

# ЛЕКЦИЯ 1. Методи на информационното моделиране

Информационното моделиране е важна дейност при разработването на компютърно базирани информационни системи.

Информационният модел засяга част от организацията и нейното обкръжение и предоставя средства, с които информацията се записва и начини, по които информацията се обработва.

Информационните модели като цяло приемат формата на entity-relationship модели, диаграми за поток на данните (поточни диаграми) и история на entity.

Информационният модел се проверява от потребителите за да е сигурно, че техните изисквания за информационната система са разбрани коректно и пълно, и той се използва по-късно за проектиране на компонентите за информационна обработка от информационната система за тази организация.

Информационните модели на организацията често включват спецификация на изискванията. Стремежът към повторното използване на части от съществуващите системи, или реинженеринг на системата, е много труден без наличието на такива модели. Ако моделите съществуват в стандартна форма, комуникацията между потребителите и системните разработчици може да се подобри.

"Сближаването" засяга преместване от разнообразни, съществуващи методи към "един най-добър метод", докато „хармонизирането“ предполага съществуването на референтен метод, с който съществуващите методи имат явни съответствия.

Въпреки това, ако резултата от „сближаването“ е на достатъчно високо равнище то може да бъде еквивалентно на хармонизация.

# Основни цели

Нашата първа цел е да анализираме и сравним възможностите на информационното моделиране за избраните методи за разработване на системи, за да разберем техните прилики и разлики, чрез референтната рамка от общи концепции и гледни точки за моделиране.

Втората ни цел е да покажем как такава референтна рамка може да се използва за метод на хармонизация чрез представяне въз основа за определяне на метод за съответствия.

Пет от методите, които сравняваме са информационния инженеринг (IE), структурирани системи за анализ и метод за проектиране (SSADM), Metodologica Informatica (MEIN), MERISE, и Coad и Yourdon's Обектноориентиран анализ (OOA).

# Втора цел

Ще установяваме, дали методите имат широк или тесен обхват един спрямо друг и ще сравним обектноориентирания с традиционните методи.

Ще очертаем насоки за разработчика, където има избор на моделиране, за да бъдат направено и накратко посочено как направения избор може да се разглежда като предимства или недостатъци, свързани с набор от критерии за качество на информационното моделиране.

Ще обсъдим различните ситуации, които могат да благоприятстват или да не благоприятстват даден избор за моделиране. Такива насоки могат да бъдат част от една бъдеща, по-усъвършенствана референтна рамка.

В допълнение, за да подпомогнем разбирането на концепциите на метода и да покажем как те могат да бъдат използвани по интегриран начин, ще обсъдим общите принципи за изграждане на информационни модели, а също ще предоставим коментари и принципи, които описват как сме изградили специфични модели за примери от практиката.

# Традиционни методи

Четири от петте метода, които ще сравним са представител на "традиционнния" подход към развитие на системите. Този подход е в широка употреба през последните двадесет години и е основа за по-голямата част от софтуерни инструменти, които в момента на разположение за подкрепа на развитието на системата.

Три от методите, SSADM (Великобритания), MERISE (Франция) и Mein (Испания), са избрани, тъй като те са най-широко използваните методи в съответните европейски страни, докато четвъртия метод, софтуерен инженеринг, се използва широко в международен мащаб. В допълнение, тези методи са предмет на обсъждане в Euromethod, и са най-подробни по отношение на информация, моделиране, и са най-подходящи за нашето сравнение.

# Обектноориентирани методи

Обектноориентирани методи имат предимства пред традиционните подходи. За да се проучат тези твърдения, петият метод с когото сравняваме е обектноориентиран анализ (OOA) (Coad и Yourdon 1991 г.), който е добре познат обектноориентиран метод за моделиране на информация.

# Какво моделира информационния модел

Има различни позиции, между които моделирация може да се колебае, когато решава какво ще се моделира, и позицията, която той ще заеме ще зависи от неговата гледна точка за естеството на действителността, което се нарича онтологичен поглед.

В контекста, с който сме свързани, гледната точка на наивния реализъм е, че организациите са обективни явления, които съществуват независимо и се възприемат по един и същ начин от всеки наблюдател, и модела за тази информация е модел на тази обективна реалност.

Към другия край на онтологичния спектър е това, което можем да наречем погледа на „субективизма“, който е че организацията е явление, чиято "реалност" е продукт на възприятията на наблюдателя, и следователно е индивидуална за всеки наблюдател. При този подход няма обективна реалност и информационния модел е модел на възприеманията на един или повече наблюдатели на организацията.

И двета възгледа са по-скоро крайни, и подхода който ще използваме е на социалния конструктивизъм. Той гледа на организацията като на социално изградено явление, чито много аспекти, могат да се възприемат по различен начин от различните наблюдатели, но които са в състояние, чрез преговори, обсъждане или някакъв друг метод, да формулират общ поглед за период от време, съдържащ най-малко несъответствия, които могат да се използват като основа за изграждане на информационния модел.

Ще затворим това кратко описание на тази важна тема, като посочим важното предположение, което правим: процесът на постигане на обща гледна точка за организацията е завършен преди започването на дейността по моделиране и тази обща гледна точка е на разположение за тази дейност.

# Какво е метод?

Метод за разработка на системи, или метод, може да бъде дефиниран, както следва:

*Метод е интегриран набор от дейности и продукти за спецификацията и създаването на информационна система.*

Процесът на прилагане на метод за разработване на информационна система се нарича *процес на развитие на системата*.

Определението се отнася до спецификацията или генерирането като някои методи не обхващат всички системи по процеса на развитие, както и произвеждането на спецификации, а не реалните системи.

Има няколко термина, които се употребяват. Те са до голяма степен еквивалентни като методологични методи, разработване, методи и подходи за разработване на системи. Техниката обикновено е по-специфична, като се позовава на добре дефинирана задача, която се занимава с определена област от процеса на разработване.

## Фази и продукти

За да се намали сложността на метод, той обикновено се разделя на различни фази, състоящи се от дейности и по-малки задачи. Дейностите и фазите генерираят продукти, които могат да бъдат, например - описанието на изискванията на потребителя, част от спецификацията, парче от програмен код или операционна система.

## Концепция на модел, моделиране и моделиране

Един модел може да се дефинира общо като абстрактно представяне на част от реалността. Ще използваме термина модел, за да се позовем на абстрактно представяне на част от организация, която представлява продукт на частта за информационното моделиране на метод. Той е общ за тези модели, които включват аспекти на организационната среда.

Терминът моделиране се отнася до дейността на изграждане на модел, докато концепцията на моделирането е концептуален "градивен елемент", който е използван за изграждане на модел и това е осигурено от информационното моделиращата част на метод.

Трябва да отбележим, че терминът "модел" може да се използва по много различни начини в литературата за информационните системи.

# Цели на методите

Като цяло, методите имат за цел създаването на информационни системи, които притежават високо качество и производителност.

Концепцията на високото качество е, че системата трябва да отговаря на изискванията на всички свои потребители, а на висока производителност концепцията е, че системата трябва да бъде разработена навреме и в рамките на бюджета.

Една некачествена система може да стане неизползваема или да бъде саботирана. Тя ще бъде загуба на ресурси за организацията и дори може да има ефект поставянето на организация извън бизнеса, особено ако засяга критична нейна част. Система с лоша производителност може да обезсили нейната полза, ако тя струва повече от планираното.

# Еволюция на методите

## Ранна ера (преди 1970)

В този период, са били идентифицирани само дейности по анализ и програмиране. Потребителските изисквания са анализирани в програмните спецификации от анализаторите на системи, които са ги насочили към програмистите. Дейностите са само слабо засегнати и само диаграма за прогреса на програмата е единственият стандарт.

Подчертаната дейност е била програмирането, като ефективността при изпълнението на програмата, по отношение на скоростта на изпълнение или на основния размер на паметта, са били основните съображения, поради хардуерните ограничения.

Пример за метод от този тип във Великобритания е препоръчен от Националния компютърен център, описан от Daniels и Yeates (1969).

## Структурирани методи - 1970 Структурно програмиране

Дейностите на фазата на програмирането постепенно става все по-определенна, което отчасти се дължи на основен принцип, че само три конструкции - линейност, разклоненост и цикъл - са достатъчни за да се пишат програми с една входна и изходна точка (Bohm and Jacopini 1966; Dijkstra 1968).

## Структуриран дизайн

При функционалния подход, Стивънс и др. (1974) предлага декомпозиция на система на определен брой йерархични функции или модули, показвайки ги на структурна диаграма и прилагайки правила за свързване и сближаване за управление на проектирането. Поточните диаграми са използвани за да се идентифицират основните трансформации на данни и подсистеми.

При данновия подход (Jackson 1975 г.; Warnier 1974), структурата на входните и изходните данни, е била използвана, за да се определи структурата на програмата по отношение на нейните йерархични функции.

## Структурен анализ

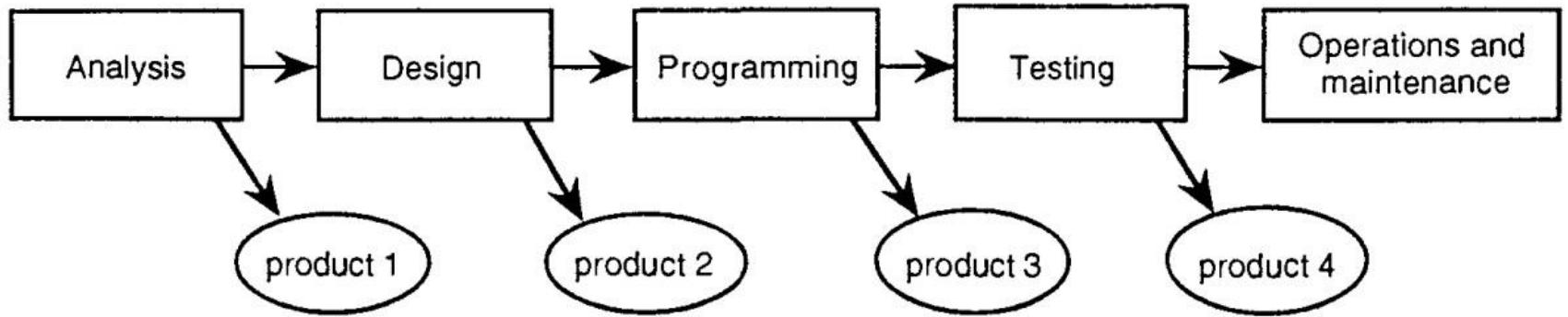
Той разграничава фазата на анализ от проектирането, с цел да осигури нетехническо описание на изискванията на потребителите. Той е съсредоточен върху диаграмата на потока от данни (DFD) (De Marko 1979; Gane and Sarson 1979; Yourdon and Constantine 1979), който моделира изискванията по отношение на йерархично разлагане на процеси и потоци от данни, вместо на програми и файлове на фазата на проектиране. Най-високо ниво на потока от данни на диаграмата дава преглед на системата за изясняване на изискванията. Осигурява улеснения за по-подробна спецификация, като например миниспецификации, таблици за решения, дървета за решения и речник на данните.

## Модел „лавина“

Описани са тези фази и продукти от общ модел на развитие на системите, показан на фигура 1.

Моделът „лавина“ добавя все повече и повече детайли през следващите фази, създавайки все по-голям обем документация.

Разделението между fazите е важно, тъй като създава концепцията на абстракция. Ранните fazи и техните продукти са на високо равнищена абстракция и се фокусират върху това какви са изискванията на потребителите, като се абстрагират от по-ниско ниво fazи, които са загрижени как да бъдат изпълнени изискванията.



фиг. 1. Модел „лавина“

## Ориентирани към данните методи - 1980

Структурните методи предполагат, че само потребителите искат да компютъризират текуща система, за да получат по-висока ефективност. Въпреки това, новите системи често изискват значително участие на потребителя и дискусия. Това води до акцент върху фазата на анализа.

В допълнение, структурните методи са процесоориентирани, с ефекта, че спецификациите често са твърде подробни и компютърно-ориентирани за потребителските разбирания.

## Ориентация към данни

Това е моделиране на данни или обекти. То притежава ефекта на по-ясно разграничаване на етапа на анализ чрез разглеждане на изолирани от процеси обекти, което е по-абстрактно и окуражава нови начини да се мисли за организацията.

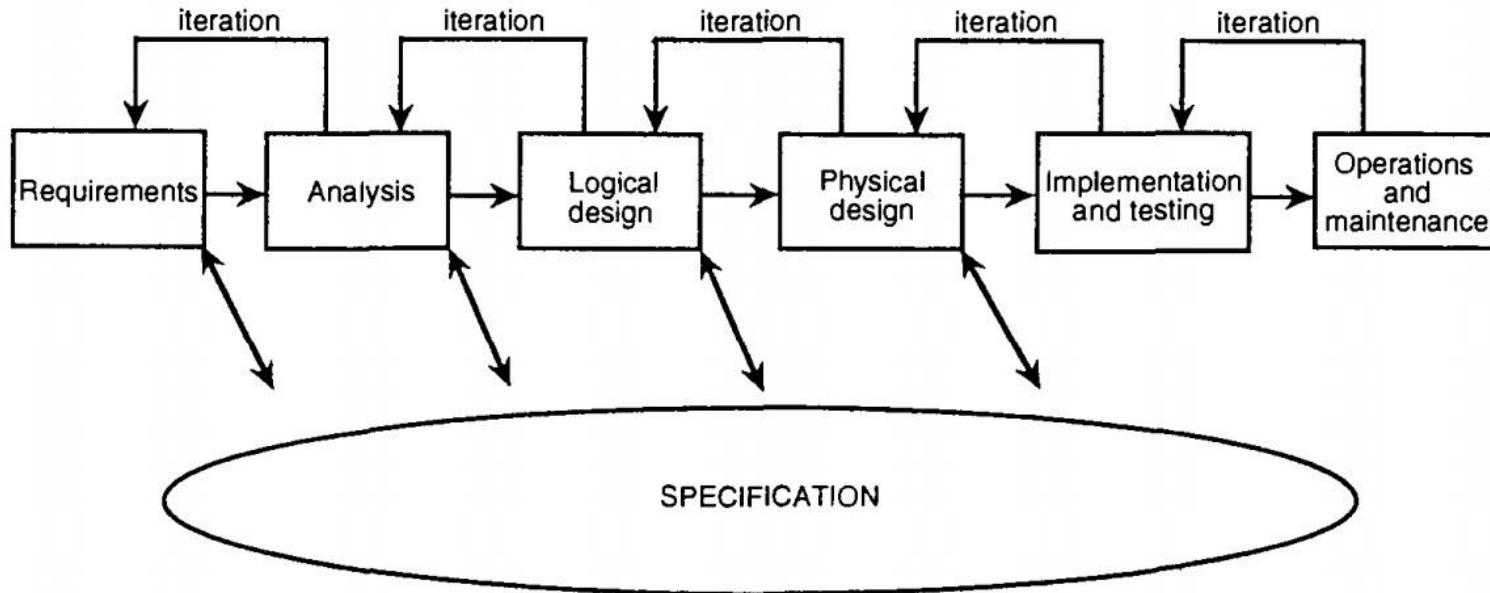
Ориентиране към потребителя ~ 1980

Съществуващо едно нарастващо разбиране, че много системи не отговарят на изискванията на потребителите. Необходимо е потребителите да се включат по-активно в процеса, така че те биха могли да проверят спецификацията. Следователно, продуктите са правени по-лесни за разбиране.

По-голям обхват от дейности за определяне на изискванията на потребителите се появи, като стратегическо планиране и определяне на разходите и ползите, които са полезни за по-новите видове системи, посочени по-горе.

## Итеративен модел

Моделът „лавина“ се превърна в един итеративен модел на развитие на системите (фиг. 2). Итерацията е между една или повече фази, както и тя добавя акцент върху ранните фази, което води до появата на изисквания за определена фаза. Различните продукти, показани на фиг. 1, сега са част от една по-интегрирана спецификация.



фиг. 2. Итеративен модел за разработване на система

## Итеративни фази на модел

Типичните дейности във всяка фаза могат да бъдат описани, както следва:

- *Изисквания.* Предизвиква и улавя изискванията от страна на потребителите.
- *Анализ.* Анализ на изискванията на потребителите в детайли и изразяването им точно в спецификацията.

Тази фаза основно се занимава с информационни дейности за моделиране, които изграждат информационни модели на важната част от организацията и околната среда, с акцент върху елементите, за които трябва да се запише информацията, и елементите, които изпращат и получават информацията и начина, по който информацията се обработва.

- *Логическа архитектура.* Спецификацията се използва за производство на проект за компютърно-базирана система, която ще служи като основа на внедрена информационна система. Често взаимодействието между човека и компютърната система се проектира тук.

*Физически дизайн.* Физическият дизайн е насочен към хардуера, софтуера, човешки и организационни компоненти.

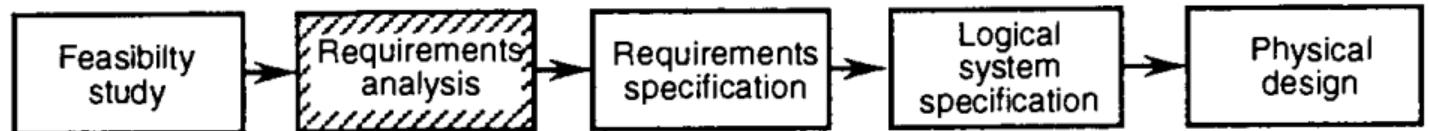
*Внедряване и тестване.* Системата е внедрена в софтуер и човешки процедури и е тествана.

*Поддръжка.* Поддръжката се състои от дейности, които искат и правят промени в системата.

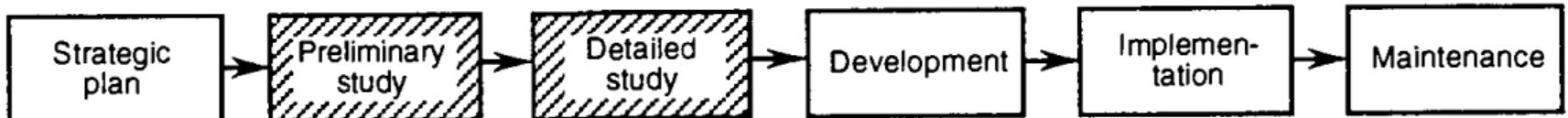
## Традиционни методи

Четири от методите, които ще разгледаме са Информационен инженеринг, SSADM, MEIN и MERISE. Те са примери за методи ориентирани към данните и фиг. 3 показва техните основни фази.

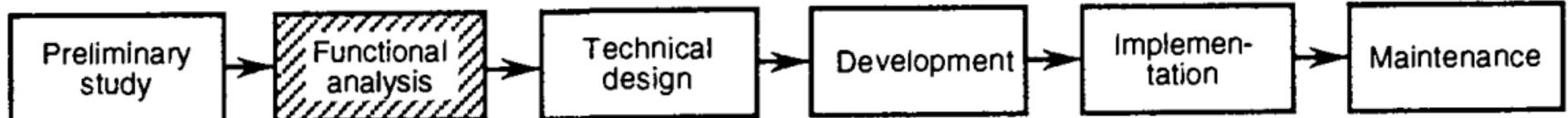
Интересуваме се предимно за информационните дейности за моделиране. Те се намират във фазата на анализ. Моделирането на информацията е важно, тъй като произвежда посочване на информацията и нейната обработка.



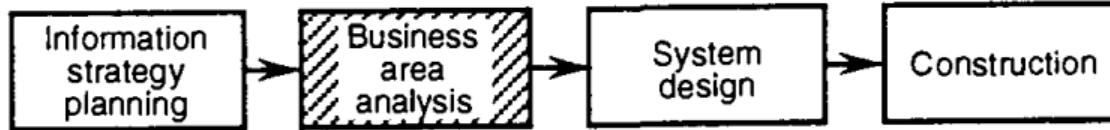
Structured Systems Analysis and Design (SSADM)



MERISE



Metodologica Informatica (MEIN)



Information Engineering (IE)

Information modelling phase

Фиг. 3. Основни фази на Информационното инженерство, SSADM, MERISE и MEIN.

Всички методи имат предходна фаза, в която се определят изисквания, и последваща фаза, когато спецификацията се използва за логически дизайн.

## Обектноориентирани методи

Ще сравним обектноориентирания (ОО) метод, обектноориентирания анализ (Coad и Yourdon 1991 г.) с четирите методи ориентирани към данни, които сме нарекли "традиционнни" методи.

ОО подхода е еволюиран от първите ОО програмни езици, като например Simula 1967 и Smalltalk през 1970 г., с Object Pascal, Ada и C++, C# като по-съвременни примери за обектноориентирано програмиране; принципите за модулна спецификация (Парнас 1972) и работа на проектно равнище могат да се намерят по ниво проектиране в Jackson System Development (JSD) (Jackson 1983).

По-съвременните приложения на ОО принципите на проектиране и анализ на степента на развитие на системите е довело до обектноориентиран дизайн (OOD), както личи от Booch (1991) и обектноориентирания анализ, (OOA), които са представени в Coad и Yourdon (1991), Shlaer и Mellor (1988) и Rumbaugh et al. (1991). На архитектурата на базите данни е повлияла работата на Chen (1976) с връзки между обектите (ER). На транзакционният анализ е повлияла работата на Brodie и Silva (1982).

ОО методи навлизат в широка употреба в края на 80-те години на 20-ти век и обикновено са фокусирани върху отделните фази. Например, обектноориентиран анализ, обектноориентиран дизайн и обектноориентирано програмиране са фокусирани съответно върху фазите на анализа, дизайна и изпълнението.

Избираме Coad и Yourdon OOA за сравнение, тъй като той е на ниво анализ и се отнася главно за изграждане на информационен модел. Обектноориентираните методи са от особен интерес за нашето сравнение, тъй като те предлагат различна парадигма за информационно моделиране. Целта им е подобна на данновоориентираните методи, като техния фокус е върху обектите или данните, но те групират процеси по обекта, на който тези процеси работят.

Традиционните методи имат различни начини за групиране на процеси, и не интегрират процес с обект така силно.

## ИНФОРМАЦИОННО МОДЕЛИРАНЕ

Информационното моделиране изгражда информационния модел, който обикновено се състои от две основни компоненти – структура на модела и процесите на модела.

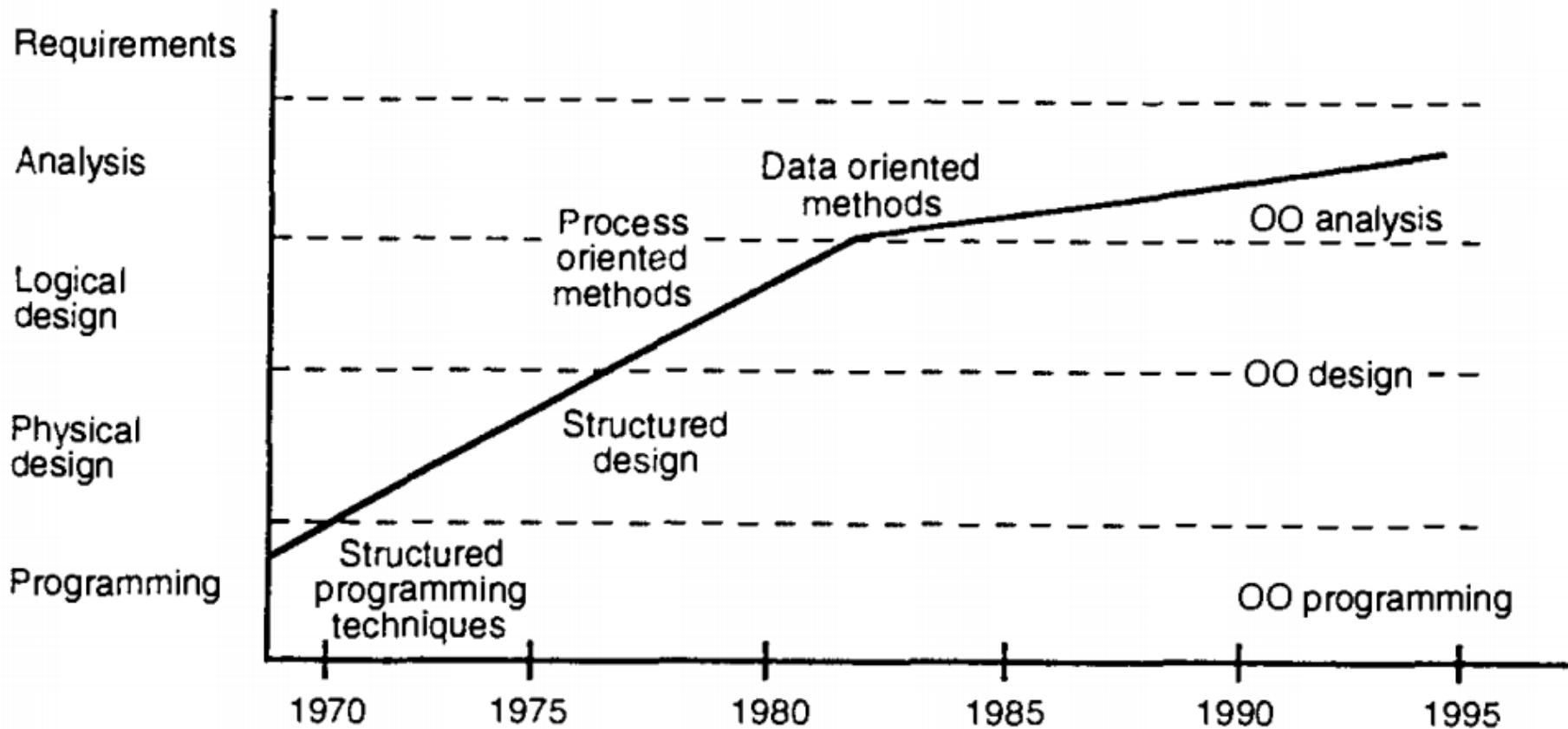
Структурата на модела описва организационните и заобикалящите организацията елементи, за които се налага да се записва информация, като често се използват концепциите за същност, атрибут и връзка, и се показват на релационнообектовата диаграма.

Процесите на модела описват елементите, свързани с обработката на информация, използвайки понятия като процес, събитие, поток данни и ги изразяват в термините на структурни моделни елементи. Например, процеси на модела са данновия поток или разделянето на процесите.

Възможно е да има трети компонент към информационния модел – „управление на модела“, което ограничава стойностите на елементите в структурата на модела.

Информационният модел е на концептуално ниво, така че данните, които засягат структурата на моделните обекти, атрибути и връзките няма да се получат преди да се премине проектната фаза на развитието на системата. На проектната фаза процесите ще бъдат проектирани да работят с тези данни, вместо с отделни елементи на структурата на модела.

# Резюме на еволюцията на методите



Дали методите отговарят на целите си?

### Методи за подобрения

Ще резюмираме начина, по който са се развили методите за постигане на високо качество и производителност.

- Фазите и дейностите, свързани с тях стават по-ясни, като заменят интуицията на разработчика за ориентиране и позволяват да се управлява по-добре процеса. Това намаля грешките на разработчика.
- Продуктите са определени по-ясно, така че изискванията на потребителите могат да се моделират правилно и точно във всички необходими подробности. Станали са по-прости, така че разработчиците да правят по-малко грешки и потребителите да могат да ги разберат и проверят.
- Ранните фази са станали по-абстрактни, улеснявайки включването на потребителите, което води до по-точното определение и улавяне на изискванията на потребителите, а също и помага за проверка на изискванията.
- Обектноориентираните методи обещават по-плавен преход между различните етапи, като през цялото време се използва обектноориентираната парадигма.

Ще разгледаме само онези проблеми, които се отнасят до по частта за информационното моделиране на методи, относно висококачествените цели на методите. Общите проблеми са, както следва:

Методите се различават по своя обхват и не могат да моделират всички аспекти на изискванията.

Съществува терминологичен проблем, тъй като методи могат да използват различна терминология за една и съща концепция и еднакви термини за различни концепции, които объркват разработчици и потребители.

Не съществува базов набор от концепции за моделиране, така че не е възможно да се осигурят добри насоки за разработчика относно дейности или продукти.

Така лесно могат да се допуснат грешки при моделирането.

Моделите се изразяват във форма, която е трудна за разбиране от потребителите, правейки валидирането неефективно и водещо до ниско качество на системата.

## По-нови методи и разглеждани проблеми

Ще разгледаме частта, която е свързана с информационното моделиране. Ще се концентрираме върху компютърно-базирания аспект на евентуалната система.

### Широк критицизъм

Много критики и предложения за подобряване на метода са свързани с идентифициране на различни видове методи (Hirschheim и Klein, 1989) или подобряване на "твърди" методи, като ги променя в "меки" методи. Примери за по-широк вид критицизъм на методи са следните:

Методи предполагат изисквания, които са фиксирали и известни в началото на процеса на развитие. Въпреки това е добре известно, че изискванията се променят в процеса на разработка, поради фактори като околната среда или изискванията на потребителите, научили повече за технологията или тяхната организация.

Ако метод се съсредоточи върху грешен проблем в организацията, това би довело до осигуряване на система, която всъщност не е необходима, или не позволява достатъчно участие на потребителя, за изискванията, като за тях няма да бъдат определени правилно и напълно и валидирани спецификации.

Методи, които наблягат на техническите аспекти на информационните системи, пренебрегват по-човешкия социален и психологически контекст, в който те трябва да бъдат интегрирани.

Методите отнемат твърде много време, като разработчиците се съсредоточават върху сравнително маловажни задачи, като например подробно попълване на формуляри.

## Насоченост на критиките

Като отговор на критиките, които посочихме по-горе, се появяват нови или подобрени методи.

## Променящите се изисквания

Еволюционния подход (Crinnion 1991), е метод, който предполага разделяне на системата на няколко части, всяка от тези части е разработена отделно, така че потребителите имат възможност да научат постепенно техните изисквания. На по-общо равнище, Boehm (1985) е предложил спирален модел на разработване на системи, в които явно се включва итерация.

## Грешен проблем / участие на потребителя

За намиране на ключови проблемни области в организацията могат да бъдат използвани фактори за критичен успех (Flin и Arce, 1995) или анализ на организационната цел (Yu, 1993 г.). Прототипиният подход (Ramesh и Luqi, 1993) предлага, като най-добро включване на потребители които да наблюдават експеримент с реалната система вместо абстрактни спецификации. Съществуват няколко метода, които включват софтуерни инструменти, които да направят прототипи на част или цялата система.

## Само технически аспекти

„Меки“-те методи са насочени в тази област, и често обхващат широк набор от въпроси. Има методи, които предизвикват и договарят често сложни бизнес изисквания (Checkland, 1981), тъй като въвеждането на компютърните технологии изисква радикална преоценка на бизнес процеси (Hammer, 1990). Други методи имат за цел да предотвратят отрицателни социални и психологически ефекти, произтичащи от въвеждането на непозната или нежелана технология (Mumford, 1983 г.; Harker и др., 1990). По-конкретно, някои методи се съредоточават върху "полезността" на евентуалната система за нейните потребители (Gould и др., 1991 г.).

## Прекалено много време

Много методи се поддържат от CASE (Computer Aided Software Engineering) инструменти (Flynn и др., 1995), които управляват големи обеми от данни и автоматизират повтарящи се задачи в процеса на разработка.

## СТАНДАРТИ

### Тенденции

По-нова характеристика на развитието на метода е, че са предприети стъпки към стандартизация. Източниците за стандартизация се различават, и могат да бъдат индустриски, международни организации или органи, като Европейската комисия. В допълнение, обхватът на стандарта може да варира от най-високо равнище преглед на процеса на развитие на системите за подробно описание на различни видове продукти.

Много е вероятно стандартизацията да се увеличава в бъдеще, тъй като тя носи предимства, като например намаляване на разходите, опростено обучение и обещава по-висококачествени системи.

## Съществуващи стандарти

Стандартът на Великобритания BS 5750 (Британски институт за стандарти, 1987) и неговия еквивалент, международния стандарт ISO 9001, е много общ стандарт, който обхваща управлението на всеки производствен процес, и много организации изискват от своите доставчици да се съобразят с този стандарт за производство на софтуер.

Схемата TickIT (TickIT 1992), първоначално спонсорирана от Министерството на търговията и индустрията във Великобритания, е процесът на сертифициране, който е по-специфичен за развитието на системите. Той оценява качеството на системата за управление на процеса на разработване на системи в една организация, като прилага стандарта ISO 9001-3, който е насочен към аспекти на управлението на софтуерното производство.

В САЩ, Capability Maturity Model (CMM), разработен под спонсорството на Министерството на от branата (Humphrey, 1989) определя пет равнища на зрялост на процеса на развитие на системи в една организация. Всяко равнище на модела се характеризира с определени свойства, които се отнасят до процедурите, използвани за управление на процеса. Така например, организацията на равнище 1 притежава *ad hoc* или хаотичен процес на развитие, без формални процедури или механизъм за оценка на разходите.

## Обзор на Euromethod

Текущото състояние на Euromethod (CSTA, 1994; Franckson, 1994) се занимава с определяне на резултатите, които да бъдат обменяни между клиента и доставчика по време на всички аспекти на развитие на системите, от офертата за възлагане на обществена поръчка до операционната система (Jenkins, 1992). Въпреки това, акцентът е по-скоро в началните, отколкото по-късните етапи на този процес.

Седем метода са избрани като основа за развитието на Euromethod: SSADM (Великобритания), MERISE (Франция), SDM (Холандия), DAFNE (Италия), Vorgehensmodell (Германия), MEIN (Испания) и Информационния инженеринг. Euromethod версия 0 е доставена през юни 1994 г. (ЕМ, 1994 г.), и след пробния период, версия 1 е от 1996 година. Първоначално тя се използва за рекламирани обществени проекти за възлагане на обществени поръчки на Европейския съюз.

## Отчетните резултати

Резултати се определят по отношение на общите понятия на модела на отчетните резултати, и се състои от три основни вида: целева област, план за доставка и резултати от проектната област.

Резултатите от целевата област са свързани с развитието на информационната система и се състоят от описание на информационната система и оперативните елементи. Планът за доставка описва отношенията клиент-доставчик по време на процеса на разработка, дефиниращи началната, крайната и междинните стъпки на организацията и информационната система и последователността на точките, в които се взима решение. Проектната област разчита на процеса на разработка и се състои от проектните планове и отчетите по проекта.

## IS-описания

IS-описание очертава елементите на информационна система и е еквивалент по смисъла на нашите означения на продукта. Тя може да се състои от един или повече от шест гледни точки, които в термините на IS-свойствата, описват различни, макар и припокриващи се множества от системни елементи. Заедно, гледните точки дефинират концепциите и терминологията описваща информационната система.

Три гледни точки са свързани с информационната система, а други три са свързани с компютърната система. Вижданията за информационната система определят представянето на информационния ресурс, използваните и създадените актори и техните процеси с връзките между тях. Те са изгледи на: бизнес информация, бизнес процес и работна практика. Изгледите на компютърната система дефинират представянето на данните, структурата и функционалността на компютърната система. Те са погледа върху данните на компютърната система, нейните функции и архитектура.

Гледните точки за информационната система са на равнището на фазата на анализа, както вече описахме те дават представа за равнището на детайлност на бизнес процесите, което състои от следните IS-описания: бизнес процес, стартиращи събития, генериирани събития, декомпозиция на процес, динамични зависимости, бизнес правила и използване на информация.

## Хармонизация

Euromethod вижда две стъпки по пътя към хармонизиране на метода. Първо, той описва продуктите и концепциите на различни европейски методи по отношение на понятия и свойства на получавания модел, и на второ място, той предвижда ситуация, при която методи могат да доближават тяхната терминология към Euromethod.

## ISO / IEC стандарт 12207

Този проект на стандарт (Singh, 1994) може да се разглежда като допълнение към Euromethod, тъй като не е документиран стандарт, но дефинира ключови процеси в развитието на системите. Това се прави така, че архитектурата е отворена и може да се използва с всеки модел на разработваната система.

Има три основни вида процеси: първични, поддържащи и организационни. Те покриват основните дейности по роли, като доставчик, получател, оператор, отговорник и ръководител на проекта. Архитектурата набляга на принципите на totally управление на качеството и подчертава по-късните дейности в процеса на развитие, като дванадесет от четиринацесетте дейности са свързани с проектирането на системата или софтуер.

## Бърза разработка на приложения - DSDM

Последният пример, който ще опишем ("радикална стъпка", Computing, 2 март 1995 г.) се нарича Метод за динамично развитие на системата (DSDM), и е стандарт за бързо разработване на приложения (RAD). Това е насочено към критиките, обсъдени по-горе, при която развитието на системите често е твърде бавен. DSDM осигурява на високо равнище рамка за разработване на системи, основана на тринадесет ключови принципи. Два от тези принципа са, че включване на потребителите в RAD е наложително, както и итеративно разработване и тестване, което трябва да бъде интегрирано през целия жизнен цикъл.

# ЛЕКЦИЯ 2. Структурен модел

Ще разгледаме понятия и схеми, предоставени от четирите метода за структурния модел: информационно инженерство, SSADM, MEIN и MERISE.

Ще представим и ще дефинираме всяко понятие като цяло, както и ще обсъдим съответната концепция (ако има такава) за всеки от методите. За да може нашата концепция да бъде сбита, ще предположим, че контекстът е свързан с изграждане на информационен модел на част от организация.

## Референтна рамка

Концепциите на референтната рамка за структурния модел са: обект, атриут, идентификатор, атриутни ограничения на отношенията, двоични отношения, главни ограничения, ограничение за участниците, атрибути на връзката, *n*-арни връзки, обобщения, наследяване на свойство, множествено наследяване, изключение, изтегляне, натрупване и правило.

## Структура на моделните компоненти

### Диаграми

Диаграмите, които съдържат структурните модели на методите са:

Информационен инженеринг

SSADM

MEIN

MERISE

Диаграма на обектната връзка

Логическа структура на данните (LDS)

Диаграма на обектната връзка

Концептуален модел на данните (CDM)

## Текст

В допълнение, методите предполагат, че структурата на модела, трябва да съдържа описание на всички атрибути в текстова форма, която е отделена от диаграмата, за да се избегне сложната диаграма. Въпреки това, те не описват това достатъчно подробно, така че използваме обща форма, която наричаме списък обект-атрибут, за всички методи.

## Обект и атрибут

Тук ще определим концепциите на референтната рамка относно обекти и атрибути и ще анализираме съответните концепции в методите. Концепциите на референтната рамка, които се използват са обект, атрибут, идентификатор и ограничения за атрибут на връзка.

## ENTITY

### Дефиниция

*Entity* представлява обект от интерес за дадена организация, който може да бъде конкретен или абстрактен, и който е по-скоро статичен, а не динамичен.

Нека разгледаме случая като пример фирмата за водопроводни части Аквадукт. Например, във фирмата Аквадукт, клиентите имат банкови сметки и доставчиците части за доставки. Тук, клиентът, доставчикът, потребителският акаунт и част са обекти от интерес за организация, където клиента и доставчика са конкретни обекти и банковата сметка на клиент е абстрактен обект.

За по-точна дефиниция ние използваме термините „entity“ и „entity instance“:

Типът „entity“ представлява набор или клас от „entity“-та в една организация, които поделят едни и същи характеристики.

„Entity“ инстанцията представлява определено „entity“ в една организация, която е член на „entity“ множество или клас.

За да се избегне тромавата терминология, ще следваме обичайната практика и да използват термина „entity“ за да се позовем на неговия тип.

Това определение на „entity“ е доста неясно и наистина не помага, когато се опитваме да идентифицираме „entity“ в една организация като част от изграждането на информационен модел.

CUSTOMER  
queue the other side

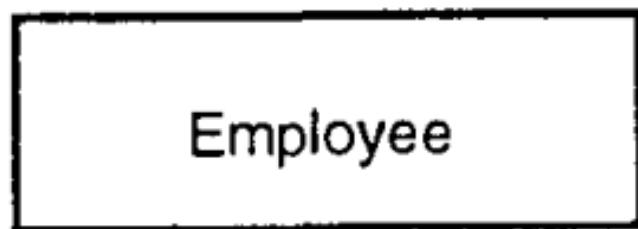


Инстанции на „entity“ поделят  
едни и същи характеристики

IE

За IE, „entities“ са хора, предмети или понятия, които са от значение за фирмата и за които искаме да съхраняваме информация. Те могат да бъдат материални или нематериални" (Martin, 1990).

Графичното представяне на „entity“ в IE е правоъгълна кутия съдържаща неговото име, написано вътре.



фиг. 2.1. „Entity“ служител (IE).

IE описва и пресичащи се „entity“, които не се различават концептуално от нормалното „entity“.

## MEIN

За MEIN, „entity“ е „реален или абстрактен обект, който съществува и е различим; той може да бъде човек, или фирма или различим обект“ (MEIN II, 1991). Също така „entity“ е нещо, за което искаме да съхраним информация.

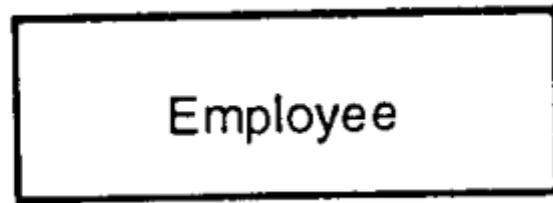
MEIN класифицира „entity“ като обикновено, слабо или асоциативно. Обикновеното „entity“ е такова че инстанциите му могат да се идентифицират чрез самите себе си. Това е, защото то има идентификатор на самия себе си.

Графичното представяне на обикновено „entity“ е правоъгълна кутия с неговото име, написано вътре, както е показано на фигура 2.2.

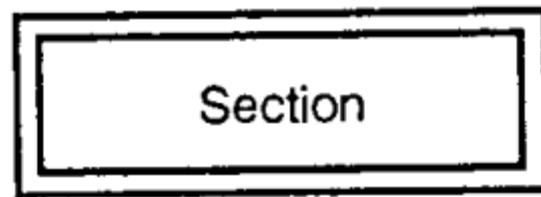
Едно слабо „entity“ е това чито срещания са идентифицирани от вече асоциирани едно или повече „entities“. Идентификаторът на слабо „entity“ обикновено се образува от собствен идентификатор, както и идентификатор на свързани „entity“. Например, една секция на фирмата Аквадукт изисква името на департамента за идентификация, както и собственото си име, за да се постигне уникалност.

Графичното представяне на слабо „entity“ е правоъгълна кутия с двойна линия и името на „entity“, написано вътре, както е на фигура 2.3.

MEIN също така дефинира асоциативно „entity“, което ще бъде обсъдено по-нататък.



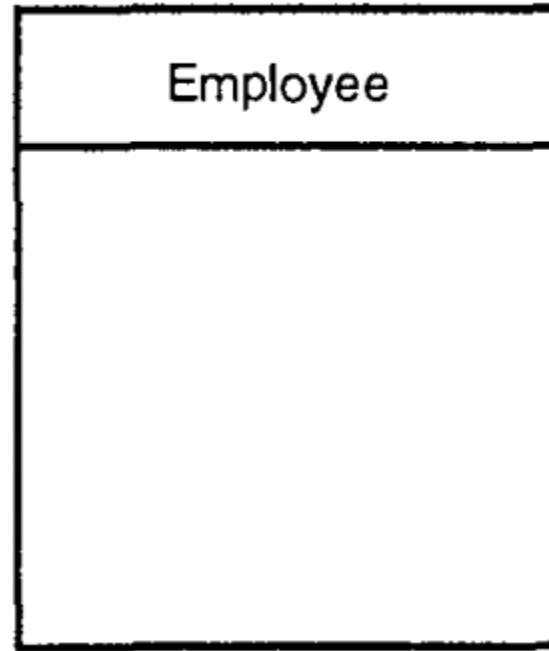
фиг. 2.2. Нормално „entity“  
служител (MEIN).



фиг. 2.3. Секция на слабо „entity“.

## MERISE

За MERISE "„entity“ е понятие, което е полезно за нуждите на управлението на фирмата" (Quang и Chartier-Kastler, 1991). Алтернативно, „„entity“ е абстрактен или конкретен обект във вселената на дискурса“ (Tardieu и др., 1983).

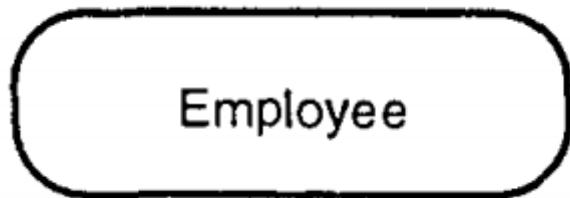


фиг. 2.4. „Entity“ служител  
(MERISE).

## SSADM

За SSADM, " „entity“ е обект или концепция, реален или абстрактен, който е от значение за изследваната област на бизнеса" (SSADM, 1990).

„Entity“ е графично представено от кутия с име, написано отвътре, както е на фигура 2.5.



фиг. 2.5. „Entity“ служител (SSADM).

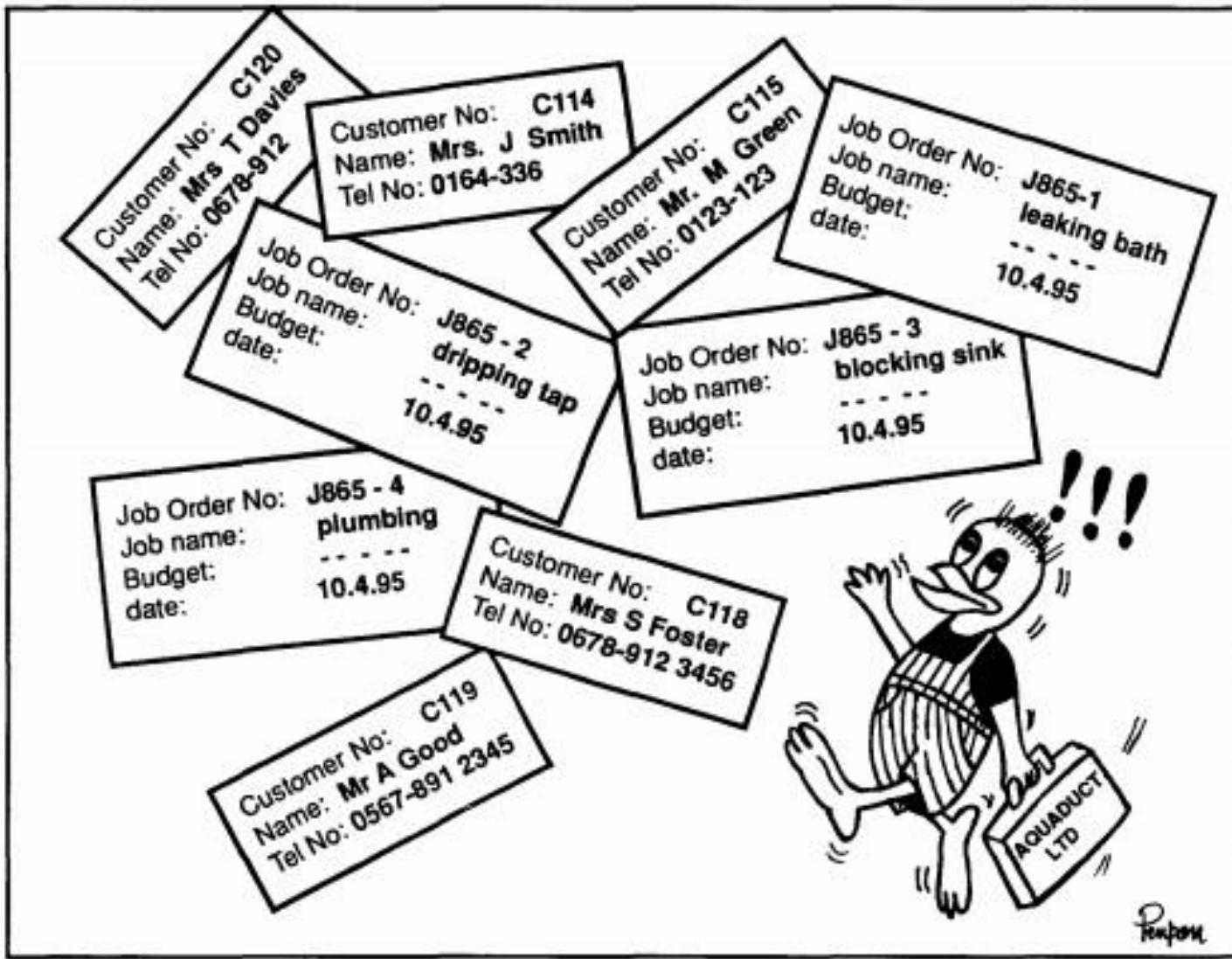
## АТРИБУТ

Определения

Атрибут

Атрибутната стойност представлява дескриптор или „entity“ инстанция например, като възраст „45“ или името „Смит“ на дадено лице. Типът на атрибута представлява множество или клас на „entity“ дескриптори, които описват едно и също тип „entity“.

Примери на често срещаните видове атрибути са име, възраст, адрес и номер на социална осигуровка, които могат да са типични дескриптори на „entity“ тип човек.



Атрибутите описват „entity“  
по-детайлно

Що се отнася до „entity“ и типа „entity“, ще следваме нормалната практика и ще използваме термина атрибут, за да се позоваваме на типа атрибут.

## **Идентификатор**

Идентификаторът в простата си форма, атрибут, всяка стойност, на който еднозначно идентифицира една инстанция на „entity“. Така например, всяка стойност на атрибута „номер на социалната осигуровка“ ще идентифицира еднозначно инстанция на „entity“ личност.

Идентификаторът е важен, тъй като неговите стойности са използвани от потребителите за връзка и оттук за получаване на актуализирани „entity“ инстанции. От разработчиците във фазата на дизайна на системната разработка, да вземе решение за представяните данни от „entity“.

Идентификаторът може да бъде прост или съставен. Ако тя е прост, той обикновено е един атрибут. Съставният идентификатор може да бъде съставен от два или повече атрибути или от смесица от атрибути и „entities“.

## **Ограничения на атрибутни отношения**

Те ограничават допустимите комбинации на „entity“ и атрибут, като съществуват два основни типа ограничение: важност и опционалност на атрибута на връзката.

