по време на изпълнение (runtime), то говорим за динамични софтуерни архитектури.

Автономни софтуерни системи(ACC) - Autonomic Computing [IBM, 2003] - Компютърна среда, която може да се самоуправлява и динамично да се променя според изискванията и целите на бизнеса. Самоуправляващите се среди може да извършват автономни дейности, без нуждата от намеса на ИТ специалисти

Свойства на автономните системи:

- **Самоконфигуриране** (Self-configuring)
- **Самолекуване** (Self-healing)
- **Самооптимизиране** (Self-optimizing)
- **Самозащита** (Self-protecting)

Самоконфигурирането -динамизъм - Самоконфигуриращите се компоненти се променят динамично, вследствие на промени в средата с която взаимодействат. Сред възможните промени са: Свързване на нови компоненти; Изключване на същесвтуващи компоненти; Промени в характеристиките на системата

Самолекуване - С може да открива и диагностицира проблеми, и сама реагира на тях. Самолекуващите се компоненти откриват дефекти и извършват действия като: Променят собственото си състояние; Променят други компоненти, с които взаимодействат

Самооптимизиране- Системите или техни компоненти може да наблюдават и автоматично да променят ресурси, за да отговорят на нуждите на потребителите и бизнеса. Някои възможни промени са: Преразпределяне на ресурси, за да се подобри коефициента на употребата им; Подсигуряване на времето за изпълнение на специфични бизнес транзакции; Текущата практика е оптимизацията да се извършва от операционните системи

Самозащита - компонентите може да предвиждат, откриват, идентифицират заплахи и да предприемат действия за да намалят уязвимостта си. Това може да се прави поне за най-известните заплахи към сигурността, които включват: Неоторизиран достъп и употреба; Вируси и троянски коне; Атаки от тип “denial-of-service”

Примери за действия на автономни софтуерни системи:

- Самоконфигуриране: Инсталация на липсващ софтуер;

- Самолекуване- Промяна на конфигурация, например локация на някакъв компонент, за да може той да бъде намерен;

- Самооптимизиране - Промяна на натоварването в следствие на увеличаване или намаляване на капацитета на компонент/ресурс;

- Самозащита- Извеждане на определени ресурси offline ако бъде засечен опит за проникване

Архитектура на автономни софтуерни системи - Класическо (хардуерно) инженерно решение - Принцип на обратната връзка (Control Loop). **Архитектурата се изгражда на пет нива**: Managed Resources (MR); Touch-Points (TP); Touch-Point Autonomic Managers (TPAM); Orchestrating Autonomic Managers (OAM); Manual Manager (MM)

Managed Resources (MR) - Хардуерен или софтуерен компонент, притежаващ интерфейс, посредством който може да се управлява: Server; Storage unit; Database; Application server; Service; Network router; Personal computer; Application and etc. MR може да има вградена собствена управляваща архитектура

Touch-points(TP) - Модули, в които са реализирани сензор и ефектор, за даден MR - аналог на датчик и управляващ механизъм. Осигуряват управляващ интерфейс за достъп и настройка на разпределени MR. Пр: Log files; Events; Commands; Application programming interfaces; Configuration files

Сензори и ефектори

Съставни части на сензора- Свойства, които предоставят информация за текущото състояние на MR и са достъпни чрез типични “get” методи. Събития (events, messages, notifications), които настъпват, когато състоянието на MR се променя

Съставни части на ефектора -Операции (методи), които позволяват да се променя състоянието на MR. Операции,

реализирани в TPAM, които позволяват на MR да прави заявки към своя TPAM

Touch-point autonomic managers (TPAM) - Работят директно с MR чрез техните TP. Реализират специфични интелигиентни обратни връзки (control loops). Използват се определени политики за управление на поведението на интелигиентните обратни връзки

Orchestrating autonomic managers(OAM): TPAM управляват само ресурсите (MR), за които пряко отговарят; OAM координират действията на TPAM

Software product line (SPL) - набор от софтуерни системи, които имат общи функционалности (features), които удовлетворяват специфичните нужди на определен пазарен сегмент и са разработени от набор основни ресурси според предварително предначертан план

Проблемни области и продуктови линии - Продуктовите линии представят решения на задачите в проблемните области. Продуктовите линии намират успешно приложение в следните области: Мобилни телефони, С за управление на технологични процеси, Транспортни С, Финансови С, Битова електроника

Разработка чрез продуктови линии (ПЛ) - ПЛ са аналог на поточните линии при производството на хардуер от унифицирани елементи. Широко се използват в областта на вградените С, където изделята представляват комбинация от софтуер и хардуер. ПЛ е добър подход, когато дадена организация разработва набор продукти, които имат много общи функционалности. Често се случва дадена организация да цели да овладее голям пазар и всеки продукт в ПЛ е ориентиран към специфичен пазарен сегмент.

Продуктови линии и самостоятелни С - Когато някоя от основните функционалности в ПЛ се промени, тази промяна се отразява във всички производни С. ПЛ може да еволюира и тогава всички С в нея също еволюират. Самостоятелните С нямат горните предимства

Ползи от продуктовите линии - Увеличена производителност; Разходите за разработка и поддръжка на се разпределят между всички продукти в ПЛ; Крайната цена и време за разработка на нов продукт се понижава; Дефекти в основнатите функционалности на ПЛ се отстраняват веднъж за всички продукти

Производство на продукт: Под продукт на ПЛ разбираме конкретна софтуерна С. Продуктът представлява своеобразен екземпляр (инстанция) на ПЛ. Това става на две стъпки: **Селекция**: отстраняват се ненужните елементи (assets) и се уточняват възможните вариации; **Разширяване**: добавят се допълнителни елементи (вероятно разработени от нулата). Селекцията е основна стъпка. Трудността идва от това как да подберем основните елементи на ПЛ. Възможните решения се основават на ключови думи, атрибути и т.н. В момента най-често използвания подход се базира на функционалности (features).

Функционалност (Feature): Може да се разглеждат като абстракция на характерни понятия от терминологията на дадена проблемна област; Може да се използват като средство за комуникация между заинтересованите лица; Пр: “forward” , “reply” и “reply all” може да се разглеждат като features на email клиент

Ако има голямо количество общи компоненти в различните версии е разумно да се използва ПЛ

Product Lines for Evolution - Продуктите в ПЛ може да са резултат и на промени във времето.