## **Главни атрибутни отношения**

Те определят броя на стойностите на атрибут, който може да бъде свързан с една инстанция на свързаното „entity“. Те могат също да дефинират броя на инстанциите на „entity“, които могат да бъдат асоциирани с една стойност на свързания атрибут. Има четири общи видове релации, едно: едно, едно: много, много: един и много: много, където смисълът на много е една, или повече от една, стойност или инстанция.

## **Опционални атрибутни отношения**

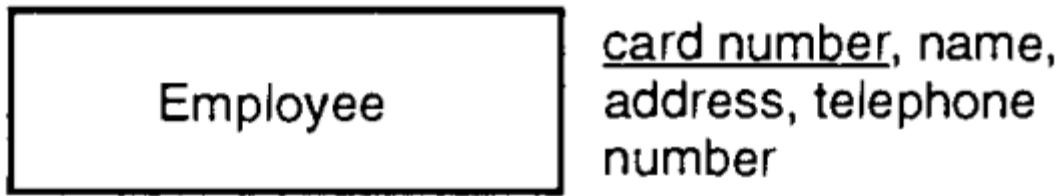
Това дефинира дали или не всички инстанции на „entity“ трябва да са асоциирани със стойности на атрибут, и дали или не всички стойности на атрибута трябва да бъдат асоциирани със инстанции на „entity“. Ако те трябва да бъдат асоциирани, тогава се наричат **задължителни**, но ако те могат да са асоциирани, това се нарича **опционални**.

Ще отбележим, че главните атрибутни отношения, както и опционалните определят специфични аспекти на атрибутните връзки всеки момент на съществуването си в организацията. Тези аспекти могат да се различават, когато връзката се разглежда в продължителен период от време.

Например, въпреки че служителите могат да имат само един адрес във всеки един момент, един служител може да се мести на много различни адреси за определен период от време.

IE

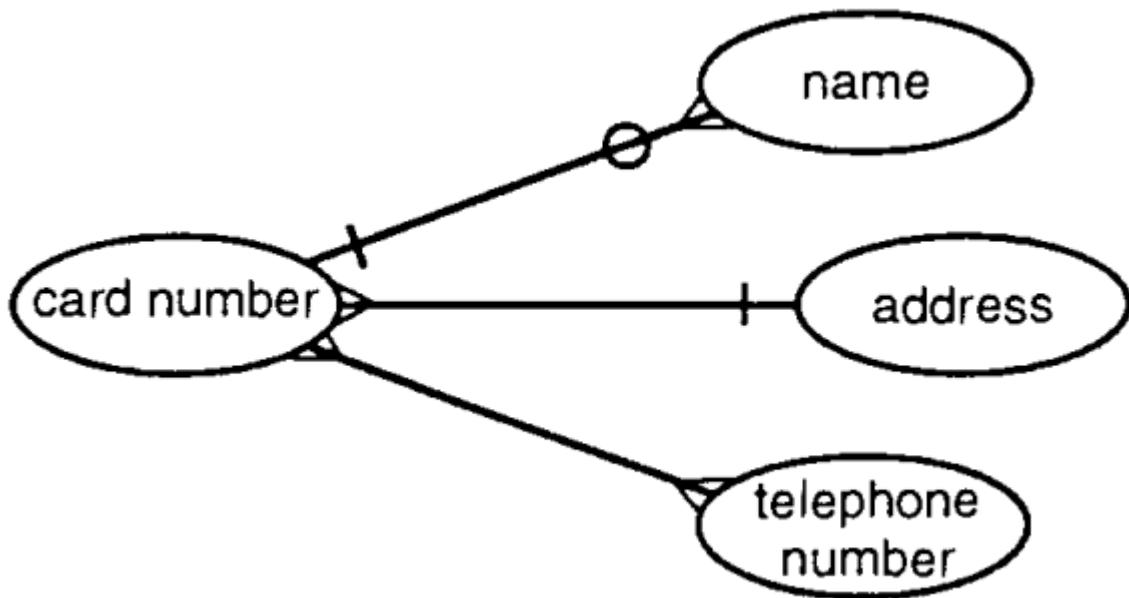
В IE един атрибут се нарича даннов елемент.  
"Елемент от данни е този, който съдържа  
само една част от информацията (стойност на  
атрибут) за „entity“ инстанция и не може да  
бъде разделен на части, които имат свое  
собствено значение".



фиг. 2.6. „Entity“ служител с атрибути показващи атрибута номер на картата като идентификатор (IE).

Кардиналността – символът „гарванов крак“ представлява много, докато късата линия представлява едно. Може да има много стойности, от името на атрибута, но само една стойност на атрибута адресира асоцииране с „entity“.

Опционалните – малкия кръг представлява възможността за връзка, докато липсата му представлява задължителна връзка. Името на атрибута не е задължително, но името на адреса на атрибута е задължително.



фиг. 2.7. Бабъл диаграмата показва атрибути на „entity“ служител с атрибутни ограничения на връзките. Атрибутът „номер на картата“ е идентификатор, представляващ „entity“ служител (IE).

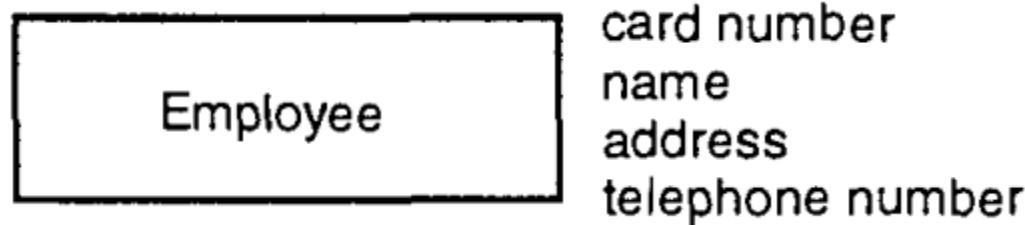
За IE, идентификаторите се наричат *първични ключове*, докато *неключови атрибути* са тези, които не еднозначно да идентифицират „entity“, но са обикновени атрибути на това „entity“.

Първични ключове и някои неключови атрибути могат да бъдат представени чрез графичен модел, със записването им до тяхното свързано „entity“.

Идентификаторът е пръв в списъка и е подчертан (Martin и McClure, 1985). Един пример за графично представяне на атрибути е показан на фиг. 2.6.

IE позволява всички четири вида кардиналност, така че една „entity“ инстанция може да има една или много стойности на атрибутите, както и обратното. В допълнение, атрибута за взаимовръзка може да бъде специфициран като задължителен или опционален. Това, обаче, може да бъде дефинирано само за *едно:едно* или *едно:много* кардинални типове, специфицирани от посоката на „entity“.

Тези ограничения не са показани на диаграмата, но са посочени отделно. IE може да направи това графично чрез диаграма наричана бабъл диаграма, и например за „entity“ служител и неговите атрибути са показани на фиг. 2.7, където атрибутите са показани в балони, а отношенията между идентификаторите за номера на картата, представляващ „entity“ служител, и другите атрибути са представени графично като ребра свързващи балоните.



фиг. „Entity“ служител с атрибути  
(MEIN).

## MEIN

Атрибута в MEIN се дефинира като "основна и неделима единица на информацията, използвана за да се направи разграничаване на „entity“ (MEIN II, 1991)". Когато атрибут е също и идентификатор, той се нарича първичен ключ.

Атрибутът на „entity“ може да приеме стойност от набор от възможните стойности наречена *домейн*. В допълнение, атрибута може да бъде прост или съставен. Съставният атрибут е съставен от няколко атрибути, например адресния атрибут може да се съдържа атрибут къща или апартамент, улица и пощенски код.

Атрибутите са документирани отделно само с най-важното показано до „entity“, на което те принадлежат. Фиг. 2.8 показва пример на атрибути на „entity“.

MEIN не предоставя начин за разграничаване на идентификатор от други атрибути. В допълнение, няма разлика между графичното представяне на съставен и прост атрибут. MEIN не определя важността на връзките между атрибутите или опционалността.

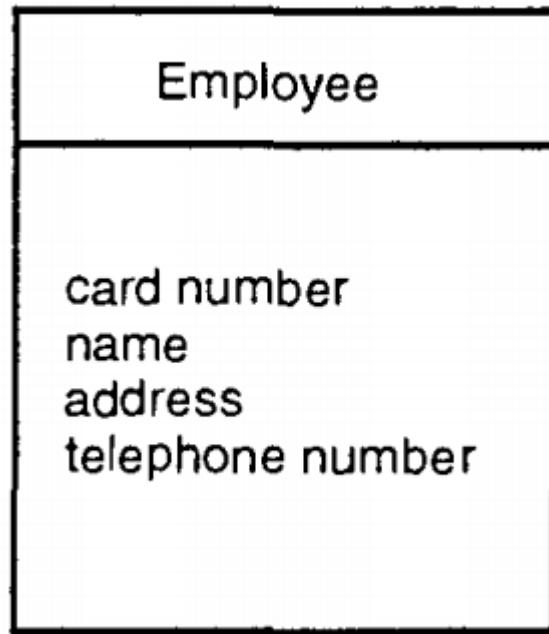
## MERISE

В MERISE, атрибута (също наричан свойство) „е елементарен даннов елемент, който характеризира „entity”“ (Quang и Chartier-Kastler 1991). Подобна дефиниция е „атрибут е представяне на свойство на „entity“ или свойство на асоциация между „entities““ (Tardieu и др., 1983).

Само най-важните свойства са показани на диаграма. Техните имена могат да бъдат написани в празната част на прозореца, който представлява „entity“, както е на фиг. 2.9.

На Фигура 2.9, идентификаторът може да бъде първи в списъка на свойствата, или можем да посочим кои от свойствата са идентифицирани чрез техните имена в различен стил (Quang и Chartier-Kastler, 1991) или чрез подчертаване на важно свойство (Planche 1992 ).

Що се отнася до MEIN, не са дефинирани кардиналните отношения на атрибут и опционалните.



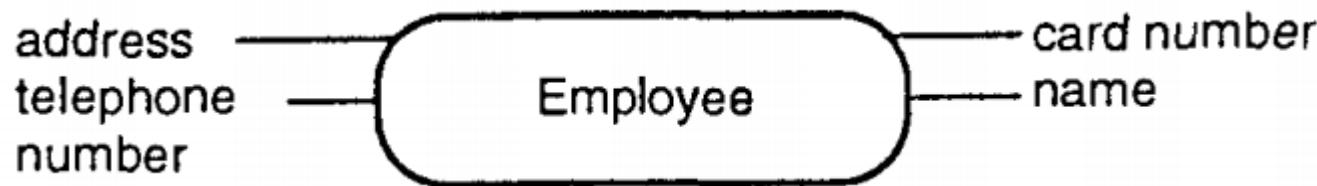
фиг. 2.9. „Entity“ служител с атрибути, показващи атрибута номер на картата като идентификатор (MERISE).

## SSADM

За SSADM, "атрибут е характеристика на „entity“, което е всеки детайл, който служи да се опише, идентифицира, класифицира, в количествено отношение или да изрази състоянието на „entity“" (SSADM, 1990).

За SSADM, атрибута описва само едно „entity“. Идентификаторите не се различават графично от неидентифицирани атрибути. Когато „entity“ има само няколко атрибута, те могат да бъдат показани чрез графичен модел чрез записването им до „entity“, на което те принадлежат. Пример е показан на фиг. 2.10.

SSADM позволява само едно:едно и много:едно (от посоката на „entity“) кардинални типове, ограничавайки „entity“ инстанция да притежава само една стойност на атрибут. И задължителната и optionalната спецификация е на разположение за optionalността на атрибутните отношения, но е само за „entity“.



фиг. „Entity“ служител с атрибути  
(SSADM).

## СПИСЪК „ENTITY“-АТРИБУТ

Таблица 2.1. показва списък „entity“-атрибут, който описва атрибути, свързани с тях „entities“ и съответните ограничения. Няма стандарт за това описание, така че използваме формата на Таблица 2.1. за всички методи. Подробно са описани три „entities“ и техните атрибути, които илюстрират всичките пунктове относно атрибутите.

*Идентификатор.* Идентификаторът е показан в таблица 2.1., чрез подчертаване на съответния атрибут или атрибути. В частта „entity“ има съставен идентификатор, показан от „/“ отделяща съответните атрибути, като и „име“ и „име на раздел“ са необходими атрибути, за да се идентифицира раздел. Това е защото имената на секциите не са уникални в рамките на различни части.

*Съставен атрибут.* Адресът е показан като съставен, и се състои от една къща или апартамент, улица и пощенски код.

*Ограничения на връзките на атрибута.* Те са показани по посока на „entity“-атрибут.

**Table 2.1** Entity–attribute list

Entity	Attribute	Attribute relationship constraints
Department	<u>department name</u>	one:one; mandatory
Employee	<u>card number</u>	one:one; mandatory
	name	one:many; optional
	address (house/apt, street, postcode)	many:one; optional
	telephone number	many:many; optional
Section	<u>department name/section name</u>	one:one; mandatory

## РЕЗЮМЕ

### „Entity“ и атрибут

„Entity“ и атрибут се дефинират от всички методи, използващи подобни термини и смисълът е същият. Таблица 2.2. обобщава метод съответен за концепцията на „entity“ и атрибута. MEIN е единственият метод дефиниращ слабо „entity“.

Методите са в съответствие с техните дефиниции за „понятието“ идентификатор. Таблица 2.3. показва методи, които имат съставни атрибути и идентификатори. Когато е записана „\*“, съответния тип на атрибута е показан в графичния модел.

**Table 2.2** Entity and attribute

<b>Concept</b>	<b>Method</b>			
	<b>IE</b>	<b>MERISE</b>	<b>MEIN</b>	<b>SSADM</b>
<b>Entity</b>	yes	yes	yes	yes
<b>Attribute</b>	yes	yes	yes	yes

**Table 2.3** Identifier and composite attribute

<b>Methods</b>	<b>Attributes</b>	
	<b>Identifier</b>	<b>Composite</b>
<b>IE</b>	yes*	-
<b>MEIN</b>	yes	yes
<b>MERISE</b>	yes*	-
<b>SSADM</b>	yes	-

## Ограничения на атриутните отношения

### Различия

Налице са съществени различия относно концепцията на ограниченията на атриутните отношения. Таблица 2.4. показва тези, които са разрешени от методите.

**Кардинални атриутни отношения.** IE позволява всичките четири типа, докато SSADM позволява само два типа.

**Опционални атриутни отношения.** IE позволява това, но само по посоката на „entity“ и когато крациналното атриутно отношение е едно:едно или едно: много. SSADM позволява това за всяка кардиналност, но само по посоката на „entity“.

**MEIN и MERISE.** Те не дефинират кардиналност или опционалност на атриутната връзка.

**Table 2.4** Attribute relationship cardinality and optionality

	Attribute relationship	
Method	Cardinality	Optionality
IE	one:one, one:many, many:one, many:many	mandatory/optional (in one:one and one:many only)
MEIN	—	—
MERISE	—	—
SSADM	one:one, many:one	mandatory/optional

**Table 2.5** Graphical representation of entity and attributes (including identifiers)

Method	Entity with attributes	
IE	Employee	<i>card number</i> , name address telephone number
MEIN	Employee	card number name address telephone number
MERISE		<div data-bbox="1480 558 1768 649" style="border: 1px solid black; padding: 5px;">Section</div> <i>weak entity</i>
SSADM	<p>The diagram shows an oval-shaped entity box labeled "Employee". Four lines extend from the entity to four rectangular boxes, each containing an attribute name: "address", "telephone", "number", and "name".</p>	<i>card number</i> <i>name</i>

## ЗАКЛЮЧЕНИЯ

Референтни рамкови концепции

„Entity“, атрибут и идентификатор. Всички методи ги поддържат.

Ограничения на атрибутните връзки. Няма метод, който да ги поддържа напълно.

Обхват на методите

IE и SSADM имат по-широк обхват, отколкото MEIN и MERISE, тъй като те осигуряват ограничения на атрибутните връзки (макар и с ограничения).

IE и SSADM по този начин могат да специфицират повече подробности относно „entity“-атрибутните връзки, отколкото MEIN и MERISE.

# ЛЕКЦИЯ 3. Взаимоотношения

В тази лекция ще обсъдим концепциите на референтната рамка на бинарна релация, релационни атрибути, кардинални ограничение и ограничение за участие.

Ще представим накратко концепцията на  $n$ -арна релация, но дискусията ще се фокусира най-вече върху двоични релации.

## РЕЛАЦИЯ

### Определения

#### Релация

Релацията представлява връзка между „entity“ типове в една организация. „Entity“ типовете могат да са различни или едни и същи.

Броят на „entity“ типовете, които участват в една релация определя нейната степен. Бинарна релация имаме когато броя на участващите „entity“ типове е равен на две, а  $n$ -арна релация е налице когато броя на участващите „entity“ е равен на  $n$ .

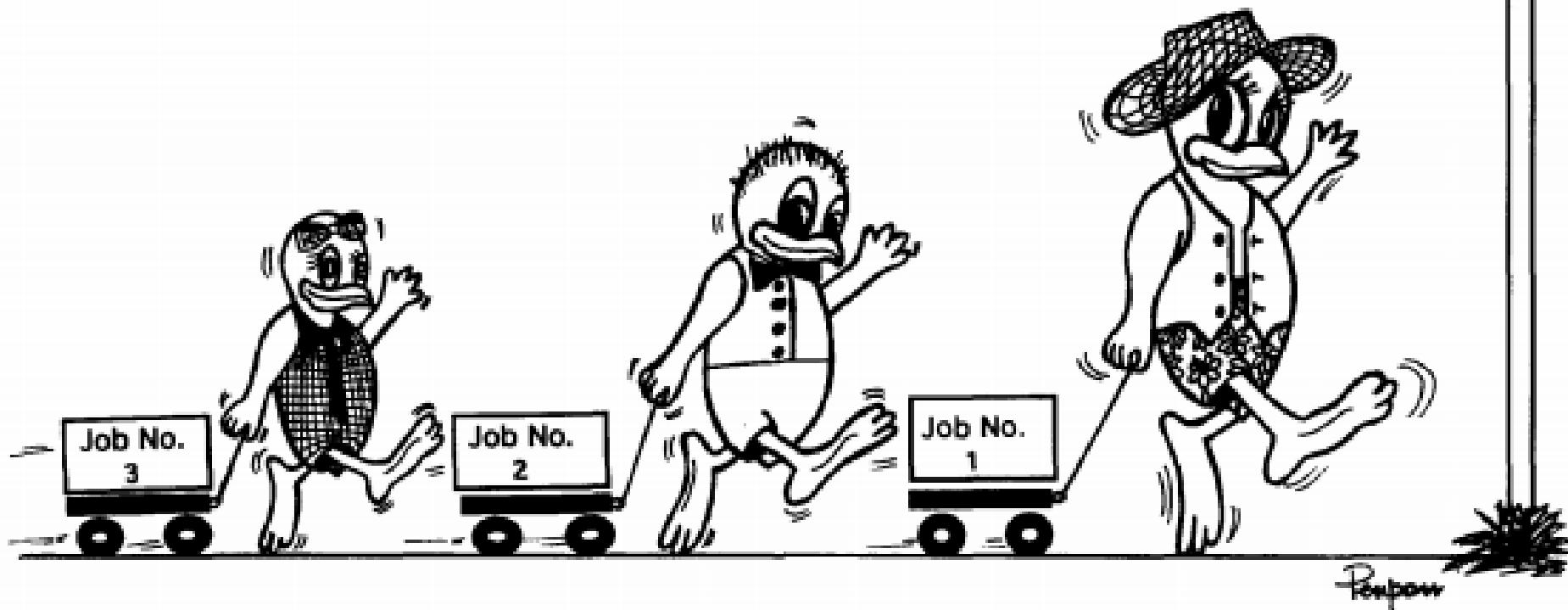
Една релация има име, което трябва да е уникално, тъй като повече от една релация може да съществува между едни и същи видове „entity“. По-точно понятието релация се дефинира по следния начин:

*Типът релация представлява смислена връзка между „entity“ типове в една организация.*

*Релационната инстанция представлява смислена връзка между инстанции на „entity“ типове в една организация.*

За да се опрости терминологията, използваме термина релация, за да се позовем на типа релация.

**CUSTOMER**  
queue the other side



Instances of an entity may have relationships with instances of another entity.

## Релационен атрибут

*Релационния атрибут* е атрибут, който е свързан с релация. Например, ако даден служител има релация "работи за" организация, данните за тази релация могат да започнат да се записват. Това се моделира от релационния атрибут.

## IE

В IE една релация е известна като „асоциация“. За IE, асоциация е "смислена връзка между два обекта" (Martin, 1990). Също така, терминът асоциация се използва за да покаже, че съществува връзка между две различни „entities“ или „entities“ от същия тип" (Finkelstein, 1989). В IE са разрешени само бинарни асоциации.

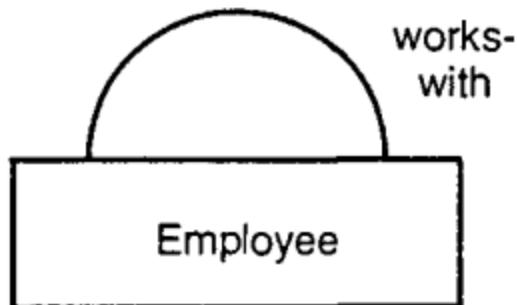
IE използва терминът "етикет" (Martin, 1990) вместо името на връзка, и използва два различни етикета, за да изрази всяка от посоките на релацията, гледана от всяка от двете „entities“.

Графично представяне на една релация е линия, свързваща двете свързани „entities“. Фиг. 3.1. показва пример за релация свързваща двойки от „entities“ инстанции от един и същ тип, които се наричат рекурсивни. Тук е необходим само един етикет.

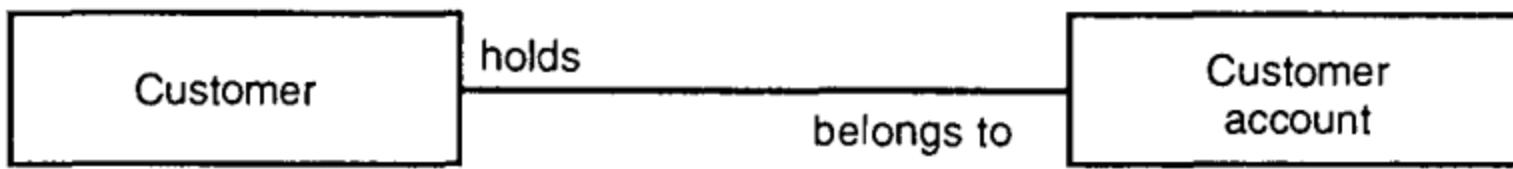
Фиг. 3.2. представлява двоична релация, свързваща две различни „entities“, с два различни етикета, за да изразят всяка посока на релацията. Етикетите са написани до „entities“, за които се прилагат.

## MEIN

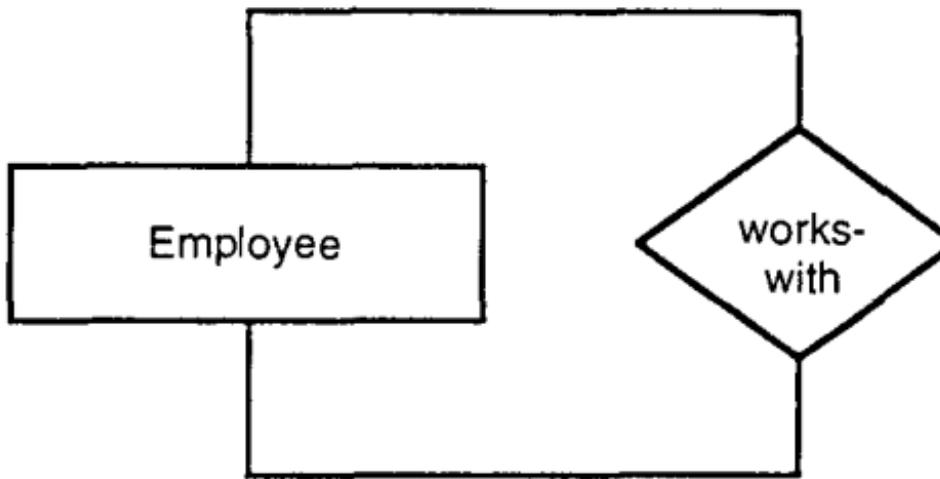
За MEIN релацията, известна също като асоциация, е "кореспонденция между две или повече „entities“, които могат да бъдат различни, или от един и същи тип"(MEIN II, 1991). В MEIN съществуват N-арни релации.



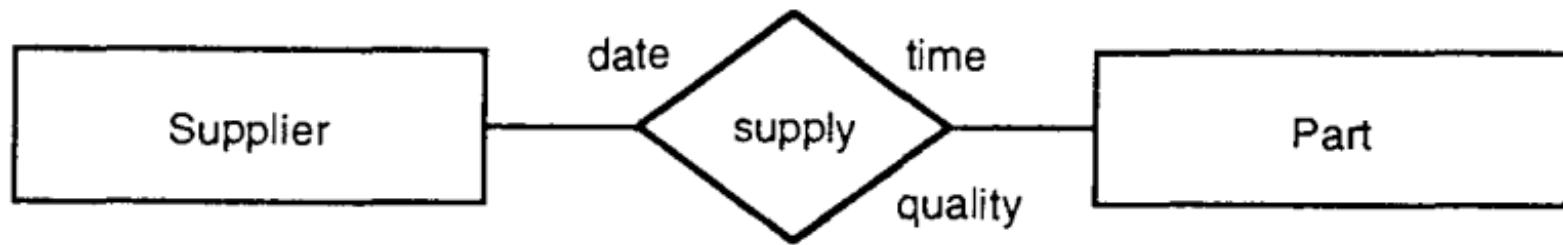
**Figure 3.1** Recursive relationship works-with. Employee works-with employee (IE).



**Figure 3.2** Binary relationship between customer and customer account. The two different labels 'holds' and 'belongs to' are used to specify each direction of the relationship (IE).



**Figure 3.3** Recursive relationship works-with. Employee works-with employee (MEIN).



**Figure 3.4** Binary relationship supply between supplier and part. Supply has attributes date, time and quality (MEIN).

Друга характеристика на MEIN е, че една релация може да има атрибути. MEIN използва само едно име за една релация. Графичното представяне на една релация е във формата на кутия с диамантена форма, съдържаща името си, свързано със съответните „entities“ чрез линии. Пример на рекурсивна връзка може да се види на фиг. 3.3.

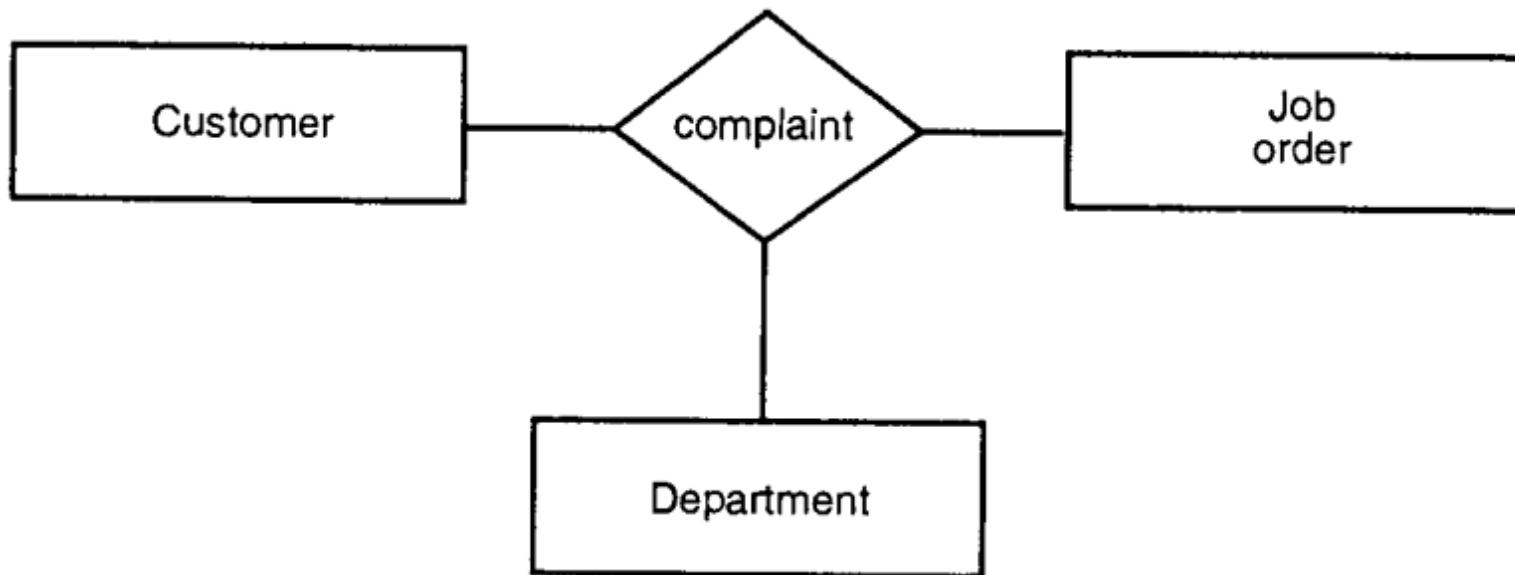
Бинарна релация между две различни „entities“ е показана на фигура 3.4, което също показва атрибути на релацията, които са написани чрез символ на релацията.

Фигура 3.5 показва пример на  $N$ -арна релация, когато различни „entities“ участват в тернарна релация. Смисълът на тази релация е, че клиентът се оплаква от работата в отдела. Например, жалбата може да е поради лоша изработка или дефектни части.

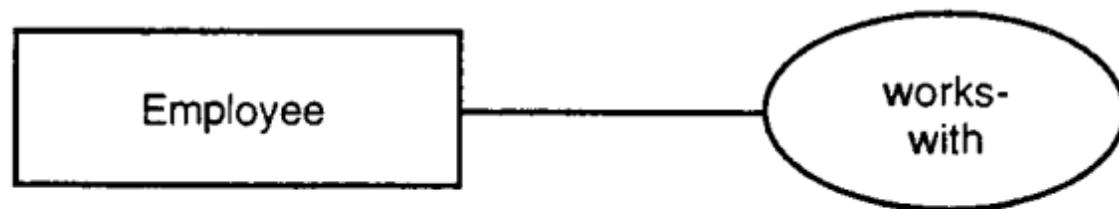
## MERISE

За MERISE, една релация е "семантична връзка между няколко „entities“ независимо от вида на обработката" (Quang и Chartier-Kastler, 1991) или "една релация се възприема като асоциация между „entities“" (Rochfeld, 1987).

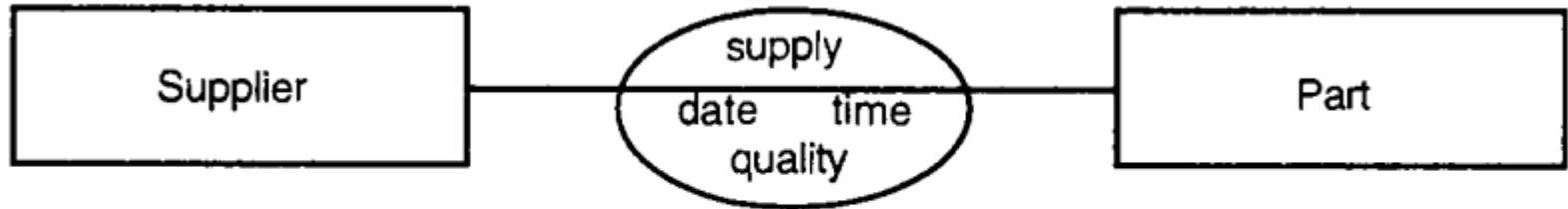
За MERISE могат да бъдат определени бинарни, а също и  $N$ -арни релации. MERISE използва терминът измерение, а не степен що се отнася до броя на „entities“, участващи в една релация (Tardieu и др., 1983).



**Figure 3.5** N-ary relationship complaint between the participating entities department, job order and customer (MEIN).



**Figure 3.6** Reflexive relationship works-with. Employee works-with employee (MERISE).



**Figure 3.7** Binary relationship supply between supplier and part. Supply has attributes date, time and quality (MERISE).

В MERISE релацията използва само едно име и то може да има атрибути. Графично представяне на една релация е елипса, в която е написано името, свързани чрез линии със съответните „entities“.

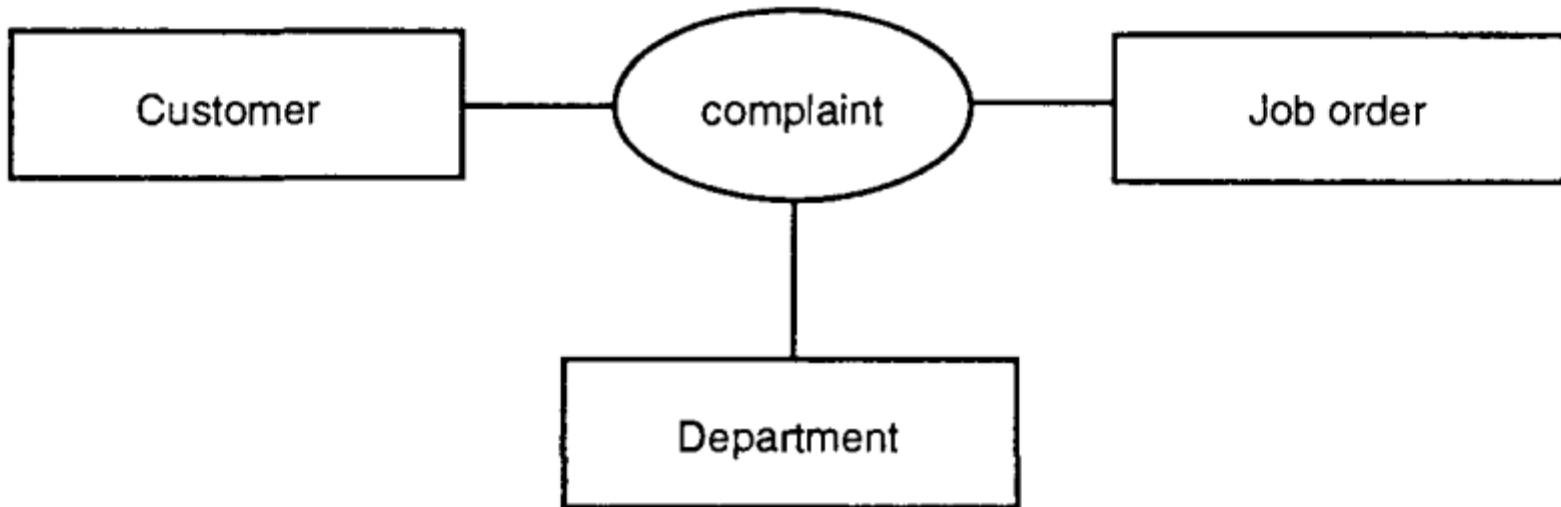
Фигури 3.6 и 3.7 представляват различни случаи на двукомпонентни релации. На първата са свързани инстанции на „entity“ от един и същи тип чрез рекурсивна релация (наричана *рефлексивна* в MERISE). Във втората, релация свързва две различни „entities“ и притежава три атрибута.

На фигура 3.8 са показани MERISE модели на *N*-арна релация за MEIN.

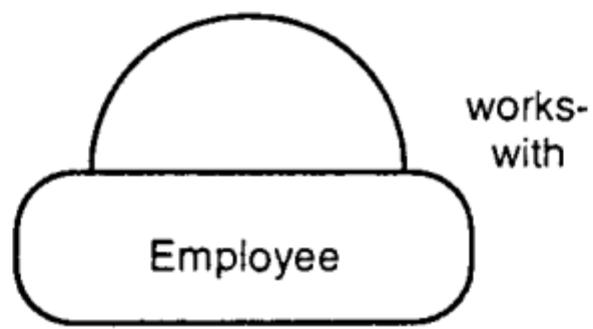
## SSADM

В SSADM, се допускат само бинарни релации и се използва термина *фраза на релационния линк* вместо името на релацията. Графично представяне на релацията е линия, свързваща двете свързани „entities“, и имената на фразите на релационния линк са написани на съответните „entities“. SSADM не позволява релационни атрибути.

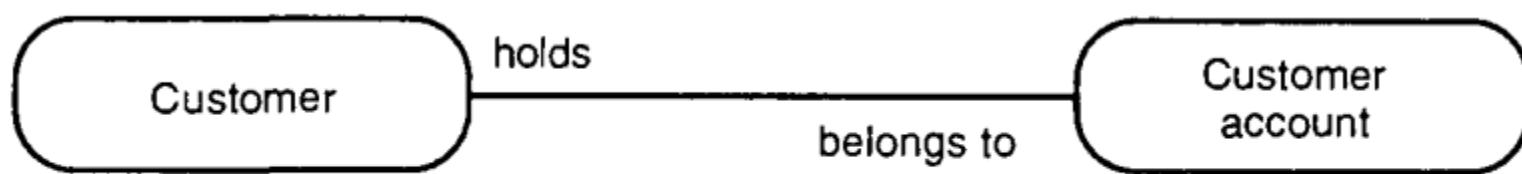
Рекурсивната релация, известна като еволвента, е показана на фигура 3.9, и се нарича „ухо на прасе“, заради формата си. Фигура 3.10 показва пример, когато две различни „entities“ участват в една релация.



**Figure 3.8** N-ary relationship complaint between the participating entities department, job order and customer (MERISE).



**Figure 3.9** Recursive relationship works-with. Employee works-with employee (SSADM).



**Figure 3.10** Binary relationship between entities customer and customer account (SSADM)

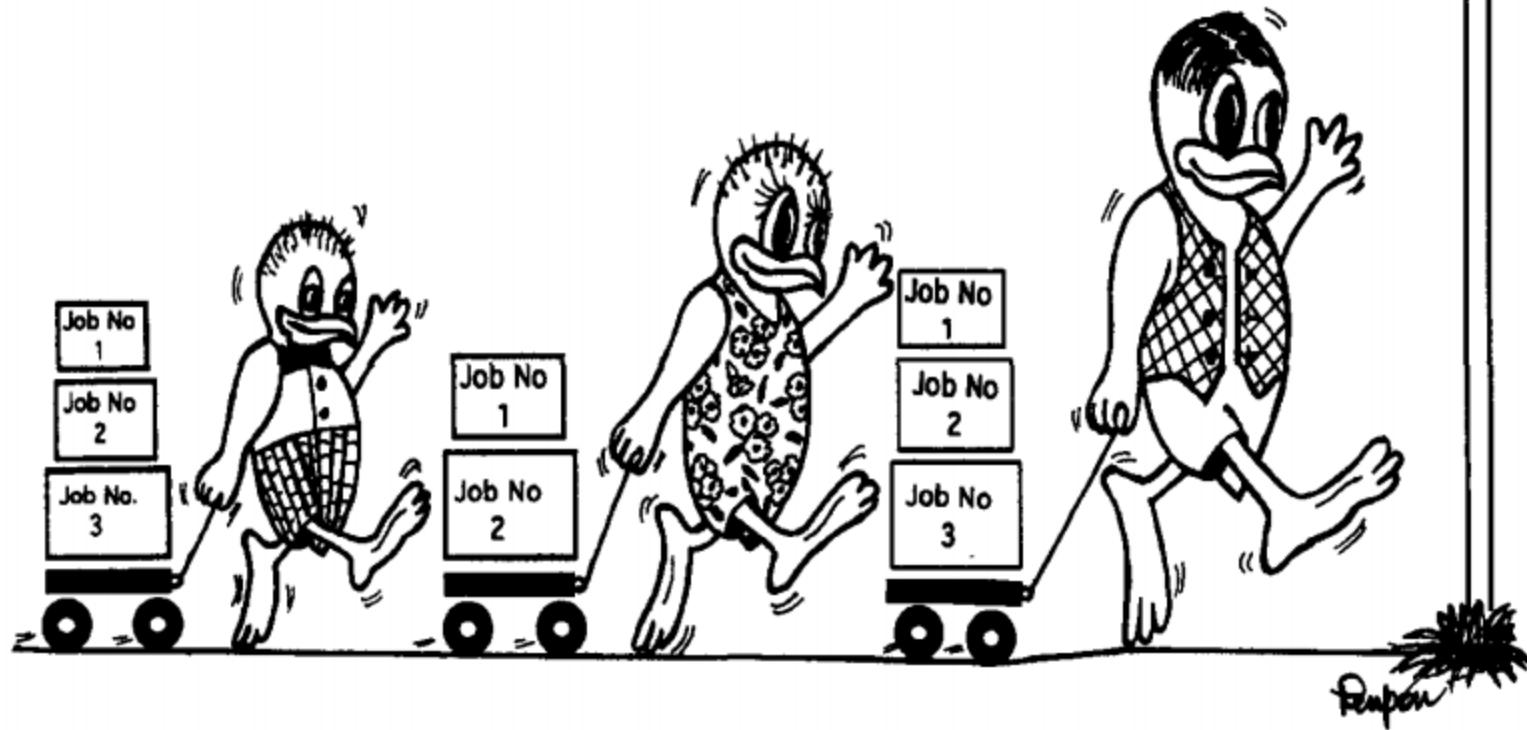
## Кардинални ограничения

### Дефиниция

Кардиналното ограничение е свързано с броя на инстанциите на „entity“, които могат да бъдат свързани с инстанции на друго „entity“ в една релация, и можем да дефинираме ограниченията, по-точно:

*Кардинално ограничение на „entity“ A в една релация R(AB) дефинира броя на инстанциите на „entity“ B, които могат да бъдат свързани с една инстанция на „entity“ A.*

**CUSTOMER**  
queue the other side



Instances of an entity may be related to different numbers of instances of another entity.

Ще използваме краткият термин кардиналност, вместо термина кардинални ограничения. Най-общите кардинални стойности са една или много, където смисъла на много е една, или повече от една инстанция, и кардиналното ограничение трябва да е вярно по всяко време от живота на „entity“.

Ако комбинираме общите кардиналности на две „entities“ в бинарна релация, то те имат четири възможни стойности, посочени като кардинални отношения. Четирите стойности са: едно към едно (1:1), един към много (1: N или 1: M), много към един (N: 1 или M: 1) и много към много (M: N).

IE

За IE, термина кардиналност се отнася до това "колко от едно „entity“ е асоциирано с колко от друго „entity“" (Martin 1990).



**Figure 3.11** 1:N/N:1 cardinality. A customer is associated with many customer accounts and a customer account is associated with one customer (IE).



**Figure 3.12** 1:1 cardinality. An advertising campaign is associated with one project and a project is associated with one advertising campaign (IE).



**Figure 3.13** M:N cardinality. A purchase order is associated with many parts and a part is associated with many purchase orders (IE).

Кардиналното ограничение се специфицира чрез **максималния кардинален показател** в IE, които могат да приемат стойности 1 или  $N$ , където  $N$  означава много.

Максималния кардинален индикатор =  $N$ . Когато *instancieta* на „entity“ предприятие  $A$  може да е свързана с много инстанции на „entity“  $B$ , максималната стойност на кардиналния индикатор за  $A$  е  $N$ . Това е представена графично с „гарваново стъпало“ поставено близо до „entity“  $B$  върху линията представляща  $R(AB)$ . Фигура 3.11 показва тази ситуация, когато инстанция на клиент може да бъде свързана с много инстанции на клиентския акаунт. Това е един  $1:N$  кардинално съотношение, от посоката на клиента.

*Максимален кардинален индикатор = 1.* Когато инстанция на „entity“ A може да бъде свързано само с една инстанция на „entity“ B, максималната стойност на кардиналния индикатор за A е 1. Това е представено графично с чертичка, под прав ъгъл, поставена най-близо до „entity“, пресичайки линията представляваща R(AB). Фигура 3.11 показва ситуация, когато инстанция на потребителски акаунт („entity“ A) може да бъде свързана само с една инстанция на клиент („entity“ B). Това е пример на N: 1 кардинално съотношение, от посоката на потребителския акаунт.

Можем да отбележим, че едно N: 1 кардинално съотношение е обратно на 1:N съотношение.

IE работят по този начин, което се отбелязва с термина „look across“, (Ferg, 1991) на кардиналността, такава че кардиналността на „entity“ A в една релация R(AB) дефинира броя инстанции на „entity“ B, че инстанция на „entity“ A вижда когато 'looks across' релацията R. На фигури 3.12 и 3.13 са показани примери за 1:1 и M: N кардинални съотношения.

## MEIN

В MEIN, термина *степен на асоцииране* съответства на кардинално ограничение, относно участие в асоциация на всяко от съответните „entities“ (MEIN II, 1991).

Кардиналността в MEIN се изразява с помощта на концепцията на кардинално съотношение. MEIN също приема стила „look across“ на релационната кардиналност.

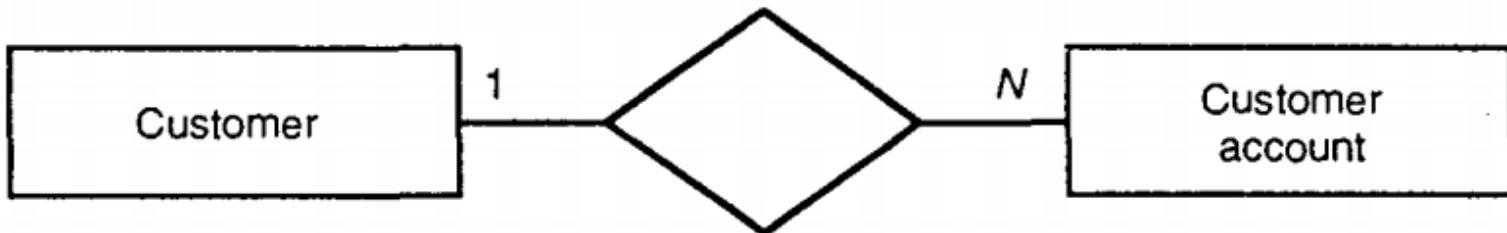
Едно към много ( $1:N$ ). Едно към много ( $1:N$ ) кардинално съотношение в релация  $R(AB)$ , изразена в  $A \rightarrow B$  посока, е когато инстанцията на „entity“  $A$  може да бъде свързана с много инстанции на „entity“  $B$ , и една инстанция на „entity“  $B$  може да е свързана със само една инстанция на „entity“  $A$ .

Фигура 3.14 показва пример за графично представяне на кардиналното съотношение  $1:N$ , където знакът „ $1$ “ до „entity“ показва, че инстанция на друго „entity“ може да бъде свързана със само една инстанция на това „entity“. Знакът  $N$  или  $M$  показва, че инстанция на друго „entity“ може да бъде свързана с много инстанции на това „entity“.

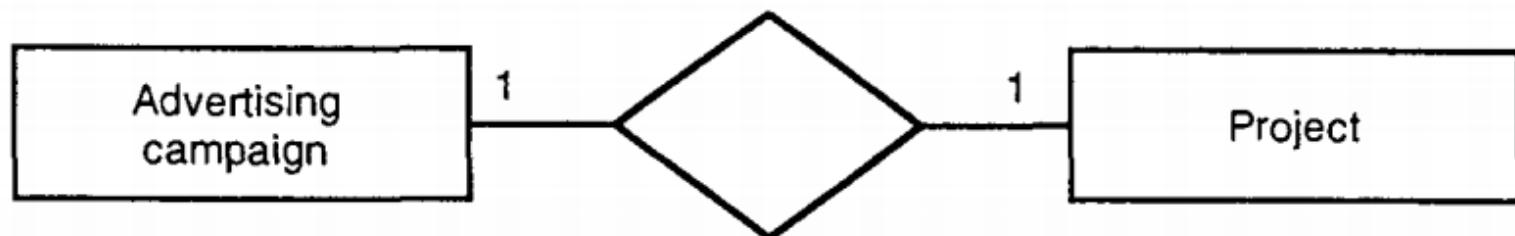
*Едно към едно (1:1).* Едно към едно (1:1) кардинално съотношение в една релация R(AB) е налице, когато инстанция на „entity“ A може да бъде свързана със само една инстанция на „entity“ B, и инстанция на „entity“ B може да бъде свързана само с една инстанция на „entity“ A.

За графично представяне на кардинално съотношение едно към едно, знака „1“ се поставя до двете „entities“ както е показано на фигура 3.15.

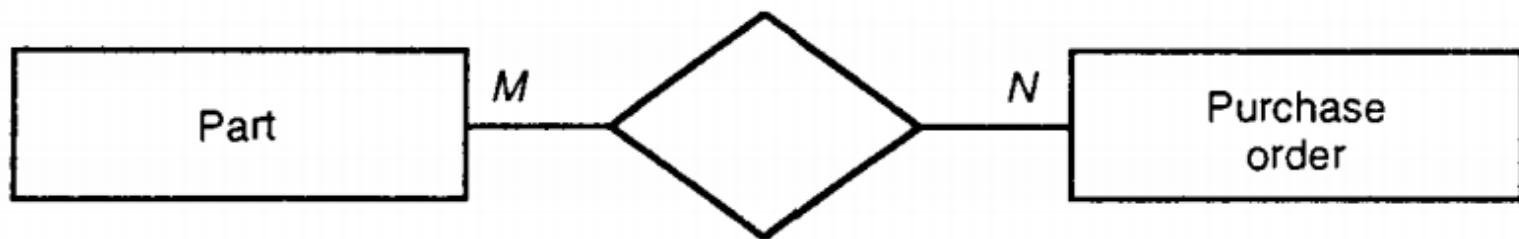
*Много към много (M: N).* Кардинално съотношение много към много (M: N) в една релация R(AB) е когато една инстанция на „entity“ A може да бъде свързана с много инстанции на „entity“ B и една инстанция на „entity“ B може да бъде свързана с много инстанции на „entity“ A. Такъв пример е показан на фигура 3.16.



**Figure 3.14** 1:N cardinality. A customer is associated with many customer accounts and a customer account is associated with one customer (MEIN).



**Figure 3.15** 1:1 cardinality. An advertising campaign is associated with one project and a project is associated with one advertising campaign (MEIN).



**Figure 3.16** M:N cardinality. A purchase order is associated with many parts and a part is associated with many purchase orders (MEIN).

## MERISE

Концепцията за кардиналност в MERISE е подобна в едно отношение на тази за IE, тъй като тя се отнася за спецификацията на минималния и максималния брой инстанции. Въпреки това, приликата свършва там, тъй като това е броят на инстанциите на релация в която посочените инстанции на „entity“ могат да участват. Това се нарича „свързаност“ (Planche, 1992).

Кардиналният стил в MERISE е по-различен и се нарича стил на участие (Ferg, 1991), като „entity“ инстанцията е свързана с релационната инстанция вместо да е свързана с инстанция от друго „entity“.

Но както MERISE референциите посочват явно, че MERISE релационна инстанция може да се свърже към не повече от една инстанция на свързани „entities“ (Collongues и др., 1989; Rochfeld, 1987) стила на участие е концептуално еквивалентен на „lookacross“ стила.

MERISE осигурява кардинални параметри – минимум ( $x$ ) и максимум ( $y$ ), посочени заедно като ( $x, y$ ). Максималният кардинален параметър изразява смисъла на кардиналното ограничение,  $x$  параметъра засега не се обсъжда. Максималния кардинален параметър може да приеме стойност 1 или  $N$ , където  $N$  означава много.

Максимална кардиналност = 1. Максималната кардиналност на „entity“ A в една релация R(AB) е 1, когато инстанция на „entity“ A може да участва в максимум една инстанция на релацията R. За да изразим графично това, пишем израз (x, N) за „entity“ A.

Максимална кардиналност = N. Максималната кардиналност на едно „entity“ в релацията R(AB) е голяма, когато инстанция на „entity“ A може да участва в много инстанции на релацията R. Графично това се изразява като се запише израза (x, N) за „entity“ A.

Фигура 3.17 показва пример на тези две ситуации, където инстанция на потребителски акаунт може да участва в само една инстанция на релацията с потребител, и инстанция на клиент може да участва в много инстанции на релацията с клиентския акаунт. Кардиналността на съотношението на релацията е еквивалентно на 1: N, от посоката на клиента.

Основната разлика между „lookacross“ и участващите стилове е само една от повърхността на графично представяне. Стилът на „lookacross“ представлява кардиналността, на пример, „entity“ A до „entity“ B, докато стила на участие представлява кардиналността на „entity“ A до „entity“ A.



**Figure 3.17** 1:N/N:1 cardinality. A customer is associated with many customer accounts and a customer account is associated with one customer (MERISE).



**Figure 3.18** 1:1 cardinality. An advertising campaign is associated with one project and a project is associated with one advertising campaign (MERISE).



**Figure 3.19** M:N cardinality. A purchase order is associated with many parts and a part is associated with many purchase orders (MERISE).

Релацията между две „entities“, и двете с кардиналност ( $x, 1$ ), е еквивалентно на кардинално съотношение „едно към едно“ (1:1), както е показано на фигура 3.18. Ако две „entities“ са с кардиналност ( $x, N$ ) и участват в една релация, както е на фигура 3.19, това е еквивалентно на кардинално съотношение „много към много“ ( $M: N$ ).

## SSADM

В SSADM, концепцията на кардиналното ограничение е известна като *степен на релацията*, и се отнася до броя на инстанции на „entity“, с които могат да бъдат свързани инстанции на друго „entity“.

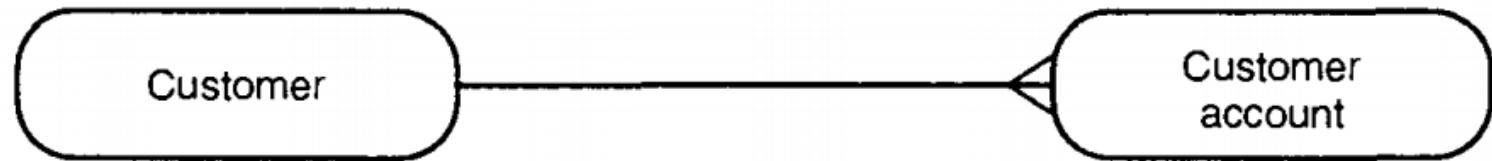
Кардиналният стил, който приема SSADM е стила „lookacross“, (Ferg, 1991). Дефинирани са четирите често срещани типове релационни кардинални съотношения.

*Един към много.* За графично представяне на кардинално отношение „едно към много“ ( $1: N$ ), знака "крак врана", сложен на линията от „entity“ показва, че една инстанция на друго „entity“, може да бъде свързана с много инстанции на това „entity“. Равнинната линия от „entity“ показва, че инстанция на друго „entity“ може да бъде свързана със само една инстанция на това „entity“. Пример е показан на фигура 3.20.

*Едно към едно.* SSADM представя графично кардинално съотношение „едно към едно“ (1:1) с равнинна линия, която присъединява „entity“ A към „entity“ B с „тризъбеца“, както е показано на фигура 3.21.

*Много към много.* Кардинално съотношение „много към много“ ( $M: N$ ) е графично представено с линия, която ще се присъедини „entity“ A към „entity“ B с „тризъбеца“ в двата края, както е показано на фигура 3.22.

*Фиксирана степен.* Понякога е фиксиран броя на инстанциите на „entity“, които могат да бъдат свързани с инстанция на друго „entity“. За да представим фиксирана степен графично, броя на инстанциите на „entity“ B, които могат да бъдат свързани с една инстанция на „entity“ A е написан до „entity“ B. Пример за това е показан на фигура 3.23.



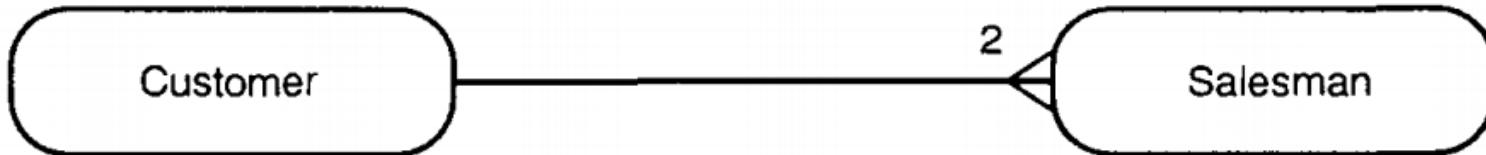
**Figure 3.20** 1:N/N:1 cardinality. A customer is associated with many customer accounts and a customer account is associated with one customer (SSADM).



**Figure 3.21** 1:1 cardinality. An advertising campaign is associated with one project and a project is associated with one advertising campaign (SSADM).



**Figure 3.22** M:N cardinality. A purchase order is associated with many parts and a part is associated with many purchase orders (SSADM).



**Figure 3.23** Fixed degree. A customer is associated with exactly two salesmen (SSADM).

## ОГРАНИЧЕНИЕ НА УЧАСТИЕ

### Дефиниция

Ограничението на участие на „entity“ определя дали всички инстанции на „entity“ трябва да участват в релации с инстанции на друго „entity“.

Има два типа ограничения за участие, *пълно и частично*, което определяме чрез разглеждане на участието на инстанциите на „entity“ A в релацията R(AB).

### Пълно участие

Когато всички инстанции на „entity“ A трябва да участват в релация с инстанции на „entity“ B, участието на A в релацията R(AB) се нарича пълно.

В тази ситуация, която често се използва термина *задължително участие*, инстанция на A не може да съществува, без да е свързана с „entity“ B.

### Частично участие

В случаите, когато не задължително всички инстанции на „entity“ A да участват в релацията с инстанциите на „entity“ B, участието на A в релацията R(AB) се нарича частично.

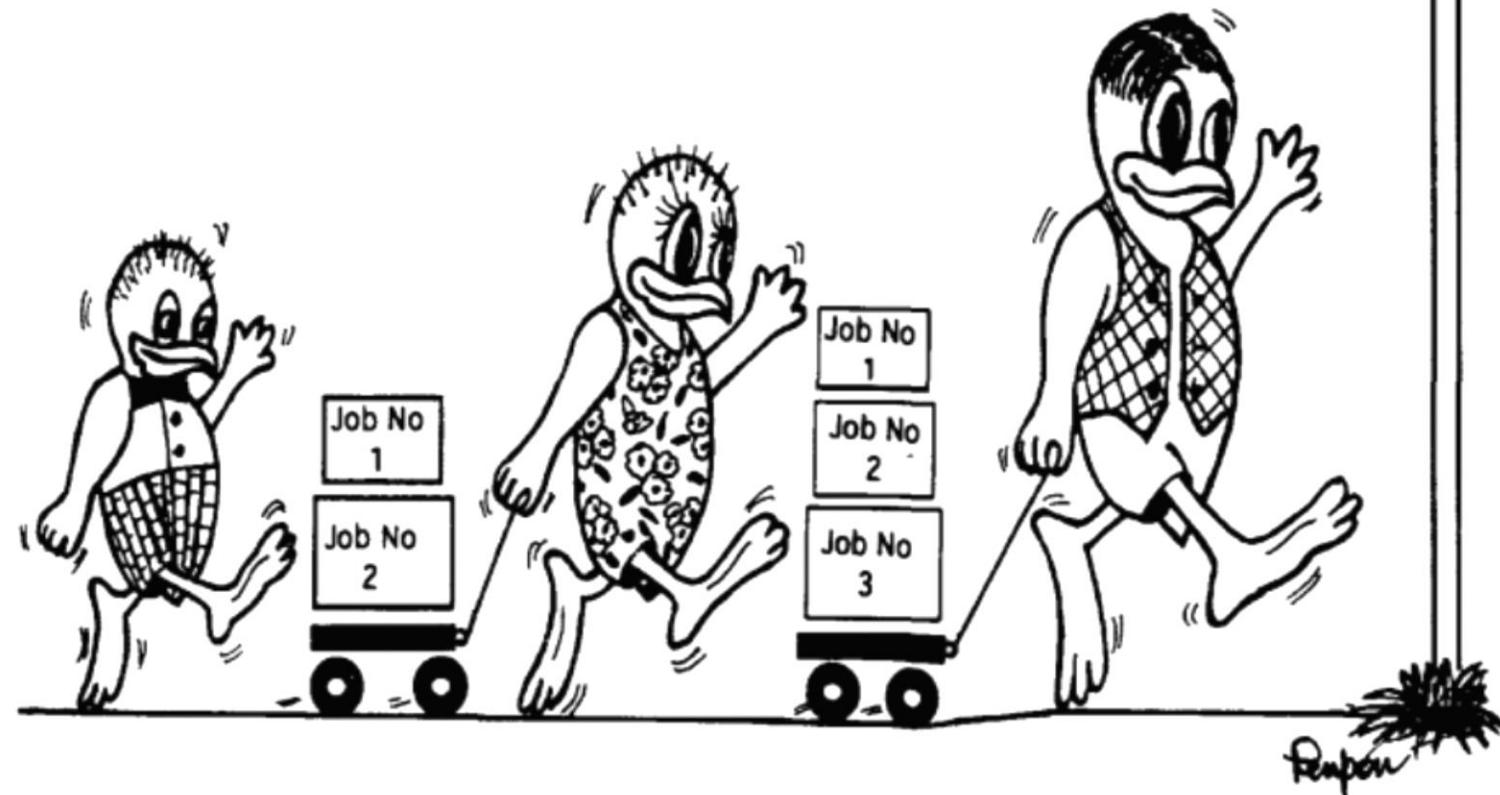
В тази ситуация, често наричана *опционална*, инстанция на „entity“ A може да съществува без да е свързана с „entity“ B. Ограничението за участие трябва да е истина, по всяко време през живота на „entity“.

IE

В IE, ограничението на концепцията за участие се определя от минималния кардинален индикатор, за едно „entity“ в една релация, която може да приеме само две стойности: 0 или 1. Когато стойността е 0, тя е еквивалентна на optionalното участие на „entity“ в релацията. Когато стойността е 1, това е еквивалентно на mandatoryното участие на „entity“ в релацията.

Третата концепция също се дефинира като "по желание с тенденция към задължително" или "преходна асоциация" (Finkelstein, 1989). Тази концепция позволява участието на „entity“ в една релация да бъде optionalно за даден период от време, и след това предполага mandatoryното участие. Стилът на изразяване на ограничението за участие в IE е „lookacross“, (Ferg, 1991).

**CUSTOMER**  
queue the other side



Instances of an entity may or may not participate in a relationship with instances of another entity.



**Figure 3.24** A job order may be associated with a project, but a project must be associated with a job order (IE).



**Figure 3.25** A supplier may be recorded initially without a supplier account, but should be associated with one later (early version of IE).

За графично представяне на задължителното участие на „entity“ A в релацията  $R(AB)$ , се използва къса линия, под прав ъгъл, пресичаща релационната близо до „entity“ B и за да се представи опционалното участие на „entity“ A се използва един малък кръг пресичащ линията на релацията близо до „entity“ B. На фигура 3.24 е показан пример.

За представяне на опционална тенденция към задължително участие за „entity“ A, къса линия и малък кръг, пресичащ асоциационната линия се поставя в непосредствена близост до „entity“ B. Фигура 3.25 показва пример.

## MEIN

MEIN осигурява ограничение за участие.

## MERISE

Споменахме кардиналните параметри ( $x, y$ ) за „entity“ в една релация и подчертахме максималния кардинален параметър ( $y$ ).

Развиваме описанието на минималния кардинален параметър, тъй като той съответства на концепцията за ограничения за участие, специфицирайки минималният брой на инстанции на релация към която всяка инстанция на „entity“ трябва да е свързана.



**Figure 3.26** A job order may be associated with a project, but a project must be associated with a job order (MERISE).

Минималната кардиналност за „entity“ може да приеме една от двете стойности: 0 или 1. Ако стойността е 0, то е еквивалентно на допълнително участие на „entity“ в релацията, и ако стойността е 1, то е еквивалентно на задължително участие на „entity“ в релацията. Стилът на изразяване на ограничение за участие в MERISE е стила на участие (Ferg, 1991).

За да се представи графично задължителното участие на „entity“ в релацията  $R(AB)$ , изразът  $(1, y)$  е записан до „entity“ A; докато при опционалното участие на „entity“ A, израза  $(0, y)$  е написан в непосредствена близост до „entity“ A. Пример е показан на фигура 3.26. Някои автори (Rochfeld, 1987; Tardieu и др., 1983) използват термина пълно и частично за отнасяне до ограничения за участието от двете страни на релацията разглеждана като едно цяло.

## SSADM

SSADM използва термините *са задължително* и *опционално*, за да се опише двата вида ограничение за участие.

За да се изрази графично задължително участие, SSADM използва непрекъсната линия близо до задължителното „entity“, а за optionalno участие, SSADM използва пунктирана линия.

## РЕЗЮМЕ

Бинарни релации кардинални ограничения

Всички методи предоставят тези понятия, въпреки че всички те използват различни термини за кардинално ограничение.

Ограничение на участие

MEIN не предоставя тази концепция. Всички други методи я предоставят, въпреки че използват различни термини.

Релационен атрибут

MEIN и MERISE го предоставят, а IE и SSADM не го предоставят. Въпреки това, взаимоотношенията с атрибути могат да бъдат моделирани в IE и SSADM чрез сечения или свързани „entities“.

Таблица 3.1 показва съответствието между референтната рамка и концепциите на метода, както и разликите в терминологията.

**Table 3.1** Relationship concepts

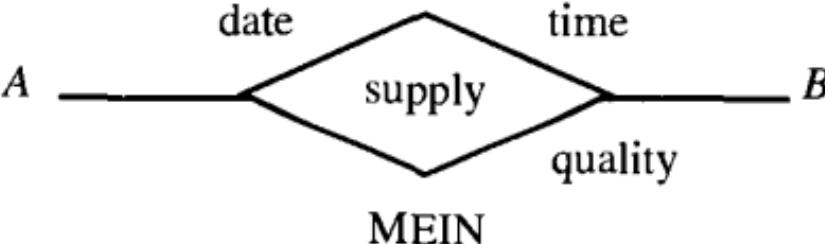
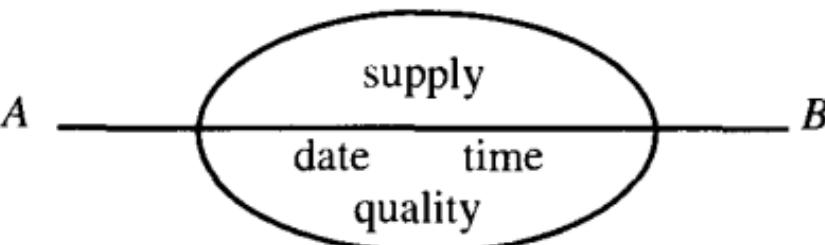
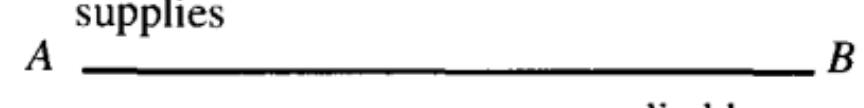
Concept	Methods			
	IE	MEIN	MERISE	SSADM
Binary relationship	yes	yes	yes	yes
Term to refer to a relationship among same entity types	recursive	recursive	reflexive	pig's ear
Relationship name	label (two labels allowed)	relationship name (only one)	relationship name (only one)	relationship link phrase (two names allowed)
Relationship with attribute	–	yes	yes	–
Relationship constraint definition	cardinality and participation	cardinality	cardinality and participation	cardinality and participation
Cardinality style	lookacross	lookacross	participation	lookacross
Participation	mandatory and optional	–	mandatory and optional	mandatory and optional
Participation style	lookacross	–	participation	participation

## Минимални различия

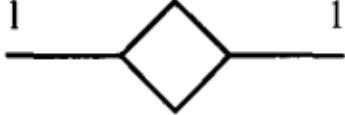
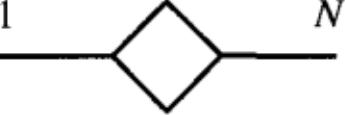
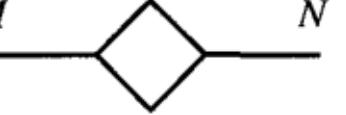
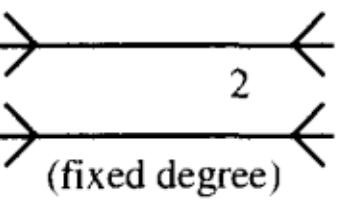
Mein и MERISE позволяват само едно име за една връзка, докато IE и SSADM позволяват едно име за всяка посока на релацията.

Таблиците от 3.2 до 3.4, показват различни, повърхностни графични представяния на различни концепции.

**Table 3.2** Graphical representation of a relationship, including relationships with attributes

With attributes	Without attributes
 <p>MEIN</p>	 <p>IE</p>
 <p>MERISE</p>	 <p>SSADM</p>

**Table 3.3** Cardinality types

	Graphical representation of relationship cardinality		
Method	one to one (1:1)	one to many (1:N)	many to many (M:N)
IE			
MEIN			
MERISE			
SSADM			

## ЗАКЛЮЧЕНИЕ

Методите се различават по отношение на релационните понятия.

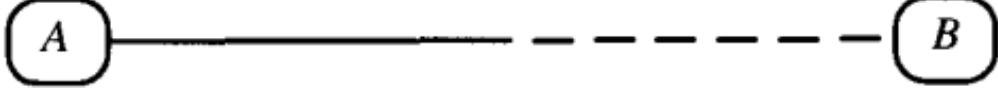
Референтни рамкови концепции

*Бинарни релации и кардинални ограничения.* Всички методи ги предоставят.

*Ограничение на участие.* MEIN не предоставя това.

*Релационен атрибут.* IE и SSADM не предоставят това.

**Table 3.4** Graphical representation of participation constraint

Method	Relationship participation	
	$A \rightarrow B$ mandatory	$B \rightarrow A$ optional
IE		
MEIN	not defined	
MERISE		
SSADM		

## Обхват на метода

- MEIN не поддържа концепцията за ограничение на участието.
- MEIN и MERISE, позволяват само едно име за релация. Това може да редуцира смысла показан на диаграмата.

## Графични представления

На повърхността, графичните представления правят методите да изглеждат много различно, но представените концепции са еквивалентни.

# ЛЕКЦИЯ 4. Множествена релация

Започнахме дискусия за възможностите на методите за моделиране на релация между две „entities“, използвайки концепцията на бинарното отношение.

Въпреки това, чест случай в организации е когато, които има релации, с участието на повече от две „entities“. Такива релации се определят като *множествена релация*. Прави се разлика между множествени релации в организацията и концепцията на *n*-арна релационна референтна рамка. Ще опишем съответното понятие използвано от методи за моделиране на различни типове на множествена релация.

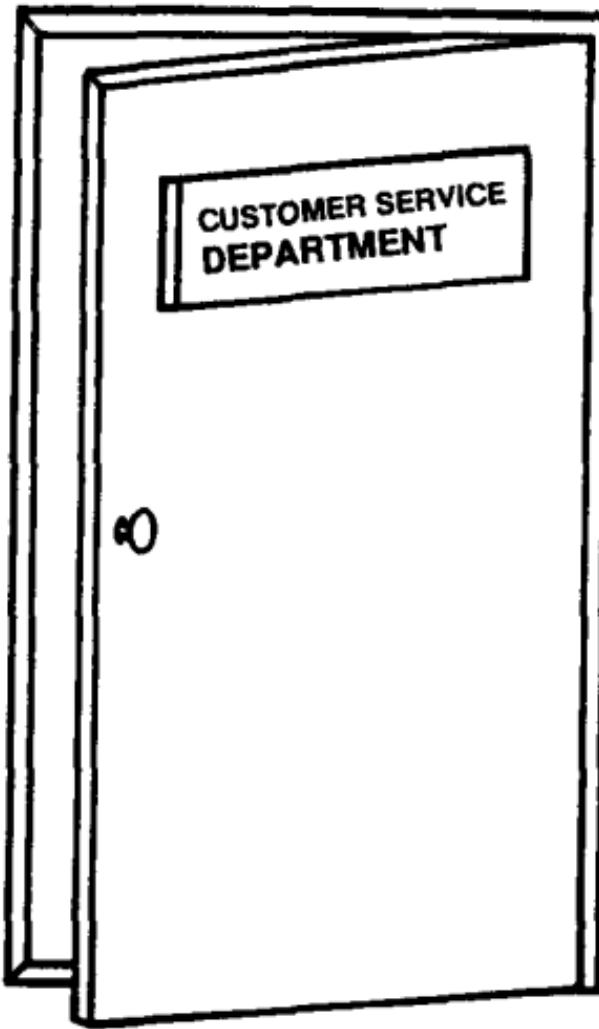
## ОПРЕДЕЛЕНИЕ

Определението на множествена релация е следното:

Множествена релация е връзка между повече от две „entities“, която се счита в приложението като модул или комбинация, която семантично не може да бъде разделена.

„Entities“, които са в такава комбинация ще наричаме *участващи „entities“*. Общата дефиниция на концепцията за релацията, дадена в Лекция 3. се отнася и за N-арна релация.

Ще обсъдим как методите моделират множествени релации. MEIN и MERISE предвиждат специална концепция за *N*-арно отношение, докато IE и SSADM предоставят само бинарна релационна концепция. Ще видим как се справят методите с множествени релации, при които се прилагат различни кардинални и ограничения за участие.



Roupon

A multiple relationship concerns a relationship between two or more entities.

## МОДЕЛИРАНЕ НА МНОЖЕСТВЕНА ВРЪЗКА ЧРЕЗ ИЗПОЛЗВАНЕ НА N-АРНАТА РЕЛАЦИОННА КОНЦЕПЦИЯ

### MEIN и MERISE

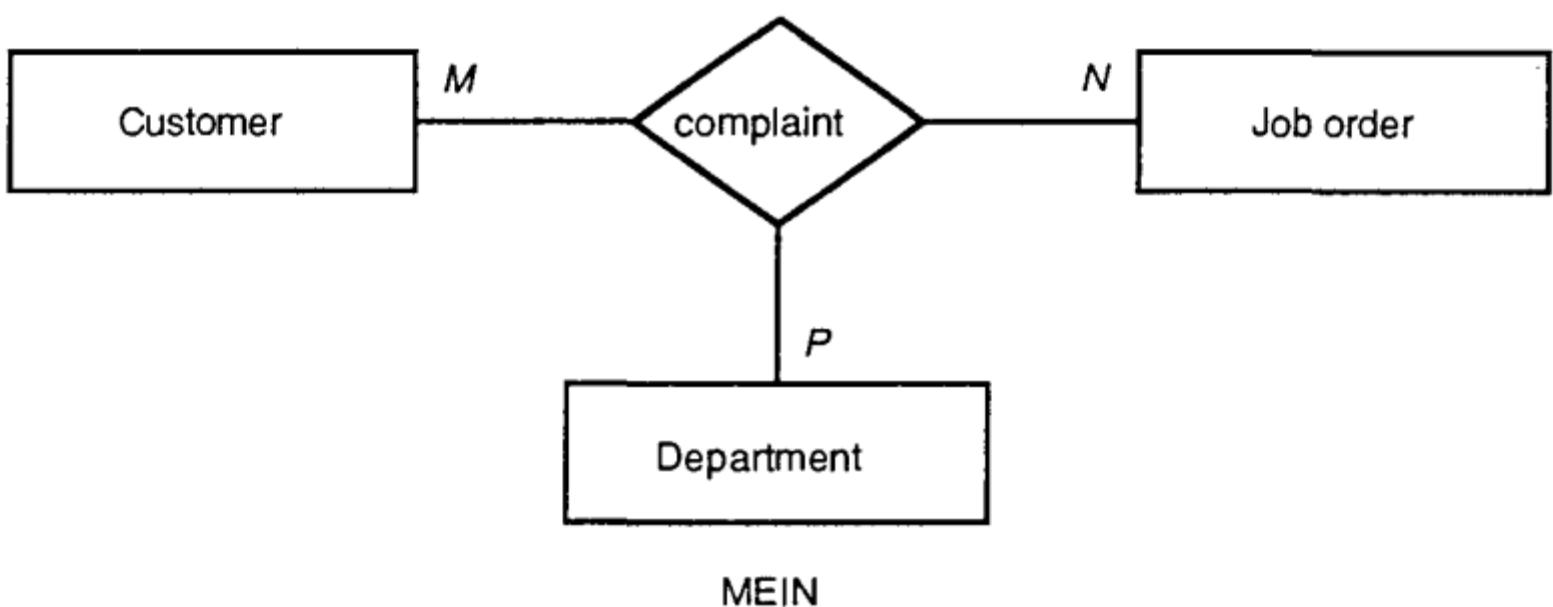
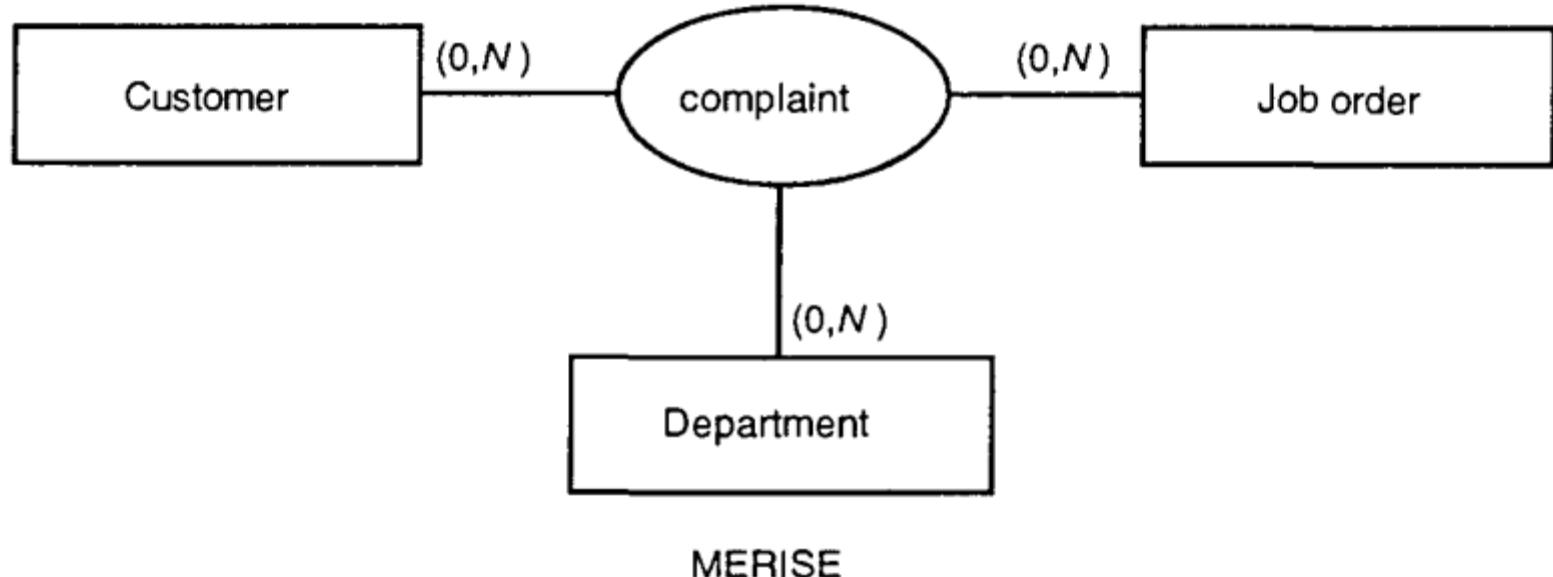
В първият пример на множествена релация беше въведена в Лекция 3, и се отнася до три свързани „entities“, клиенти, реда на работа и отдела, който сега ще опишем по-подробно.

Смисълът на тази връзка е, че клиентът може да направи оплакване в отдела за реда на работа. Кардиналните съотношения между всички двойки от „entities“ са  $M: N$ , такива, че, например, клиентът може да се оплаче от много поръчки за работа, отдела може да бъде упрекнат, от много клиенти, както и обратното.

Важният пункт е, че искаме да сме в състояние да моделираме факта, че например е налице комбинация от „entities“, в която даден клиент т се оплаква на определен отдел за определен ред на работата. Концепцията на  $N$ -арната релация в MEIN и MERISE моделира смисъла на тази комбинация от „entities“, както и релацията, жалбата, както е показано на фигура 4.1.

Имплицитно, съвместна инстанция на всяка от участващите „entities“ уникално идентифицира една инстанция на оплакването.

На Фигура 4.1 за MEIN знаковете  $M$ ,  $N$  и  $P$  графично представляват  $M: N$  кардинални съотношения между „entities“. За MERISE,  $M: N$  кардиналните съотношения са представени чрез максималната кардинална стойност  $N$  за всяко „entity“, която определя, че една инстанция на „entity“ може да участва в много инстанции на релацията, всяка инстанция на което е комбинация от една инстанция на всяка от участващите „entity“.



**Figure 4.1** Multiple relationship complaint with participating entities customer, job order and department, using the  $n$ -ary relationship concept (MEIN and MERISE).

## МОДЕЛИРАНЕ НА МНОЖЕСТВЕНИ РЕЛАЦИИ, ЧРЕЗ КОНЦЕПЦИЯТА НА БИНАРНАТА РЕЛАЦИЯ

IE и SSADM предоставят само бинарни релационни концепции за моделиране на релации. За да се избегнат проблеми при моделиране на множествени релации с тази концепция е необходимо да се създаде „абстрактно entity“ и да се свържат с него участващите „entity“.

След това се описват двата основни подхода за моделиране на множествена релация чрез използване на бинарната релационна концепция: *N*-арния подход и вложението двоичен подход.

### Концептуални проблеми при двоични релации

Разглеждаме множествена релация с три участващи „entity“ клиенти, реда за работа и отдел. Фигура 4.2 показва как тази връзка може да се моделира от IE и SSADM чрез използване на двоична релационна концепция за връзка само с участващите „entity“.

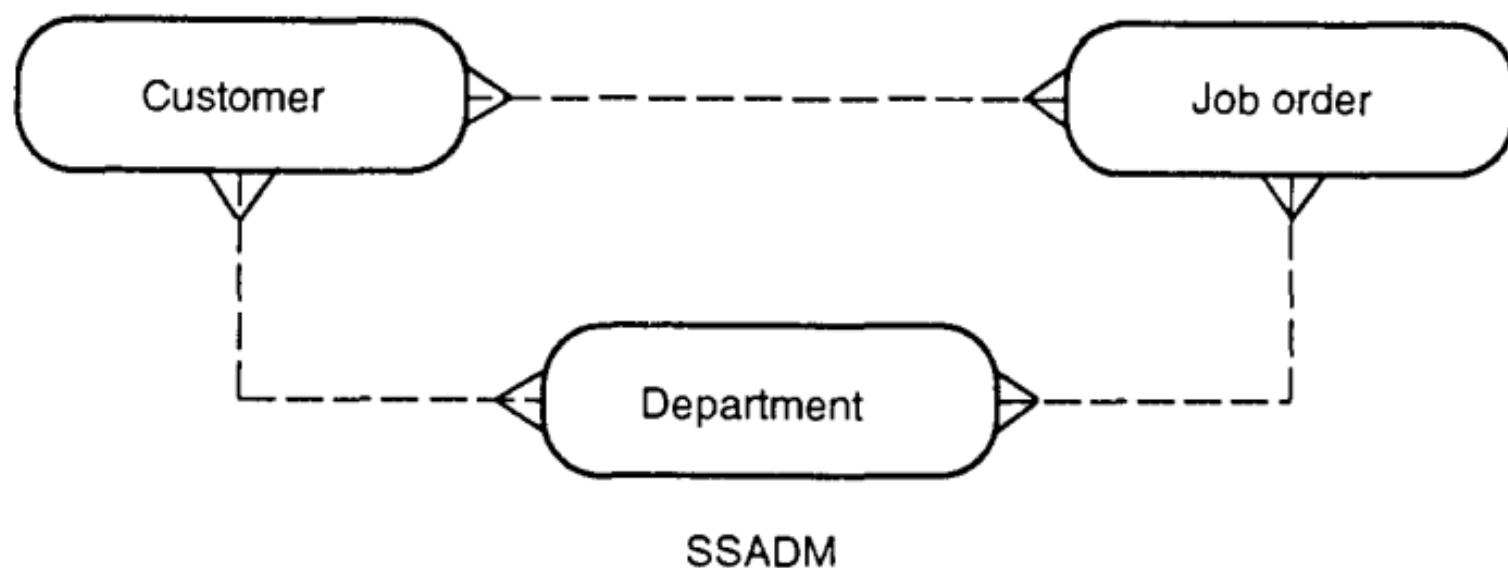
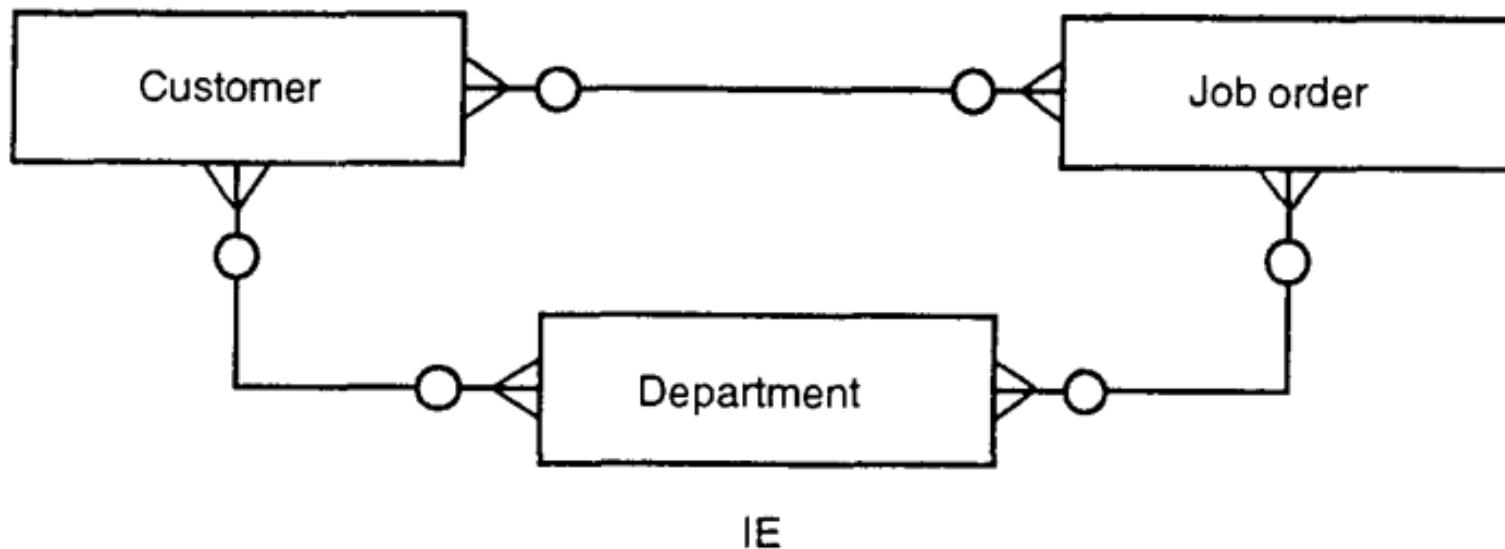
Нека разгледаме ситуация, в която двама клиента C1 и C2 се оплакват за реда на работа J1; C1 се оплаква на отдел D1 и C2 се оплаква на отдел D2. Нашият модел се нуждае да записва за даден клиент, от кой номер на работа има оплакване до кой отдел.

Фигура 4.3 представлява „entity“ и релационните инстанции, които са в състояние да се записват от модела на фигура 4.2. Има два записани факта:

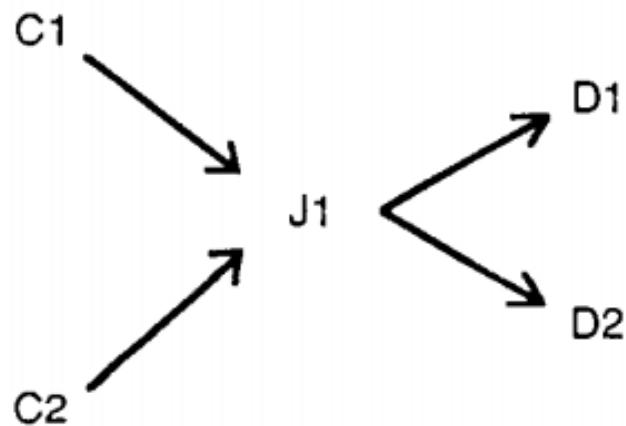
Бинарната релация между клиента и реда на работа на фигура 4.2 може да записва факта, че клиентите C1 и C2 се оплакват от реда на работа J1.

Бинарните отношения между реда на работа и отдела на фигура 4.2 може да запишат факта, че реда на работа J1 е свързан с отделите D1 и D2.

Въпреки това, ние не можем да направим извод от тези два факта от кой отдел D1 или D2, клиент C1 се оплаква относно реда на работа J1.



**Figure 4.2** Binary relationships with participating entities customer, job order and department (IE and SSADM).



**Figure 4.3** Job order J1 has two related customers and two departments, but does not record which customers are related to which departments.

Релацията между клиента и отдела на фигура 4.2, не може да помогне, тъй като, например, клиентът C1 може да се е оплакал на отдел D1 относно за друг номер на работа J3.

Като цяло, не можем да моделираме множествена релация правилно чрез двоични релации само между участващите „entity“. Информацията, която желаем да запишем засяга всички участващи „entity“ като модул, и не може да се запише като съвкупност от информацията, съдържаща се в бинарните релации. Комбинацията не може да бъде разделена.

#### N-арен подход за моделиране на множествена релация

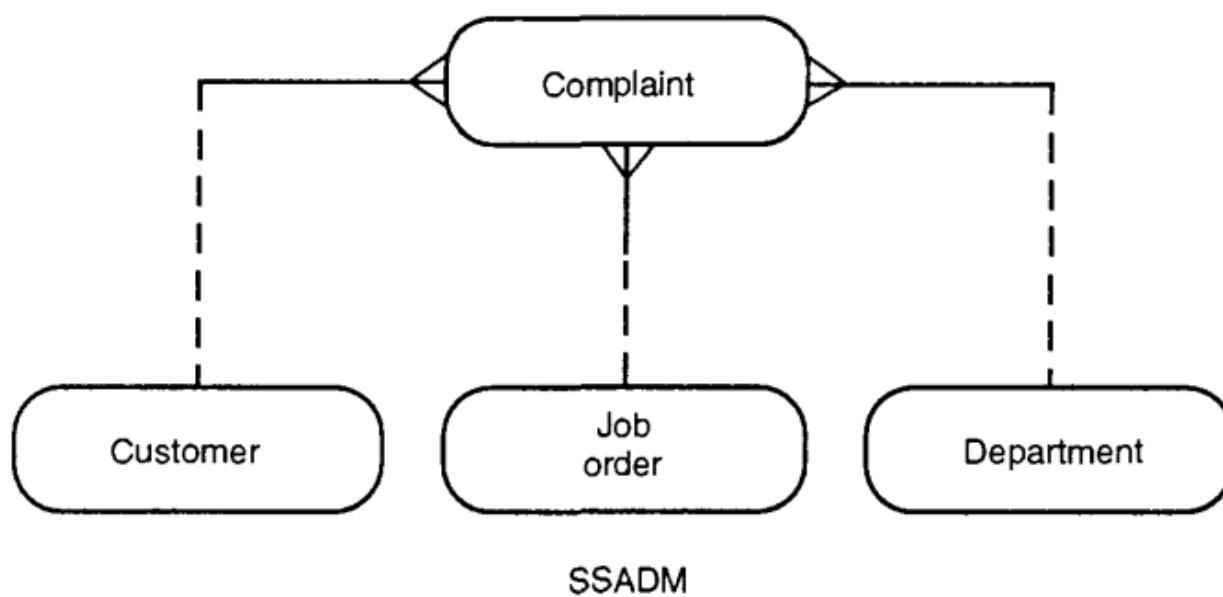
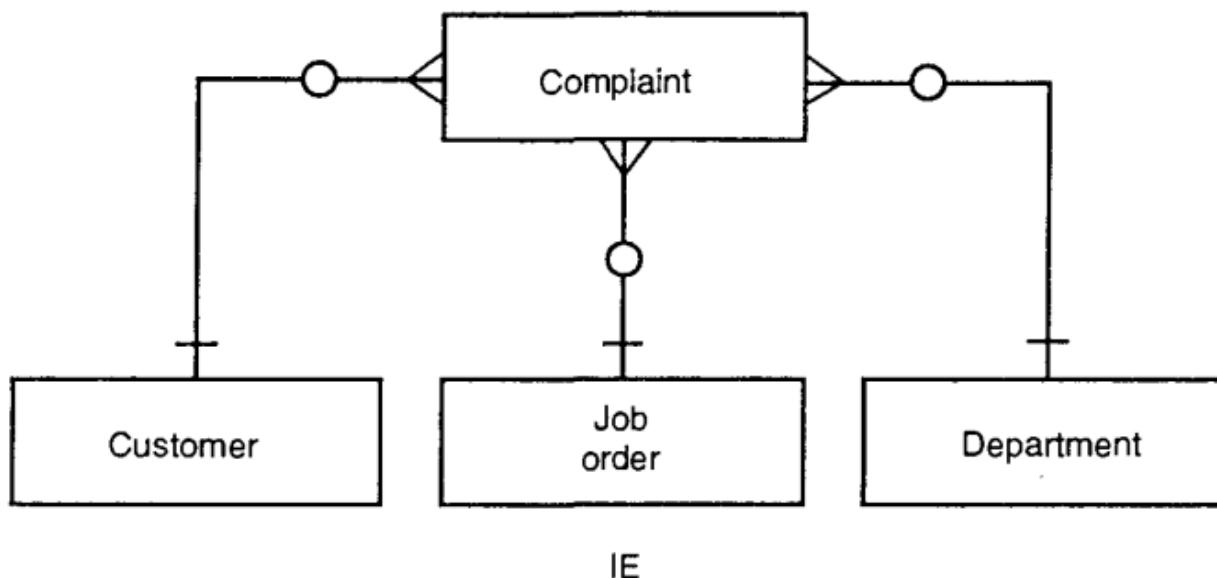
В N-арния подход, ново, абстрактно „entity“, което ние именоваме „оплакване“ е изградена от „entity“ клиенти, реда на работа и отдел. Жалбата се свързва с участващите „entity“ с бинарни релации.

Смисълът на една инстанция на оплакване е, че един клиент се оплаква за реда на работа за един отдел. Участието на абстрактното „entity“ в тези релации е задължително, и неговото кардинално съотношение с участващите „entity“ е N:1. Фигура 4.4 показва това. Необходимата информация се записва, като инстанция на клиент е асоциирана с няколко инстанции на оплакване, от които всяка инстанция е асоциирана с една инстанция на реда на работа и една инстанция на отдел.

Фактът, че съвместна инстанция за всеки един от клиентите, ред на работа и отдел, еднозначно идентифицира един случай на жалба се моделира в IE и SSADM чрез специфициране на „entity“-атрибутен списък, идентификаторът на жалба ще бъде комбинация на идентификаторите на участващите „entity“.

## Понятие за комбинирано „entity“

В примера по-горе, е естествено срещащо се име, жалба, за релацията, с която потребителите са запознати и която се изисква да се запази в модела. Ще използваме термина *комбинационно (комбинирано)* „entity“ и то е именован, абстрактен обект, използван от потребителя, за да изрази комбинацията от участващите „entity“.



**Figure 4.4** Multiple relationship complaint with participating entities customer, job order and department, using the binary relationship concept (IE and SSADM).

Въпреки това, такова „entity“ не може винаги да присъства в една множествена релация, и дизайнерът може да се наложи да измисли име за абстрактното „entity“, което се изисква за да свърже участващите „entity“. Например, да предположим, че проблемът от практиката има допълнително изискване да запише кои служители (E) за кои клиенти (C) работят, на кои номера на работи (J), ECJ може да се използва като име на ново абстрактно „entity“, което можем да назовем с термина "дизайнер entity".

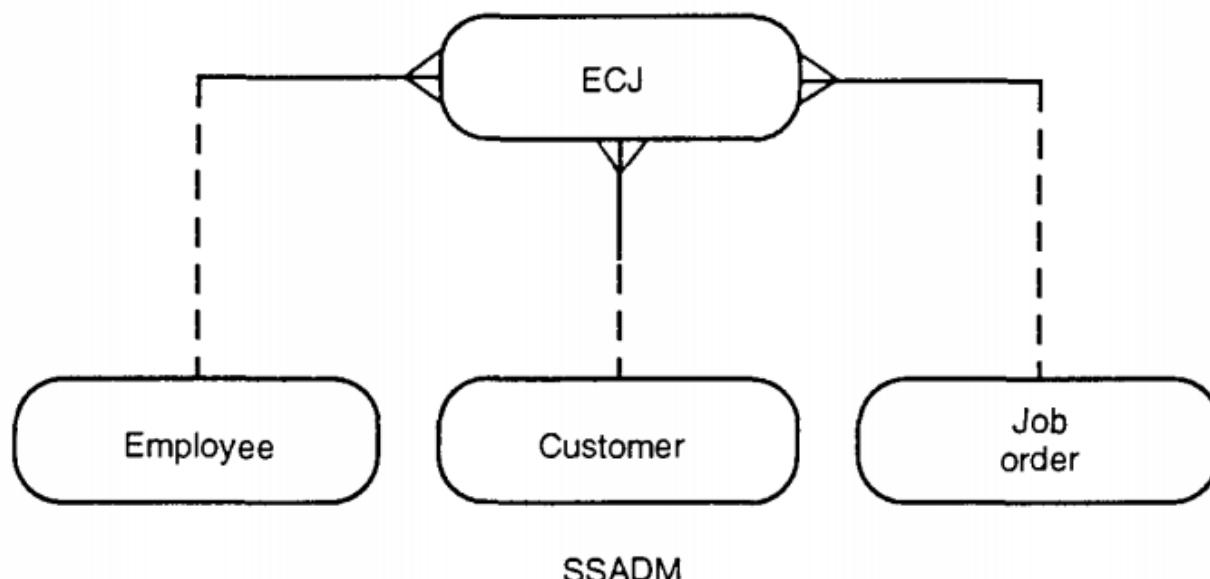
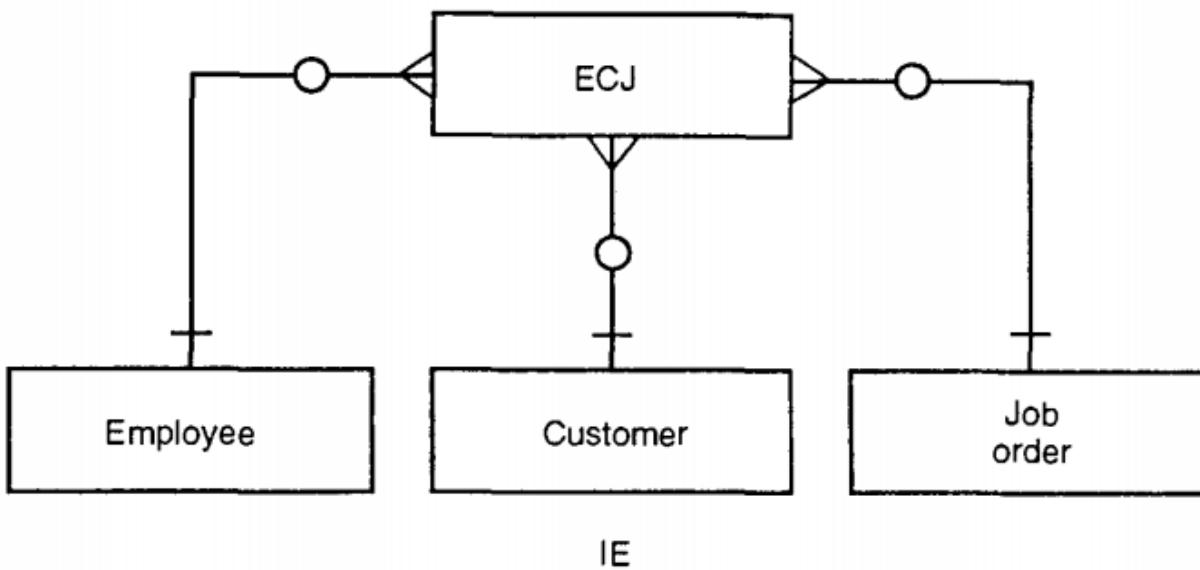
В този случай, моделирането е идентична с това в IE и SSADM, където *абстрактното „entity“ ще бъде ECJ*, както е показано на фигура 4.5.

#### Вложен бинарен подход за моделиране на множествена релация

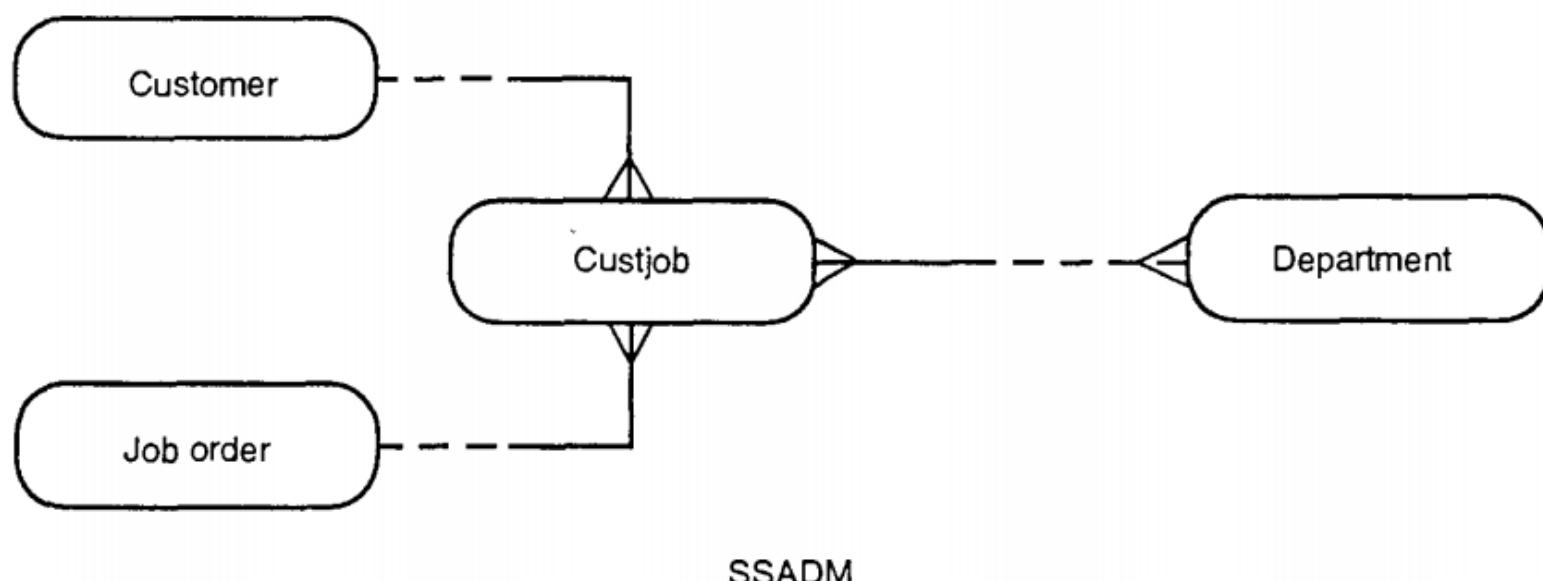
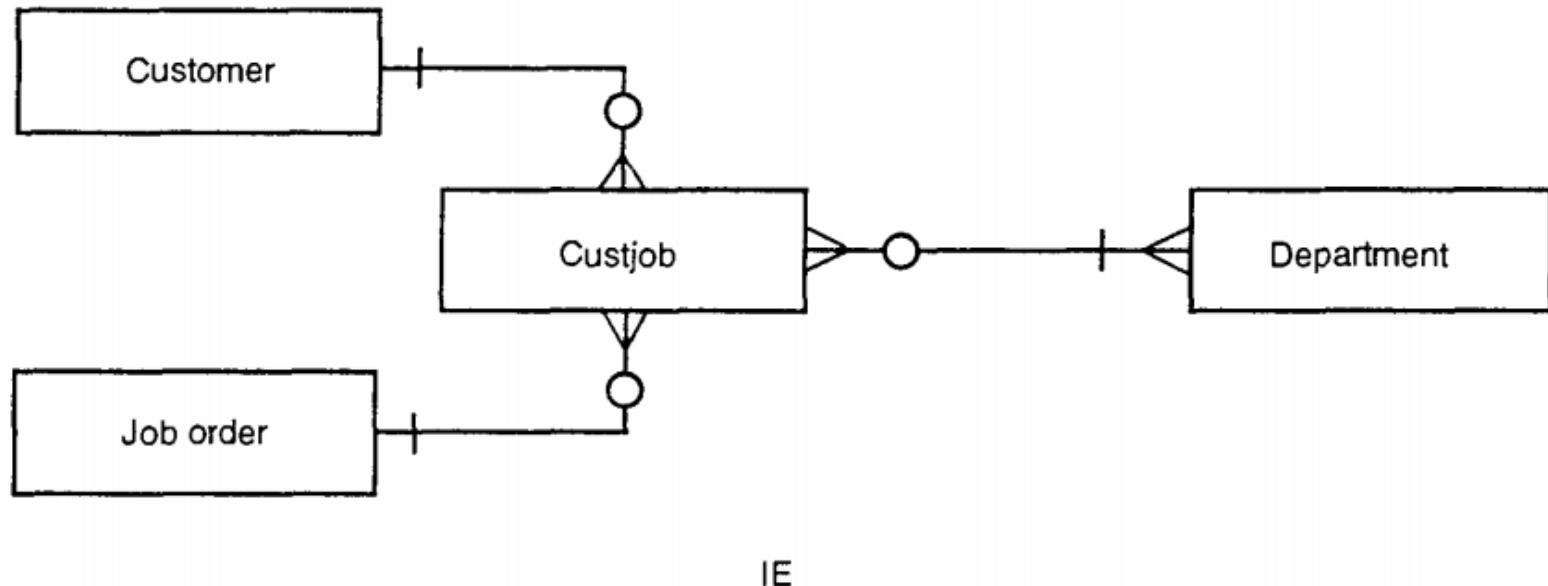
Този подход обединява две „entity“ и формира абстрактно „entity“, което е зависимо от тяхната комбинация. В този пример, ние избираме клиента и номера на работа, създавайки абстрактно „custjob entity“, със смисъла на уникална комбинация на оплакване на един клиент за един ред на работа. Това е свързано с трети „entity“ отдел с помощта на бинарна релация. Резултата на този подход е показан на фигура 4.6.

Участието на абстрактното „entity“ в тези релации е задължително. Кардиналното съотношение на абстрактното „entity“ в тази релация с невложени „entity“ е  $M: N$ , в контраст с фигура 4.5.

Клиента и реда на работа по този начин са "вложени", за да формират едно абстрактно "дизайнер entity", което ще означим с термина „custjob“. Идентификаторът на custjob трябва да бъде определен по подходящ начин, за да покаже, че неговите инстанции са се идентифицирали еднозначно чрез съвместна инстанция на всеки един от клиентите и номера на работа. Необходимата информация се записва, като инстанция на отдела се свързва с няколко инстанции на custjob, като всяка инстанция е асоциирана с точно една инстанция на клиента и номера на работа.



**Figure 4.5** Multiple relationship with participating entities employee, customer, and job order, using the binary relationship concept and the 'designer' entity ECJ (IE and SSADM).



**Figure 4.6** Multiple relationship with participating entities customer, job order and department, using the binary relationship concept (IE and SSADM).

## Ефективност на концепцията за бинарната релация

Трябва да се отбележи, че както е описано по-горе, нито един от методите не осигурява подходящи графични средства за представяне на идентификаторите на абстрактни „entities“, което намалява ефективността при използване на тази концепция за моделиране на множествени релации. Например, на Фигура 4.6, да предположим, че друг „entity“ проект е бил свързан с бинарна релация към custjob. Как бихме могли да знаем от диаграмата, дали или не съществува съвместен идентификатор на custjob?

## Кардиналност на множествена релация

### Въведение

Досега само най-простиия случай е използван за пример на множествена релация, където единствените кардинални ограничения, които сме прилагали са кардинални съотношения „много към много“ ( $M: N$ ). Как методите се справят с моделиране на множествен релации, където кардиналностите са различни?

За да се обсъди това, избираме два примера, базирани на една и съща множествена релация, които ще покажат обхвата и ограниченията на  $N$ -арната релационна концепция в методите. Варираме кардиналностите на участващите „entities“ и след това показваме моделиране на получените множествени релации.

Разглеждаме следните два примера:

Пример 1. Инстанция на едно „entity“ може да бъде свързано само с една инстанция на други „entity“.

Пример 2. Две „entity“ заедно определят трето „entity“.

Ще използваме представянето на множествената релация, в която участват „entities“ мениджъри, номер на работа и клиент, като наш базов пример. Инстанцията на представянето представлява факта, че един мениджър може да направи презентация за опита свързан с номера на работа по отношение на клиента.

Представянето е комбинация на „entity“, чието име се запазва. Една от причините за това може да се окаже, че това име отличава релациите от други множествени релации, които могат да съществуват, включвайки същите участващи „entity“, като например „номинирани с жалби“ или "качествен контрол".

## Пример 1

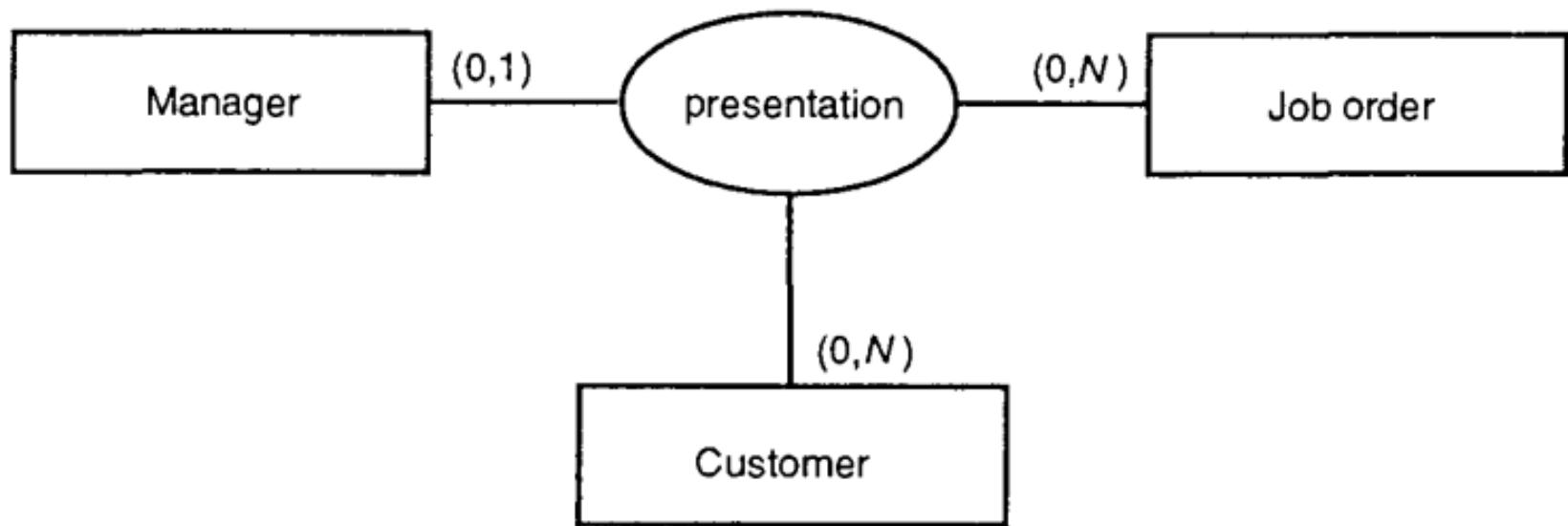
За първия пример, разглеждаме ситуация, когато един мениджър може да направи презентация за само един номер на работа и един клиент. В този случай, има две  $N: 1$  кардинални съотношения (менеджър: номер на работа, менеджър: клиент), така че не всички кардинални съотношения са  $M: N$ .

### MEIN

В MEIN не е възможно да се моделира тази ситуация чрез използването на  $N$ -арна релационна концепция, като кардиналния стил „lookacross“ означава, че за една дадена кардинална стойност до „entity“ не може да знаем кое „entity“ е „looking across“ това „entity“. Вместо това трябва да се използват бинарни релации, както ще опишем за IE и SSADM.

### MERISE

Тъй като MERISE използват участващия стил, те могат да моделират релация чрез  $N$ -арна релационна концепция, както е показано на фигура 4.7.



**Figure 4.7** Multiple relationship presentation using the  $n$ -ary relationship concept. A manager can make a presentation about only one job order concerning only one customer (MERISE).

## IE и SSADM

Бинарните релации във всеки от методите могат да бъдат използвани за моделиране на този пример, и Фигура 4.8 показва как това може да стане в SSADM и IE.

Инстанцията на мениджър е свързана само с една инстанция на комбинираното „entity“ презентация, което от своя страна е свързано само с една инстанция на номер на работа и една инстанция на клиента. Така инстанция на презентация отчита факта, че един мениджър може да направи презентация за един номер на работа касаещ един клиент.

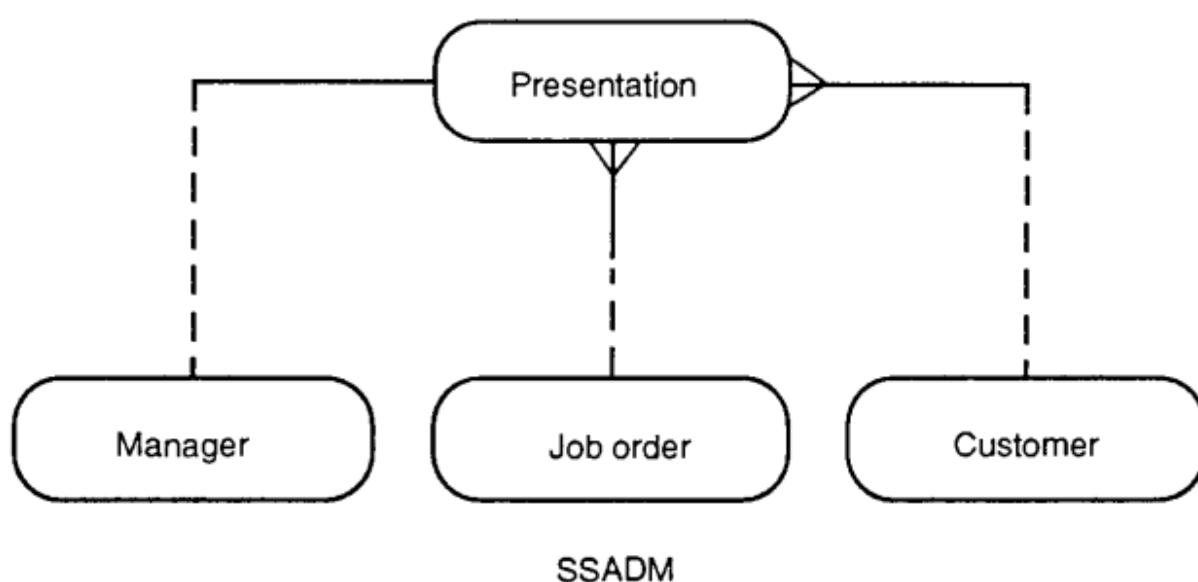
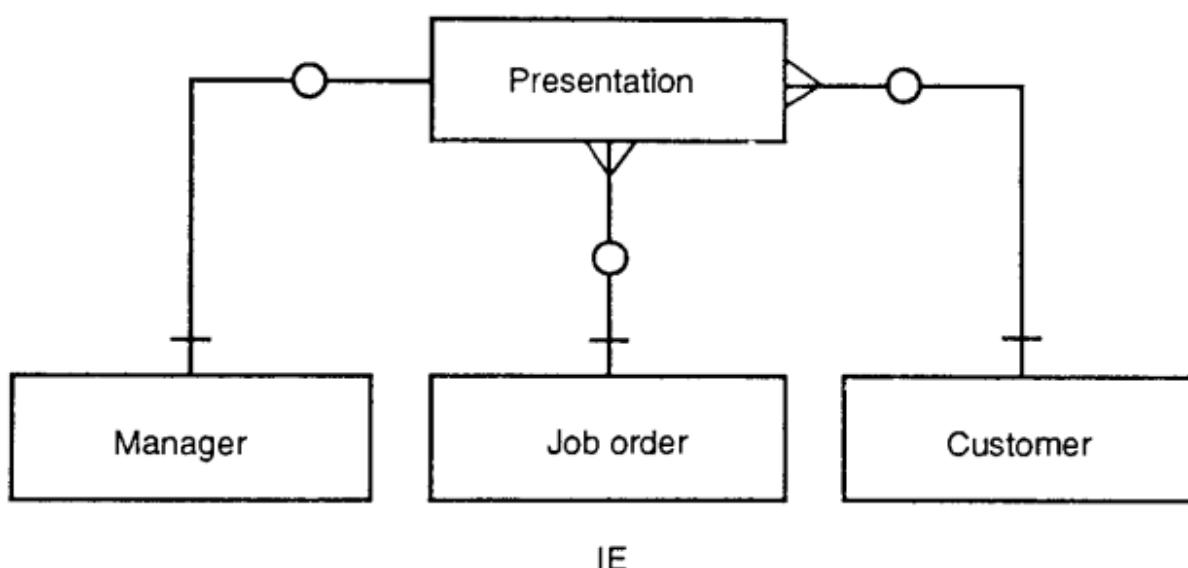
Трябва да използваме комбинация на „entity“ във връзка с участващи „entity“ за правилно моделиране на ситуацията. Ако пропуснем комбинация на „entity“, тогава ще загубим името на презентацията. Този пропуск може да доведе до грешки, особено когато проектираме транзакции, базирани на „entity“ и релации. Такива транзакции не могат, например, да вмъкват инстанции на всички четири „entity“ заедно, както се изисква по смисъла на релацията.

## Пример 2

Във втория случай, разглеждаме ситуация, в която две „entity“ определят трето „entity“. Променяйте кардиналността на примера, така че определен клиент и номер на работа еднозначно определят мениджъра. Така презентация относно опита на определен номер на работа на определен клиент винаги се дава от същия мениджър. В допълнение, кардиналните съотношения на всички участващи „entity“, са  $M:N$ .

## MEIN

Този пример може да се моделира в MEIN, чрез използване на N-арната релационна концепция, както е на фигура 4.9, където значението на кардиналната стойност 1 до мениджър „entity“ е, че съвместна инстанция на всеки клиент и номер на работа еднозначно определя мениджър инстанция.



**Figure 4.8** Multiple relationship presentation using the binary relationship concept. A manager can make a presentation about only one job order concerning only one customer (IE and SSADM).

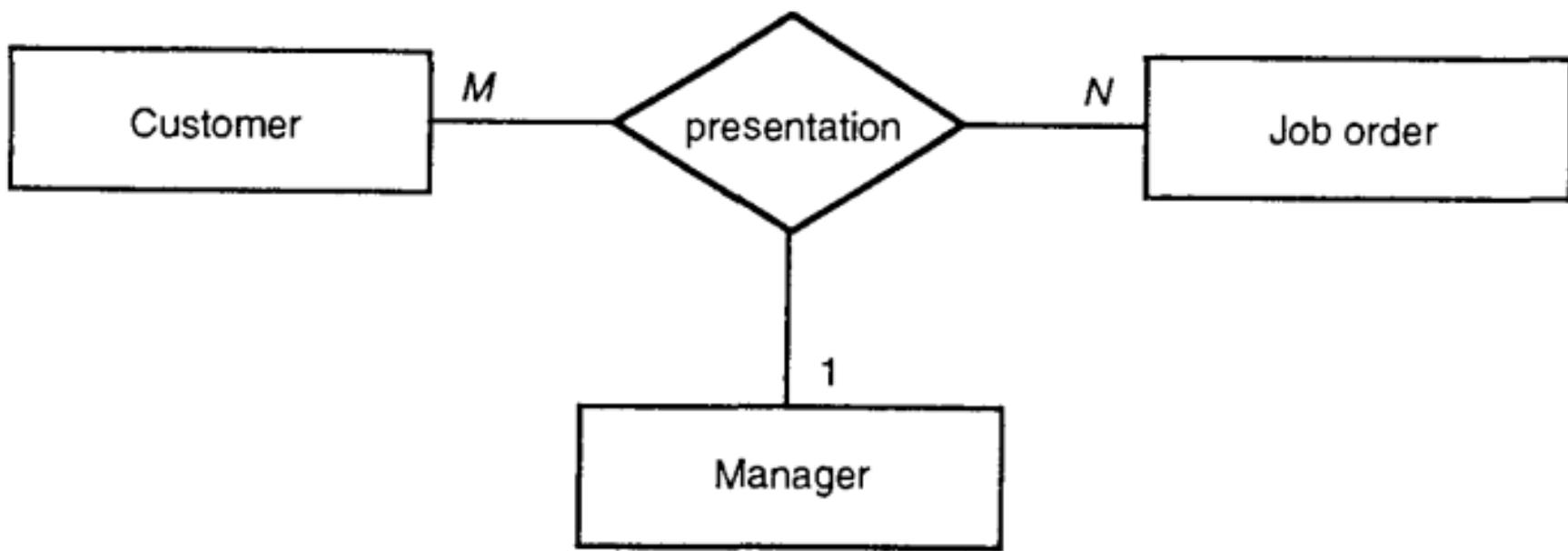
## MERISE

Тъй като MERISE използват стила на участие, те не могат да моделират тази връзка само чрез използване на концепцията на  $N$ -арна релация. MERISE може да използва ограниченията на функционалната свързаност, за да моделира тази връзка.

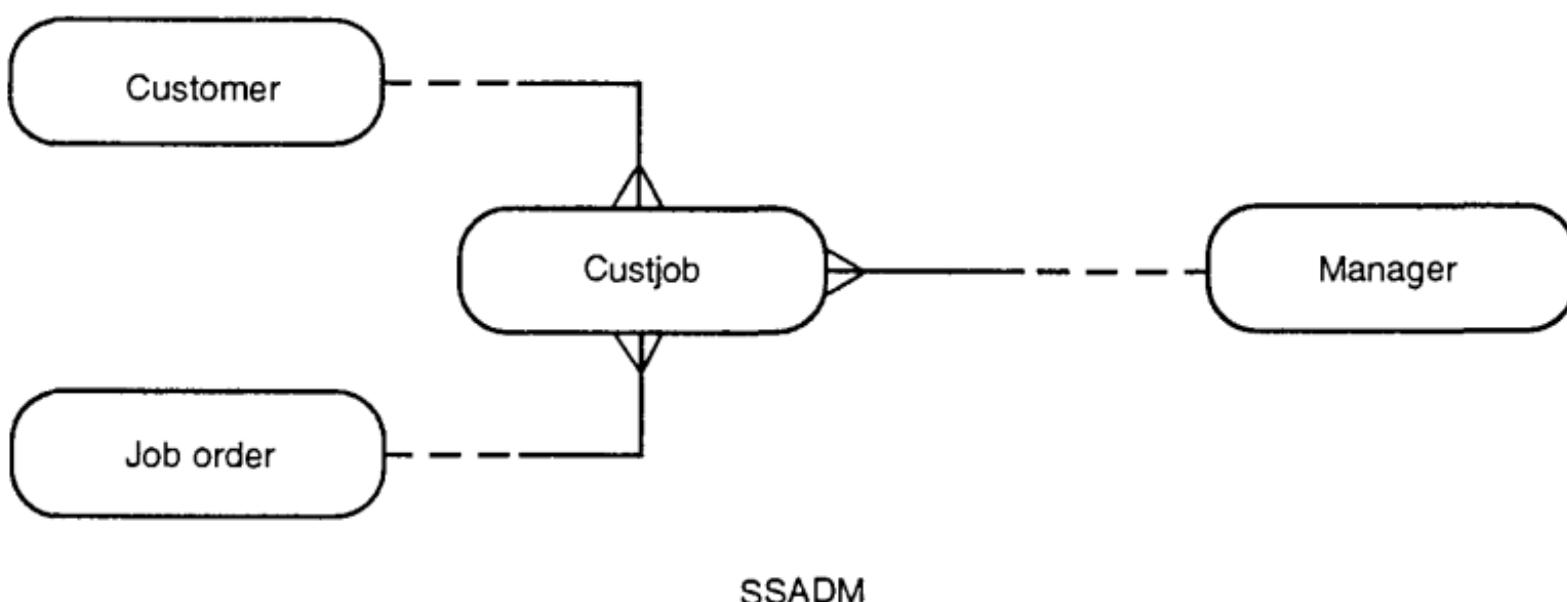
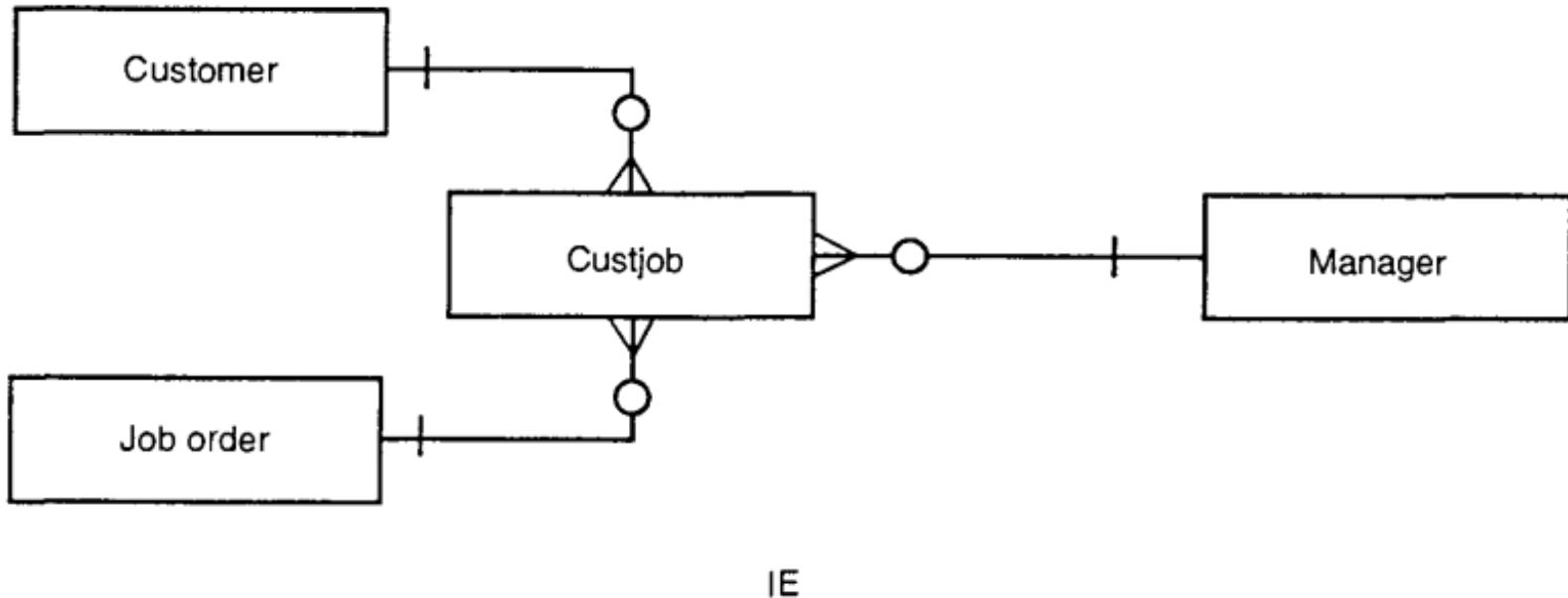
## IE и SSADM

Бинарни релации във всеки от методите могат да се използват за моделиране на този пример, и Фигура 4.10 показва как това може да се направи в SSADM и IE, чрез използване на вложен бинарен подход, където абстрактно „entity custjob“ (също изискващо подходящ идентификатор да специфицира такава съвместна инстанция на клиент и номер на работа уникално идентифицира инстанция на custjob), се дефинира, за да наложи ограничение на мениджър.

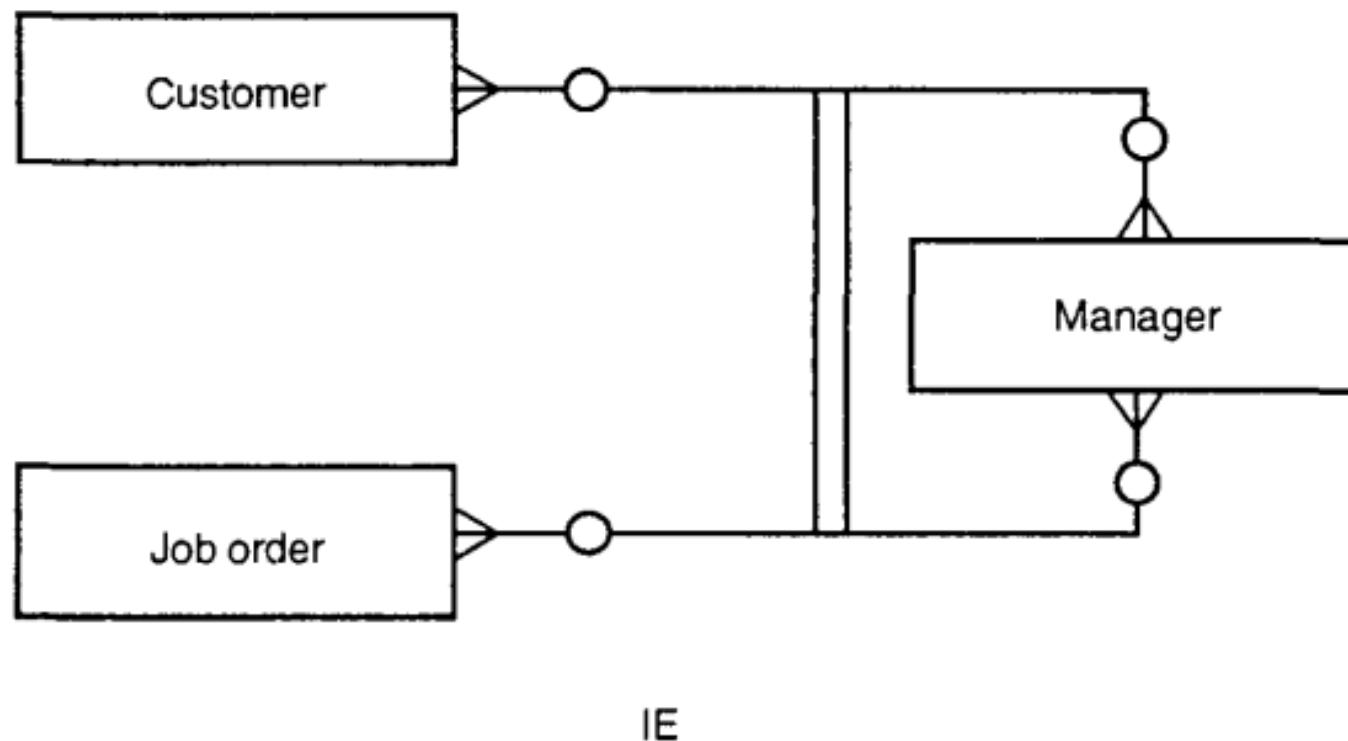
Една ранна версия на IE (Finkelstein, 1989; Martin and Finkelstein, 1981) може да моделира този пример с помощта на типа на асоциация наречена с термина *разширена асоциация*. За този конкретен случай, фигура 4.11 изразява факта, че следните две асоциации са свързани към един мениджър: customer:manager и job order:manager.



**Figure 4.9** Multiple relationship presentation using the  $n$ -ary relationship concept. A customer and a job order uniquely determine a manager (MEIN).



**Figure 4.10** Multiple relationship using the binary relationship concept. A customer and a job order uniquely determine a manager (IE and SSADM).



**Figure 4.11** Extended associations in IE (early version). A customer and a job order relate to the same manager.

## УЧАСТИЕ НА МНОЖЕСТВЕНА РЕЛАЦИЯ

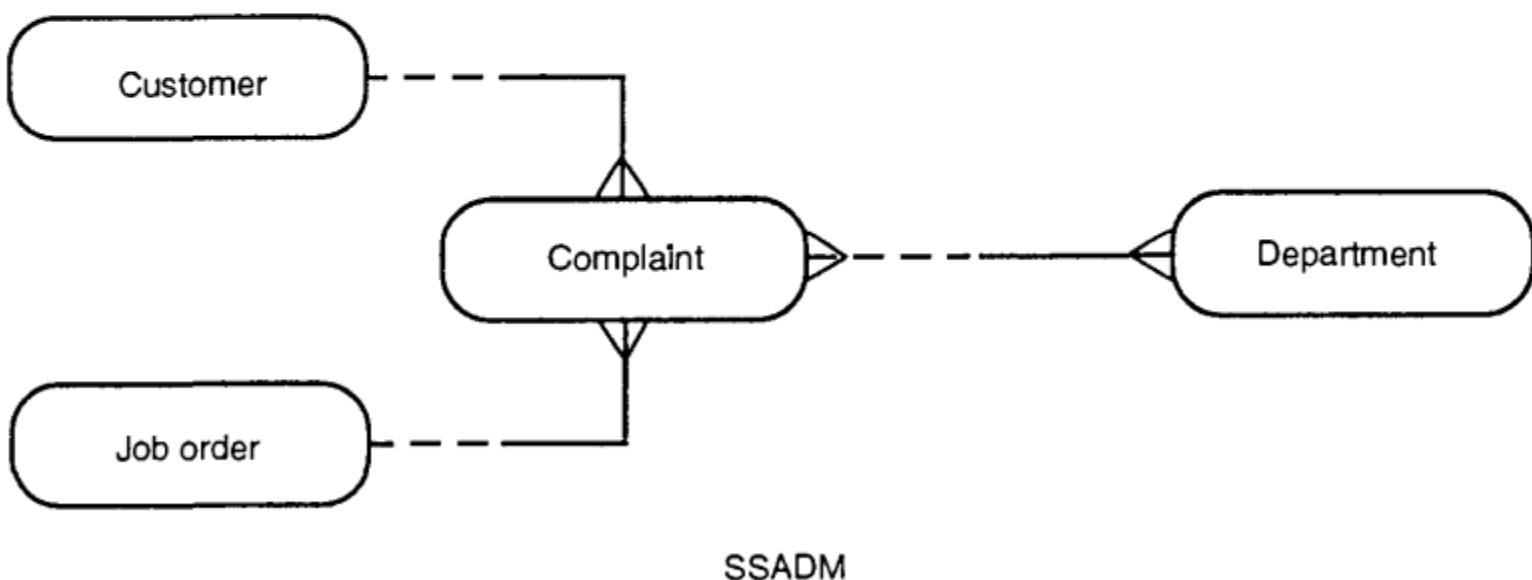
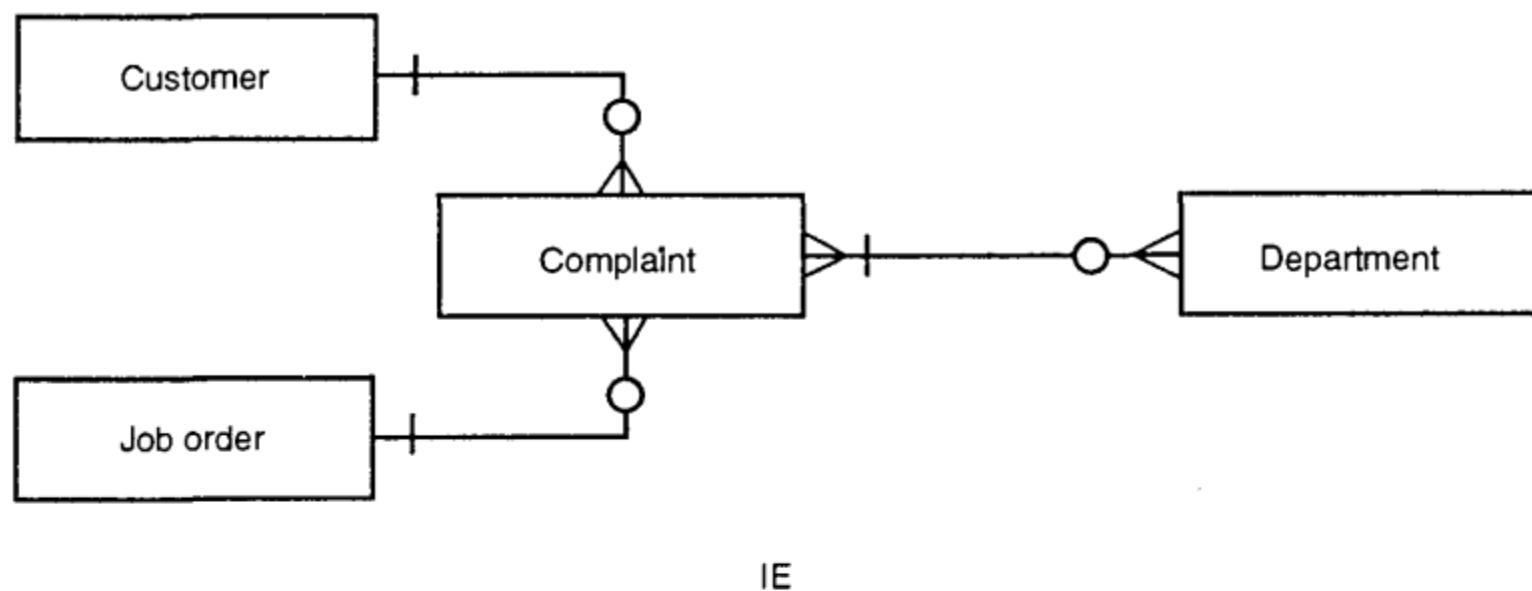
В този пример не променяме кардиналността на участващите „entities“, като вместо това варираме ограниченията за участие. Това е за да се подчертава предположение, направено в предишните примери в тази лекция, че комбинираното „entity“ винаги има задължителна релация с всички участващи „entities“.

Използваме множествената релация на фигура 4.4, относно релацията за жалба, и променяме ограничението за участие, така че, въпреки че жалбата обикновено се подава от един клиент, номера на работа и отдел, отдела е опционален за някои оплаквания.

Тази ситуация може да възникне, когато клиентът не може да се оплаче за номер на работа към конкретен отдел, и не се вземе решение за съответния отдел, докато не измине известно време, след като жалбата е пусната.

## MEIN и MERISE

Тази релация не може да се моделира с помощта на  $N$ -арни релационни концепции в MEIN и в MERISE. Негласно правило в тази концепция е, че инстанция на релацията трябва да бъде свързана към една инстанция на всяко от участващите „entities“.



**Figure 4.12** Multiple relationship using the binary relationship concept. A complaint-related customer and a job order need not be associated with a department (IE and SSADM).

## IE и SSADM

Вложеният бинарен подход, използвайки концепцията на бинарната релация, може да моделира тази ситуация, и това е показано на фигура 4.12, където комбинацията „entity“ оплакване има optionalна релация с отдел.

## РЕЗЮМЕ

### *N*-арна и бинарна релационна концепция

В първия раздел на тази лекция, обсъдихме как MEIN и MERISE могат да моделират множествени релации чрез концепцията на *N*-арните релации. След това обсъдихме как IE и SSADM предоставят само концепцията на бинарната връзка за да моделират множествени релации и сме описали *N*-арни и вложени бинарни подходи използващи това.

*N*-арният подход използва абстрактно „entity“, за да свърже участващите „entity“ в множествена релация, като използва концепцията на бинарната релация. Вложеният бинарен подход използва абстрактно „entity“, което свързва участващите „entity“ с концепцията на бинарната релация, където две „entity“ са вложени едно в друго.

След това изследваме отклоненията от "ясния" случай чрез разглеждане на промени в множествени релации, тъй като те възникват по естествен път в организациите. Обсъдихме две кардинални вариации и едно вариационно участие, всички от които дават някои проблеми за един или повече методи, когато използваме *N*-арната релационна концепция.

Таблица 4.1 показва възможностите на методите за моделиране на различни примери за множествена релация, които сме описали, включително вариации в кардиналността и участието. Неограниченият случай се отнася до примера на фигура 4.1. Кардиналните вариации се отнасят до примерите, в които кардиналното съотношение (1) е различно от *M*: *N*, и (2) две „entities“, заедно определят трето „entities“. Вариационното участие се отнася до примера, когато една инстанция на „entity“ може да съществува, без да бъде свързана с една инстанция на комбинацията на участващите „entity“.

**Table 4.1** Multiple relationship modelling

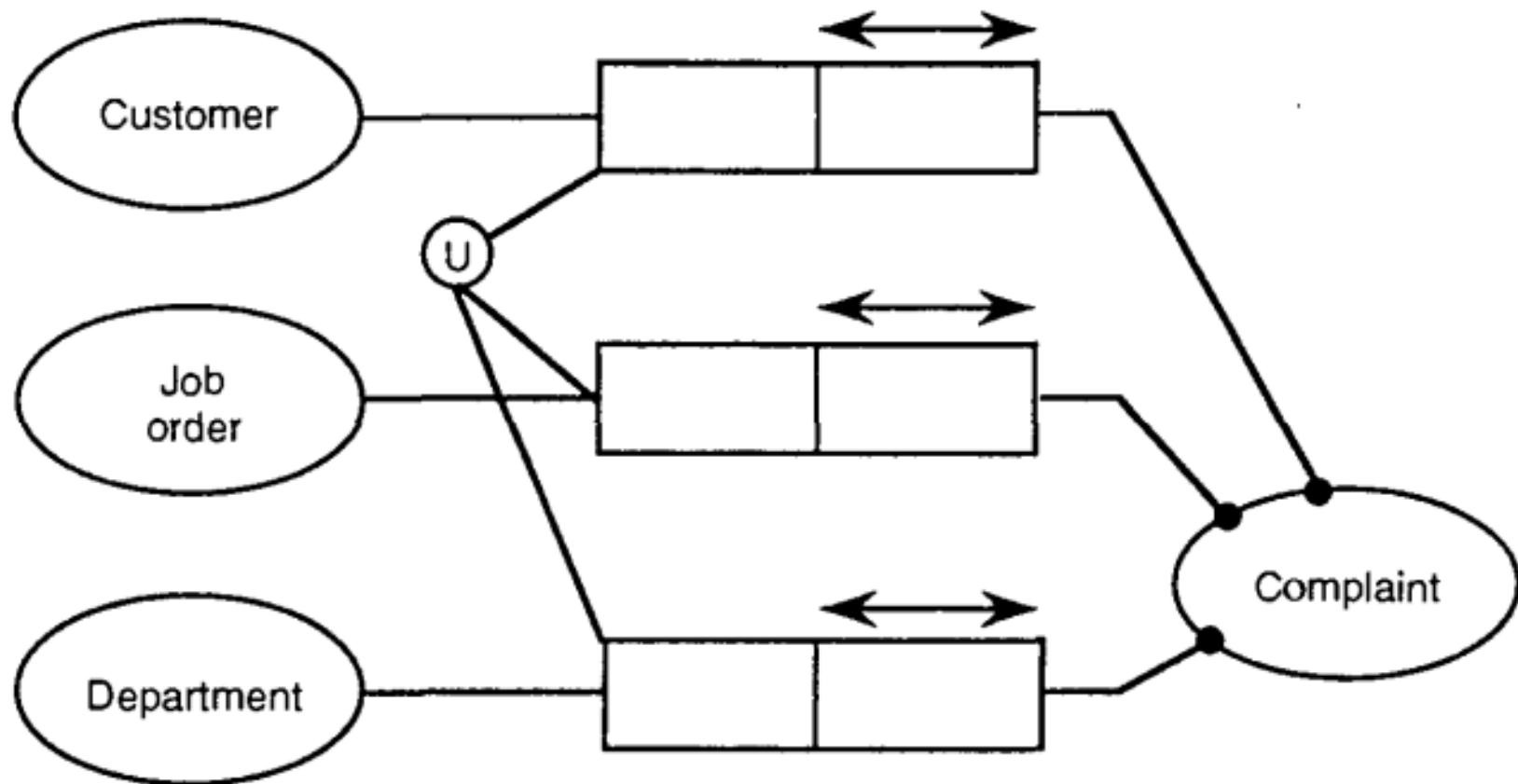
	Method					
	MEIN		MERISE		IE	SSADM
Situation	<i>n</i> -ary relationship concept	binary relationship concept	<i>n</i> -ary relationship concept	binary relationship concept	binary relationship concept	binary relationship concept
Unrestricted	yes	yes	yes	yes	yes	yes
Cardinality variation 1	no	yes	yes	yes	yes	yes
Cardinality variation 2	yes	yes	yes (needs extra constraint)	yes	yes	yes
Participation variation	no	yes	no	yes	yes	yes

## Идентификатори на концепцията на бинарните отношения

Трябва да се отбележи, че N-арните и вложените бинарни подходи при моделирането са силно зависими от коректното дефиниране на идентификатори за абстрактни „entities“.

Нито един от методите не осигурява подходящи графични средства за представяне на това, и по този начин се намалява тяхната ефективност.

NIAM (Nijssen и Halpin, 1989) е пример за метод, който прави това графично и по-ясно чрез уникално ограничение. Пример за това е показан на фигура 4.13, където уникалното ограничение (представено от "U") свързва тези „entities“, които формират съставния идентификатор на комбинираното „entity“. Това съответства на примера на фигура 4.4.



**Figure 4.13** Multiple relationship complaint using the binary relationship concept and illustrating the use of the uniqueness constraint (NIAM).

## Моделиране на насоки и критерии за качество

Понятията, разглеждани в тази лекция, повдигнаха проблема за избора на начина на моделиране, като метод може да предостави повече от една концепция за моделиране на дадена ситуация. При тези обстоятелства е необходимо ориентиране при моделирането, не е достатъчно само да се избере правилната концепция за коректно моделиране на ситуацията, но трябва и да се направи правилния избор между концепции въз основа на *критерии за качество*. Това са критерии, които могат да бъдат приложени, за да се постигне изграждане на информационен модел с високо качество.

Например, критерият за *естествеността* е, че концепциите на метода трябва да съответстват на концепциите в потребителската организация, и това може да се прилага за да се избере на  $N$ -арна релация за моделиране на неограничен случай на множествени релации, въз основа, на които  $N$ -арните отношения съответстват на множествени отношения по-директно от няколко бинарни релации.

## ЗАКЛЮЧЕНИЕ

Референтни рамкови концепции

Концепцията за  $N$ -арната релация присъства в само два от методите MERISE и MEIN.

Всички методи предоставят концепцията на бинарната връзка, която може да моделира всички представени множествени релационни вариации.

Въпреки това, бинарната релация е недостатъчна като графично представяне тъй като идентификаторите не са показани, а те са от решаващо значение за моделиране на множествени релации. Това трябва да се подобри в методите.

Необходими са указания, тъй като концепцията на  $N$ -арните отношения в някои методи не може да моделира някои видове множествени релации.

## Насоки при моделирането

Концепцията на  $N$ -арната релация е естествен подход за моделиране на множествени релации, при които кардиналността и ограниченията за участие са „изяснени“.

Когато множествени релации не са „ясни“ и включват промени в кардиналността и ограничения за участие на учащи „entities“, тогава не можем да ги моделираме с даден метод чрез концепцията на  $N$ -арната релация.

Решението относно използването на даден бинарен релационен подход може да зависи от стойностите на ограниченията за участие. Ако „entities“ трябва да бъдат свързани заедно в един и същ момент време може два подхода да са подходящи. Въпреки това, ако съществува възможност за установяване на свързване на едно „entity“ на по-късен етап, вложеният бинарен подход е единственият избор.

Таблици 4.2 и 4.3 показват основните предимства и недостатъци при използване на бинарната или  $N$ -арната концепции за множествено релационно моделиране.

**Table 4.2** Advantages and disadvantages in modelling multiple relationships with the binary relationship concept

Advantages	Disadvantages
<ul style="list-style-type: none"><li>• Can model all multiple relationship examples discussed</li></ul>	<ul style="list-style-type: none"><li>• Choice of using nested or <math>n</math>-ary approach, therefore multiple relationship modelling not natural</li><li>• Model may be overcomplex</li><li>• Have to use an abstract entity</li><li>• Identifier of abstract entity has no graphical representation</li></ul>

**Table 4.3** Advantages and disadvantages in modelling multiple relationships with the  $n$ -ary relationship concept

Advantages	Disadvantages
<ul style="list-style-type: none"><li>• Multiple situation modelling is natural</li><li>• No need to create abstract entity</li></ul>	<ul style="list-style-type: none"><li>• Some methods cannot model all cardinality and participation variations shown</li></ul>

# ЛЕКЦИЯ 5

## СИСТЕМЕН АНАЛИЗ, КОНСТРУИРАНЕ НА МОДЕЛ И СИМУЛАЦИЯ

-  **Конструиране на модел**
-  **Поведение на модела**
-  **Моделиране на технически  
системи**
-  **Пример: Поддръжка осигурена от  
средата за моделиране Simplex3**

# **ВЪВЕДЕНИЕ**

**Симулацията е подниво на голямо  
интердисциплинарно поле, което също  
предизвиква системно развитие на модела.**

**Изучаването на модели чрез симулации е  
водещо за *естествените науки, бизнес и  
икономика, биология и медицина.***

**Затова симулацията е основна, на  
интердисциплинарно ниво и не е ясно  
очертана ограничена област.**

# ВЪВЕДЕНИЕ

Въпреки широкото разнообразие на приложения, анализа на модела винаги следва образца на нивото.

Този образец формира методологичните основи при конструирането на модел.

# Конструиране на модел

Моделиращият инициализира изгледите на света като многоцветно множество от форми.

От разноцветното множество в реалния свят човешкото съзнание идентифицира част от изучаваното. Тази област се нарича **реална система**. Частта непринадлежаща на реалната система е останалата среда.

Разделянето на една част от света от останалата, с която той има взаимодействия, е резултат, проявяващ се при конструирането на модел.

Често конструкцията на модела е възможна само при разумни ясни граници и включването/изключването на определени елементи в/или от системата.

# Конструиране на модел

Примери:

В модел, описващ динамиката на популацията на хищник и жертва, е необходимо да дефинираме границите на системата по такъв начин, че възможно най-малко животни да ги нарушават и външното влияние върху популацията да е минимално.

Ако е необходимо, системата би трябвало да се разшири, за да обхване всички придвижвания на животните.

Втора възможност е да се ограничи времетраенето на изучаването на модела така, че да не са възможни премествания. Например, ако една популация през лятото е в ограничена област, тя ще се изучава само тогава.

# Конструиране на модел

Пример:

При изучаване на слънчевата система е прието слънцето и всичките му планети да се разглеждат като една система.

Въпреки, че съществуват външни влияния върху слънчевата система, те са незначителни и могат да се игнорират.

Не е удачно да се ограничи системата само от слънцето и най-близките му две планети.

Външните планети ще имат толкова силно влияние върху тази система, че не е невъзможно да се направи използваем модел.

# Конструиране на модел

Ако взаимодействията между дадена система и нейната околнна среда са силни и разделяне не е възможно, следващите по големина системи трябва да се обмислят.

Например, в даден регион не е възможно да се изучава само агрокултурата, нужно е да се изследват и взаимодействията с други сектори: работници, пазара на работна сила, конкуренцията за земята. Само с разглеждането на всички важни фактори в региона може да се получи използваем модел.

# Конструиране на модел

Първоначално нищо не може да се каже относно поведението на реалната система.

Реалната система се държи като затворена черна кутия, чиито вътрешни механизми на работа са скрити от погледа.

Входната информация към реалната система я кара да премине към ново вътрешно състояние. Това ново вътрешно състояние може да се наблюдава като се следи изходната информация.

# Конструиране на модел

Примери:

- Разглеждаме атом. Ядрото се изучава чрез бомбардирането му с алфа-частици. В резултат на този вход атомното ядро преминава в ново състояние и реагира, отдавайки частици от гама-фона;
- Разглеждаме национална икономика. При понижаване на размера на лихвите след даден период от време се наблюдава като резултат увеличение на икономическия растеж.

# Конструиране на модел

Цялата информация, опит и измервания, които могат да се получат от реалната система, формират наблюдаваните данни. Това е опитно установеното поведение на реалната система. То се представя под формата на If-Then състояния или Input-Output описания.

# Конструиране на модел

Системният анализ се опитва да развие концептуалния модел като се базира на наблюдаваните данни. Концептуалният модел съдържа изображение в човешкото съзнание за същността на реален концептуален модел. Концептуалният модел е мислен образ на реалната система.

Конструкцията на концептуалния модел се състои от дефинирането на компонентите на модела, дефинирането на структурата на модела и заместването на външното влияние.

# Конструиране на модел

- *Дефиниция на компонентите на модела.*

Първо трябва да се реши, кои обекти ще се представят в модела и как те могат да бъдат описани. Компонент на модела се състои от име, атрибути/свойства и поведение. Динамичното поведение описва как атрибутите се променят във времето.

- *Дефиниция на структура на модела.*

Описва взаимовръзките и взаимното влияние между дефинираните в модела обекти.

- *Представяния на външни влияния.*

Концептуалният модел е самосъдържаща се (самоописателна) единица. Реалните системи от друга страна са предмет на влияния от околната им среда. За външни въздействия трябва да се намери заместващо представяне.

# Конструиране на модел

Тези три стъпки използват абстракция и идеализиране.

Абстракция означава, че не всички атрибути от реалната система са представени в концептуалния модел. За тези обекти от реалната система, за които моделирация счита, че не са важни, се оставят извън модела.

Идеализация означава, че реалните условия са маркирани върху идеални обекти.

## Поведение на модела

Знанието за поведението на концептуалния модел може да се установи по два съществено различни начина. Тези два начина са известни като аналитични методи и реално моделиране (симулация).

Когато концептуалния модел е описан чрез формален език е възможно да се получат нови знания за поведението на концептуалния модел чрез използването на правила за извличане, дефинирани чрез този език.

# Поведение на модела

Пример:

Непрекъснатите от гледна точка на време концептуални модели, които са специфицирани по скорост на промяна, се представят чрез диференциални уравнения. В някои случаи математически може да се получи точното аналитично решение на тези диференциални уравнения.

Има алтернативен начин за добиване на резултати относно поведението на концептуалния модел. Можем да опитаме да разработим реална система, чието поведение да се опише чрез концептуален модел. Поведението на концептуалния модел може да бъде симулирано от тази реална система. Реалните системи, които са използвани по този начин се наричат симулационни модели.

# Поведение на модела

Експерименти със симулационен модел могат да дадат само индивидуални решения.

Резултатите, които описват поведението на концептуалния модел могат да се получат аналитично или да се установят чрез използване на симулационен модел. За тези концептуални модели, за които съществува и аналитичното решение и симулационния модел, могат да се сравняват резултатите получени по двата начина и за всеки отделен случай те могат да се сравняват. Резултатите от тях не би трябвало да са идентични. Най-голямата стойност на разликата им би дала оценка на грешката, която се получава когато се изгради симулационния модел и когато се извърши експеримента.

## Поведение на модела

От особена важност е доказателството, че даден концептуален модел потвърждава представянето на поведението на системата, която се изучава. Това е познато като моделиране.

Директно сравнение на реалната система и модел не е възможно. Валидността се базира единствено на сравнение на данните получавани от системата и тези получавани от модела.

# Поведение на модела

За перфектно съответствие между системните данни и данните на модела е особено важно да се има в предвид, че:

- Измерванията върху реалната система са обект на грешки. Разликата между полученото от модела и експериментално получените стойности може да се измери.
- Концептуалният модел се получава от реалната система чрез абстракция и идеализъм и съдържа далеч по-малко променливи и затова поведението му е различно.
- Ако данните на модела са получени чрез симулационен модел ще се появят грешки при конструирането и експериментирането на модела. Тези грешки включват грешки от закръгляването, възникнали при представянето на числа с плаваща точка в компютъра и грешки направени при числено решаване на диференциални уравнения.

# Поведение на модела

Реалната система и модела могат да си съответстват само при наличен толеранс.

В случаи, когато поведението на концептуалния модел показва това със задоволителна точност, данните от модела могат да се отнесат към реалната система.

Най-важните констатации са могат да се резюмират:

- Изучаването на реалната система не е директно наблюдавано чрез човешкото съзнание, а е под формата на начални неструктурирани данни.
- Концептуалният модел е представяне на системата, което е резултат на човешките разсъждения. Концептуалният модел е представата за реалната система изградена мислено.

# Моделиране на технически системи

В науката се залага ограничена част от реалността, наречена реална система. Разработва се концептуалния модел на тази реална система, който би трябвало да обясни или предскаже поведението на реалната система.

В инженерството често се срещаме с различни ситуации. Тук реалната система все още не съществува – тя ще се проектира или разработи и тогава ще се изгради. Въпреки това общите принципи, основни за научното изучаване на модели, могат да се разширят, за да се прилагат и за технически системи.

При планирането и проектирането на техническите системи първата задача е да се определят изискванията, които системата трябва да удовлетвори. Това означава първоначално да се дефинират: желаното изпълнение, цената и ограничаващите условия. Стъпката, която води до тази спецификация се нарича описание на изискванията.

# Моделиране на технически системи

Конструкционният план се разработва базирайки се на изискванията. Това трябва да бъде достатъчно детайлно, за да позволи техническата система да се изгради недвусмислено.

Проектният план може да се използва за конструиране на техническа система. След като системата се изгради и е готова за стартиране, могат да се направят измервания и тогава да се сравнят с изискванията. Ако съответствието между актуалните системни данни и изисквания е незадоволително, проектът и реалната система трябва да се модифицират.

# Моделиране на технически системи

За да се избегне скъпата корекционна стъпка е препоръчително първо да се провери изпълнението на проекта, използващ модела.

На тази стъпка трябва да се отбележи, че проектът обикновено не може да служи като основа на модела, тъй като моделът съдържа опростеност като резултат от абстракциите и идеализирането.

Данните, получени от модела могат да се сравнят с изискванията.

# Пример: Поддръжка осигурена от средата за моделиране Simplex3

Симулационната система *Simplex3* поддържа за потребителя следните равнища:

- *Концептуален модел*

Описателният език на модела *Simplex MDL* позволява представяне на концептуалния модел.

- *Конструкция на модела*

Започвайки от описанietо на концептуалния модел се генерира симулационен модел. Това се случва с компилационен процес, в който описанietо на модела *Simplex MDL* се конвертира в изпълнима програма.

- *Симулационен модел*

На разположение е библиотека на модела, която управлява компонентите на модела и техните различни версии.

- *Експериментиране с модела*

С модела могат да се извършат различни експерименти.

- *Данни на модела*

Резултатите от експериментите на модела могат да се анализират, подгответ за визуализиране и архивират.

# ЛЕКЦИЯ 6

## КОНСТРУКЦИЯ НА МОДЕЛА

- ❑ Моделът Biotope\_1
- ❑ Експериментиране с моделът Biotope\_1
- ❑ Версия и състояние на модела

# **ВЪВЕДЕНИЕ**

**Работата със симулационни модели може в общност да се раздели в следните три основни области:**

- Конструиране на модела;**
- Експериментиране с модела;**
- Обработка на резултатите.**

# Моделът Biotope\_1

Ще разгледаме описаннието на модела Biotope\_1 в симулационната система Simplex3.

Моделът изучава отношенията между популацията на зайци (жертвии) и лисици (хищници). Наблюденията показват, че популациите на зайците и на лисиците варират.

Когато зайците се увеличават бързо, наличната храна за лисиците расте, така че след известно време техният брой също нараства. Обратно, намаляване популацията на зайците води до намаляване на броя на лисиците. Малък брой на лисиците означава добри условия на живееене за зайците.

# Моделът Biotope\_1

Раждаемостта на зайците се дължи на естествените превишавания на смъртността, тогава, на отсъствието на лисици популацията на зайците ще нарасне, според следното диференциално уравнение:

$$\text{Hare}' = a * \text{Hare}$$

Това означава, че броят на допълнителните зайци за единица време е пропорционален на текущия брой.

В допълнение на естествената смъртност, размера на смъртността на зайците зависи от срещите между зайци и лисици.

$c = (\text{Брой срещи}) / (1 \text{ единица време} (\text{ЕВ}) * 1 \text{ Заек} * 1 \text{ Лисица})$   
Броят на зайците и лисиците за една единица време (ЕВ) е “ $c$ ”.

# Моделът Biotope\_1

Следва, че:

За даден брой зайци и лисици, общия брой срещи за единица време е даден чрез произведението с \* Hare \* Fox.

Така наречените брой жертви BFacH и BFacF определя как броя на зайците и лисиците се променя като резултат от срещите. Ако е прието, че един заек е изяден, когато протече среща, тогава задаваме FacH = 1. Ако само един заек е изяден на всеки две срещи, тогава ще зададем BFacH = 0.5. Ако на лисицата е нужно да изяде четири заека, за да оцелее, тогава фактора на жертвата BFacF ще има стойност 0.25.

BFacH \* с \* Hare \* Fox е броя на зайците, които са намалели от популацията за единица време като резултат от изяддането им от лисиците.

# Моделът Biotope\_1

Подобно, факторът BFacF \* с \* Hare \* Fox дава увеличение на броя лисици като резултат от срещите.

$$\text{Hare}' = a * \text{Hare} - \text{BFacH} * c * \text{Hare} * \text{Fox}$$

$$\text{Fox}' = -b * \text{Fox} + \text{BFacF} * c * \text{Hare} * \text{Fox}$$

Ако приемем, че на всяка среща броя на зайците намалява с един, например  $\text{BFacH} = 1$ . Ако десет заека са нужни, за да поддържат една лисица жива, то  $\text{BFacF} = 0.1$ .

Приемайки  $\text{BFacH} = \text{BFacF}$  се получава форма на уравнения, познати като уравнения на Lotka – Volterra:

# Модельт Biotope\_1

$$\text{Hare}' = a * \text{Hare} - d * \text{Hare} * \text{Fox}$$

$$\text{Fox}' = -b * \text{Fox} + d * \text{Hare} * \text{Fox}$$

Мерните единици са много важни при  
конструкцията на модела.

# Моделът Biotope\_1

```
1  BASIC COMPONENT Biotope_1

2  USE OF UNITS
3      UNIT[NumH] = BASIS
4      UNIT[NumF] = BASIS
5      TIMEUNIT = [a]

6  DECLARATION OF ELEMENTS

7  CONSTANTS
8      a (REAL[1/a]) := 1.75 [1/a],
9      b (REAL[1/a]) := 1.25 [1/a],
10     c (REAL[1/(NumH*NumF*a)])
11        := 0.0375 [1/(NumH*NumF*a)],
12     BFacH(REAL[NumH]) := 1.0 [NumH],
13     BFacF(REAL[NumF]) := 0.1 [NumF]

14 STATE VARIABLES
15 CONTINUOUS
16 Hare (REAL[NumH]) := 400 [NumH],
17 Fox (REAL[NumF]) := 37 [NumF]

18 DYNAMIC BEHAVIOUR
19 DIFFERENTIAL EQUATIONS
20 Hare' := a*Hare - BFacH*c*Hare*Fox;
21 Fox' := -b*Fox + BFacF*c*Hare*Fox;
22 END

22 END OF Biotope_1
```

Моделът Biotope\_1 в средата Simplex3

# Моделът Biotope\_1

В моделът Biotope\_1, са представени две нови основни единици за броя на зайци и лисици. В описанието на този модел, времето Т е прието да бъде в единици години.

Константи са:

$$a=1.75 \quad [1/a]$$

$$b=1.25 \quad [1/a]$$

$$c=0.0375 \quad [1(NumH * NumF * a)]$$

$$BFacH = 1.0 \quad [NumH]$$

$$BFacF = 0.1 \quad [NumF]$$

Начални условия са:

$$Hare(0) = 400 \quad [NumH]$$

$$Fox(0) = 37 \quad [NumF]$$

# Експериментиране с моделът Biotope\_1

Моделът Biotope\_1 показва спецификацията на модела, който е конструиран.

Моделът е много прост – състои се само от една компонента.

# Експериментиране с моделът Biotope\_1

За да е възможно да се работи с компонента в нейната текуща версия, компонентата трябва да е присъединена към модел.

# Експериментиране с моделът Biotope\_1

Създаване на експерименти и изпълнения.

Сега могат да се извършат различни експерименти с моделите в каталога Models.

Избира се каталога Experiments и се избира команда New experiment... Ако е необходимо експериментът, може да се преименува.

Един експеримент обикновено се състои от различни симулационни изпълнения. Всяко изпълнение може да се свърже със свой собствен модел от каталога Models. По този начин е възможна работата с различни модели по време на един експеримент и директно да се сравнят резултатите от всеки.

# Експериментиране с моделът Biotope\_1

Може да се види, че изпълнението Run1 съдържа прекъсвания, контролни параметри, наблюдения, симулационни резултати и протоколи.

Симулационните резултати към тази точка на прекъсване може да се анализират и покажат. Могат да се направят промени на променливите на модела.

Показва вътрешната структура на изпълнение с прекъсвания.

Симулационното изпълнение започва при  $T = 0$  с Break0. Симулацията в случая продължава до  $T = 10$ . Всякакви промени на един или повече моделни променливи се запазват във файла за промени Chg. Основавайки се на състоянието описано чрез End и на промените, съхранявани в Chg, е изчислено новото начално състояние, с което започва следващия сегмент на изпълнение.

# Експериментиране с моделът **Biotope\_1**

Контролните параметри контролират как протича изпълнението на симулацията. Те могат да се видят и модифицират в каталога Control parameters.

Наблюдателите предлагат възможността за записване стойностите на избраните променливи на модел по време на изпълнение на симулация. Те могат да бъдат анализирани и показани по-късно.

# Експериментиране с моделът Biotope\_1

Забележки:

Контролните параметри и наблюдатели са валидни за цялата продължителност на изпълнение и те могат да бъдат създадени или модифицирани само в началото на изпълнението – Break0.

Създадени са два различни наблюдателя, наречени Obs1 и Obs2. Тези наблюдатели следят променливите на състояния Hare, Fox, и Hare', Fox'.

Видът на наблюдателя може да се дефинира, например Complete timeseries за записване на пълни динамични редове.

# Експериментиране с моделът Biotope\_1

Специфицират се следните атрибути:

- *Име на наблюдателя*

Името на наблюдателя може да се избере свободно.

- *Край*

Посочват се времената за започване и край на записвания интервал.

- *Стъпка*

При малки размери на стъпката се забавя симулацията. При големи размери на стъпката се достига до грубо и неточно представяне на резултатите. В примера размера на стъпката е 0.1.

# Експериментиране с моделът Biotope\_1

Наблюдателят Obs2 се създава по подобен на Obs1 начин със същите атрибути за Hare' и Fox'.

Когато в каталога Observer е избран прозорецът на съдържанието се показва всичката информация съответна на двета наблюдателя.

# Експериментиране с моделът Biotope\_1

Забележка:

Ако не се създават никакви наблюдатели не се записват данни и накрая е налично само крайното състояние End. В този случай поведението на променливите, зависещо от времето, не може да се визуализира.

# **Експериментиране с моделът Biotope\_1**

Каталога **Simulation results** показва  
динамични редове записани от  
наблюдателя.

# Експериментиране с моделът Biotope\_1

Стандартно графично на симулационните резултати може да се получи чрез избирането на желаните динамични редове и натискане на бутона за показване в лентата с инструменти.

Получаваме първите графични резултати чрез избиране на динамичните редове Obs1 # Biotope\_1 / Hare и Obs1 # Biotope\_1 / Fox.

# Експериментиране с моделът Biotope\_1

Забележка:

Всеки симулационен резултат може да се избере и могат да се правят сравнения на изпълненията.

# Експериментиране с моделът Biotope\_1

В примера динамичните редове се представят чрез линейна диаграма и таблица.

Тя показва стойностите на променливите Hare и Fox на всички времеви точки (времевата стъпка на наблюдателя е 0.1).

# Експериментиране с моделът Biotope\_1

Забележка:

Можете да са отворени едновременно, много резултатни прозорци.

Ако е избран само един симулационен резултат, могат да се употребят разнообразни математически методи за анализ. В примерът се разглеждат динамичните редове Obs1 # Biotope\_1 / Fox.

Когато е избран метода за анализ Fast Fourier Analysis, се появяват входни полета, с които методите могат да се параметризират.

# Експериментиране с моделът Biotope\_1

Параметризирани модели.

Симулационните резултати могат да се покажат и анализират на всяка точка на прекъсване. Допълнително стойностите на моделните променливи могат да бъдат модифицирани.

В експериментът разгledан тук има прекъсване Break1 и време  $T = 10$ . Директорията Break1 начално съдържа само крайното състояние End.

След Break1 е избрана и команда Model parameters е била осъществена се появява прозорец. Лявата част на прозореца показва пълен списък на променливите, използвани в този модел.

# Експериментиране с моделът Biotope\_1

Стойностите на променливите могат да се променят. Резултатния прозорец, съдържа входни полета за желаната стойност на променливата.

# Експериментиране с моделът Biotope\_1

Например, промяна стойността на променливата Hare от текущото 353.311048 на 444. След потвърждаването на измененията всички промени са съхранени във файл на промените Chg на прекъсването Break1. Резултатът може да бъде прегледан чрез избиране Chg в прозореца на съдържанията.

Изпълнението на симулацията може да бъде продължено.

Когато продължението на изпълнението на симулацията е завършено, Run1 съдържа ново прекъсване с ново крайно състояние End, наречено Break2.

# Експериментиране с моделът Biotope\_1

Определението на езика на модела SIMPLEX MDL поддържа четири типа компоненти:

- Основни компоненти.

Тези компоненти могат да се изпълняват независимо или да бъдат част от йерархичния модел. Моделът Biotope\_1 показва цялостна основна компонента.

- Компоненти от високо равнище.

Simplex3 осигурява йерархично моделиране. Компонентите от високо равнище съдържат структурно описание, когато се описват връзките между второстепенните компоненти.

- Подвижни компоненти.

Подвижните компоненти представлят обекти, които могат да се местят от едно на друго място.

- Функционални компоненти.

За да улесни яснотата и структурата на спецификацията на модела, функциите и подфункциите могат да се дефинират.

# Експериментиране с моделът Biotope\_1

Прекъсвания.

Изпълненията на програмата могат да бъдат прекъснати произволен брой пъти. В началото на изпълнението съществува само Break0. Това начало се състои само от крайното състояние, което има своя стойност по подразбиране.

Възможно е да изменим състоянието End.

# Експериментиране с моделът Biotope\_1

Командите Show and state, Show change и Show initial state могат да се използват за получаване на кореспондираща информация.

Сега може да се стартира изпълнението на симулацията. Когато изпълнението на симулацията завърши в подходящата точка от време, е достигнато прекъсване Break1. Това първоначално съдържа състоянието за край.

По-нататъшни промени на Break1 генерират нов файл на промените. Състоянието за край End и файла на промените Chg са използвани за генериране за новото начално състояние, с което може да започне нов сегмент на изпълнение.

# Експериментиране с моделът Biotope\_1

Ако не са направени промени на Break, оригиналното състояние за край ще се използва като начално състояние за следващ сегмент на изпълнение.

Забележка:

Break се състои от крайно състояние на предишно изпълнение на сегмент и промените и новите начални състояния за следващия изпълним сегмент.

Основавайки се на състоянието за край и промените се генерира ново състояние на следващия сегмент на изпълнение.

Всяко прекъсване може да се направи стартираща точка на ново изпълнение.

# Експериментиране с моделът Biotope\_1

Break1 може да се избере от няколко  
възможни прекъсвания и да се използва  
като Break0 в ново изпълнение Run2.

# Експериментиране с моделът Biotope\_1

Тази функционалност е особено важна, когато се изучават няколко различни набора от параметри.

# Версия и състояние на модела

Описанието на всяка версия на компонента на модела се извършва от SIMPLEX MDL.

Версия на компонента се компилира на две фази в обектния код. Получава се изпълним симулационен модел.

# Версия и състояние на модела

Процеса на компилиране.

Описанието на модела в SIMPLEX MDL се превежда в C-код от SIMPLEX MDL компилатор, след което се превежда в обектен код от C-компилатор.

# Версия и състояние на модела

Една версия може да е в разнообразни състояния, което зависи от позицията ѝ при компилационния процес.

Възможни са преминавания между състоянията на версията. Те или се стартират от потребителя или се изпълняват автоматично чрез симулационната система.

- Непроверена.

Версията е в това състояние веднага след създаването.

Допълнително това състояние е присъединено от симулационната система към версииите, чийто MDL-код токущо е бил изменен. Версията се представя само в SIMPLEX MDL форма.

- Проверена.

На тази стъпка, компилацията чрез MDL-компилаторът е завършила успешно. Моделът се представя като C-код.

- Подготвена.

На тази стъпка, C-кода е бил преведен чрез C-компилатор в обектен код.

# Версия и състояние на модела

Забележка:

Състоянието SyntaxOK е налице само при юерархичните модели. То е необходимо, защото една компонента може да съдържа подкомпоненти или самата тя да е подкомпонент на компонента от по-високо равнище.

# Версия и състояние на модела

Компилацията се стартира от потребителя чрез избиране на желаната версия на компонентата и избиране на подходяща команда.

Командата `Check version` превежда MDL описанието в С-код, чрез това преминаването на компонентата от състояние Непроверен в състояние Проверено. Ако има грешки в MDL-описанието, преводът се спира и се извежда съобщение за грешка.

Командата `Prepare version` води версията от състояние Проверена в състояние Подготвена. Използва С-компилатор, за да преведе С-кода в обектен код.

Една компонента може да се представи в различни версии. Текущата версия е една от тези версии.

Ако се избере самата версия, текущата версия може да се провери и подготви. Командите `Check current version` и `Prepare current version` са достъпни за компоненти.

# Версия и състояние на модела

Състояния на модели.

За да е възможна работата с компонента или в случай на йерархичен модел – йерархия от компоненти трябва да се създаде модел.

# Версия и състояние на модела

Модели могат да попаднат в две различни състояния – конфигурирана или инсталлирана.

Забележка:

За йерархични модели, които се състоят от повече от една компонента е нужно да се избере само компонентата от най-високо равнище.

В състоянието Конфигурирана, моделът се състои само от име на модел и каталог с версии, или в случай на йерархичен модел версии, които ще принадлежат на модела. Състоянието се въвежда чрез избиране на компонента и използване на команда Install model.

# Версия и състояние на модела

Един модел се премества от състояние Конфигуриран в състояние Инсталиран, използващ командата `Install model`.

Тази команда се достига чрез десния клавиш на мишката, след като се избере моделът да бъде инсталиран. Компилираните версии на компонентите се свързват със системата за изпълнение, за да се създаде изпълнима програма.

Командата `Reset` изтрива тази изпълнима програма.

Остава само каталога, съдържащ компонентата версия. Това състояние на переход се извършва автоматично, когато потребителят промени версията на компонентата, принадлежаща на модела, докато тези модификации изискват изпълняваната програма да се прекомпилира.

# Версия и състояние на модела

Замисъл на версията.

Всеки компонент може да има различни версии. Версите са независими една от друга.

Едно сходство илюстрира ситуацията:

Всеки член на семейство си има своя идентичност и име. Те отговарят на версия Членове на едно и също семейство, имат обща фамилия, която ги присъединява в единица от по-високо равнище. Семейството (фамилията) отговаря на компонентата (име).

За да се идентифицира уникално член на едно семейство са необходими и фамилията и индивидуалното първо име. Подобно обект на модел се описва от неговите компоненти и имена на версията.

В Simplex3 една версия може да се декларира като текуща версия, което я различава от другите. Текущите версии са достъпни чрез техните компонентни имена.

# Версия и състояние на модела

Когато компонента се разшири, принадлежащата ѝ версия става видима. Текущата версия може да се открие чрез избиране на компонентната директория.

Прозорецът на съдържанията показва всички компоненти, включително и версииите им.

Пример:

Моделният обект Biotope\_1 / Version0 ще се направи текуща версия. Тогава е достатъчно просто да се даде компонентното име Biotope\_1, за да се получи достъп и да се редактира точно тази версия.

# Версия и състояние на модела

Решаващото преимущество на концепцията на версията се изяснява, когато се използват юерархичните компоненти. Не са необходими никакви промени на описанието на модела, когато се промени една версия.

Забележка:

Ако един компонент се състои само от една версия, това автоматично е текущата версия. Всички команди, които са валидни за версиите са валидни също и за компонентите.

# Версия и състояние на модела

Първите и последните редове трябва да съдържат новото име на компонентата „Biotope\_Test”, вместо „Biotope\_1”.

Засегнати са следните два реда:

BASIC COMPONENT Biotope\_Test

END OF Biotope\_Test

- Проверка и подготовка на компонентата:

Подготовката включва проверката. В този случай може да с използва направо команда Prepare version.

# Версия и състояние на модела

Упражнение1:

Създайте нова компонента, наречена Biotope\_2Priv. Това се състои от допълнително събитие, отговарящо на компонентата Biotope\_1.

Когато броя на зайците става под 300, 10% от лисиците се премахват.

Това събитие има следната форма в SIMPLEX MDL:

ON ^Hare < 300[NumH]^

DO

Fox^ := Fox – Fox/10;

DISPLAY (“ Vreme T %f \ Broi na premahnatite lisici: %f \n\n”, t,  
Fox/10);

END

Събитието трябва да се вмъкне след ключовата дума DYNAMIC BEHAVIOUR.

Упражнение2:

Вместо разиването на новата компонента като независима компонента, именувана Biotope\_2 Priv, създайте я като Version3 на компонентата Biotope\_1.

# Версия и състояние на модела

Команден способ.

Най-лесно е да се използват менютата. Това е по-бавния способ. По-бързия способ е да се използва командния способ.

Въвеждане на команди.

Командите се въвеждат в penultimate line на SIMPLEX прозорец със заглавието Command Interpreter. Тук желаните команди заедно с подходящите им параметри могат да се въведат от клавиатурата.

# Версия и състояние на модела

Валидни са следните правила:

1. Разделителят между името на командата и параметрите е символът интервал.

Пример:

SelBank Biotope

2. Разделителят между параметрите е символът интервал.

Пример:

AddVar Window Hare Fox

SetVar Hare 500

Общия команден синтаксис има следния вид:

Cd1 Par1 Sep Par2 Sep... ParN;

където Cd1...CdN са команди в експерименталната среда,  
Par1...ParN са съответни параметри и Sep се използва за  
означаване на символа интервал.

# Версия и състояние на модела

3. Поправки се правят чрез поставяне курсора от дясно на грешния символ, използвайки курсорните клавиши. При натискане на клавиша Delete този символ ще се изтрие. Тогава нов символ може да се въведе пред курсорът.
4. Всички команди, които се изпълняват се запазват по ред. Чрез натискане на стрелките нагоре и надолу списъка команди може да бъде обхождан и предишни команди могат да бъдат пренесени като команди за редактиране и въвеждане. Командите, които се появяват в полето могат да бъдат модифицирани. При натискане на курсорния клавищ върху командния вход се получава ново командно поле.
5. Когато обектите се създават, преименуват и копират, новото име е чувствително за големи и малки букви. По-късно потребителят може да използва и главни и малки букви.

Пример:

SelBank Biotope  
is identical to  
selbank biotope

# Версия и състояние на модела

6. Имената на командите могат да се съкращават докато са уникални. Съкращенията не могат да се използват за параметри.

Пример:

SelBank Biotope

is identical to

selb biotope

7. Цялото съдържание на командното поле се въвежда, когато се натисне клавиша Return. Позицията на курсора в командния ред се пренебрегва.

# Версия и състояние на модела

Библиотеки на модела.

Всички компоненти, които потребителя разглежда за принадлежащи, заедно могат да се обединят от библиотека на модела. Отделните потребители могат да имат няколко моделни библиотеки, които всички принадлежат на неговия личен private каталог от библиотеки. Не е възможен достъп до private каталог от библиотеки и затова също и до моделните библиотеки на други потребители.

В допълнение на private каталогите от библиотеки, принадлежащи на индивидуален потребител, е достърен public каталог от библиотеки. Всички потребители имат достъп до този каталог от библиотеки. Public каталога от библиотеки се използва основно за размяна на компоненти между потребители.

# ЛЕКЦИЯ 7

## ОСНОВИ НА ЕЗИКА ЗА ОПИСАНИЕ НА МОДЕЛИ **SIMPLEX MDL**

-  **Структура на основните компоненти**
-  **Алгебрични уравнения**
-  **Събития**
-  **Повишаване на времето в дискретно-времеви модели**
-  **Разделяне на пространството на състоянията**
-  **Масиви**

# **ВЪВЕДЕНИЕ**

- Запознаване с най-важните елементи на  
моделния описателен език **SIMPLEX MDL**;
- Структура на основните компоненти;
- Алгебрични уравнения;
- Събития;
- Разделяне на пространството на  
състоянията;
- Масиви.

# Структура на основните компоненти

Основните компоненти са основните, изграждащи блокове в Simplex3. Те съдържат описание на динамичното поведение на модела. Компонентите от по-високо равнище свързват основните компоненти, за да се оформи структура, но те не съдържат описание на динамично поведение.

# Структура на основните компоненти

Моделът CedarBog.

Моделът CedarBog описва затлачването на езеро от време на време. Моделът е описан първо в R. B. Williams, в статията му „Компютърна симулация на енергиен поток в езерото Кедър“, списание „Системен анализ и симулация в Екологията“, Academic Press 1971 г.

Първоначално моделът използва три променливи на състояния:

$p$ : растения (plants);

$h$ : тревопасни животни (herbivores);

$c$ : месоядни животни (carnivores).

# Структура на основните компоненти

- Променливите имат мерни единици „Тонове биомаса“.

Мъртвата органична материя, която е разположена на дъното езерото се представя с  $o$ .

Загубата на биомаса в околната среда е представена от променливата  $e$ .

Моделът се описва от пет променливи:  $p, h, c, o$  и  $e$ .

# Структура на основните компоненти

Енергията от слънчева светлина се означава със sun.

Тя се описва от следното алгебрично уравнение:

$$\text{Sun} = 95.9 * (1 + (0.635 * \sin(2 * \pi * T))).$$

Мерните единици за слънчева радиация са  $\text{kJ/m}^2$ .

## Структура на основните компоненти

Слънчевата радиация води до увеличаване на биомасата на растенията за единица време.

Превръщащият фактор Bio\_Fac е необходим за да превръща слънчевата светлина в увеличаваща се растителна биомаса за единица време:

$$\text{sun\_bio} = \text{sun} * \text{Bio\_Fac}$$

# Структура на основните компоненти

Превръщаия фактор Bio\_Fac се дава от уравнението:

$$\begin{aligned} Bio\_Fac &= \frac{sun\_bio}{sun} = \frac{\frac{t}{a}}{\frac{kJ}{m^2}} = \frac{\frac{10^3 * kg}{3,15 * 10^7 * s}}{\frac{10^3 * Nm}{m^2}} = \frac{\frac{kg}{3,15 * 10^7 * s}}{\frac{kg * m * m}{s^2 * m^2}} = \frac{s}{3,15 * 10^7} = 3,17 * 10^{-8} s \\ &= 3,17 * 10^{-8} s = 3,17 * 10^{-8} * (3,17 * 10^{-8}) * a \approx 10^{-15} * a \end{aligned}$$

Това превръщане лесно се осъществява автоматична от Simplex3, ако единиците за Bio\_Fac са дадени като  $(t * m^2) / (a * kJ)$ . В описанието на модела това е записано като:

Bio\_Fac (REAL [t \* m<sup>2</sup>] / (a \* kJ) ]): = 1[(t \* m<sup>2</sup>) / [(a \* kJ)]

# Структура на основните компоненти

Следните уравнения описват връзките между растенията, тревопасните и месоядните животни.

$$p' = \text{sun\_bio} - 4.03 * p$$

$$h' = 0.48 * p - 17.87 * h$$

$$c' = 4.85 * h - 4.65 * c$$

# Структура на основните компоненти

Първото равенство означава, че има експоненциално намаляване на биомасата на растенията, както и увеличение породено от слънчевата светлина, което кара растенията да растат.

Подобно второто равенство описва скоростта на промяна на биомасата на тревопасните. Намалението на биомасата е възпрепятствано от увеличението на консумирането (унищожаването) на растенията. Поведението на месоядните е аналогично.

# Структура на основните компоненти

Загубата на биомаса в околната среда и мъртвата органична материя се дава с:

$$o' = 2.55 * p + 6.12 * h + 1.95 * c$$

$$e' = 1.00 * p + 6.90 * h + 2.70 * c$$

Двете равенства дават скоростта на:

- промяната на биомасата, която се съдържа в мъртвия органичен материал;
- губещата се биомаса в околната среда.

$O$  и  $e$  са пропорционални на биомасата на растенията, тревопасните и месоядните животни.

# Структура на основните компоненти

Променливата  $T$  представя симулационното време.  
Прогресът на симулационно време се контролира  
автоматично от системата за контрол на изпълнението на  
симулационната среда.

$T$  не е задължително да се дава от потребителя.

Не е задължително да се декларира  $T$ .

## Моделно описание за компонента CedarBog.

```
1 BASIC COMPONENT CedarBog

2 USE OF UNITS
3 TIMEUNIT = [a]

4 DECLARATION OF ELEMENTS
5 CONSTANTS
6     Pi      (REAL)    := 3.14,
7     Bio_Fac (REAL[a]) := 1E-15 [a]

8 STATE VARIABLES
9 CONTINUOUS
10    p (REAL[t]) := 0 [t],
11    h (REAL[t]) := 0 [t],
12    c (REAL[t]) := 0 [t],
13    o (REAL[t]) := 0 [t],
14    e (REAL[t]) := 0 [t]

15 DEPENDENT VARIABLES
16 CONTINUOUS
17    sun      (REAL[kJ/m^2]) := 0 [kJ/m^2],
18    sun_bio (REAL[t/a])     := 0 [t/a]

19 DYNAMIC BEHAVIOUR
20    sun := 95.9 [kJ/m^2] * (1 + 0.635 * SIN (2[1/a]*Pi*T));
21    sun_bio := sun * Bio_Fac;

22 DIFFERENTIAL EQUATIONS
23    p' := sun_bio - 4.03[1/a] * p;
24    h' := 0.48[1/a] * p - 17.87[1/a] * h;
25    c' := 4.85[1/a] * h - 4.65[1/a] * c;
26    o' := 2.55[1/a] * p + 6.12[1/a] * h + 1.95[1/a] * c;
27    e' := 1.00[1/a] * p + 6.90[1/a] * h + 2.70[1/a] * c;
28 END

29 END OF CedarBog
```

## Структура на основните компоненти

Преди симулационното изпълнение Run1 се дават стойности на контролните параметри. На фигурана са показани подходящи стойности за модела MCedarBog. Промените се правят чрез избиране на обект Control parameters и викане на командата Configure control parameters. След въвеждане на параметрите, трябва да се извика функцията Check.

# Структура на основните компоненти

Създават се наблюдатели с цел в тях да се запишат и представят резултатите на избраните динамични редове. В нашия случай, променливите са Слънце (sun), Растения (p), Месоядни (c) и Органични (o).

# Структура на основните компоненти

Всички моделни променливи трябва да имат начални стойности. Симулационното изпълнение продължава до  $T = 2.0$ . Динамичните редове, които са записани в каталога *Simulations results*, могат да бъдат показани и анализирани.

Подкаталога *Statistics* в директория *Protocols* осигурява информация за напредъка на симулационното изпълнение - например броя на стъпките или средната стойност на дължината на стъпката.

# Структура на основните компоненти

Синтаксис за описание на основни компоненти.

Синтаксиса за описание на основните компоненти е:

```
basic_component ::= BASIC COMPONENT identifier  
                  [mobile_subclass_declaraction]  
                  [unit_definition_part]  
                  [local_definitions]  
                  declaration_of_elements  
                  [dynamic_behaviour]  
                  END OF identifier
```

Една основна компонента се състои от име (Identifier), декларационна част и описание на динамичното поведение.

# Структура на основните компоненти

Квадратните скоби посочват, че обградената секция не се нуждае да бъде представяна.

Редът на секциите трябва да се спазва:

mobile\_subclass\_declaration

Тук се декларират класове за подвижни компоненти, които се използват в основните компоненти.

unit\_definition\_part

# Структура на основните компоненти

На променливите на модела и числата могат да се посочат мерни единици.

local \_definitions

Секцията съдържа изброени променливи, списък функции и списък на случайни разпределения.

declaration\_of \_elements

Тук се декларираят константи, променливи, произволни променливи, индикатори и региони, използвани в модела.

# Структура на основните компоненти

Ключовата дума BASIC COMPONENTS е последвана от името на компонентата.

Декларационната част съдържа само declaration\_of\_elements.

Синтаксисът на декларацията на елементите е:

declaration\_of\_elements ::= DECLARATION OF ELEMENTS

- [list\_of\_constants]
- [list\_of\_state\_variables]
- [list\_of\_dependent\_variables]
- [list\_of\_sensor\_variables]
- [list\_of\_random\_variables]
- [list\_of\_transition\_variables]
- [list\_of\_sensor\_indications]
- [list\_of\_locations]
- [list\_of\_sensor\_locations]

# Структура на основните компоненти

Тези елементи могат да се разделят в четири групи:

- Променливи на модела;
- Случайни променливи;
- Индикатори;
- Региони.

Променливите на модела описват атрибутите, които всяка компонента притежава. Всяка променлива на модела изисква:

- Употреба;
- Поведение като функция на времето;
- Тип.

## Променливи на състоянието

Използват се за описание на динамични свойства на компонента. Тяхната първа производна относно времето може да се използва в диференциалните уравнения. Техните стойности могат да бъдат модифицирани от дискретни събития.

# Структура на основните компоненти

Зависима променлива:

Зависимите променливи се дефинират чрез уравнения. Дясната част на алгебрично уравнение може да съдържа променливи на състоянието, други зависими променливи или времето.

Сензорна променлива:

Сензорните променливи са свързани с променлива в друга компонента, използвайки връзка в компонента от по-високо равнище. Сензорната променлива позволява достъп до променливи от други компоненти. Може да не се дават инициализационни стойности на сензорните променливи.

# Структура на основните компоненти

Времево поведение на моделните променливи:

- дискретно

Стойността на тези променливи може да се промени от отделни събития;

- непрекъснато:

Тези променливи се променят и по непрекъснат и по дискретен начин. Непрекъснатите променливи се описват от диференциални уравнения.

Типове променливи на модела:

- цял;
- реален;
- логически;
- изброим.

# Структура на основните компоненти

Моделът *CedarBog* не съдържа никакви *submit\_declaration* или *local\_definitions*. Нужно е само *declaration\_of\_elements*.

Списъкът от елементи се състои от константи, променливи на състоянието и зависими променливи.

Времевото поведение трябва да се определи за всички променливи на състоянието и зависими променливи.

В този случай и двете са непрекъснати.

Всички моделни променливи трябва да получат тип. Тук всички са от тип *Real*.

# Структура на основните компоненти

Синтаксиса на динамичното поведение има следната форма:

```
dynamicBehaviour ::= DYNAMIC BEHAVIOUR  
                      statement_sequance
```

```
statement_sequance ::= statement { statement }
```

Фигурните скоби означават, че съдържанието им може да е празно, или че те могат да се повтарят неограничен брой пъти.

```
statement ::= algebraicEquation  
                  | differentialEquations  
                  | regionDefiningStatement  
                  | eventDefiningStatement
```

# Структура на основните компоненти

Реда на запис на алгебричните уравнения, диференциалните уравнения и събитията е произволен.

- Алгебрични уравнения:

Алгебричните уравнения са дефиниращи уравнения за зависимите променливи от лявата страна на ‘:=’;

- Диференциални уравнения:

Позволени са диференциални уравнения от първи ред;

- Събития:

Събитията описват промените на променливите на състоянията.

Стартирането на събитие се случва, когато състоянието на модела отговаря на определените в събитието условия.

# Структура на основните компоненти

Моделът *CedarBog* съдържа алгебрични уравнения за зависимите променливи *sun* и *sun\_bio*.

**Забележка:**

*SIMPLEX MDL* е декларативен и непроцедурен когато става дума за описание на динамика.

След ключовата дума DIFFERENTIAL EQUATIONS се записват диференциалните уравнения за непрекъснатите променливи *p*, *h*, *c*, *o* и *e*.

# Структура на основните компоненти

## Забележки:

- Ключовите думи са част от синтаксиса. Те винаги се записват с главни букви в SIMPLEX MDL. Елементите на декларацията са разделени чрез запетайки, освен за последния. Изразите завършват винаги с точка и запетая.

Примерния модел CedarBog е непрекъснат.

# Алгебрични уравнения

Алгебричните уравнения в SIMPLEX MDL дефинират зависима променлива. Дясната страна на дефиниращото уравнение може да съдържа променливи на състоянието, зависими променливи или време.

# Алгебрични уравнения

Пример. Декларирали са две зависими променливи A и B.

DEPENDENT VARIABLES

DISCRETE

A (REAL) : = 0,

B (REAL) : = 0

Нека те са дефинирани от следните две алгебрични  
уравнения:

DYNAMIC BEHAVIOUR

A : = 2;

B : = A + 1;

# Алгебрични уравнения

От тези дефиниции, тези зависими променливи винаги имат следните стойности:

$A = 2$

$B = 3$

Независимост на реда означава, че алгебричните равенства могат да се напишат така:

$B := A + 1;$

$A := 2;$

Трябва да е вярно все още и:

$A := 2 \quad B := 3$

Ясно е, че алгебричните уравнения не трябва да се бъркат с изразите в процедурните езици за програмиране.

# Алгебрични уравнения

За да се уверим, че алгебричните уравнения винаги дават същите резултати, независимо от реда им, те се оценяват итеративно (последователно) докато стойностите на зависимите променливи повече не се различават от тези на предишната итерация.

Първо показваме какво се случва, когато равенствата са написани в следния ред:

A := 2;

B := A + 1;

# Алгебрични уравнения

Нови стойности се изчисляват от старите. След второто изчисляване на алгебричните уравнения, стойностите на А и В нямат промяна и стойностите

$$A = 2 \quad B = 3$$

се взимат като финални.

Бяха нужни две итерации. Последната итерация се използва само за проверка дали итерацията на алгебричните уравнения може да се прекъсне.

Ако алгебричните уравнения се напишат в следния ред:

$$B := A + 1; \quad A := 2;$$

# Алгебрични уравнения

Сега са необходими три итерации за изчисляването.

Максималния брой итерации може да се определи от потребителя, използвайки контролния параметър *MaxCyc*, който може да се промени в каталога *Control parameters* във всяко изпълнение чрез извикване на командалата *Configure control parameters*. Подходящото поле във входния прозорец може да се открие в горната лява страна на секцията *Algebraic equations* под името *Max.#Cycles*. Той има стойност по подразбиране 50.

Итерацията прекъсва, когато стойностите на зависимите променливи не се променят с повече от *EPS*. Стойността на *EPS* може да се промени в полето *Epsilon*. То има стойност по подразбиране 1E-10.

# Алгебрични уравнения

Забележки:

- Тъй като алгебричните уравнения могат да се оценят много често, препоръчително е те да са подредени подходящо в описанието на модела. Това може да спести времето на изчисление;
- Независимостта на реда от алгебрични уравнения е необходима, за да е възможно да се раздели една компонента на подкомпоненти по произволен начин. Независимостта на реда гарантира резултатите да са идентични, независими от реда, в който подкомпонентите са извикани.

# Алгебрични уравнения

Възможно е следното разширение на модела Biotope\_1. Представено е ограничение на пашата, което осигурява, че броят на зайците не превишава MaxCap = 1500. Вместо

$$\text{Hare}^{\prime} = a * \text{Hare}$$

сега е налице:

$$\text{Hare}^{\prime} = (\text{MaxCap} - \text{Hare}) / \text{MaxCap} * a * \text{Hare}$$

Скоростта на нарастване за зайците стига до нула когато броят на зайците приближава MaxCap.

# Алгебрични уравнения

На първа итерация получаваме:

$$\text{Hare\_Inc} = 0 \quad \text{Cap} = 0.733$$

$$\text{Fox\_Dec} = -46.25 \quad \text{Hits} = 555$$

На втора итерация получаваме:

$$\text{Hare\_Inc} = 513.33 \quad \text{Cap} = 0.733$$

$$\text{Fox\_Dec} = -46.25 \quad \text{Hits} = 555$$

Simpex3 може да изведе детайлен протокол на оценката на алгебричните уравнения и преходите между състоянията. За да се активира тази функция се активира бутона Options в долната дясна част на прозореца на контролните параметри в секция Protocol.

В протокола са се отпечатали само тези стойност, които са се изменили от последната ситуация. Ако дадена стойност не присъства в протокола това означава, че тя остава непроменена от предходния цикъл.

# Алгебрични уравнения

BASIC COMPONENT Biotope\_3

USE OF UNITS

UNIT[NumH] = BASIS

UNIT[NumF] = BASIS

TIMEUNIT = [a]

DECLARATION OF ELEMENTS

CONSTANTS

a (REAL[1/a]) := 1.75 [1/a], # Скорост на нарастване на зайците

b (REAL[1/a]) := -1.25 [1/a], # Скорост на нарастване на лисиците

c (REAL[1/(NumH\*NumF\*a)]) # Вероятност на срещите  
:= 0.0375 [1/(NumH\*NumF\*a)], #

MaxCap(INTEGER[NumH]):= 1500[NumH],# Максимален капацитет

BFacH(REAL[NumH]) := 1.0[NumH], # Коефициент на плячка  
зайци

BFacF(REAL[NumF]) := 0.1[NumF] # Коефициент на плячка  
лисици

# Алгебрични уравнения

STATE VARIABLES

CONTINUOUS

Hare (REAL[NumH]) := 400[NumH],

Fox (REAL[NumF]) := 37[NumF]

DEPENDENT VARIABLES

CONTINUOUS

Cap (REAL), # Свободен капацитет

Hare\_Inc (REAL[NumH/a]), # Увеличение на зайците

Fox\_Dec (REAL[NumF/a]), # Намаляване на лисиците

Hits (REAL[1/a]) # Срещи

# Алгебрични уравнения

## DYNAMIC BEHAVIOUR

```
Hare_Inc := Cap * a * Hare;  
Cap      := (MaxCap - Hare) / MaxCap;  
Fox_Dec  := b * Fox;  
Hits     := c * Hare * Fox;
```

## DIFFERENTIAL EQUATIONS

```
Hare' := Hare_Inc - BFacH * Hits;  
Fox'  := Fox_Dec + BFacF * Hits;
```

END

# Алгебрични уравнения

## DYNAMIC BEHAVIOUR

Cap := (MaxCap - Hare) / MaxCap;

Hare\_Inc := Cap \* a \* Hare;

Hits := c \* Hare \* Fox;

Hare' := Hare\_Inc - BFacH \* Hits;

END

Fox\_Dec := b \* Fox;

## DIFFERENTIAL EQUATIONS

Fox' := Fox\_Dec + BFacF \* Hits;

END

# Алгебрични уравнения

Забележка:

Позволени са и алгебричните уравнения от следния вид:

$$y = y - 0.5 * (y - (x/y))$$

Стойността на  $y$  се оценява итеративно докато или максималния брой итерации е достигнат, или разликата между старите и новите стойности на  $y$  са станали по-малки от EPS. Това позволява формиране на рекурентни алгебрични уравнения.

Позволено е алгебричното уравнение  $A = A + 1$ . Уравнението  $A := A + 1$  се изчислява многократно, докато се достигне MaxCyc, увеличавайки стойността на  $A$  с 1 всеки път.

# Събития

Едно събитие причинява промяна в състоянието за променливите на състоянията в дискретни точки от време.

Събитието се състои от следните три части:

- Стартиращ механизъм;
- Процедурна част; (незадължителна)
- Описанието на промените на състоянието.

Синтаксиса има формата:

```
event_defining_statement ::= triggering_mechanism  
                           [procedural_part]  
                           transitions_part
```

# Събития

Има две възможности за стартиращия механизъм:

- Удовлетворени стойности на условие;
- Чрез индикатор.

Синтаксиса на стартиращия механизъм е:

triggering\_mechanism ::= WHENEVER expression  
/ON indication {OR indication}

# Събития

## Стартиращия механизъм и конструкцията WHENEVER.

Конструкцията WHENEVER причинява случването на събитие, когато логическия израз след нея има стойност TRUE.

Това означава, че определено състояние на модела, което е представено от логическия израз, причинява промяната в състоянието на модела.

Важно е да се отбележи, че промените на състоянието в събитие, определено от WHENEVER конструкция, се изпълняват докато стойността на логичния израз е TRUE. Промените на състоянието трябва да покажат, че истинната стойност се променя на FALSE, за да се избегне безкраен цикъл.

Обратно, ON конструкцията причинява събитие да се случи само, когато истинната стойност на условие се промени от FALSE на TRUE. Ако стойността остане TRUE, събитието не се пуска от ON конструкцията.

# Събития

Събитията се изпълняват поради стартиращия механизъм.

В допълнение към незадължителната процедурна част, събитието съдържа описание на промените на състоянието.

Това описание включва:

- Дефиниране на промяната на състоянието;
- Сигнални команди;
- Дефиниране на региони за събития;
- Команди за визуализация.

# Събития

Синтаксисът е следния:

```
transition_part ::= DO [TRANSITION[S]
                        transition_statement_sequence
                        END
transition_statement_sequence ::= transition_statement
                                { transition_statement }
transition_statement ::= state_transition_definition
                        | signal_satement
                        | event_region_defining_statement
                        | display_statement
state_transition_definition ::= state_variable_assinment
                                transfer_statement
state_variable_assinment ::= selected_element
                            ‘^’ ‘:=’ ex[ression ‘;’]
```

# Събития

Избираме като илюстрация модела CedarBig. Моделът се разширява от едно събитие. Мъртвата органична материя на дъното на езерото надхвърля  $o = 30[t]$ .

Като резултат от това събитие, стойността на променливата на състоянието е поставена да е  $o = 0 [t]$  и събитието не се изпълнява.

# **Събития**

Пример. Моделът Empty с едно събитие.

## **BASIC COMPONENT Empty**

### **USE OF UNITS**

**TIMEUNIT = [a]**

### **DECLARATION OF ELEMENTS**

### **CONSTANTS**

**Pi (REAL) := 3.14,**

**Bio\_Fac (REAL[a]) := 1E-15 [a]**

# Събития

## STATE VARIABLES

### CONTINUOUS

```
p (REAL[t]) := 0 [t],      # Растения  
h (REAL[t]) := 0 [t],      # Тревопасни  
c (REAL[t]) := 0 [t],      # Месоядни  
o (REAL[t]) := 0 [t],      # Запас  
e (REAL[t]) := 0 [t]       # Загуба на среда
```

## DEPENDENT VARIABLES

### CONTINUOUS

```
sun   (REAL[kJ/m^2]) := 0 [kJ/m^2],  # Solar radiation  
sun_bio (REAL[t/a])  := 0 [t/a]
```

## DYNAMIC BEHAVIOUR

```
sun := 95.9 [kJ/m^2] *(1+0.635 * SIN (2[1/a] * Pi *T));  
sun_bio := sun * Bio_Fac;
```

# Събития

## DIFFERENTIAL EQUATIONS

$p' := \text{sun\_bio} - 4.03[1/a] * p;$

$h' := 0.48[1/a] * p - 17.87[1/a] * h;$

$c' := 4.85[1/a] * h - 4.65[1/a] * c;$

$o' := 2.55[1/a] * p + 6.12[1/a] * h + 1.95[1/a] * c;$

$e' := 1.00[1/a] * p + 6.90[1/a] * h + 2.70[1/a] * c;$

END

WHENEVER  $o > 30 [t]$

DO

$o^{\wedge} := 0 [t];$

END

END OF Empty

# Събития

Пример. Еднократно изпълнение на събитие.

BASIC COMPONENT PrintSun

USE OF UNITS

TIMEUNIT = [a]

DECLARATION OF ELEMENTS

CONSTANTS

Pi (REAL) := 3.14,

Bio\_Fac (REAL[a]) := 1E-15 [a]

# Събития

STATE VARIABLES

DISCRETE

Printed (LOGICAL) := FALSE

CONTINUOUS

p (REAL[t]) := 0 [t],	# Plants
h (REAL[t]) := 0 [t],	# Herbivores
c (REAL[t]) := 0 [t],	# Carnivores
o (REAL[t]) := 0 [t],	# Deposits
e (REAL[t]) := 0 [t]	# Loss to environment

# Събития

DEPENDENT VARIABLES

CONTINUOUS

sun (REAL[kJ/m^2]) := 0 [kJ/m^2], # Solar radiation

sun\_bio (REAL[t/a]) := 0 [t/a]

DYNAMIC BEHAVIOUR

sun := 95.9 [kJ/m^2] \*(1+0.635 \* SIN (2[1/a] \* Pi \*T));

sun\_bio := sun \* Bio\_Fac;

# Събития

DIFFERENTIAL EQUATIONS

p' := sun\_bio - 4.03[1/a] \* p;

h' := 0.48[1/a] \* p - 17.87[1/a] \* h;

c' := 4.85[1/a] \* h - 4.65[1/a] \* c;

o' := 2.55[1/a] \* p + 6.12[1/a] \* h + 1.95[1/a] \* c;

e' := 1.00[1/a] \* p + 6.90[1/a] \* h + 2.70[1/a] \* c;

END

WHENEVER (sun < 95.9 [kJ/m<sup>2</sup>]) AND NOT (Printed)

DO

DISPLAY("T= %f Radiation falls below %f \n", T, sun);

Printed^ := TRUE;

END

WHENEVER (sun >= 95.9 [kJ/m<sup>2</sup>]) AND (Printed)

DO

Printed^ := FALSE;

END

END OF PrintSun

# Събития

Позволени са сравнения в логически израз за конструкцията  
**WHENEVER.**

Допустими са операции:

,,”=” : равно;

,,”<” : различно;

,,”<” : по-малко от;

,,”<=” : по-малко или равно;

,,”>” : по-голямо от;

,,”>=” : по-голямо или равно.

# Събития

Логическия израз за WHENEVER конструкцията може се съдържа времето  $T$ .  $T$  е от тип REAL, стойността му винаги е достъпна и не се нуждае от декларация.

# Събития

Примери:

Ако събитието се регистрира, когато симулационния часовник  $T$  достигне стойността 10, тогава трябва да се използват следните условия :

WHENEVER  $T \geq 10$ ;

Моделът Empty може да бъде разширен. В модела залагаме, че преди  $T = 1.0$  не може да настъпи изследваното събитие.

Логическото условие има следната форма:

WHENEVER ( $T \geq 1.0 [a]$ ) AND ( $o > 30 [t]$ )

Като следствие на това ограничение,  $o$  първоначално ще расте. В момента  $T = 1 [a]$ , стойността се нулира  $o = 0 [t]$ , докато в този момент време „ $o$ ” е вече значително по-голямо от  $30 [t]$ . Оттук нататък, намалението се появява всеки път, когато „ $o$ ” достигне стойността  $30 [t]$ , докато  $T > 1.0 [a]$ .

# Събития

## Стартиращ механизъм и ON конструкцията

Всички условни крайни промени на състояния могат да се опишат, използвайки WHENEVER конструкция. ON конструкцията е полезно допълнение. То позволява едно събитие да бъде стартирано само веднъж при дадена индикация.

Синтаксиса има следния вид:

triggering\_mechanism ::= WHENEVER expression  
                                 | ON indication {OR indication}

indication ::= indexed\_identifier  
                         | START  
                         | "^^" expression "^^"

# Събития

Примери:

В модела Empty стойността за  $o$  е дефинирана като  $o=0$  [t], винаги когато е изпълнено условието  $o>30$  [t]. Тази процедура може да бъде записана използвайки конструкцията WHENEVER, тъй като чрез установяването на  $o$  като  $o=0$  [t], условието става лъжа и събитието не се изпълнява втори път. Възможно е да се замени WHENEVER конструкцията с ON конструкцията.

Това води до следния запис:

ON  $^o>30$  [t] $^$

DO

$o^:=o$  [t];

END

# Събития

Ако например, условието  $o > 30$  [t] води до еднократно действие, което намалява броя на растенията с 10%. Описанието трябва да е следното:

ON  $^o > 30$  [t]<sup>^</sup>

DO

$p^{\wedge} := p - p / 10;$

END

# Събития

В моделът PaintSun, извеждането на слънчевата енергия е много по-просто с използването на ON-конструкцията. Логическата променлива Printed повече не се нуждае от анулиране. Когато радиацията падне под  $95.9 \text{ [kJ/m}^2\text{]}$ , събитието се изпълнява само веднъж. Описанието има следния вид:

ON ^sun<95.9 [kJ /m ^ 2]^

DO

DISPLAY (“ $T = \% f$  Енергията пада под  $\% f \backslash n$ ”, T, sun);  
END

# Събития

Индикаторите се декларират в декларационната част:

Пример:

DECLARATION OF ELEMENTS

STATE VARIABLE

A (INTEGER) := 0

TRANSITION INDICATORS

Indic

DYNAMIC BEHAVIOUR

ON ^ T >= 10 ^

DO SIGNAL Indic;

END

ON Indic

DO A ^ := 1;

END

# Събития

В момента  $T = 10$ , е поставен индикатора Indic.

Възможна е употребата на стандартен указател START.

Пример:

ON START

DO

$A^:=0;$

END

В този случай, първо се обработва събитие при започването на симулационното изпълнение.

START индикатора е необходим, защото промяна в логическата стойност на условие не може да се установи, докато няма предишна стойност.

Забележка:

Индикатора START е стандартен. Не е необходимо да се декларира.

# Събития

Преходи в събитие.

Събитията се състоят от стартиращ механизъм, процедурната част и преходи. Възможни са следните конструкции за преходите:

- Промени на състояние;
- Командата SIGNAL;
- Разделение на пространството на състояния от събитие;
- Командата DISPLAY.

Синтаксиса е:

```
transition_statement ::= state_transition_definition  
                      | signal_statement  
                      | event_region_defining_statement  
                      | display_statement
```

# Събития

Частта state\_transition\_definition описва възможностите за промени на състояние.

Има две възможности за определяне на промяната на състоянието:

- Промени на състоянието за променливи на състоянието;
- Преместващи команди за подвижни компоненти.

Преходите между състояния при събития са позволени за непрекъснати и за дискретни променливи на състоянията.

Забележка:

Зависима променлива, чиято стойност се изменя само от събития, трябва да се декларира като дискретна. Непрекъснатите променливи на състоянията могат да бъдат променени от събитие.

# Събития

Синтаксисът за промени на състояния за дискретни или непрекъснати променливи на състоянията има следната форма:

state\_transition\_definition ::= state\_variable\_assignment  
| transfer\_statement

state\_variable\_assignment ::= selected\_element '^' ':' '='  
expression ';' ;

# Събития

Примери:

- Моделът Empty съдържа следната променлива на състоянието:  
o ^ := 0 [t];

Променливата „o” е непрекъсната. Тя може да се променя от събитие;

- В моделът PrintSun е поставена дискретната променлива Printed. Printed е обявена като дискретна променлива на състоянието от тип LOGICAL, например:

Printed ^ := TRUE;

Следващата възможност за промяна на състояние е команда SIGNAL. Тя поставя индикатор в събитие.

Командата SIGNAL се прилага към потребителски дефиниран индикатор. Стандартните индикатори START и STOP са на разположение по подразбиране и не е нужно да бъдат декларирани. Последният кара симулацията да спре.

Това позволява формулировката на произволен стоп-критерий.

# Събития

Пример:

Моделът CedarBog се изпълнява до  $T = 2$  [a].

По-нататък стоп критерий може да бъде, когато залежите (отлаганията) достигнат стойност  $o = 150$  [t]. Допълнителното събитие ще изглежда така:

ON       $^o > 150$  [t] $^A$

DO

    SIGNAL STOP;

END

Синтаксисът на командалата SIGNAL има следния вид:

signal\_statement ::= SIGNAL indexed\_identifier ';'  
                  | SIGNAL STOP ';'

# Събития

По-нататъшна възможност е да се раздели пространството на състоянията от събитие. Това осигурява възможността за специфициране на различни преходи, зависещи от състоянието на модела.

Синтаксисът е:

```
event_region_defining_statement ::= IF expression  
                                DO transition_statement_sequence END  
                                { ELSEIF expression  
                                  DO transition_statement_sequence END }  
                                { ELSE expression  
                                  DO transition_statement_sequence END }
```

Пример:

Разглеждаме модела CedarBog. Чрез извършване на действия в интервали от 0.2 [a], количеството на растенията се запазва между 22 [t] и 26 [t]. Това изисква събитие, което се стартира редовно.

Промяната на състояние зависи от състоянието на модела в съответния момент от време.

# Събития

Условие	Действие
$p < 20[t]$	20% добавени растения
$20[t] < p < 22[t]$	10% добавени растения
$22[t] < p < 26[t]$	Растения без непромяна
$26[t] < p < 28[t]$	10% растения премахнати
$p > 28[t]$	20 % растения премахнати

# Събития

Събитието има следната форма:

WHENEVER T >= Tstep

DO

  TSTEP<sup>^</sup> := T +0.2;

  IF p < 20 [t]

    DO p<sup>^</sup>:= p + p/5; END

  ELSEIF (p > = 20 [t]) AND (p < 22 [t])

    DO p<sup>^</sup>:= p + p/10; END

  ELSEIF (p > = 26 [t]) AND (p < 28 [t])

    DO p<sup>^</sup>:= p - p/10; END

  ELSEIF p > = 28 [t]

    DO p<sup>^</sup> := p - p/5;END

END

**Забележки:**

Възможно е разделянето на пространство на състоянията при събитие. Ако събитие е стартирано, преходите са зависими от състоянието на модела.

Съществува възможността за специфицирането на различни динамики, основани на състоянието на модела.

# Събития

Синтаксиса е:

```
display_statement ::= DISPLAY '(' string { ',' expression } ')' ';'  
string ::= " " { ascii_character } " "
```

Символът на низ съответства на този на printf команда в езика C. Броят и типът на изразите, които трябва да съответства на низа.

Бележка:

Тук не се изпълнява семантична проверка. Неправилното използване на команда може да доведе до грешка по време на изпълнението като segmentation violation, bus error или floating point overflow.

# Събития

За различните типове се използват следните формати:

INTEGER : % ld

REAL : %f

LOGICAL : %d

Enumerated type : %d

Time T : %f

Стойностите на променливи от логически тип и изброими типове могат само да се изведат чрез тяхното вътрешно представяне. Те съответстват на цели числа в описанието на декларацията.

Специални символи:

\n Нов ред

\f Нова страница

# Събития

Примери:

DISPLAY (“ Arrival of a customer \n ”);

DISPLAY (“ Arrival at time T = %f \n ”, T);

DISPLAY (“ I = %ld X = %f L = %d E= %d \n ”,  
I, X, L, E);

# I:INTEGER, X: REAL

# L:LOGICAL

# E: Enumerated type

DISPLAY (“ Area = %f \n ”, X\*Y);

# Събития

Процедурна част на събитие.

Събитията може да съдържат възможна процедурна част. Тя позволява алгоритми да се включват в моделното описание. Процедурната част се състои от декларативна част и част с изразите. Резултатите се съхраняват във временни променливи и се пренасят в декларативната секция на събитията. Временните променливи не са дефинирани извън събитието. Когато събитието се изпълни втори път, те се инициализират отново. Ако декларацията им не им дава начална стойност, те се инициализира с 0 или FALSE.

# Събития

Процедурната секция може да съдържа:

- Присвоявания;
- Условия;
- Цикли;
- Команди за изход;
- Команди за визуализация.

# Събития

Синтаксис:

procedural\_assignment ::= indexed\_identifier ‘:=’ expression ‘;’

Пример:

ON^T >= TNext^

DECLARE

    X(REAL)                # Дължина на вектора (x1,x2)

DO PROCEDURE

    X := SQRT (x1\*x1 +x2\*x2);

END

DO TRANSITIONS

    Y1^ := x1/x;      # Присвояване на променливите на състоянието

    Y2^ := x2/x;      # Формира се единичния вектор (y1,y2)

END

# Събития

Упражнения:

Представете стоп критерий в модела CedarBog.

Симулацията трябва да се изпълнява поне докато  $T = 20$  [a]. Трябва да приключи след  $T = 20$  [a], когато растенията са със стойност  $p = 30.0$  [t].

Условието за спиране е:

$(T \geq 20 \text{ [t]}) \text{ AND } (p \geq 30.0 \text{ [t]})$

# Събития

Езерото се освобождава от залежи на всеки 0.5 времеви единици.

Упътване:

Въвежда се нова променлива на състоянието TNext, която отбелязва кога се случва събитието. Допълнителното събитие има вида:

WHEREVER T >= TNext

DO

$o^\wedge := 0 [t]$

$TNext^\wedge := TNext + 0.5 [a] ;$

END

## Събития

Езеро се чисти от залежите на всеки 0.5 единици време.

Това се прави само, ако е нужно, т.е. когато  $o > 50 [t]$ .

Упътване:

Събитието има вида:

WHENEVER ( $T \geq T_{\text{Next}}$ ) AND ( $o > 50 [t]$ )

DO

$o^\wedge := 0 [t]$

$T_{\text{Next}}^\wedge := T_{\text{Next}} + 0.5 [a];$

END

# Събития

Нарастване на времето в модели с крайно време.

В Simplex3 стартиращата контролна част на изпълняваната система е отговорен за управлението на преходите между различните състояния.

В SIMPLEX MDL едно състояние винаги се характеризира от време и цикъл.

По всяко време и при всеки цикъл променливата на състоянието е в точно едно състояние.

$z(tn, ki)$  Състояние

$tn$  Време

$ki$  Цикъл

Зависим от времето преход в състоянието:

$$z(tn + 1, k1) = ft(z(tn, ki))$$

Преход в условно състояние:

$$Z(tn, ki + 1) = fk(z(tn, ki))$$

Преход в състояние, зависещо от времето е налице когато стойността на променлива на състоянието се промени в дискретен момент време.

# Събития

Промени на цвета на светофар в зависимост от времето:

Време	Състояние
T = 0	Червено
T = 5	жълто
T = 6	зелено
T = 11	жълто
T = 12	червено
и т.н.	

## Събития

С начална стойност TNext=0 и състояние жълт,  
събитието ще има следната форма:

WHENEVER T>=TNext

DO

    State^ := ‘red’;

    TNext := 12;

END

Времето Т ще присъства винаги в събитие,  
зависещо от времето.

# Събития

Събитието води до ново състояние на модела, което се достига при цикъл 1. Фактът, че новото състояние на модела не става валидно до цикъл 1 е отбелязан със символа ‘^’.

В началото на симулацията, при цикъл 0, имаме начални стойности, които се изменят от зависещо от времето събитие при цикъл 1, получавайки ново състояние за модела.

# Събития

Също при събитие променливата TNext получава нова стойност.

Това осигурява, че условието на събитието още веднъж ще стане истина при  $T = 12$ , така че събитието отново ще се регистрира.

Условните събития се появяват, когато при специален цикъл, състоянието на модела дава възможност за по-нататъшни събития, които водят до ново състояние на модела при следващия цикъл.

# Събития

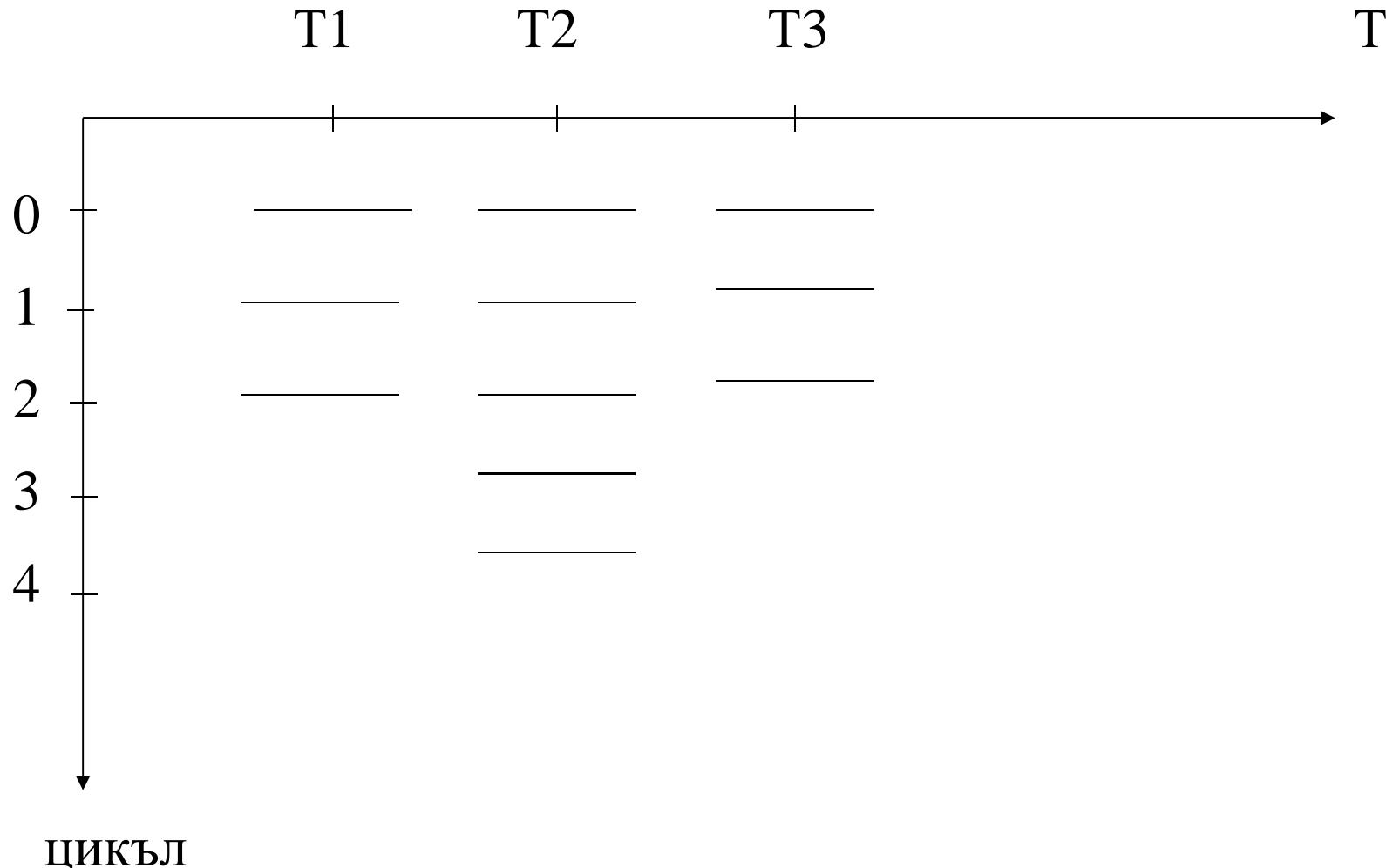
В този случай, условното събитие важи за състоянието на модела при цикъл 1 и го променя в ново състояние в цикъл 2.

Възможно е състоянието на модела при специално време и цикъл да дава възможност на повече от едно събитие.

Всички тези събития започват от оригиналното непроменено състояние. Когато всички събития се изпълнят, цикълът напредва и новото състояние става валидно.

# Събития

Цикли и времеви стъпки за време  $T_1$  – състояния в цикли 0, 1 и 2.  
При цикъл 0 имаме начално състояние.



## Събития

Състоянията водят до цикъл 1. Новото, изменено състояние на модела активира по-нататъшни събития, които водят до състоянието в цикъл 2.

Ако не са възможни повече условни събития, последователността от цикли свършва и симулационния часовник напредва поради следващото зависимо от времето събитие.

# Събития

## Моделът брояч.

Частици пристигат при един Гайгеров брояч с експоненциално разпределение със среден интервал от пет времеви единици. Всеки път, когато се открие частица, броячът се увеличава с единица. Броячът на откритите частици се отпечатва когато е кратен на десет.

# Събития

BASIC COMPONENT Counter

DECLARATION OF ELEMENTS

STATE VARIABLES

Count (INTEGER) := 0,

TNext (REAL) := 0,

NPrint (INTEGER) := 0

RANDOM VARIABLES

Interval (REAL) : EXPO

(Mean:=5,LowLimit:=0.15,UpLimit:=25)

# Събития

## DYNAMIC BEHAVIOUR

```
# Събитие 1
WHENEVER T >= TNext
DO
    Count^ := Count + 1;
    TNext^ := T + Interval;
END
```

```
# Събитие 2
ON ^IMOD(Count, 10) = 0^
DO
    DISPLAY ("Counter = %d \n", Count);
    NPrint^ := NPrint + 1;
END

END OF Counter
```

# Събития

При време  $T = 0$  и  $cycle = 0$  се удовлетворяват началните стойности. Докато симулационния часовник  $T$  е равен на  $T_{Next}$ , действа събитието. Времето на следващия момент  $T_{Next}$  е дефинирано в събитието от генератора на случайни числа, който доставя експоненциално разпределени случайни числа.

В момента  $T = 0$  и  $cycle = 1$  променливите на състоянието имат следните стойности:

Counter = 1

$T_{Next} = T_{Random}$

# Събития

Докато не се изпълни указанието за събитие 2, то събитие 2 не се случва. За време  $T = 0$  са налице само циклите 0 и 1. Докато при време не са възможни по-нататъшни събития, симулационния часовник напредва към следващото време, при което се очаква събитие, зависещо от времето. Това е времето съхранявано в променливата  $T_{Next}$ .

След като пристигне десетата частица се активира второ събитие. Започвайки от нова, увеличена стойност на брояча при цикъл 1, извеждането може да се извърши. В цикъл 2 се съдържа новото състояние на модела с увеличена стойност на  $NPrint$ .

# Събития

## **Моделът тест.**

Моделът Test съдържа три дискретни променливи на състоянията A, B и C и променлива на състоянието TNext за нарастването на времето.

# Събития

BASIC COMPONENT Test

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

A (INTEGER) := 0,

B (INTEGER) := 0,

C (INTEGER) := 0,

TNext (INTEGER) := 0

TRANSITION INDICATORS

End

# Събития

## DYNAMIC BEHAVIOUR

# Събитие 1

WHENEVER T >= TNext

DO

DISPLAY ("A= %d, B= %d, C= %d \n", A, B, C);

TNext<sup>^</sup> := TNext + 1;

A<sup>^</sup> := 2;

END

# Събития

# Събитие 2

ON ^( $A = 2$ )^

DO

DISPLAY ("A= %d, B= %d, C= %d \n", A, B,  
C);

$A^\wedge := 1$ ;

$C^\wedge := 1$ ;

END

# <- Цикли и последващи събития

# Събития

# Събитие 3

ON ^( $A = 2$ ) AND ( $B = 0$ )^

DO

    DISPLAY ("A= %d, B= %d, C= %d \n", A,  
    B, C);

    B^ := A;

END

# Събития

```
# Събитие 4
ON ^ (B = 2) ^
DO
    DISPLAY ("A= %d, B= %d, C= %d \n", A, B,
C);
    A^ := 0;
    B^ := 0;
    C^ := 0;
    SIGNAL End;
END
```

# Събития

# Събитие 5

ON End DO

    DISPLAY ("A= %d, B= %d, C= %d \n", A, B, C);

END

END OF Test

# <- Цикли и последващи събития

# Събития

В началото на симулационното изпълнение при  $T=0$  имаме следното началното състояние:

Takt0			
A	B	C	TNext
0	0	0	0

## Събития

Ако сега проверим петте възможни събития ще видим, че стартиращия механизъм за събитие 1 ще позволи събитие 1 да се осъществи. Това събитие води до ново състояние на модела. Това ново състояние на модела принадлежи на цикъл 1.

# Събития

След първото събитие моделът е в следното състояние:

Takt1			
A	B	C	TNext
2	0	0	1

# Събития

Проверка на събитията показва, че първото събитие е активирало последващите събития 2 и 3. И двете събития започват от състоянието на модела в цикъл 1. Промените на съставните променливи не стават действителни до следващия цикъл.

# Събития

Състоянието на модела при цикъл 2 е:

Takt2			
A	B	C	TNext
1	2	1	1

# Събития

Това състояние при цикъл 2 задейства събитие 4.

При цикъл 3 имаме следното състояние:

Trakt3			
A	B	C	TNext
0	0	0	1

# Събития

Поставя се индикатор END.

След като е поставен указател END, събитие 5 може да продължи в цикъл 4, където крайното състояние на модела се извежда. Няма по-нататъшни действия.

Проверка на събитията показва, че никакви по-нататъшни събития не се задействат при  $T = 0$ . Контролът на изпълнението автоматично ще увеличи времето на симулационния часовник към следващата точка от време, при която могат да се появят събития, в този случай при  $T = 1$ .

# Събития

Също състоянието на модела в предишния момент време при цикъл 0 е валидно в момента  $T = 1$ .

Събитие 1 може да се обработи, базирайки се на това състояние в цикъл 0, след като  $T_{Next} = 1$  и неговото условие е изпълнено.

Чрез избиране обекта Protocol в каталога Protocols, изходът произведен от DISPLAY командите може да се види в прозореца на съдържанието. Те могат също да се изведат още с команда Print от менюто File.

# Събития

Това дава следните резултати:

1 DISPLAY :             $A = 0, B = 0, C = 0$

2 DISPLAY :             $A = 2, B = 0, C = 0$

3 DISPLAY :             $A = 2, B = 0, C = 0$

4 DISPLAY :             $A = 1, B = 2, C = 1$

5 DISPLAY :             $A = 0, B = 0, C = 0$

# Събития

Ред 1 произлиза от събитие 1 и показва оригиналното състояние при цикъл 0. Събитие 1 дава състоянието от цикъл 1. Двете събития 2 и 3 работят върху основното състояние при цикъл 1. По тази причина, редове 2 и 3, които произлизат от събития 2 и 3 са идентични.

Преходите между състоянията описани от събития 2 и 3 оказват ефект при цикъл 2. Събитие 4 така се основава на състоянието на модела, че се извежда в ред 4. Събитие 5 действа върху състоянието в цикъл 3 и генерира ред 5.

# Събития

Най-важните факти могат да се обобщят:

- При всяка точка от време всички събития, които са позволени ще се изпълнят.
- Всички събития, които действат върху състояние на модела при определен цикъл, сами по себе си не променят състоянието на модела. Промените на състоянията, причинени от събития стават валидни при следващия цикъл.
- Тъй като за всички събития важи едно и също състояние на модела, редът по който те се обработват не е важен. Механизмът на цикъла в SIMPLEX MDL води до събития, независещи от реда, в който са описани.

# Събития

Следния пример ще обясни как събитията не влияят на състоянието на модела по време на текущия цикъл, но оказват влияние на следващия цикъл:

Инициализираме променливата на състоянието A с  $A = 3$ .

STATE VARIABLE

DISCRETE

A (INTEGER) : = 3

След това имаме следното събитие:

ON ^ A = 3 ^

DO

    A ^ := 5;

    DISPLAY (“A = % d \ n”, A);

END

## Събития

В цикъла 0 имаме  $A = 3$  и събитието е стартирано. В този цикъл се изпълнява команда DISPLAY. Извеждането показва  $A = 3$ .

По време на същото събитие участва присвояването  $A = 5$ . То не оказва влияние до цикъл 1. Ако е нужно новата стойност да се изведе, тогава команда DISPLAY трябва да се изпълни един цикъл по-късно:

```
ON ^A = 3^  
DO  
    A ^ := 5;  
    SIGNAL Print;  
END  
ON Print  
DO  
    DISPLAY ("A = % d \ n", A);  
END
```

При цикъл 0 са дадени стойности на променливата на състоянието A и индикатора Print. Новите стойности стават валидни. Сега DISPLAY показва изход  $A = 5$ .

# Събития

Особено важно е, че указател за ON израз е валиден само за един цикъл. В моделът Test индикаторът End е поставен в събитие 4 и така става ефективен в следващия цикъл. Това води до стартиране на събитие 5. В края на цикъл 4, индикатора автоматично се нулира от стартирация времеви контрол, така че събитие 5 няма да се изпълнява отново в следващия цикъл.

Събитието се изпълнява в цикъл, в който логическия израз е получил булевата стойност TRUE.

## Събития

Едно събитие ще се изпълни в цикъл  $n$ , ако:

Предишен цикъл  $n-1$ : логическо условие FALSE

Цикъл  $n$ : логическо условие TRUE

Ако по време на последния цикъл стойността на логическото условие още веднъж се промени от FALSE на TRUE, тогава събитието ще се стартира още веднъж.

## Събития

Обратно WHENEVER израз ще доведе до събитие, което се стартира във всеки цикъл, в който логическото условие има стойност TRUE.

С ON израз събитието ще се изпълни само в цикъл 1 и 5, тъй като само тогава стойността се променя от FALSE на TRUE.

Ако се използва WHENEVER израз, тогава събитието се стартира в цикли 1, 2 и 5.  
Следващата таблица демонстрира това:

# Събития

Моделът QueueD

Ще бъде изграден модел на M/M/1 опашка, който се състои от източник, опашка, сървър и контейнер.

# Събития

Източникът генерира идентични задачи през експоненциално разпределени интервали, със средна дължина 15 ВЕ. Така създадените задачи въвеждат опашката.

Сървърът съдържа 1 място, където работите могат да се обработват. Времето за обработка е експоненциално разпределено със средна стойност 10 ВЕ.

След обработката задачите напускат сървъра и отиват в контейнера. Когато задачите се съберат в контейнера, те се унищожават.

# Събития

Моделът QueueD съдържа следните преходи между състояния:

ПРЕХОДИ	УСЛОВИЯ
Генериране на задача	Достигнато е време на пристигане
Задачата напуска опашката, влиза в сървъра и се обработва	Сървърът е свободен и опашката съдържа поне 1 задача
Задачата напуска сървъра и влиза в контейнера	Сървърът е зает и е достигнат края на времето за обработка
4 задачи са унищожени в контейнера	4 задачи са се натрупали в контейнера

# Събития

Тъй като всички работи са идентични е достатъчно да се декларират променливите на състоянията, представлящи броя на задачите в опашката, сървъра и контейнера.

# Събития

BASIC COMPONENT QueueD

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

NQueue (INTEGER) := 0,

NServer (INTEGER) := 0,

NSink (INTEGER) := 0,

TArrive (REAL) := 0,

TWork (REAL) := 0,

Protocol (LOGICAL) := FALSE

# Събития

## RANDOM VARIABLES

Arrive (REAL) : EXPO  
(Mean:=15,LowLimit:=0.5,UpLimit:=75),  
Work (REAL) : EXPO  
(Mean:=10,LowLimit:=0.32,UpLimit:=50)

## DYNAMIC BEHAVIOUR

```
# Създава се работа
WHENEVER T >= TArrive
DO
    NQueue^ := NQueue + 1;
    TArrive^ := T + Arrive;
    IF Protocol
        DO DISPLAY ("T= %f New customer \n",T); END
    END
```

# Събития

```
# Начало на обработката
WHENEVER (NServer = 0) AND (NQueue > 0)
DO
    NQueue^ := NQueue - 1;
    NServer^ := NServer + 1;
    TWork^ := T + Work;
    IF Protocol
        DO DISPLAY ("T= %f Customer enters
server\n",T); END
    END
```

# Събития

```
# Край на обработката
WHENEVER (T >= TWork) AND (NServer = 1)
DO
    NServer^ := 0;
    NSink^ := NSink + 1;
    IF Protocol
        DO DISPLAY ("T= %f Customer leaves
server\n",T); END
    END
```

# Събития

```
# Уничожаване на работа
WHENEVER NSink >= 4
DO
    NSink^ := 0;
    IF Protocol
        DO DISPLAY ("T= %f 4 customers
destroyed\n",T); END
    END
END OF QueueD
```

# Събития

	Време T=0	Време T=4	Време T=5
Цикъл 0	NQueue=0 Nserver=0 NSink=0 TArrive=0 TWork=0	NQueue=0 Nserver=1 NSink=0 TArrive=5 TWork=4	NQueue=0 Nserver=0 NSink=1 TArrive=5 TWork=4
	NQueue=1 Nserver=0 NSink=0 TArrive=5 TWork=0	NQueue=0 Nserver=0 NSink=1 TArrive=5 TWork=4	NQueue=1 Nserver=0 NSink=1 TArrive=17 TWork=4
Цикъл 1	NQueue=0 Nserver=1 NSink=0 TArrive=5 TWork=4		NQueue=0 Nserver=1 NSink=1 TArrive=17 TWork=8
Цикъл 2			

# Събития

Състоянието на модела е определено от началните стойности в момент 0, цикъл 0. Това състояние на модела позволява събитие 1 да се стартира.

Като следствие от събитие 1 получаваме състоянието на модела при цикъл 1, който стартира събитие 2. Това събитие преобразува състоянието на модела от цикъл 1 в ново състояние в цикъл 2.

В момента  $T = 0$  не могат да участват други условни събития. Контролът на изпълнение на Simplex3 търси зависими от времето събития. Събитието със следващото най-малко време е събитие 3, което е изпълнимо при  $T = 4$ . Симулационното време напредва към  $T = 4$ .

# Събития

Състоянието в момента  $T = 4$  и цикъл 0 е идентично на това при  $T = 0$ , цикъл 2. Събитието 3 променя това състояние в ново при цикъл 1.

От момента  $T = 4$  не са възможни по-нататъшни условни събития, симулационното време отива към следващото събитийно време  $T = 5$ , когато събитие 1 може да се изпълни.

# Събития

Първо за цикъл  $K = 0$  се изпълняват събитията, зависещи от времето. Новите стойности на дискретните променливи на състоянието формират новото състояние на модела в цикъла  $K=1$ . От това състояние са възможни няколко последващи събития, ако са изпълнени съответстващите условия. Симулационното време се увеличава автоматично от изпълнението на времевия контрол на Simplex3.

$T_{\text{Arrive}}^{\wedge} := T + \text{Arrive};$

$T_{\text{Work}}^{\wedge} := T + \text{Work};$

# Събития

## Крайни, дискретно-времеви автомати.

Крайните автомати (КА) се характеризират от крайна редица от стойности на променливите на състоянията и зависими променливи, и от преходите между състоянията, които се извършват само в дискретни моменти от време.

Като пример за КА ще разгледаме автомат за продажба на цветя.

Обхватът за входния набор  $X$  се състои от 4 елемента:

$$X = \{x_0, x_1, x_2, x_3\}$$

$$\{\text{nothing}, \text{fCash}, \text{rCash}, \text{fill}\}$$

## Събития

Обхватът за променливата на състоянието Z е набор от 4 елемента:

$$Z = \{z_0, z_1, z_2, z_3\}$$

{0 bunches, 1 bunch, 2 bunches, 3 bunches}

Обхватът на външната променлива Y има следните стойности:

$$Y = \{y_0, y_1, y_2\}$$

{empty, cash, flower}

Локалната функция на преходите на KA може да се опише чрез използване на матрица на преходите.

# Събития

Локална функция на преходите – f

	Z <sub>0</sub>	Z <sub>1</sub>	Z <sub>2</sub>	Z <sub>3</sub>	Функция на автомата
X <sub>0</sub>	Z <sub>0</sub>	Z <sub>1</sub>	Z <sub>2</sub>	Z <sub>3</sub>	Няма пари – старо състояние
X <sub>1</sub>	Z <sub>0</sub>	Z <sub>1</sub>	Z <sub>2</sub>	Z <sub>3</sub>	Неточни пари – старо състояние
X <sub>2</sub>	Z <sub>0</sub>	Z <sub>0</sub>	Z <sub>1</sub>	Z <sub>2</sub>	Точни пари – намаление на броя букети с 1, ако е възможно

# Събития

Локалната функция на преходите определя новото състояние в цикъла  $k_{i+1}$  в момента от време  $t_n$  чрез старото състояние. Имаме:

$$z(t_n, k_{i+1}) = f(z(t_n, k_i), x(t_n, k_i))$$

Това означава, че машината, която е в състояние  $z_2$  при цикъл  $k_i$  въвежда състояние  $z_1$ , когато входът е равен на  $x_2$ . Новото състояние  $z_1$  е въведено в цикъл  $k_{i+1}$ .

$z2$ : Съдържание на машината 2 букета

$x2$ : Вход коректни пари

$z1$ : Съдържание на машината 1 букет

# Събития

Изходната функция  $g$  описва изхода на машината в състояние  $z$  с вход  $x$  в цикъла  $k_i$ .

Може да се види, че входа и изхода са поставени в един и същи цикъл. Преходът на състоянията за  $z$  се случва един цикъл по-късно. Имаме:

$$Y(t_n, k_i) = g(z(t_n, k_i), x(t_n, k_i))$$

# Събития

Изходна функция  $g$ :

	Z <sub>0</sub>	Z <sub>1</sub>	Z <sub>2</sub>	Z <sub>3</sub>	Поведение на автомата
X <sub>0</sub>	Y <sub>0</sub>	Y <sub>0</sub>	Y <sub>0</sub>	Y <sub>0</sub>	Няма пари – няма изход
X <sub>1</sub>	Y <sub>1</sub>	Y <sub>1</sub>	Y <sub>1</sub>	Y <sub>1</sub>	Неточни пари – връщане на пари
X <sub>2</sub>	Y <sub>1</sub>	Y <sub>2</sub>	Y <sub>2</sub>	Y <sub>2</sub>	Точни пари – 1 букет, ако е възможно
X <sub>3</sub>	Y <sub>0</sub>	Y <sub>0</sub>	Y <sub>0</sub>	Y <sub>0</sub>	Ново зареждане – няма изход

# Събития

При цикъл  $k_0$  е налице началното състояние. В цикъла  $k_1$  са известни входът  $X$  и изходът  $Y$ , които са базирани на старото състояние.

След обработване на входа и изпълнение на изхода в следващия цикъл 2 се определя новата стойност на променливата на състоянието.

# Събития

BASIC COMPONENT Vending

## LOCAL DEFINITIONS

VALUE SET Input: ('nothing', 'f\_Cash', 'r\_Cash', 'fill')

VALUE SET Output: ('empty', 'Cash', 'Flowers')

## DECLARATION OF ELEMENTS

### STATE VARIABLES

X (Input) := 'nothing', # Вход

Z (INTEGER) := 3, # Състояние

TStep (INTEGER) := 0 # Напредване на времето

# Събития

## DEPENDENT VARIABLES

Y (Output) := 'empty' # Изход

## RANDOM VARIABLES

Rand (INTEGER) : IUNIFORM (LowLimit := 0, UpLimit :=4)

## TRANSITION INDICATORS

StateCh, # Промяна на състояние

Print # Отпечатване на ново състояние

# Събития

DYNAMIC BEHAVIOUR

WHENEVER T >= TStep

DO DISPLAY("TStep %d \n", TStep);

DISPLAY("Old state: Number of bunches = %d \n", Z);

IF Rand = 0 # случаен вход

DO X^ := 'nothing'; END

ELSIF Rand = 1

DO X^ := 'f\_Cash'; END

ELSIF Rand = 2

DO X^ := 'r\_Cash'; END

ELSIF Rand = 3

DO X^ := 'r\_Cash'; END

ELSE

DO X^ := 'fill'; END

SIGNAL StateCh;

TStep^ := TStep + 1;

END

# Събития

```
# Промяна на състоянието
ON StateCh
DO
    DISPLAY ("Input X = %d  Output Y = %d \n", X, Y);
    IF X = 'fill'
        DO
            Z^ := 3;
        END
    ELSIF X = 'r_Cash'
        DO
            IF Z > 0
                DO Z^ := Z - 1; END
            END
        SIGNAL Print;
    END
END
```

# Събития

# Алгебрични уравнения за изход

IF X = 'fill'

DO Y := 'empty'; END

ELSIF X = 'f\_Cash'

DO Y := 'Cash'; END

ELSIF X = 'r\_Cash'

DO

IF Z > 0

DO Y := 'Flowers'; END

ELSE

DO Y := 'Cash'; END    END

# Събития

ELSE

DO Y := 'empty'; END

# Отпечатване на новото състояние

ON Print

DO

DISPLAY ("New State: Number of bunches Z =  
%d \n\n", Z);

END

END OF Vending

# Събития

Началните стойности при  $T = 0$  и цикъл 0 са :

$X = \text{'nothing'}$ ,  $z = 3$ ,  $y = \text{'empty'}$

Състоянието на модела се определя изцяло от променливите на състоянията  $x$  и  $z$ . Новото състояние при всяка точка от време се определя от предишното състояние чрез локалната функция на преходите  $f$ .

Във всеки момент от време съставните променливи при цикъл 0 са непроменени. В следващия цикъл 1 променливата  $X$  има съответна стойност.

# Събития

Променливата на състоянието z получава новата си стойност в цикъл 2.

Изходът у произтича от състоянията на променливите x и z. Затова у е зависима променлива. Всеки път, когато x и z променят стойностите си контролът на изпълнението в Simplex3 осигурява, че зависимата променлива в изходната функция g се преизчислява.

# Събития

Може да се види, че във всеки момент от време взима участие пълна процедура, състояща се от вход, изход и промяна на състояние. След като всички промени на условията на състоянието се изпълняват, времето се придвижва напред и започва нова процедура.

Изходите, създадени от DISPLAY командата могат да се проверят чрез избиране на обект Protocol в каталога Protocols в прозореца на съдържанията. Може да се изведе чрез командата Print от менюто File.

Забележка: В протокола се записват стойностите на VALUE SET променливите под формата на техните вътрешни числени представления, дефинирани от потребителя, т.е.  $y = 2$  вместо  $y = \text{'flowers'}$ .

# Събития

Следващият пример показва първите 5 времеви  
стъпки на симулационното изпълнение:

>> T=0.000 : Start of simulation

DISPLAY: Tstep 0

DISPLAY: Old state: Number of bunches = 3

DISPLAY: Input X= 2 Output Y =2

DISPLAY: New State: Number of bunches Z = 2

# Събития

DISPLAY: Tstep 1

DISPLAY: Old state: Number of bunches = 2

DISPLAY: Input X = 2 Output Y =2

DISPLAY: New State: Number of bunches Z =1

DISPLAY: Tstep 2

DISPLAY: Old state: Number of bunches = 1

DISPLAY: Input X = 1 Output Y =1

DISPLAY: New State: Number of bunches Z =1

# Събития

DISPLAY: Tstep 3

DISPLAY: Old state: Number of bunches = 1

DISPLAY: Input X = 3 Output Y =0

DISPLAY: New State: Number of bunches Z =3

DISPLAY: Tstep 4

DISPLAY: Old state: Number of bunches = 3

DISPLAY: Input X = 2 Output Y =2

DISPLAY: New State: Number of bunches Z =2

## Събития

DISPLAY: Tstep 5

DISPLAY: Old state: Number of bunches = 2

DISPLAY: Input X = 2 Output Y =2

DISPLAY: New State: Number of bunches Z =1

>> T = 5.000 : End time reached

Забележка:

Циклите гарантират независимостта на реда на събитията. Това означава, че събитията могат да се напишат в произволен ред и да се разпределят произволно между компонентите.

# Събития

Разделяне на пространството на състоянията.

Налице са следните възможности за описание на  
синтаксиса за динамично поведение

statement ::= algebraic\_equation

- | differential\_equations
- | region\_defining\_statement
- | event\_defining\_statement

Като допълнение на алгебричните уравнения,  
диференциалните уравнения и събития има езикови  
конструкции, разделящи пространството на  
състоянията. Разделянето на пространството на  
състоянията означава определяне на различни  
динамики, зависещи от състоянието на модела.

# Събития

Ключовите думи IF, ELSEIF и ELSE се използват, за да специфицират секциите с различни динамики.

Да разгледаме модела Plane, който описва движенията на топка върху две равнини под ъгъл една с друга.

Състоянието на модела в този случай е описано от двете съставни променливи  $x$  и  $v$ . Пространството на състоянията е двумерно.

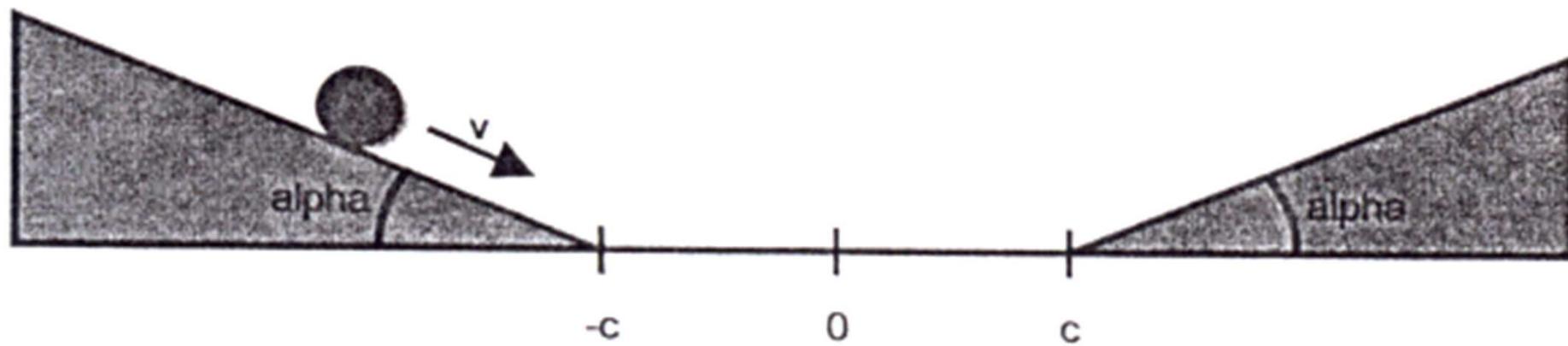
Трите условия, разделящи пространството на състоянията на 3 части:

$$x < -c$$

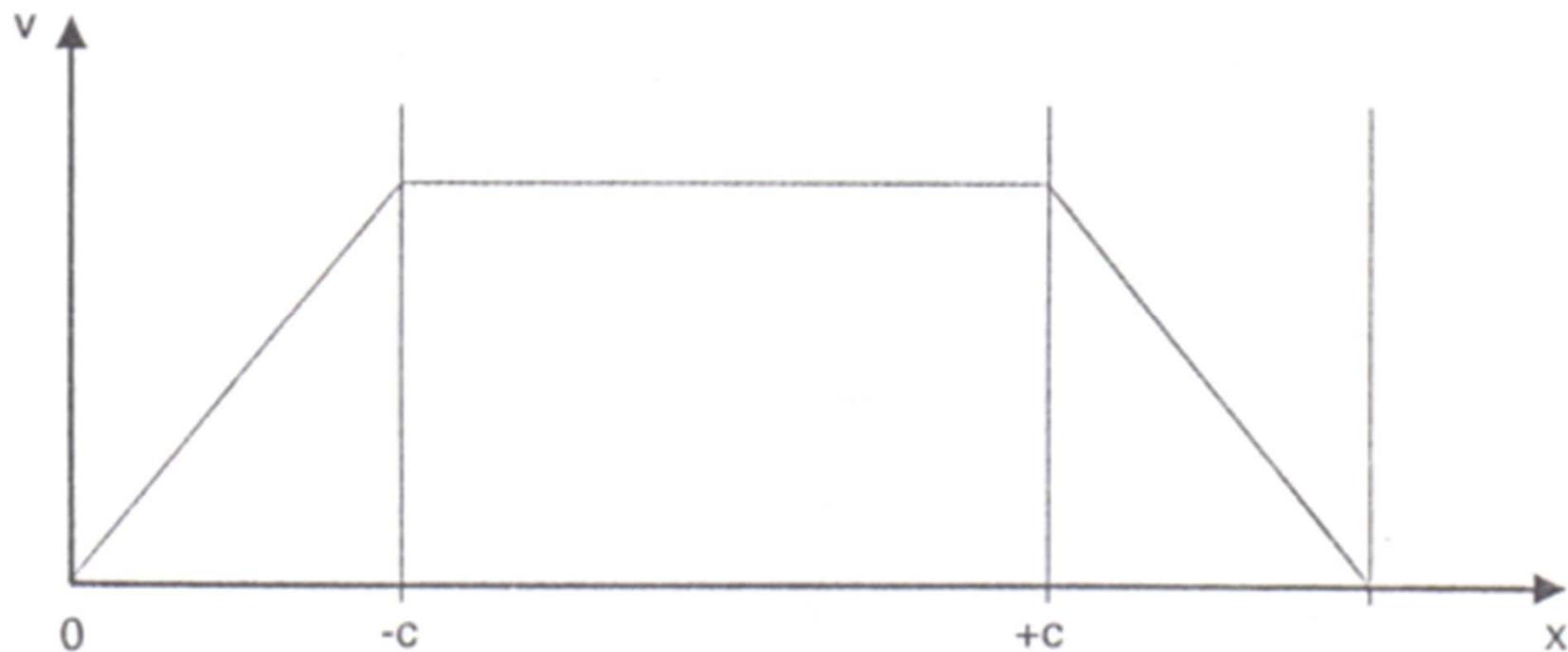
$$x > c$$

$$-c \leq x \leq c$$

# Събития



# Събития



# Събития

BASIC COMPONENT Plane

USE OF UNITS

TIMEUNIT = [s]

DECLARATION OF ELEMENTS

CONSTANTS

g (REAL[m/s<sup>2</sup>]) := 9.81 [m/s<sup>2</sup>], # гравитационно ускорение

alpha (REAL) := 0.52, # ъгъл на наклон

c (REAL[m]) := 1.5 [m] # хоризонтална секция

STATE VARIABLES

CONTINUOUS

x (REAL[m]) := -5 [m], # x-координата

v (REAL[m/s]) := 0 [m/s] # скорост

# Събития

## DYNAMIC BEHAVIOUR

# ляво-наклонена равнина

IF x < -c

DO

## DIFFERENTIAL EQUATIONS

v' := g \* SIN(alpha);

x' := v \* COS(alpha);

END

END

# Събития

```
# дясното-наклонена развина  
ELSIF x > c  
DO  
    DIFFERENTIAL EQUATIONS  
        v' := -g * SIN(alpha);  
        x' := v * COS(alpha);  
    END  
END
```

```
# хоризонтална равнина  
ELSE  
DO  
    DIFFERENTIAL EQUATIONS  
        v' := 0 [m/s^2];  
        x' := v;  
    END  
END
```

```
END OF Plane
```

# Събития

## Синтаксиса е:

**dynamicBehaviour** ::= DYNAMIC BEHAVIOUR  
                          statementSequence

`statement_sequence ::= statement { statement }`

statement ::= algebraic\_equation

## | differential\_equations

| region\_defining\_statement

| event\_defining\_statement

**region\_defining\_statement** ::= IF expression

**DO** statement\_sequence **END**

{ELSEIF expression

DO statement\_sequence END }

[ELSE expression

DO statement\_sequence END]

# Събития

Region\_defining\_statement може да съдържа алгебрични уравнения, диференциални уравнения, събития и разделяния на пространството на състоянията.

Моделът Vending е пример за разделяне пространството на състоянията с алгебрични уравнения. В зависимост от променливата на състоянието x се определя зависимата променлива y:

```
IF X = 'fill'  
    DO Y := 'empty'; END  
ELSEIF X = 'f_Cash'  
    DO Y := 'Cash'; END  
ELSEIF X = 'r_Cash'  
    DO  
        IF Z > 0  
            DO Y := 'Flowers'; END  
        ELSE  
            DO Y := 'Cash'; END  
        END  
    ELSE  
        DO Y := 'empty'; END
```

# Събития

Масиви.

Всички количества могат да се дефинират като масиви до три измерения. Всяка стойност е достъпна чрез един или повече индекси. Индексите започват от 1 и стигат до стойността, определена в декларацията. Най-големият индекс е броя на елементите на масива в съответното измерение.

Пример:

ARRAY [3] [5] X (Real) # двумерен масив с 15 елемента

# Събития

Индекси.

Всеки индивидуален елемент се идентифицира чрез индекс или списък от индекси.

Синтаксис:

Index\_identifier ::= [ index\_or\_index\_set  
                      { index\_or\_index\_set } ]

index\_or\_index\_set ::= ['expression']  
                      | '{ index\_set }'

index\_set ::= basis\_set ['|' expression ]

basis\_set ::= index\_range  
              | identifier OF index\_range  
              | ALL  
              | ALL identifier

Index\_range ::= unsigned\_integer  
                  unsigned\_integer '..' unsigned\_integer

unsigned\_integer ::= unsigned\_number  
                  | identifier

## Събития

Индивидуален индексен израз се затваря в квадратни скоби, а набор от индекси във фигурни скоби.

Индексен израз е числов израз от тип INTEGER.

Набор от индекси отбелязва непрекъсната област от цели числа. Тази област може да се ограничи от условие, което следва след символа '|'.

Непрекъсната област от индексен набор може да се даде чрез долна и горна граници или с ключова дума ALL, която отбелязва всички възможни индекси на идентификатора в съответното измерение.

Идентификаторът може да се използва заедно с индексния набор за представяне на всички индивидуални индекси.

# Събития

Моделни величини са:

- Константи;
- Случайни променливи;
- Зависими променливи;
- Сензорни променливи.

Всички моделни величини могат да се представят  
чрез масиви.

# Събития

Примери:

Елемент на масив се идентифицира чрез индекса си

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

ARRAY [3] X (INTEGER) := 0,

ARRAY [3] [5] Y (INTEGER) := 1

DYNAMIC BEHAVIOUR

WHENEVER X[1] >= 0

DO

X[1]^ := 2;

Y[1] [1]^ := 2;

END

# Събития

Вместо индекс може да се използва израз, за да се оцени индекса

DECLARATION OF ELEMENTS

CONSTANTS

k (INTEGER) := 1

STATE VARIABLES

DISCRETE

ARRAY [3] X (INTEGER) := 0

DEPENDENT VARIABLES

DISCRETE

ARRAY [3] [5] Y (INTEGER)

DYNAMIC BEHAVIOUR

Y [k] [k+1] := X[2\*k+1];

WHENEVER X[1] >= 0

DO

X[k]^ := 5;

END

Израза за оценяване на индекс трябва да даде INTEGER стойност. Излизане  
извън границите на масива води до грешка при изпълнението.

# Събития

Вместо отделен индекс може да се използва набор от индекси.

Промяната на състоянието се извършва за всички индекси:

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

ARRAY [6] X(INTEGER) := 0,

ARRAY [3] [5] Y(INTEGER) := 0

DYNAMIC BEHAVIOUR

WHENEVER X[1] >= 0

DO

X {1..3}^ := 1;

X{4..6}^ := 2;

Y{1..2} {3..5}^ := 3;

END

# Събития

Ключовата дума ALL отбелязва всички възможни индекси

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

ARRAY[5] X (INTEGER) := 0;

ARRAY [3] [5] Y (INTEGER) := 1

DYNAMIC BEHAVIOUR

WHENEVER X[1] >= 0

DO

X {ALL}^ := Y {ALL} [1];

END

В това събитие се извършват 3 присвоявания, като се препокриват три елемента от вектора X.

# Събития

Представя се идентификатор на индексите. Този индекс може да се използва при заявлението:

DECLARATION OF ELEMENTS

STATE VARIABLES

CONTINUOUS

ARRAY [5] X (REAL) := 0,

ARRAY [3] [5] Y (REAL) := 1

DYNAMIC BEHAVIOUR

DIFFERENTIAL EQUATIONS

X {i OF 1..2}' := 3 \* i;

Y {ALL} {Index OF 1..3}' := X [Index];

Y {ALL} {Index OF 4..5}' := X [Index] + 1;

END

Някои елементи на масивите X и Y са дефинирани като диференциални уравнения. Идентификаторите i и присъстват в дясната страна на диференциалното уравнение.

# Събития

Пълният набор от индекси може да се отбележи също чрез  
ключовата дума ALL

DECLARATION OF ELEMENTS

STATE VARIABLES

CONTINUOUS

ARRAY [5] X (REAL) := 0,

ARRAY [5] [3] Y (REAL) := 1

DYNAMIC BEHAVIOUR

DIFFERENTIAL EQUATIONS

X {ALL j}': = j \* X {j};

Y {k} {ALL k}': = k \* X[k];

END

Диференциалните уравнения се дефинират за всички елементи от масива X. В масива Y са засегнати елементите Y[k] [1], Y[k] [2] и Y[k] [3].

# Събития

Възможно е да се ограничи индексния набор. Засягат се само онези индекси, за които логическият израз има стойност TRUE.

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

ARRAY [5] X (INTEGER) := 0,

ARRAY [5] [3] Y (INTEGER) := 1

DYNAMIC BEHAVIOUR

WHENEVER X [1] = 0

DO

X {ALL i | Y [i] {1} > 0}^ := 1;

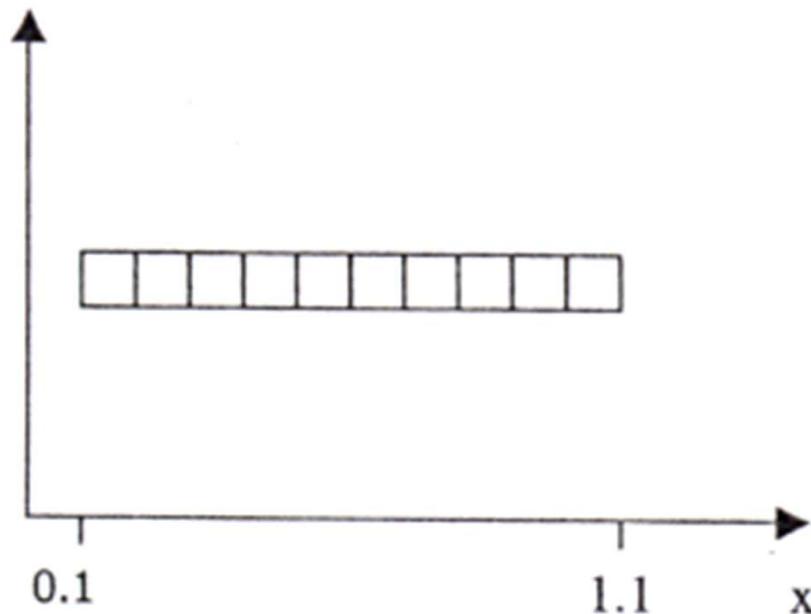
END

# Събития

Параболични частни диференциални уравнения  
Избираме уравнението на топлопроводимостта  
за хомогенна пръчка като пример за  
параболично диференциално уравнение.  
Прието е, че страните на пръчката са  
изолирани. Промяна на топлинната енергия се  
регистрира само в двата края на пръчката.

# Събития

Фигурата показва тънка пръчка с дължина 1, чиито краища са разположени върху оста X на 0.1 и 1.1. Страните на пръчката са изолирани, така че не се обменя никаква енергия с околната среда. Това означава, че топлина може да се движи само в пространството X. Затова това е едномерен проблем.



# Събития

Левият край на пръчката при  $x = 0.1$  се пази на константната температура от  $0^0 \text{ C}$ . В началото – цялата пръчка има температура  $u = 0$ . В момента  $T = 0$  температурата на десния край при  $x = 1.1$  е  $u = 2$ .

Очаква се, че пръчката се затопля бавно и температурата накрая ще достигне устойчиво състояние, в което левият край има температура  $u = 0^0$  и десния край има температура  $u = 20$ . В момента  $T = 10$  температурата все още е далеч от равновесие. В момента  $T = 500$  е налице гладко линейно разпределение.

# Събития

Температурата може да се наблюдава в една избрана точка от пръчката по време на симулацията.

## Събития

Топлинният обмен е описан от следните частни диференциални уравнения:

$$\frac{\partial u}{\partial t} = D \cdot \frac{\partial^2 u}{\partial x^2}$$

Това уравнение е частен случай от по-общото уравнение.

$$\frac{\partial u}{\partial t} = g\left(t, u, \frac{\partial u}{\partial x}, \frac{\partial^2 u}{\partial x^2}\right)$$

# Събития

Промяната по всяко време  $t$  зависи от диференциалните коефициенти

$$\frac{\partial u}{\partial x}, \frac{\partial^2 u}{\partial x^2}$$

Двата диференциални коефициента могат да се приближат чрез разлики.

За да направим това разделяме  $X$  направлението на  $n$  интервала с дължина  $k$ . Триточкова апроксимация се използва за всяка точка от мрежата.

# Събития

Диференциален коефициент от първи ред:

$$\left( \frac{\partial u}{\partial x} \right)_i = \frac{u_{i+1} - u_{i-1}}{2k} \quad i \neq 1, n$$

За лявата (дясната) граница имаме:

$$\left( \frac{\partial u}{\partial x} \right)_1 = \frac{-3u_1 + 4u_2 - u_3}{2k}$$

$$\left( \frac{\partial u}{\partial x} \right)_n = \frac{u_{n-2} - 4u_{n-1} + 3u_n}{2k}$$

# Събития

Диференциален коефициент от втори ред:

$$\left( \frac{\partial^2 u}{\partial x^2} \right)_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{k^2} \quad i \neq 1, n$$

# Събития

Където с  $u_i$  е отбелаязана стойността на функцията в момента време  $t_j$  за  $x$ -координатата  $x_i$ :

$$u_i = f(x_i, t_j)$$

Приемаме следните начални условия за топлинния трансфер:

$$u(x, t) = 0 \quad 0.1 \leq x < 1.1 \quad t = 0$$

$$u(x, t) = 2 \quad x = 1.1 \quad t = 0$$

В момента време  $t = 0$  температурата  $u$  е равна на нула в цялата пръчка. Само на десния край при  $x = 1.1$  имаме  $u = 2$ .

## Събития

Приемаме за граничните условия, че температурата в краищата на пръчката е константна през време:

$$u(x, t) = 0 \quad x = 0, t \geq 0$$

$$u(x, t) = 2 \quad x = 1, t \geq 0$$

Ще изучим промените на температурата в пръчката при дадените начални и гранични условия.

Разделяме пръчката на 10 интервала с дължина 0.1. Получаваме 11 координатни точки.

За всяка точка имаме обикновено диференциално уравнение от първи ред:

# Събития

$$\frac{du_i}{dt} = D \cdot \left( \frac{\partial^2 u}{\partial x^2} \right)_i$$

## Събития

За което се изпълнява:  $D = 0.001 \text{ [m}^2/\text{min}]$

Диференциалния коефициент  $\frac{\partial^2 u}{\partial x^2}$  ще бъде

приближен чрез метод описан по-нататък.

Желаем да изчислим промяната на температурата на 11 различни региона чрез диференциално уравнение от първи ред.

# Събития

Имаме следната процедура:

- Изчисляваме диференциалния коефициент:

$$\frac{\partial^2 u}{\partial x^2}$$

базиран на състоянието на модела в момента  $t$ .

# Събития

Първо се оценява диференциалният коефициент

$$\frac{\partial^2 u}{\partial x^2}$$

за всяка точка  $i$  от мрежата чрез приближение с три точки. Това е записано във вектора  $u_2$  по следния начин:

# Събития

$$u_{-2[1]} = \frac{\partial^2 u(x, t)}{\partial x^2}, \quad x = 0.1$$

$$u_{-2[2]} = \frac{\partial^2 u(x, t)}{\partial x^2}, \quad x = 0.2$$

$$u_{-2[11]} = \frac{\partial^2 u(x, t)}{\partial x^2}, \quad x = 1.1$$

# Събития

С така определените диференциални коефициенти

$$\frac{\partial^2 u}{\partial x^2}$$

може да се опише диференциално уравнение.

От описанието на модела се вижда, че всички точки от мрежата са инициализирани с нула.

Първата промяна на състояние е чрез събитие, в което са поставени началните стойности

$$u(x,t) = 2 \text{ за } x = 1.1 \text{ t} = 0.$$

# Събития

- Единадесетте диференциални коефициента

$$\left( \frac{\partial^2 u}{\partial x^2} \right)_i$$

$i = 1..11$

са дефинирани от стойността на променливата на състоянието  $u$ . Те ще се изчислят чрез алгебрични уравнения. Отделно се третират граничните условия при  $x = 0.1$  и  $x=1.1$ .

# Събития

Втората частна производна от интервала  $u_2[i]$  може да се използва, за да определи диференциалния коефициент  $u[i]'$ .

Забележка:

Стабилността на численото интегриране на частните диференциални уравнения изисква повече грижи. За уравнението на топлопроводимостта стабилността зависи от следната стойност:

$$\alpha = \frac{D \cdot \Delta t}{\Delta x^2}$$

$$\alpha < 0.1$$

където трябва да е изпълнено

# Събития

BASIC COMPONENT Rod

USE OF UNITS

TIMEUNIT = [min]

DECLARATION OF ELEMENTS

CONSTANTS

D (REAL[m^2/min]) := 1.0 E-3 [m^2/min]

# Събития

STATE VARIABLES

CONTINUOUS

ARRAY[11] u (REAL[Cel]) := 0 [Cel]

# температура в точка x

DEPENDENT VARIABLES

CONTINUOUS

ARRAY[11] u\_2 (REAL[Cel/m^2]) := 0 [Cel/m^2]

# 2ра производна в точка x

# Събития

## DYNAMIC BEHAVIOUR

```
u_2 [1]      :=  
(u[3] - 2*u [2] + u [1] ) / (POW(0.1, 2) * 1[m^2]);  
u_2 [11]     :=  
(u[11] - 2*u[10] + u [9] ) / (POW(0.1, 2) * 1[m^2]);  
u_2 {i OF 2..10} :=  
(u[i+1] - 2*u [i] + u[i-1]) / (POW(0.1, 2) * 1[m^2]);
```

# Събития

ON START

DO

  u[11]^ := 2 [Cel];

END

DIFFERENTIAL EQUATIONS

  u {i OF 1..10}' := D \* u\_2[i];

  u [1]'       := 0 [Cel/min];

  u [11]'       := 0 [Cel/min];

END

END OF Rod

## Събития

Могат да се използват стойностите по подразбиране на контролните параметри, за да се изпълни симулацията на модела Rod.

В примера симулацията се изпълнява до  $T = 500$ . В прозореца Analyse and present функцията TimeCut се избира заедно с подходяща точка от време. Динамичните редове Obs#/Rod/u[10] и Obs1#/Rod/u[11] се поставят на края на списъка.

# ЛЕКЦИЯ 8

## РЕГИОНИ И ПОДВИЖНИ КОМПОНЕНТИ

-  **Модела Опашка**
-  **Транспортни модели**
-  **Стратегии**

# **ВЪВЕДЕНИЕ**

**Елементите на модел, които дефинират състояние на основните компоненти могат да се разделят на следните 4 групи:**

- Моделни променливи;**
- Случайни променливи;**
- Индикатори;**
- Региони.**

# ВЪВЕДЕНИЕ

Регионите са места за съхранение на подвижни компоненти. Подвижна компонента е компонента, която има декларационна част, но няма част на динамичното поведение. Свойствата на подвижна компонента могат да се променят само външно от основна компонента.

Примери за подвижни компоненти:

- Клиенти;
- Обработвани детайли;
- Палети;
- Превозни средства;
- Задачи;
- Съобщения.

# ВЪВЕДЕНИЕ

Регионът може да съдържа една или повече подвижни компоненти. Компонентите се подреждат по специфициран критерий (първи влязъл – пръв излязъл, последен влязъл – първи излязъл и т.н.). По този начин регионът може да се използва в модела, за да моделира обикновено обслужване и опашка. Simplex3 прави статистика върху съдържанието на регион, което е достъпно за потребителя.

Пример за региони:

- Буфери;
- Съхраняващи области;
- Обработващи региони;
- Маршрутни участъци за превозни средства;
- Места за сядане в превозни средства;
- Опашки за клиенти;
- Пощенски кутии за съобщения.

# ВЪВЕДЕНИЕ

Преместването на подвижните компоненти и изменението на атрибутите им е възможно само в събития, специфицирани в основните компоненти.

Подвижните компоненти могат да се създават, местят и унищожават, и техните свойства могат да се модифицират.

Примери:

- Регионът с име Queue получават нова подвижна компонента от класа Customer.

Queue<sup>^</sup> : ADD 1 New Customer; #създаване

- Подвижна компонента от клас Customer се изтрива от региона, именуван Server и се унищожава. В този случай компонентата се поставя на второ място.

Server<sup>^</sup> : REMOVE Customer [2]; #унищожаване

# ВЪВЕДЕНИЕ

- Първата подвижна компонента, която е в регион Queue се премества в регион Server и се нарежда в опашката.

Server<sup>^</sup> : FROM Queue GET Customer [1]; #преместване

- Първата подвижна компонента от регион Queue се изпраща в регион Server

Queue<sup>^</sup> : TO Server SEND Customer [1]; # преместване

- Всички компоненти от клас Customer имат атрибут, наречен Tdone. Тук се присвоява крайното време за обработка. Атрибутът Tdone на първата компонента в регион Server получава стойност  $T + 10$ .

Server : Customer [1].Tdone<sup>^</sup> :=  $T + 10$ ; #промяна на атрибут

# ВЪВЕДЕНИЕ

- Когато първата подвижна компонента се премести от регион Queue в регион Server, нейните атрибути TDone получават нова стойност.

Server<sup>^</sup> : FROM Queue GET Customer [1]

CHANGING

TDone<sup>^</sup> := T + Tserver; #промяна на атрибут

END

- Всички подвижни компоненти, които са в регион Server се унищожават и се заменят от 3 нови подвижни компоненти.

Server<sup>^</sup> : PLACE 3 New Customer; #унищожаване и поставяне на нова стойност

# Модела Опашка

Как се декларират регионите и как се  
създават подвижните компоненти?

Какви стратегии за подреждане в опашка  
са популярни?

# Модела Опашка

Сървъри и опашки.

Ще създадем модел на M/M/1 опашка, състоящ се от източник, опашка, сървър и контейнер. Разликата се състои във факта, че в текущия модел Queue задачите имат атрибути и затова трябва да се моделират чрез подвижни компоненти. Това не беше необходимо в модела QueueD. Тъй като всички задачи в модела QueueD бяха идентични беше достатъчно просто да се приложи брояч, който съответно беше увеличаван и намаляван.

# Модела Опашка

- Източника генерира задачи през експоненциално разпределени интервали време със средна стойност 15 времеви единици. Тези задачи са разпределени в 3 равномерно разпределени приоритетни класа 1, 2 и 3, като приоритет 3 е най-висок.
- Така интегрираните задачи влизат в опашката чрез стратегията PFIFO. Това означава, че те са сортирани в намаляващ ред на приоритет. Задачите с равен приоритет са подредени съобразно тяхното време на пристигане, т.е. чрез стратегията FIFO.
- Сървърът има 1 обслужваща станция, която може да обработва само една задача в един момент. Обслужващото време е експоненциално разпределено със средна стойност 10 времеви единици.

# Модела Опашка

BASIC COMPONENT Queue1

MOBILE SUBCOMPONENTS OF CLASS Customer1

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

TArrive (REAL) := 0,

TWork (REAL) := 0,

Protocol (LOGICAL) := FALSE

# Модела Опашка

## RANDOM VARIABLES

Arrive (REAL) : EXPO (Mean := 15),  
Work (REAL) : EXPO (Mean := 10),  
Prio (INTEGER) : IUNIFORM (LowLimit:=1, UpLimit:=3)

## LOCATIONS

WaitP (Customer1 ORDERED BY DEC Priority) := 0  
Customer1,  
Station (Customer1) := 0 Customer1,  
Sink (Customer1) := 0 Customer1

END OF Queue1

# Модела Опашка

DYNAMIC BEHAVIOUR

# Създаване на задача

WHENEVER T >= TArrive

DO

WaitP<sup>^</sup> : ADD 1 NEW Customer1

CHANGING

Priority<sup>^</sup> := Prio;

END

TArrive<sup>^</sup> := T + Arrive;

IF Protocol

DO DISPLAY ("T= %f New Customer \n",T); END

END

# Модела Опашка

```
# Начало на обработката
WHENEVER (NUMBER(Station)=0) AND
  (NUMBER(WaitP)>0)
DO
  Station^ : FROM WaitP GET Customer1[1];
  TWork^ := T + Work;
  IF Protocol
    DO DISPLAY ("T= %f Customer enters Station
\n",T); END
  END
```

# Модела Опашка

```
# Край на обработката
WHENEVER (T >= TWork) AND
  (NUMBER(Station)=1)
DO
  Station^ : TO Sink SEND Customer1[1];
  IF Protocol
    DO DISPLAY ("T= %f Customer leaves Station
      \n",T); END
  END
```

# Модела Опашка

```
# Унищожаване на задача  
WHENEVER NUMBER(Sink) >= 4  
DO  
    Sink^ : REMOVE Customer1{ALL};  
END  
  
END OF Queue1
```

# Модела Опашка

След обслужване задачите напускат обслугващия пункт и влизат в контейнера. Когато контейнерът съдържа 4 задачи те се унищожават.

Моделът Queue съдържа следните преходи между състояния:

# Модела Опашка

Преход между състояния	Условие
Създаване на задача	Достигнато е времето на пристигане
Задача се премества от опашката към обслугващия пункт	Станцията е празна и опашката съдържа поне 1 задача
Започва обработката на работата	
Задача се изтрива от обслугващия пункт	Станцията се заема и времето за обслужване приключва
4 задачи са унищожени	Контейнерът съдържа 4 задачи

# **Модела Опашка**

**Подвижната компонента Customer1**

**MOBILE COMPONENT Customer1**

**DECLARATION OF ELEMENTS**

**STATE VARIABLES**

**DISCRETE**

**Priority (INTEGER) := 1 # Priority**

**END OF Customer1**

# Модела Опашка

## Деклариране на региони.

Регионите принадлежат на елементите на модела. Те трябва да се декларират в секцията DECLARATION OF ELEMENTS във всички основни компоненти, в които се появяват.

```
declarations_of_elements ::= DECLARATION OF ELEMENTS  
                           [list_of_constants]  
                           [list_of_state_variables]  
                           [list_of_dependent_variables]  
                           [list_of_sensor_variables]  
                           [list_of_random_variables]  
                           [list_of_transition_indicators]  
                           [list_of_sensor_indicators]  
                           [list_of_locations]
```

# Модела Опашка

[list\_of\_sensor\_locations]

list\_of\_locations ::= LOCATION[S]

                  location { ‘,’ location }

location ::= dim\_identifier ‘(‘ identifier

                  [ordering\_criteria\_list] ‘)’

                  ‘:=’ unsigned\_number     identifier

ordering\_criteria\_list ::= ORDERED BY ordering\_criterion

                  { ‘,’ ordering\_criterion }

ordering\_criterion ::= FIFO | LIFO

                  | INC identifier

                  |;DEC identifier

# Модела Опашка

Всеки регион си има име.

След това се дава името на класа на подвижните компоненти, които региона може да съдържа. Следва списък на възможностите на една или повече стратегии за подреждане.

Възможните стратегии за подреждане са:

FIFO: Първи добавен – първи излиза;

LIFO: Последен добавен – първи излиза;

INC идентификатор: нарастващ атрибут;

DEC идентификатор: намаляващ атрибут.

Ако не е дадена стратегия за подреждане, тогава се приема FIFO.

# Модела Опашка

Критериите INC и DEC се отнасят за атрибути на подвижни компоненти, които ще бъдат намерени в региона.

Ако първия критерий не дава уникално място за пристигане на подвижната компонента, тогава се използва следващия критерий в списъка, за да стесни възможностите. Списъкът може да съдържа до пет критерия. Ако дори след прилагането на петия критерий няма уникално решение, тогава се използва FIFO.

Критериите FIFO и LIFO могат да се появят само в края на списъка.

Подобно на променливите на състоянията, регионите трябва да се инициализират. За да се постигне това се използва присвояващия оператор, последван от броя на подвижните компоненти. Този брой може да е нула.

# Модела Опашка

Примери:

- Регионът Server може да съхранява подвижни компоненти на клас Customer и се инициализира с един Customer.

LOCATION Server (Customer) : = 1 Customer

- Регионът Queue съхранява подвижните компоненти на клас Customer в намаляващ ред на атрибута си Priority. Подвижните компоненти с по-висок приоритет се поместват по-напред в опашката. Ако двама или повече клиенти имат еднакъв приоритет те се сортират според FIFO критерия.

LOCATION Queue (Customer ORDERED BY DEC Priority) : = 0 Customer

## Модела Опашка

- Регионът Queue съдържа подвижните компоненти на клас Customer в намаляващ ред на атрибута Priority и тогава за клиентите с еднакъв приоритет, нарастващ ред на атрибута Age. Клиентите с равен приоритет и възраст са подредени съобразно LIFO.

LOCATION Queue (Customer ORDERED BY DEC Priority, INC Age,  
LIFO) :=5 Customer

Като допълнение към индивидуалните региони могат да се създават масиви от региони.

- Това определя пет региона.

LOCATIONS ARRAY [5] WaitP (Customer ORDERED BY INC Prio)  
:= 0 Customer

- Регионите позволяват всички типове на индексиране.

WaitP[3]

WaitP{2..4}

WaitP{ALL}

## Модела Опашка

Работното пространство се определя автоматично от всеки регион, който е деклариран, който се използва при симулационното изпълнение, за да съхранява статистическа информация. Тези променливи не са достъпни в модела. Техните стойности могат да се записват от наблюдатели. Когато се създаде наблюдател, разширяването на регион дава достъп до различни статистики за този регион.

Тези стойности включват:

# Модела Опашка

Име	Значение
N	Текущия брой на подвижните компоненти на региона
NLeave	Броя на подвижните компоненти, които са въведени и напуснали региона
NMax	Максималния брой подвижни компоненти в региона във всеки един момент
NMean	Средния брой подвижни компоненти
DWMean	Средното задържане на подвижните компоненти в региона
DEmpty	Времето, за което региона е празен
DOccupy	Времето, за което региона е зает от поне една подвижна компонента
TLeave	Времето, за което последната подвижна компонента напуска региона

# Модела Опашка

Описание на подвижните компоненти.

Ако основна компонента съдържа подвижни компоненти.

Синтаксиса има вида:

basic\_component ::= BASIC COMPONENT identifier

[ mobile\_subclass\_declaratiion ]

[ unit\_definition\_part ]

[ local\_definitions ]

declaration\_of\_elements

[ dynamic\_behaviour ]

END OF identifier

mobile\_subclass\_declaratiion ::= MOBILE SUBCOMPONENT [S] OF  
CLASS

identifier { ‘,’ identifier }

# Модела Оашка

Регионът може да съхранява само подвижни компоненти от един клас. Името на този клас се дефинира в декларацията на региона.

`Mobile_subclass_declaration` информира основната компонента относно всички възможни класове на подвижни компоненти.

Подвижната компонента се описва по подобен начин на основната компонента, но без динамичното поведение, докато подвижните компоненти имат свои динамики, и промените на състоянието им стават факт само чрез основните компоненти.

С изключение на динамиките, подвижните компоненти могат да съдържат всякакви елементи на модела.

# Модела Опашка

Синтаксисът за описание на компонента има вида:

mobile\_component ::= MOBILE COMPONENT identifier

[mobile\_subclass\_declaration]

[unit\_definition\_part]

[local\_definitions]

declarations\_of\_elements

END OF identifier

mobile\_subclass\_declarations ::= MOBILE SUBCOMPONENT[S]

OF CLASS

Identifier { ‘,’ identifier }

# Модела Опашка

unit\_definition\_part ::= USE OF UNITS

[unit\_definition { unit\_definition }]

[time\_unit\_definition]

local\_definitions ::= LOCAL DEFINITION[S]

[dimension\_constant\_declaration\_part]

[enumeration\_set { enumeration\_set }]

[tabular\_function { tabular\_function }]

[tabular\_distributions { tabular\_distributions }]

# Модела Опашка

declaration\_of\_elements ::= DECLARATION OF ELEMENT[S]

- [list\_of\_constants]
- [list\_of\_state\_variables]
- [list\_of\_dependent\_variables]
- [list\_of\_sensor\_variables]
- [list\_of\_transitions\_indicators]
- [list\_of\_sensor\_indicators]
- [list\_of\_locations]
- [list\_of\_sensor\_locations]

# Модела Опашка

Примерът с подвижната компонента Customer1 показва, че в този случай от многото възможности е необходима само една променлива на състоянията от тип INTEGER.

Може да се види, че регионите са възможни също и при подвижните компоненти. Това означава, че подвижна компонента, която се намира в основна компонента може в себе си да съдържа региони за съхраняване на различен клас от подвижни компоненти. По този начин е възможно да се моделират превозни средства, които се движат от регион в регион и които съдържат подвижни компоненти.

# Модела Опашка

С въвеждането на подвижни компоненти имаме следната йерархия:

Подвижни компоненти:

- Без динамики;
- Поставени в региони;

Основни компоненти:

- Описание на динамики;

Компоненти от по-високо равнище:

- Свързване на основни компоненти;
- Структурно описание;

# Модела Опашка

Моделните елементи на подвижните компоненти са достъпни с цел да се позволи да бъдат изменяни и да се осигури достъп до текущите им стойности.

Моделните елементи на подвижни компоненти се идентифицират от име и път, което започва с регион в основна компонента. Синтаксис:

```
selected_element ::= indexed_identifier { ‘:’  
indexed_identifier ‘.’ indexed_identifier }
```

# Модела Опашка

Регионът е последван от двоеточие и след това от класа идентификатор на подвижната компонента. Това трябва да се съпроводи от индекс, който посочва мястото на текуща подвижна компонента при този регион.

Точката е последвана от идентификатор на моделен елемент на подвижната компонента. Този моделен елемент може да е променлива или регион. Ако е регион избраният път може да се продължи.

# Модела Опашка

Пример:

- В момента  $T = 10$  приоритетът на задачата, расположена в отрез 5 на региона WaitP се променя на 1.

ON  $^T = 10$

WaitP : Customer [5].Prio $^\wedge = 1$ ;

END

- Когато има подвижна компонента в първия отрез в регион WaitP с приоритет по-голям от 1, този приоритет се поставя да е Prio = 1.

WHENEVER WaitP : Customer [1].Prio > 1

DO WaitP : Customer [1].Prio $^\wedge := 1$ ;

END

# Модела Оашка

Пример 2 не е верен във всяка ситуация. Възможно е да се направи опит за достъп до подвижна компонента, която не присъства в региона.

В пример 2 непрекъснато се проверява следното условие:

**WHENEVER WaitP : Customer [1].Prio > 1**

Тестът също се провежда, когато регионът WaitP е празен и не присъстват никакви подвижни компоненти. В такъв случай ще се регистрира грешка при симулационното изпълнение и то ще спре.

За да се избегнат такива грешки се препоръчва пространството на състоянията да се раздели на подпространства. Условието в израза **WHENEVER** трябва да се проверява само, когато може да се гарантира, че в региона присъства подвижна компонента.

Това може да се постигне чрез стандартната функция **NUMBER**. Тази функция връща броя на подвижните компоненти в региона. Тя е от тип **INTEGER**.

# Модела Опашка

Следващото решение е по-добро:

```
IF NUMBER (WaitP) > = 1
    DO WHENEVER WaitP : Customer [1].Prio >1
DO WaitP : Customer [1].Prio^ := 1;
    END
END
```

# Модела Опашка

Забележка:

ON израза не позволява разделянето на пространството на състоянията както е в пример 3. Ако има възможност да се направи опит за достъп до празен регион, тогава трябва да се използва WHENEVER израз.

Когато се обръщаме за достъп към подвижна компонента в регион има други възможности освен прякото определяне на нейното място.

Името на класа на подвижната компонента може да се последва от индекс или набор от индекси. Това прави възможни всички операции за подвижните компоненти.

# Модела Опашка

Примери:

- Всички подвижни компоненти в региона Server получават приоритет Prio = 1.

Server : Customer {ALL}.Prio<sup>^</sup> := 1;

- Всички подвижни компоненти до позиция  $i \leq 5$  получават приоритет Prio = 1.

Server : Customer {ALL i | i <= 5}.Prio<sup>^</sup> := 1;

- Подвижните компоненти на позиции 4, 5 и 6 получават приоритет Prio = 1.

Server : Customer {4..6}.Prio<sup>^</sup> := 1;

# Модела Опашка

Забележка:

Индексните множества са позволени само за лявата страна на изразите за присвояване.

Индексните множества не са позволени в събитийно-стартиращ механизъм. За изразите ON и WHENEVER се позволява само фиксиран индекс.

# Модела Опашка

Описание на динамичното поведение.

В основните компоненти създаването, унищожаването и преместването на подвижни компоненти от регион в регион се прави чрез събития, за които могат да се използват и ON и WHENEVER изрази. Възможни са следните команди, последвани от условие или индикатор:

- PLACE : унищожава и реинициализира;
- ADD: създава;
- REMOVE: унищожава;
- GET: донася;
- SEND: изпраща.

# Модела Опашка

Всеки от тези изрази позволява също изменението на елементите на модела, принадлежащи на подвижни компоненти, за които се прилагат изразите. Това се постига със следната конструкция:  
**CHANGING <change of element> END**

# Модела Опашка

В каква точка при обработването на събития участва движението на подвижни компоненти?

event\_defining\_statement ::= triggering\_mechanism

[procedural\_part]

transitions\_part

triggering\_mechanism ::= WHENEVER expression

| ON indication {OR indication}

transitions\_part ::= ON [TRANSITON[S]]

transition\_statement\_sequence

END

# Модела Опашка

transition\_statement\_sequence ::= transition\_statement  
{ transition\_statement }

transition\_statement ::= state\_transition\_definition  
| signal\_statement  
| event\_region\_defining\_statement  
| display\_statement

state\_transition\_definition ::= state\_variable\_assignment  
| transfer\_statement

# Модела Опашка

transfer\_statement ::= place\_statement transfer\_end

| add\_statement transfer\_end

| remove\_statement transfer\_end

| get\_statement transfer\_end

| send\_statement transfer\_end

place\_statement ::= selected\_element '^' ':'

PLACE expression NEW identifier

# Модела Опашка

add\_statement ::= selected\_element '^' ':' ADD  
expression NEW identifier

remove\_statement ::= selected\_element '^' ':'  
REMOVE indexed\_identifier

get\_statement ::= selected\_element '^' ':'  
FROM selected\_element  
GET indexed\_identifier

# Модела Опашка

```
send_statement ::= selected_element '^' ':'
                  TO selected_element
                  SEND indexed_identifier
transfer_end      ::= ';' '
                  | transfer_extension
transfer_extention ::= ';' '
                     | CHANGING
                     state_transition_definition
                     { state_transition_definition }
                     END
```

# Модела Опашка

Забележка:

Избраният елемент от ляво на двоеточието отбелязва региона, в който нещо се случва. Символът ‘^’ се използва, за да подчертава времевото поведение. Съответната промяна на състоянието на състоянието се извършва в следващия цикъл. В това отношение регионите се държат подобно на променливите на състоянието.

Промяна на състоянието на променлива на състоянието:

Tdone^ := T +10;

Промяна на състоянието за регион: WaitP^ : ADD 1 New Customer;

# Модела Оашка

Ако се декларира масив от региони са позволени всички типове индекси.

Примери:

- Петият регион от масива Server донася първата подвижна компонента от регион Queue.

Server [5]^ : FROM Queue GET Customer [1];

- Всяка обслужваща станция в масив Server донася подвижна компонента от регион Queue. Първият Server получава подвижна компонента от първата позиция и т.н. Потребителят трябва да се увери, че региона Queue съдържа достатъчен брой подвижни компоненти.

Server { ALL i }^ : FROM Queue GET Customer [i];

# Модела Опашка

- Регионът Server донася първата подвижна компонента от всеки регион в масива Queue.

Server<sup>^</sup> : FROM Queue { ALL } GET Customer[1]

- Регионът Server донася всички подвижни компоненти от всички региони в масива Queue.

Server<sup>^</sup> : FROM Queue { ALL } GET Customer { ALL }

- Регионът Server донася всички подвижни компоненти от всички региони на масива Queue, докато там има повече от 3 подвижни компоненти.

Server<sup>^</sup>:FROM Queue { ALL i | NUMBER (Queue [i])> 3 }  
GET Customer { ALL }

# Модела Опашка

- Всички подвижни компоненти в региони с повече от 3 подвижни компоненти и приоритет по-голям от 2 се преместват в регион Server.

Server<sup>^</sup> : FROM Queue { ALL i | NUMBER (Queue [i]) > 3 }

    GET Customer { ALL j | Queue [i] : Customer [j].Prio > 2 }

В основна компонента не се прави разлика дали се използват GET или SEND команди.

Следните 2 команди са еквивалентни:

Station<sup>^</sup> : FROM QueueP GET Customer [1];

QueueP<sup>^</sup> : TO Station SEND Customer [1];

Разликите са само в гледната точка към модела. По този начин моделът демонстрира дали се подчертава изпращане или получаване.

# Модела Опашка

Стартиране на модела Queue1.

Ще направим експерименти с модела Queue1.

Средно време между интервалите

$$E[A] = 1/\lambda$$

# Модела Опашка

Средно време за обработка  $E[B] = 1/\mu$

Вероятност станцията да е заета:  $p = \lambda / \mu$

Средна дължина на опашката:  $E[Q] = p^2 / (1-p)$

Средно време на изчисляване:  $E[W] = (p / \mu) / (1-p)$

Системата M/M/1, която разглеждаме показва:

$$E[A] = 15 \text{ BE}$$

$$E[B] = 10 \text{ BE}$$

$$E[Q] = 1.33 \text{ клиенти}$$

$$E[W] = 20.0 \text{ BE}$$

$$\lambda = 0.066$$

$$\mu = 0.1$$

# Модела Опашка

## Хиbridни модели.

SIMPLEX MDL е особено силен, когато се дефинират хибридни модели, съдържащи подвижни компоненти, събития и диференциални уравнения.

Хибридният модел QueueK е основан на модела Queue1. В модела Queue1 услугата се моделира просто като определено количество от време. В модела QueueK това се заменя с непрекъснат процес.

Моделът QueueK описва доставката и запълването на барели в пълнеща станция, която се състои от помпа и сензор, които следят процеса на пълнене.

Зареждащата станция може да пълни само един барел в един момент от време. Барелите, които пристигат докато помпата е заета формират опашка в региона Queue.

# Модела Опашка

Скоростта на пълнене на барел зависи от количеството течност в барела. Започвайки от начална скорост 1.0 литра/секунда се увеличава пропорционално с коефициент 0.2 към количеството в барела, докато се достигне максималната скорост от 0.6 литра/секунда.

Скоростта на запълване е:

$$\text{Rate} := \text{Min}\{0.2 * \text{Amount} + 1.0 \text{ Litre/s}; 6.0 \text{ Litre/s}\}$$

Така Rate е зависима променлива, чиято стойност е между 1.0 и 6.0 и зависи от количеството в барела.

Запълването на барела в регион Pump се описва от диференциално уравнение:

$$\text{Amount}' := \text{Rate}$$

# Модела Опашка

Когато барелът е пълен, процесът на пълнене спира. Скоростта на пълнене Rate се поставя да е 0.

Напълнените барели се изпращат в регион Store.

Количеството в барела нараства непрекъснато. Когато скоростта на пълнене достигне максималната си стойност от 6.0, тя става константа, увеличението в количеството нараства линейно.

Използва се събитие, за да се провери дали барела е пълен. Тъй като пълненето е непрекъснат процес е нужно кръстосано търсене.

Когато барелът е пълен се поставя индикатор. По този начин се моделира сензор, който дава сигнал за преместване на барел от мястото за пълнене. Барелите се представят чрез подвижна компонента CustomerK. Тази компонента съдържа променливата Volume (вместимост).

# Модела Опашка

BASIC COMPONENT QueueK

MOBILE SUBCOMPONENTS OF CLASS CustomerK

USE OF UNITS

UNIT[Litre] = BASIS

TIMEUNIT = [s]

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

TArrive (REAL[s]) := 0 [s], # Интервали време на пристигане

Protocol (LOGICAL) := FALSE

# Модела Опашка

## CONTINUOUS

Amount (REAL[Litre]) := 0 [Litre] # Текущо съдържание на барел

## DEPENDENT VARIABLES

## CONTINUOUS

Rate (REAL[Litre/s]) := 0.0 [Litre/s] # Скорост на пълнене на помпата

## RANDOM VARIABLES

Arrive (REAL[s]) : EXPO  
(Mean:=14[s],LowLimit:=0.45[s],UpLimit:=70[s]),  
Size (INTEGER) : IUNIFORM (LowLimit:=2,  
UpLimit:=6)

# Модела Опашка

TRANSITION INDICATORS

BARREL\_FULL

LOCATIONS

Queue        (CustomerK) := 0 CustomerK, #

Източник на празни барели

Pump        (CustomerK) := 0 CustomerK, # Място  
за пълнене

Store        (CustomerK) := 0 CustomerK #

Съхраняване на пълни барели

# Модела Опашка

## DYNAMIC BEHAVIOUR

```
# Създаване на барел
WHENEVER (T >= TArrive)
DO
    TArrive ^ := T + Arrive;
    Queue^ : ADD 1 NEW CustomerK
    CHANGING
        Volume^ := Size * 10 [Litre];
    END
    IF Protocol
    DO
        DISPLAY("T = %f : %d litre barrel created\n",T,Size*10);
    END
END
```

# Модела Опашка

```
# Преместване на барел от опашката към пълнещата станция  
WHENEVER (NUMBER(Pump) = 0) AND (NUMBER(Queue) > 0)  
DO  
    Pump^ : FROM Queue GET CustomerK[1];  
    IF Protocol  
        DO  
            DISPLAY("T = %f : Barrel moved from \n", T);  
            DISPLAY("      queue to filling station\n");  
        END  
    END
```

## DIFFERENTIAL EQUATIONS

```
# пълнене на барел  
Amount' := Rate;  
END
```

# Модела Опашка

```
# Алгебрични уравнения за контролиране на скоростта на пълнене
IF (NUMBER(Pump)> 0) AND (Amount <
    Pump:CustomerK[1].Volume)
DO
    Rate := MIN (0.2 * Amount * 1 [1/s] + 1 [Litre/s], 6 [Litre/s]);
END
ELSE DO
    Rate := 0 [Litre/s];
END
# Завършване на пълненето
IF (NUMBER(Pump) > 0)
DO
    WHENEVER (Amount >= Pump:CustomerK[1].Volume)
    DO
        Amount^ := 0 [Litre];
        SIGNAL BARREL_FULL;
    END
END
```

# Модела Опашка

```
# Напускане на пълнещата станция  
ON BARREL_FULL  
DO  
    Pump^ : TO Store SEND CustomerK[1];  
    IF Protocol  
        DO  
            DISPLAY("T = %f : Barrel full!\n",T);  
        END  
    END  
END OF QueueK
```

# Модела Опашка

В модела QueueK има и непрекъснати и зависещи от събития процеси, както и премествания на подвижни компоненти. Понататък се използва алгебрично уравнение, за да се продължи промяната на стойността на зависимата променлива Rate.

# **Модела Опашка**

MOBILE COMPONENT CustomerK

USE OF UNITS

UNIT [Litre] = BASIS

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

Volume (REAL[Litre]) := 0 [Litre] # Volume of the barrel

END OF CustomerK

**Подвижната компонента CustomerK**

# Транспортни модели

Регионите са позволени в декларацията на елементи в подвижна компонента. Това означава, че подвижната компонента, която е разположена в регион в основна компонента може да съдържа региони за разполагане на бъдещ клас от подвижни компоненти. По този начин е възможно да се моделира превозно средство, което се мести от регион в регион, и самото то да транспортира подвижни компоненти.

Подвижни компоненти, които са разположени в региона на транспортната среда могат и те да съдържат региони. По този начин могат да се опишат транспортни процеси от много равнища.

# Транспортни модели

Пример:

На входната станция се зарежда нисък товарител с камиони, които возят стоки. Зареждащата станция се моделира от основната компонента с регион за разполагане подвижната компонента за клас нисък товарител. Той има регион, който може да настани камиони като по-нататъшен клас от подвижни компоненти. Подвижните компоненти за представяне на камионите също имат регион, в който се разполагат подвижните компоненти представлящи стоките.

# Транспортни модели

Получаване на достъп до атрибути и описание на динамика за превозни процеси.

Ако е достъпна променлива на състоянието на подвижна компонента, която представя вещи, доставени от превозни средства, тогава трябва да се специфицира пълното име на пътя.

Пример 1:

Първата позиция на регион Station съдържа камион, който превозва коли. Тези коли са разположени в региона Ramp върху камиона. На първата кола на товарната рампа се дава скорост Speed = 5.

Station: Truck[1] .Ramp : Car[1] .Speed<sup>^</sup> := 5;

Синтаксиса има следната форма:

state\_variable\_assignment ::= selected\_element ‘^’ ‘: =’  
expression ‘;’

selected\_element ::= indexed\_identifier { ‘:’ indexed\_identifier  
‘.’ indexed\_identifier }

## Транспортни модели

Пътят може да е произволно дълъг. Това позволява подвижни компоненти да функционират като превозачи за по-нататъшни подвижни компоненти и т.н.

Позволени са командите:

PLACE, ADD, REMOVE, GET и SEND.

По-нататък моделните елементи на подвижната компонента, към която се обръщаме чрез всеки от горните изрази може да се модифицират. Това се постига чрез команда CHANGING.

# Транспортни модели

Примери:

- Регионът Station съдържа подвижната компонента Taxi. Един от елементите на модела на таксито е регион, наречен Seat.

Подвижната компонента от регион WaitP на клас Customer се поставя върху Seat на таксито.

Station: Taxi[1] .Seat<sup>^</sup> : FROM WaitP GET Customer[1]

- Регионът Station съдържа подвижната компонента Taxi. Всички подвижни компоненти на класа Customer, които са в регион Seat са преместени и унищожени.

Station: Taxi[1] .Seat<sup>^</sup> : REMOVE Customer {ALL};

## Транспортни модели

Моделът Транспорт.

Моделът Transport описва цикъл, в които такситата могат да качват и превозват пътници както и да ги оставят на мястото закъдето те пътуват.

# Транспортни модели

В началото на изпълнението всички таксита са на първата таксиметрова спирка QTaxi1. Те са индивидуално номерирани.

Клиентите пристигат на първата таксиметрова спирка WaitP. Техните интервали на пристигане са експоненциално разпределени със средна стойност 10 времеви единици.

Когато пристигне клиент на първата таксиметрова спирка, такси го откарва на граничната станция Station 1. Тук таксито може да вземе до 4 пътника. Качването отнема точно 3 времеви единици.

## Транспортни модели

След качването таксито влиза в Road1. Времето за придвижване е нормално разпределено със следните параметри:

Средно = 36 ВЕ

- Стандартно отклонение = 10 ВЕ
- Долна граница = 0 ВЕ
- Горна граница = 72 ВЕ

След изтичане на времето за придвижване 3 таксита се нареждат на опашката при спирка QTaxi2.

## Транспортни модели

Слизането отнема 2 времеви единици. Тогава празното такси кара през Road2 обратно до изчакваща точка QTaxi1. Времето за придвижване по Road2 е нормално разпределено както е и за Road1. За тестови цели се записва цялостното поведение на модела.

Начално се декларират подвижните компоненти на клас Taxi1 и Customer1. Тогава елементите са декларириани. Времето за качване на пътници и това за слизане в региони Station1 и Station2 е константа.

# Транспортни модели

BASIC COMPONENT Trans1

MOBILE SUBCOMPONENTS OF CLASS  
Taxi1,Customer1

DECLARATION OF ELEMENTS

CONSTANTS

Load(REAL)        := 3,        # Време за качване  
Unload(REAL)      := 2        # Време за слизане

# Транспортни модели

## STATE VARIABLES

```
TNext(REAL)      := 0,      # Генериране на пътник  
TLoad(REAL)      := 0,      # Край на качването  
TUnload(REAL)    := 0,      # Край на слизането  
Protocol(LOGICAL) := FALSE # Контрол на протокола
```

## RANDOM VARIABLES

```
ZArrive(REAL)   : EXPO (Mean := 10),  
ZTravel1(REAL)   : GAUSS (Mean:=36,Sigma:=10,  
                      LowLimit:=0,UpLimit:=72),  
ZTravel2(REAL)   : GAUSS (Mean:=36,Sigma:=10,  
                      LowLimit:=0,UpLimit:=72)
```

# Транспортни модели

## LOCATIONS

QTaxi1 (Taxi1) := 10 Taxi1,

Station1 (Taxi1) := 0 Taxi1,

WaitP (Customer1) := 0 Customer1,

Road1 (Taxi1 ORDERED BY INC TTravel) := 0  
Taxi1,

QTaxi2 (Taxi1) := 0 Taxi1,

Station2 (Taxi1) := 0 Taxi1,

Road2 (Taxi1 ORDERED BY INC TTravel) := 0  
Taxi1

# Транспортни модели

Нужни са променливи на състоянието за генериране на нов клиент и края на времената за качване и слизане. Променливата Protocol се използва, за да контролира извеждането на протокола. Промяната на началната ѝ стойност от FALSE на TRUE чрез командата Model parameters в контекстното меню на обект Break0 ще активира извеждането.

След това се декларират случаини променливи, които описват времената на пристиганията на клиенти и времената на придвижване по пътищата Road1 и Road2.

Наредбата за всички региони е FIFO.

Следващото такси, което ще напусне Road1 или Road2 съответно е в началото на опашката.

# Транспортни модели

## DYNAMIC BEHAVIOUR

# Събитие 1: Номериране на такситата

```
ON START
DO
  QTaxi1^ : FROM QTaxi1 GET Taxi1{ALL i}
  CHANGING
    TaxiN^ :=i;
  END
  IF Protocol
  DO
    DISPLAY("T= %f Event 1 \n",T);
    DISPLAY(" Number the Taxis \n\n");
  END
END
```

# Транспортни модели

# Събитие 2: Генериране на пътник

WHENEVER T >= TNext

DO

WaitP<sup>^</sup> : ADD 1 NEW Customer1;

TNext<sup>^</sup> := T + ZArrive;

IF Protocol

DO

DISPLAY("T= %f Event 2 \n",T);

DISPLAY(" Customer enters WaitP \n");

DISPLAY(" Next generation in T= %f \n",T+ZArrive);

DISPLAY(" Customers in WaitP= %ld  
\n",NUMBER(WaitP)+1);

END

END

# Транспортни модели

# Събитие 3: Такси спира на Спирка 1

WHENEVER (NUMBER(WaitP) > 0) AND (NUMBER(QTaxi1) > 0) AND  
(NUMBER(Station1) = 0)

DO

Station1^ : FROM QTAXI1 GET TAXI1[1];

IF Protocol

DO

DISPLAY("T= %f Event 3 \n", T);

DISPLAY(" Taxi number %ld enters Station 1 \n\n",  
QTAXI1:TAXI1[1].TAXIN);

END

END

# Транспортни модели

## # Събитие 4: Качване

```
IF NUMBER(Station1) = 1
DO
    WHENEVER NUMBER(Station1:Taxi1[1].Seat) = 0
    DO
        Station1:Taxi1[1].Seat^ : FROM WaitP GET
            Customer1 { ALL i | (i<=NUMBER(WaitP)) AND
            (i<=4)};
        TLoad^ := T + Load;
        IF Protocol
        DO
            DISPLAY("T= %f Event 4 \n",T);
```

# Транспортни модели

```
IF NUMBER(WaitP) < 4
DO
  DISPLAY("  Loading Taxi Number %ld, # Cust %ld \n",
    Station1:Taxi1[1].TaxiN, NUMBER(WaitP));
END
ELSE DO
  DISPLAY("  Loading Taxi Number %ld, # Cust %ld \n",
    Station1:Taxi1[1].TaxiN, 4);
END
DISPLAY("  Loading ends in T=% f \n\n",T+Load);
END
END
END
```

# Транспортни модели

# Събитие 5: Влизане в маршрут 1

ON ^T >= TLoad^

DO

Road1^ : FROM Station1 GET Taxi1[1]

CHANGING

TTtravel^ := T + ZTravel1;

END

IF Protocol

DO

DISPLAY("T= %f Event 5 \n",T);

DISPLAY(" Taxi Number %ld enters Road1 \n",  
Station1:Taxi1[1].TaxiN);

DISPLAY(" Journey ends in T= %f \n\n",T+ZTravel1);

END

END

# Транспортни модели

# Събитие 6: Излизане от маршрут 1

IF NUMBER(Road1) > 0

DO

WHENEVER T >= Road1:Taxi1[1].TTravel

DO

QTaxi2^ : FROM Road1 GET Taxi1[1];

IF Protocol

DO

DISPLAY("T= %f Event 6 \n",T);

DISPLAY(" Taxi Number %ld enters QTaxi2 \n\n",  
Road1:Taxi1[1].TaxiN);

END

END

END

# Транспортни модели

# Събитие 7: Таксито спира на Спирка 2

WHENEVER (NUMBER(Station2) = 0) AND  
(NUMBER(QTaxi2) >0)

DO

Station2^ : FROM QTaxi2 GET Taxi1[1];

IF Protocol

DO

DISPLAY("T= %f Event 7 \n",T);

DISPLAY(" Taxi Number %ld enters Station 2 \n\n",  
QTaxi2:Taxi1[1].TaxiN);

END

END

# Транспортни модели

# Събитие 8: Слизане

IF NUMBER(Station2) = 1

DO

WHENEVER NUMBER(Station2:Taxi1[1].Seat) > 0

DO

Station2:Taxi1[1].Seat<sup>^</sup> : REMOVE Customer1{ ALL };

TUnload<sup>^</sup> := T + Unload;

# Транспортни модели

IF Protocol

DO

    DISPLAY("T= %f Event 8 \n",T);

    DISPLAY(" Unloading Taxi number %ld, # Cust. %ld  
\n",

        Station2:Taxi1[1].TaxiN,  
        NUMBER(Station2:Taxi1[1].Seat));

    DISPLAY(" Unloading ends in T= %f \n\n",T+Unload);

END

END

END

# Транспортни модели

```
# Събитие 9: Влизане в маршрут 2
ON ^T >= TUnload^
DO
    Road2^ : FROM Station2 GET Taxi1[1]
    CHANGING
        TTravel^ := T + ZTravel2;
    END
    IF Protocol
        DO
            DISPLAY("T= %f Event 9 \n",T);
            DISPLAY("  Taxi Number %ld enters Road2 \n",
                    Station2:Taxi1[1].TaxiN);
            DISPLAY("  Journey ends in T= %f \n\n",T+ZTravel2);
        END
    END
```

# Транспортни модели

# Събитие 10: Излизане от маршрут 2

IF NUMBER(Road2) >0

DO

WHENEVER T >= Road2:Taxi1[1].TTravel

DO

QTaxi1^ : FROM Road2 GET Taxi1[1];

IF Protocol

DO

DISPLAY("T= %f Event 10 \n",T);

DISPLAY(" Taxi Number %ld enters QTAXI1 \n\n",

Road2:Taxi1[1].TaxiN);

END

END

END

END OF Trans1

# Транспортни модели

Описанието на динамиките се нуждае от 10 събития:

- Събитие 1:

Такситата се преместват от регион QTaxi1, номерират се и се връщат в регион QTaxi1. Индексът i за номериране на таксита е валиден и в следващата команда CHANGING.

- Събитие 2:

Създаване на нови клиенти. Изходът дава текущата дължина на опашката.

- Събитие 3:

Такситата влизат в спирката Station 1.

- Събитие 4:

Позволява се максимум от 4 пътници в таксито. Броя на клиентите се записва.

# Транспортни модели

- Събитие 5:

След качването таксито поема по Road1. Времето на пътуване се определя и присвоява на променливата на състоянието TTravel. Изходът съдържа броя на такситата и изчисленото време на пристигане в QTaxi2.

- Събитие 6:

Всички таксита, които са на маршрут Road1 са в региона Road1. Те са сортирани в намаляващ ред за времето на завършено пътуване. Всяко такси носи това време като съставна компонента. Може да се определи кога следващото такси напуска пътя и достига региона QTaxi2.

- Събитие 7:

Ако Station2 не е заета таксито в QTaxi2 може да влезе на спирката.

# Транспортни модели

- Събитие 8:

Всички клиенти се преместват и унищожават. По същото време се записва времето на слизане.

- Събитие 9:

Входа за пътя Road2 е аналогичен на входа за пътя Road1.

- Събитие 10:

Изхода от път Road2 и входа в точката на изчакване QTaxi1 са идентични на Събитие 6.

В допълнение към основната компонента е необходимо описанието на класа на подвижната компонента Taxi1. Подвижните компоненти на този клас могат да се намерят в региони QTaxi1, Station1, Road1, QTaxi2, Station2 и Road2.

# Транспортни модели

## MOBILE COMPONENT Taxi1

MOBILE SUBCOMPONENTS OF CLASS Customer1

### DECLARATION OF ELEMENTS

#### STATE VARIABLES

TTravel(REAL) := 0,

TaxiN(INTEGER) := 0

#### LOCATION

Seat(Customer1) := 0 Customer1

END OF Taxi1

# Транспортни модели

Когато моделните променливи се записват, команда DISPLAY извежда стойността в текущия цикъл. DISPLAY изразът не трябва да съдържа TNext, тъй като TNext няма да се свърже с правилната стойност до следващия цикъл. Вместо това трябва да се напише T + ZArrive.

Събитие 3 е подобно. Ще се отпечата броя на такситата, които се преместват от регион QTaxi1 в Station1. При текущия цикъл таксито все още е в регион QTaxi1. Затова трябва да се даде този регион.

Друга възможност е да се забави команда DISPLAY с един цикъл. Това може да се постигне като се направи извеждащата команда обект на индикатор за преместване.

## Транспортни модели

В допълнение към променливата на състояние подвижната компонента QTaxi1 съдържа региона Seat, който може да разположи подвижни компоненти от клас Customer1.

Когато подвижните компоненти не се нуждаят от атрибути, описание е празно. Въпреки това то се изиска.

# **Транспортни модели**

MOBILE COMPONENT Customer1

DECLARATION OF ELEMENTS

END OF Customer1

# Стратегии

Критерия, използван за да подреди задачите в опашка (и затова също редът, в които те ще се обработват) е познат като стратегия.

Стратегията използва приоритети, за да определи позицията на задача в опашка. Приоритета е мярка за важност, която може да определи приоритета на задача сама по себе си или може да се изчисли съобразно определени критерии.

В особено важни случаи може да е нужна опцията да освободим зает сървър, за да направим възможно задача с висок приоритет да се обработи незабавно. Механизмът, който позволява това да се направи се нарича изместване.

# Стратегии

Изместването (pre-emption) винаги изисква определено количество време. По тази причина стратегията на изместването може да реагира бързо на спешните случаи, но на цената на допълнително време за презареждане.

Има разлика между статичните и динамични приоритети. Със статични приоритети, на задача се присвоява приоритет, който не се променя по време на нейния престой в опашката. Обратно с динамичните приоритети може да се промени приоритета на задача, зависеща от текущото състояние на системата.

# Стратегии

Статични стратегии.

Тази секция описва няколко важни и основни стратегии, които използват статични приоритети

FIFO (Първи вътре, Първи вън)

FIFO стратегията дава приоритети съобразно времето на пристигане. Задачата, която е била в опашката най-дълго има най-висок приоритет. Той отговаря на концепцията за безпристрастност. Приоритетът се поставя равен на входното време:

$Prio := T_{ARRIVE}$

Задачите са подредени в опашката според приоритета си. Задача, която влиза в опашката първа има най-малката стойност за Prio и така стои при началото на опашката. Тя ще бъде първата за обработка.

PFIFO (Preemptive FIFO) означава, че задача влязла в опашката ще измести задачата, която в момента се обработва, ако има по-висок приоритет.

# Стратегии

Пример:

По-малко важна задача се премества през производственото съоръжение. Ако пристигне спешна задача, която трябва да се приеме и обработи веднага, тогава оригиналната задача ще се премахне от машината. Тя трябва да напусне машината и да се върне в опашката.

LIFO (Last In, First Out)

Обратно на FIFO, LIFO стратегията дава предимство на задачата, пристигнала в опашката последна. LIFO работи съобразно стек принципа, където пристигащите задачи се поставят на върха на стека и се премахват оттам. Приоритетът е отрицателното време на пристигане.

Prio : = -T<sub>ARRIVE</sub>

# Стратегии

## SJF (Shortest Job First)

Стратегията SJF присъединява най-високия приоритет към най-краткото време за обработка в сървъра. Тя максимализира производството, т.е. най-големият възможен брой задачи се обработва и най-голям възможен брой задачи могат да бъдат удовлетворени. Тази стратегия се основава на идеята, че е добро хрумване е да се обработват първо малките задачи преди да се премине към по-големите.

Пример: Планирането на задачи решавани с помощта на компютър позволява кратките задачи да се обработват първи. Времеемките изчислителни задачи приемат дълги времена за изчакване, защото имат нужда от по-големи ресурси.

Приоритетът е поставен равен на времето за обработка:

$$\text{Prio} := T_{\text{PROCESS}}$$

SJF позволява презареждане.

# Стратегии

Round Robin (Cyclic Strategy)

Round Robin стратегията свързва сървъра с всяка задача за определен период от време. След като този ‘отрез от време’ изтече се проверява дали задачата е завършена и може да напусне сървъра. Ако случят не е такъв, тогава тя се връща в опашката с нейното оставащо време за обработка. Тази стратегия предполага, че е добра идея да се обработват по малко всички задачи. Това е особено подходящо, ако задачата може да върши други дейности, които не се нуждаят от сървъра след като тя е била обработена за кратко време.

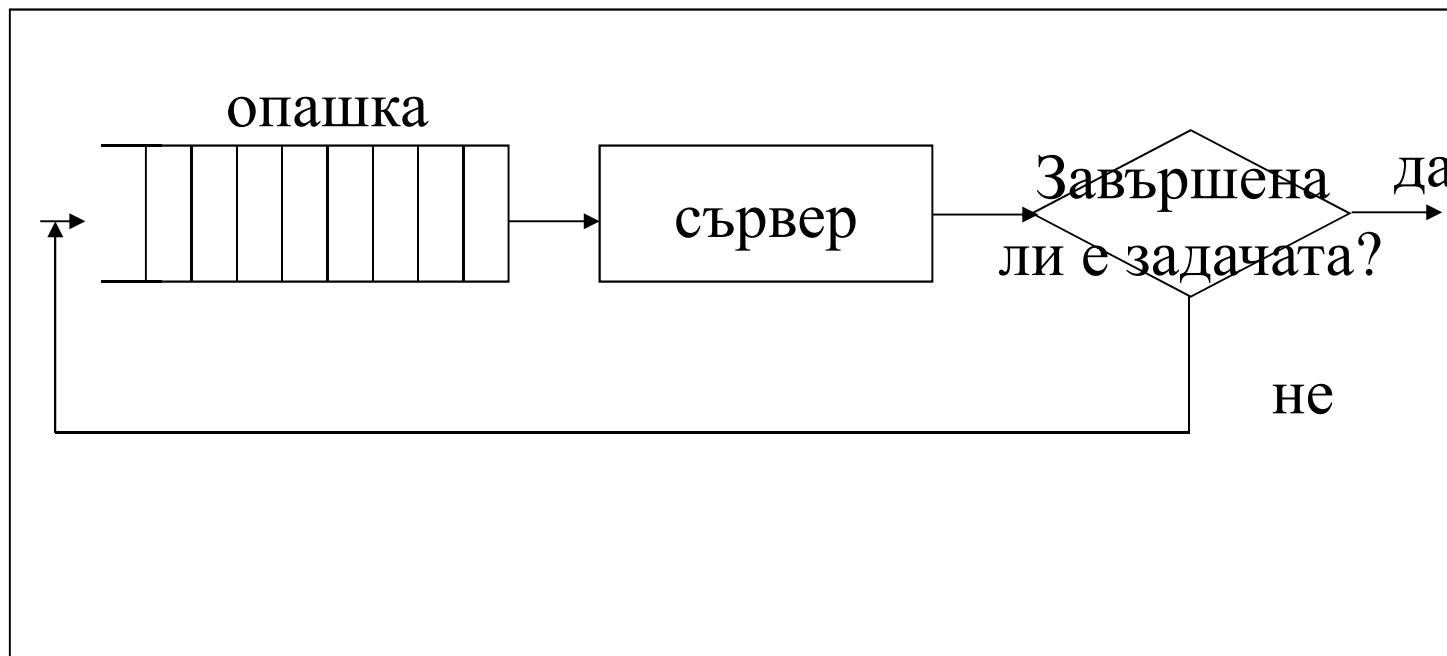
Пример:

Компютърът, работещ в режим на времеделение, свързва всяка задача с отрез от време. Задачата се обработва, но с ефективна скорост, която се намаля с толкова повече, колкото повече задачи влизат в системата.

Round Robin може да се осъществи само, ако е налично презареждане.

# Стратегии

## Стратегията Round Robin



# Стратегии

Round Robin може да се направи по-гъвкава, ако дължината на времевия отрез не е константна, а зависи от важността на задачата. Това ражда стратегията RRP (Round Robin with Priorities).

## LFB (Limited Feedback)

Цикличните стратегии изискват допълнителни ресурси. Затова в много случаи е препоръчително да се ограничи броя на задачите в стратегията Round Robin. Има различни методи за определяне на задачи, които могат да участват в Round Robin, докато други задачи се третират по различен начин.

Прост пример е LFB стратегията, където всяка задача получава  $N_{max}$  времеви отрези по начина на Round Robin. Когато всички  $N_{max}$  времеви отрези се използват, задачата влиза в различна опашка, която работи с различна стратегия, като FIFO.

Тогава задачите се премахват от тази опашка само, ако няма останали в опашката на Round Robin.

# Стратегии

Стратегиите с опашки отразяват организацията на намиращата се отдолу реална система. Много аспекти на организацията могат да се представят чрез наредба при обработване в модела, и по този начин чрез стратегия. Има толкова стратегии, колкото са формите на организация.

Пример:

- Опашката пред областта за съхранение може да се организира според нужното количество за съхранение. Например задачата с най-малко изисквания може да получи най-висок приоритет. Това означава, че задачите, имащи големи изисквания ще трябва да чакат прекалено дълго.
- Може да се използва и противоположен подход, където на големи изисквания се дава висок приоритет.
- Възможно е също да се комбинират двата метода и да се използва стратегия с редуващо се поставяне на задачи с големи и малки изисквания в опашката.

# Стратегии

Статичните стратегии свързват приоритети към задачи, когато те пристигнат в опашката.

Обратно, динамичните стратегии позволяват променянето на приоритетите на чакащите задачи в зависимост от определени условия.

Условията, които определят динамичното присвояване на приоритети може да се мени. Общо тяхно описание не е възможно.

Приоритетите трябва да се представят като функция, която взима всички системни състояния като аргументи.

# Стратегии

Следващите секции описват няколко основни динамични стратегии:

## UTL (Upper Time Limit)

Всяка задача може да получи горна времева граница, за която тя трябва да приключи. Колкото повече задачата се приближи към тази граница, толкова по-висок става приоритета ѝ. В този случай приоритета може да се определи от проста функция, която взима като аргумент времето, оставащо до границата. Това оставащо време съдържа оставащото време за обработка плюс времето за чакане.

В най-простия случай приоритета се увеличава линейно с оставащото време. Приоритетът се оценява според следната формула:

Prio := T – Tcomplete = remaining time allowed      (позволено оставащо време)

В този случай всички приоритети са отрицателни.

# Стратегии

UTLP (Upper Time Limit with Priorities)

Стратегията UTL може да се рафинира и да се подобрява с това динамичните приоритети да зависят от начален приоритет вместо всички да са идентични. UTLP стратегията увеличава приоритета за всички задачи с висок начален приоритет по-бързо от този на задачи с нисък начален приоритет.

Приоритетите могат да получат различни поведения, които зависят от началния им приоритет Prio0.

В момента  $T = 1$  задачата пристига с начален приоритет  $Prio0 = 0$ .

Според динамичните приоритети тя достига стойност  $Prio = 13$  в момента  $T = 13$ . Задачата, която влиза в системата в момента  $T = 8$  с начален приоритет  $Prio0 = 4$  достига максималния приоритет  $Prio = 20$  в момента  $T = 14$ , поради привилегированите ѝ стойности. Важният ефект е, че редът на задачите не е запазен. UTLP позволява задачите да се настигат една друга.

# Стратегии

Едно изменение на UTL заема натрупаното време на изчакване:

Prio = желано време за изчакване/натрупано време на изчакване

Коефициентът на желаното време за изчакване към натрупаното изчакващо време се намалява с увеличаване на времето за чакане. Затова задачите са подредени в намаляващ приоритет в опашката.

Всички стратегии с динамични приоритети изискващи преоценяване на приоритетите се стартират отделно. Моментът време, в което това се случва зависи от типа задача.

Два основни метода са:

- Приоритетите се изчисляват в набор от интервали;
- Приоритетите се изчисляват всеки път, когато задача трябва да се избере от стратегията.

# Стратегии

Честотата, с която приоритетите се свързват определя точността, с която те представят верните условия. Тъй като приоритетите се прави първо, това подобрение има също недостатъци.

Ако времената на обслужване са кратки сравнени с динамичното присвояване на приоритети, тогава по-късно не може да се пренебрегнат.

Приоритетите зависят от броя на задачите, засегнати от динамичните приоритети. При повече засегнати задачи е нужно по-дълго време.

# Стратегии

Стратегии на презареждане.

На модела Queue2 са направени следните изменения:

- Всяка десета задача е спешна задача с приоритет 0. Всички други задачи имат приоритет 1.
- Сървърът се презарежда. Всеки път, когато се появи спешна задача сървърът се освобождава и презаредената задача отива в края на опашката с нейното оставащо време за обработка. Подготвителното време за всяко презареждане е 1 времева единица.

# Стратегии

Регионът WaitP се подрежда според стратегията ‘намаляващ приоритет’. Тъй като няма друга информация, задачи с идентични приоритети се нареджат според FIFO стратегията.

Всеки път, когато се появи нова задача в опашката трябва да се провери дали презареждането участва или не. Това ще е случая, когато нова задача е спешна и сървърът все още не съдържа спешна задача.

Ако е необходимо презареждане се поставя индикатора Preempt. Този индикатор стартира събитие, в което текущата задача се изпраща към края на опашката и нейното оставащо време за обработка се отбелязва. По същото време се извършва присвояването  $T_{Reequip} = T + Reequip$ . Това осигурява, че спешна задача влиза в станцията след като времето за пренастройването за нея е изтекло.

За да се избегне нуждата от различаване на оставащото време за обработка и общото време за обработка, декларацията на подвижната компонента Customer2 също съдържа времето TimeA. В момента на създаване всяка задача се свързва с времето за нейната обработка, което се съхранява в TimeA.

Времето за обработка на задачата е достъпно, когато се оцени крайното време  $T_{work}$ .

# Стратегии

MOBILE COMPONENT Customer2

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

Priority (INTEGER) := 1, # Приоритет

TimeA (REAL) := 0 # Оставащо време

END OF Customer2

# Стратегии

BASIC COMPONENT Queue2

MOBILE SUBCOMPONENTS OF CLASS Customer2

LOCAL DEFINITIONS

VALUE SET StateType : ('busy', 'reequipping', 'free')

DECLARATION OF ELEMENTS

CONSTANTS

Reequip (REAL) := 1 # Време за презареждане на сървър когато спешна работа измести друга

# Стратегии

## STATE VARIABLES

### DISCRETE

```
TArrive (REAL) := 0,      # Следващо време за пристигане  
TWork   (REAL) := 0,      # Край на обработката  
TReequip (REAL) := 0,     # Край на презареждането  
Protocol (LOGICAL) := FALSE,    # Контрол на протокола  
Job      (INTEGER) := 1,      # Брояч на задачите  
State   (StateType) := 'free'  # Състояние на сървъра
```

## RANDOM VARIABLES

```
Arrive (REAL) : EXPO (Mean:=15,LowLimit:=0.5,UpLimit:=75),  
Work   (REAL) : EXPO (Mean:=10,LowLimit:=0.32,UpLimit:=50)
```

# Стратегии

## TRANSITION INDICATOR

Preempt # Индикатор за  
стартиране на презареждането

## LOCATIONS

WaitP (Customer2 ORDERED BY DEC Priority)  
:= 0 Customer2,

Station (Customer2) := 0  
Customer2,

Sink (Customer2) := 0 Customer2

# Стратегии

DYNAMIC BEHAVIOUR

# Създаване на задачи

WHENEVER T >= TArrive

DO

IF Job = 10 # Нова спешна задача

DO

WaitP<sup>^</sup> : ADD 1 NEW Customer2

CHANGING

Priority<sup>^</sup> := 0;

TimeA<sup>^</sup> := Work;

END

Job<sup>^</sup> := 1;

IF (State = 'busy') AND (Station:Customer2[1].Priority = 1)

DO SIGNAL Preempt; END END

# Стратегии

```
ELSE DO          # Нова стандартна задача
    WaitP^ : ADD 1 NEW Customer2
    CHANGING
        TimeA^ := Work;
    END
    Job^ := Job + 1;
END
```

```
TArrive^ := T + Arrive;
IF Protocol
DO
    DISPLAY ("T= %f New job \n", T);
END
END
```

# Стратегии

```
# Начало на обработката на стандартна задача
WHENEVER (NUMBER(Station)=0) AND
    (NUMBER(WaitP)>0) AND
    (State <> 'reequipping')

DO
    Station^ : FROM WaitP GET Customer2[1];
    TWork^ := T + WaitP:Customer2[1].TimeA;
    State^ := 'busy';
    IF Protocol
        DO
            DISPLAY ("T= %f Standard job enters server\n",T);
        END
    END
```

# Стратегии

```
# Стандартна задача е изместена от спешна  
ON Preempt  
DO  
    IF (State = 'busy')  
        DO  
            Station:Customer2[1].TimeA^ := TWork - T; # оставащо  
            време  
            Station^ : TO WaitP SEND Customer2[1];  
            TReequip^ := T + Reequip; # време за  
            презареждане  
            State^ := 'reequipping';  
    END
```

# Стратегии

IF Protocol

DO

DISPLAY ("T= %f Rush job arrives, \n", T);

DISPLAY (" Standard job is preempted\n");

END

END

# Стратегии

```
# Начало на обработка на спешна задача
WHENEVER (T >= TReequip) AND (State = 'reequipping')
DO
    Station^ : FROM WaitP GET Customer2[1];
    TWork^ := T + WaitP:Customer2[1].TimeA;
    State^ := 'busy';
    IF Protocol
        DO
            DISPLAY ("T= %f Rush job enters server\n", T);
        END
    END
END
```

# Стратегии

```
# Край
WHENEVER (T >= TWork) AND (State = 'busy')
DO
    Station^ : TO Sink SEND Customer2[1];
    State^ := 'free';
    IF Protocol
        DO
            DISPLAY ("T= %f Job leaves server\n", T);
        END
    END
```

# Стратегии

```
# Унищожаване на задачи
WHENEVER NUMBER(Sink) >= 4
DO
    Sink^ : REMOVE Customer2{ALL};
    IF Protocol
        DO
            DISPLAY ("T= %f 4 jobs destroyed\n", T);
        END
    END
END OF Queue2
```

# Стратегии

Стратегии с процедурна част.

Някои сложни стратегии изискват претърсване на опашката преди сървъра. Това се постига в процедурна част.

Моделът Queue3 също се базира на Queue1. В допълнение, при създаването си всяка задача се свързва с максимално време за чакане, което се съхранява в променливата WaitT на подвижната компонента Customer3. Ако това максимално време за чакане се достигне, задачата напуска опашката преждевременно без да се обработи.

# Стратегии

MOBILE COMPONENT Customer3

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

Priority (INTEGER) := 1, # Приоритет

WaitT (REAL) := 0 # Максимално време, за което  
задача може да остане в опашката

END OF Customer3

# Стратегии

BASIC COMPONENT Queue3

MOBILE SUBCOMPONENTS OF CLASS Customer3

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

TArrive (REAL) := 0, # Следващо време на пристигане

TWork (REAL) := 0, # Край на обработката

Protocol (LOGICAL) := FALSE, # Контрол на протокола

MinTime (REAL) := 0 # Следващо време на напускане

# Стратегии

## RANDOM VARIABLES

```
Arrive (REAL) : EXPO (Mean:=15,LowLimit:=0.5,UpLimit:=75),  
Work (REAL) : EXPO (Mean:=10,LowLimit:=0.32,UpLimit:=50),  
MaxW (REAL) : GAUSS (Mean:=25,Sigma:=5,  
UpLimit:=40,LowLimit:=10) # Максимално време за  
изчакване
```

## TRANSITION INDICATOR

```
MinNew # Индикатор за стартиране на  
преизчисляване на следващото време за напускане
```

# Стратегии

LOCATIONS

WaitP (Customer3) := 0 Customer3, # Входна  
опашка

Station (Customer3) := 0 Customer3, # Сървър

Sink (Customer3) := 0 Customer3 #  
Контейнер

# Стратегии

DYNAMIC BEHAVIOUR

# Създаване на задачи

WHENEVER T >= TArrive

DO

WaitP<sup>^</sup> : ADD 1 NEW Customer3

CHANGING

WaitT<sup>^</sup> := T + MaxW;

END

IF (MinTime - MaxW > T) OR

(NUMBER(WaitP) = 0)

DO

MinTime<sup>^</sup> := T + MaxW;

END

TArrive<sup>^</sup> := T + Arrive;

IF Protocol

DO      DISPLAY ("T= %f New customer\n",T);    END    END

# Стратегии

```
# Начало на обработката
WHENEVER (NUMBER(Station)=0) AND (NUMBER(WaitP)>0)
DO
    Station^ : FROM WaitP GET Customer3[1];
    IF (WaitP:Customer3[1].WaitT = MinTime)
        DO
            SIGNAL MinNew; # Стартоване на преизчисляването на
                           MinTime
        END
    TWork^ := T + Work;
    IF Protocol
        DO
            DISPLAY ("T= %f Customer enters Server\n",T);
        END
    END
END
```

# Стратегии

```
# Край на обработката  
WHENEVER (T >= TWork) AND  
  (NUMBER(Station)=1)  
DO  
  Station^ : TO Sink SEND Customer3[1];  
  IF Protocol  
    DO  
      DISPLAY ("T= %f Customer leaves server\n",T);  
    END  
  END
```

# Стратегии

# Намиране на следващото най-кратко време за изчакване, ако индикатора е поставен

ON MinNew

DECLARE

P (REAL)

DO PROCEDURE # Процедурна секция за събитие

IF (NUMBER(WaitP) > 0)

DO P := WaitP:Customer3[1].WaitT;

FOR i FROM 1 TO NUMBER(WaitP) # Цикъл за определяне на следващото време за напускане

# Стратегии

## REPEAT

IF (WaitP:Customer3[i].WaitT < P)

D0

P := WaitP:Customer3[i].WaitT;

END

## END\_LOOP

END END

# DO TRANSITIONS

## # Част на преход

MinTime<sup>^</sup> := P;

# Даване на стойност на

променлива на състояние

END

# Стратегии

```
# Чакаща работа ,губи търпение‘  
ON ^( $T \geq \text{MinTime}$ ) AND (NUMBER(WaitP) > 0)^  
DO  
    WaitP^ : REMOVE Customer3{ ALL  
        i|WaitP:Customer3[i].WaitT <= T};  
    SIGNAL MinNew;      # Стартоване на преизчисляване на  
    MinTime  
    IF Protocol  
        DO  
            DISPLAY("T= %f Waiting job leaves queue  
            prematurely!\n", T);  
        END  
    END
```

# Стратегии

```
# Уничожаване на задача
WHENEVER NUMBER(Sink) >= 4
DO
    Sink^ : REMOVE Customer3{ALL};
    IF Protocol
        DO
            DISPLAY("T= %f 4 Customers destroyed\n",
T);
        END
    END
END OF Queue3
```

# Стратегии

Първо, задачите са в региона WaitP. Независимо от това променливата MinTime се свързва с най-ранното време, при което чакащата задача може да напусне опашката прежевременно и да се унищожи.

Когато нова задача влезе в опашката, MinTime трябва да се преоценни. Просто сравнение определя дали MinTime трябва да се преоценни.

Ако симулационният часовник T е достигнал MinTime, всички задачи, чието Max време за чакане е истекло, трябва да се премахнат от опашката. По същото време стойността на MinTime трябва да се присвои отново.

Подобно MinTime трябва да се преоценни, ако задачата, която текущо определя стойността на MinTime (най-малката стойност на WaitP от всички задачи в опашката) напуска опашката, за да се обработи.

# Стратегии

В следващите два случая MinTime трябва да получи най-ниската стойност на WaitT, съдържаща се във всяка задача в опашката. Не може да се определи предварително, на коя позиция в опашката може да се намери тази задача.

По тази причина опашката трябва да се претърси. Това се извършва в събитие, което е стартирано от индикатора MinNew.

Събитието съдържа процедурна част. Декларира се променлива P, която се свързва с най-ранното време за изчакване. След напускането на процедурната част тази стойност се присвоява на MinTime.

# Стратегии

Стратегии и MDL функции.

В големи модели може да са нужни идентични функции на различни места. В този случай е добре да се използват MDL функции.

За да илюстрация на употребата на MDL функциите, ще се модифицира модела Queue3, така че търсенето на най-ранното време за изчакване да се извърши във функцията SearchW. Регионът ще бъде предаден като вътрешен параметър. Върнатият параметър е стойността на MinTime.

# Стратегии

```
# Determine next earliest waiting time if indicator is set  
ON MinNew  
DO                                # Извикване на функция  
    SearchW  
        MinTime^ := SearchW(LOCATION WaitP); # с регион  
        WaitP като параметър  
END
```

**Извадка от модел Queue4**

## Стратегии

Дефинирането на функцията SearchW е дадено в основната компонента с LOCAL DEFINITIONS:  
BASIC COMPONENT Queue4

MOBILE SUBCOMPONENTS OF CLASS  
Customer4

LOCAL DEFINITIONS

# Функция за изчисляване на следващото време за напускане

# Входен параметър е регион (WaitP)

# Връщания параметър е реален P (новото MinTime)

# Стратегии

FUNCTION SearchW (LOCATION FOR Customer4 : Job --> REAL)

DECLARE

P (REAL)

BEGIN

IF (NUMBER(Job) > 0)

DO

P := Job:Customer4[1].WaitT;

FOR i FROM 1 TO NUMBER(Job) # Цикъл за определяне на  
следващото време за напускане

REPEAT

## Стратегии

```
IF (Job:Customer4[i].WaitT < P)
    DO
        P := Job:Customer4[i].WaitT;
    END
END_LOOP
END
RETURN (P);    # Команда за връщане
END_FUNC
```

# ЛЕКЦИЯ 9

## СЪЗДАВАНЕ НА КОМПОНЕНТИ ОТ ВИСОКО РАВНИЩЕ

-  **Разделяне на модела CedarBog**
-  **Компоненти от високо равнище в  
модела CedarBog**
-  **Разделяне на модели с региони**
-  **Състояния и преходи между  
състояния в йерархични модели**

# **ВЪВЕДЕНИЕ**

**SIMPLEX MDL поддържа йерархични модели.**

**Това означава, че компонента може да се изгради от подкомпоненти, които са свързани чрез връзки (Connections).**

# РАЗДЕЛЯНЕ НА МОДЕЛА CEDARB OG

Разгледахме модела CedarBog като единична компонента. Възможно е модела да се раздели. Едно от най-важните свойства на SIMPLEX MDL е, че се позволява моделите да бъдат разделяни по произволен начин без това да влияе върху симулационните резултати.

Ще представим примерно разделяне на модела CedarBog, което съдържа слънцето като източник, езерото, околната среда и мъртвата органична материя.

# **РАЗДЕЛЯНЕ НА МОДЕЛА CEDARBOG**

## **BASIC COMPONENT Sun1**

**USE OF UNITS**

**TIMEUNIT = [a]**

**DECLARATION OF ELEMENTS**

**CONSTANTS**

**Pi(REAL) := 3.14**

# РАЗДЕЛЯНЕ НА МОДЕЛА CEDARBOG

## DEPENDENT VARIABLES CONTINUOUS

suns (REAL[kJ/m<sup>2</sup>]) := 0 [kJ/m<sup>2</sup>] #  
Слънчева енергия

## DYNAMIC BEHAVIOUR

suns := 95.9[kJ/m<sup>2</sup>] \* (1+0.635 \* SIN (2[1/a] \* Pi \* T));

END OF Sun1

# РАЗДЕЛЯНЕ НА МОДЕЛА CEDARB OG

Всяка от тези четири компоненти е независима основна компонента. Те са свързани заедно в компонентата от по-високо равнище CedarBog\_High1.

# РАЗДЕЛЯНЕ НА МОДЕЛА CEDARBOG

BASIC COMPONENT Lake1

USE OF UNITS

TIMEUNIT = [a]

DECLARATION OF ELEMENTS

CONSTANTS

Bio\_Fac (REAL[a]) := 1E-15 [a]

# РАЗДЕЛЯНЕ НА МОДЕЛА CEDARBOG

STATE VARIABLES  
CONTINUOUS

p (REAL[t]) := 0 [t], # Растения  
h (REAL[t]) := 0 [t], # Тревопасни  
c (REAL[t]) := 0 [t] # Месоядни

DEPENDENT VARIABLE  
CONTINUOUS

sun\_bio (REAL[t/a]) := 0 [t/a]

# РАЗДЕЛЯНЕ НА МОДЕЛА CEDARBOG

SENSOR VARIABLE

CONTINUOUS

sun (REAL[kJ/m<sup>2</sup>]) := 95.9 [kJ/m<sup>2</sup>] #

Слънчева енергия

DYNAMIC BEHAVIOUR

sun\_bio := sun \* Bio\_Fac;

# РАЗДЕЛЯНЕ НА МОДЕЛА CEDARBOG

DIFFERENTIAL EQUATIONS

$p' := \text{sun\_bio} - 4.03[1/a] * p;$

$h' := 0.48[1/a] * p - 17.87[1/a] * h;$

$c' := 4.85[1/a] * h - 4.65[1/a] * c;$

END

END OF Lake1

# РАЗДЕЛЯНЕ НА МОДЕЛА СЕДАРВОГ

Променливи, които правят достъпна стойността на променлива от друга компонента се наричат сензорни променливи.

Ако от вън се внесе променлива, тя трябва да се декларира като сензорна променлива.

Такъв е случая за променливата sun в компонентата Lake1. Не е необходима съответна декларация за променлива, която ще се експортира. В компонентата Sun1 няма индикация, че друга компонента ще има достъп до променливата suns.

Това означава, че всяка величина на модела в компонентата е достъпна отвън без това да се знае вътре в компонента.

Този подход се нарича glass-box-concept. Той има предимството, че други компоненти могат да декларират сензорни променливи без да е нужно да се изменя компонентата, чийто променливи са достъпни. В частност не е необходимо да се прекомпилира компонента.

# РАЗДЕЛЯНЕ НА МОДЕЛА CEDARBOG

Връзката символизира причинена зависимост, а не поток от материал или енергия. Динамичното поведение на компонентата Lake1 се влияе от променливата sun, чието поведение се определя в компонента Sun1. Това означава, че има причинна връзка от компонента Sun1 към компонента Lake1.

# РАЗДЕЛЯНЕ НА МОДЕЛА CEDARBOG

BASIC COMPONENT Organic1

USE OF UNITS

TIMEUNIT = [a]

DECLARATION OF ELEMENTS

STATE VARIABLES

CONTINUOUS

o (REAL[t]) := 0 [t] # Мъртва органична материя

# РАЗДЕЛЯНЕ НА МОДЕЛА CEDARBOG

SENSOR VARIABLES

CONTINUOUS

p (REAL[t]), # Растения

h (REAL[t]), # Тревопасни

c (REAL[t]) # Месоядни

DYNAMIC BEHAVIOUR

DIFFERENTIAL EQUATIONS

$o' := 2.55[1/a] * p + 6.12[1/a] * h + 1.95[1/a] * c;$

END

END OF Organic1

# РАЗДЕЛЯНЕ НА МОДЕЛА CEDARBOG

Ще разгледаме основните компоненти Organic1 и Environ1. Тяхното поведение се влияе от променливите на състоянието  $p$ ,  $h$  и  $c$  от компонентата Lake.

# РАЗДЕЛЯНЕ НА МОДЕЛА CEDARBOG

## BASIC COMPONENT Environ1

USE OF UNITS

TIMEUNIT = [a]

DECLARATION OF ELEMENTS

STATE VARIABLES

CONTINUOUS

e (REAL[t]) := 0 [t]

# РАЗДЕЛЯНЕ НА МОДЕЛА CEDARBOG

SENSOR VARIABLES

CONTINUOUS

p (REAL[t]),	# Растения
h (REAL[t]),	# Тревопасни
c (REAL[t])	# Месоядни

DYNAMIC BEHAVIOUR

DIFFERENTIAL EQUATIONS

$e' := 1.00[1/a] * p + 6.90[1/a] * h + 2.70[1/a] * c;$

END

END OF Environ1

# РАЗДЕЛЯНЕ НА МОДЕЛА СЕДАРВОГ

Досега бяха описани само основни компоненти. Чрез деклариране на променливи като сензорни, техните стойности се внасят отвън. Връзката все още не е описана. Все още имаме да определим кои променливи от кои компоненти ще се свържат. Тази връзка се дефинира в компоненти от високо равнище.

# **РАЗДЕЛЯНЕ НА МОДЕЛА CEDARB OG**

**HIGH LEVEL COMPONENT CedarBog\_High1**

## **SUBCOMPONENTS**

Environ1,

Organic1,

Lake1,

Sun1

**РАЗДЕЛЯНЕ  
НА МОДЕЛА CEDARB OG  
COMPONENT CONNECTIONS**

Lake1.c --> Organic1.c;

Lake1.h --> Organic1.h;

Lake1.p --> Organic1.p;

Sun1.suns --> Lake1.sun;

Lake1.c --> Environ1.c;

Lake1.h --> Environ1.h;

Lake1.p --> Environ1.p;

END OF CedarBog\_High1

# РАЗДЕЛЯНЕ НА МОДЕЛА СЕДАРВОГ

Забележка:

Динамичното поведение на модела се описва в основната компонента. Компонентите от високо равнище описват само структурата на модела.

Различни връзки могат да имат начало в един и същ елемент.

$A.s \rightarrow (B.x, C.y) ;$

е синоним на:

$A.s \rightarrow B.x;$

$A.s \rightarrow C.y;$

Не са позволени множество връзки към един и същ елемент, за да се осигури уникалност.

# РАЗДЕЛЯНЕ НА МОДЕЛА CEDARB OG

За да се изпълни модела са необходими следните стъпки:

- 1) Редактират се основните компоненти Sun1, Lake1, Organic1 и Environ1 и компонентата от високо равнище CedarBog\_High1.
- 2) Използва се команда Check Version, за да провери синтаксиса на основните компоненти, за да се преведат в С код. Това ги привежда в състояние Checked (проверена).

Командата Check Version за компонента от високо равнище CedarBog\_High1 води до изпълнението на следните действия:

- 1) Синтактична проверка на компонентата CedarBog\_High1 и всички нейни подчинени компоненти и превеждането им в С код.
- 2) Проверка на съгласуваността. Тук се проверява дали са валидни връзките, създадени в компонентата от високо равнище.

Командата Prepare version компилира всички компоненти в обектен код и ги привежда в състояние Prepared (подготвена).

# РАЗДЕЛЯНЕ НА МОДЕЛА CEDARB OG

Тогава трябва да се създаде и инсталира модел.

На него трябва да му се даде име MCedarBog-Version1. Допълнително трябва да се създаде и стартира нов експеримент ECedarBog-Version1.

Създава се наблюдател. В йерархично структуриран модел трябва да се даде пълния път на наблюдателя, за да следи съответните променливи. Това означава, че трябва да се отбележат имената на моделните величини и имената на компонентите, които той съдържа.

# Компоненти от високо равнище в модела CedarBog

Сега разделяме допълнително модела CedarBog.

Компонентата Lake се разбива на три подкомпоненти, за да представи индивидуалните пластове вода. Приема се, че наситеността на слънчевата радиация е различна във всеки пласт.

# Компоненти от високо равнище в модела CedarBog

Трите нива са идентични като структура и динамики.

Това означава, че всичките три нива могат да се разглеждат като инстанции на класа Layer2.

Всички константи са запазени заедно, заради опростеност. Трябва да се отчита, че динамичното поведение на всяко равнище да може да се различава. Единственото изменение е добавянето на константата  $f$ , която представя различната сила на слънчевата радиация. В описанието на класа Layer2 константата  $f$  се инициализира с  $f := 1$ . Тогава всяка инстанция ще получи своя собствена стойност.

# **Компоненти от високо равнище в модела CedarBog**

**Забележка:**

Компонентата Layer2 може да се изпълни също като независима основна компонента.

# Компоненти от високо равнище в модела CedarBog

BASIC COMPONENT Layer2

USE OF UNITS

TIMEUNIT = [a]

DECLARATION OF ELEMENTS

CONSTANT

f (REAL) := 1, # Слънчев фактор

Bio\_Fac (REAL[a]) := 1E-15 [a]

# Компоненти от високо равнище в модела CedarBog

STATE VARIABLES

CONTINUOUS

$p$  (REAL[t]) := 0 [t],

$h$  (REAL[t]) := 0 [t],

$c$  (REAL[t]) := 0 [t],

$o$  (REAL[t]) := 0 [t],

$e$  (REAL[t]) := 0 [t]

DEPENDENT VARIABLE

CONTINUOUS

$\text{sun\_bio}$  (REAL[t/a]) := 0 [t/a]

SENSOR VARIABLE

CONTINUOUS

$\text{sun}$  (REAL[kJ/m^2]) := 0 [kJ/m^2]

# Компоненти от високо равнище в модела CedarBog

## DYNAMIC BEHAVIOUR

`sun_bio := sun * Bio_Fac;`

### DIFFERENTIAL EQUATIONS

```
p' := f * sun_bio - 4.03[1/a] * p;  
h' := 0.48[1/a] * p - 17.87[1/a] * h;  
c' := 4.85[1/a] * h - 4.65[1/a] * c;  
o' := 2.55[1/a] * p + 6.12[1/a] * h + 1.95[1/a] * c;  
e' := 1.00[1/a] * p + 6.90[1/a] * h + 2.70[1/a] * c;
```

`END`

`END OF Layer2`

# Компоненти от високо равнище в модела CedarBog

В моделното описание за Lake2 трябва първо да регистрираме списъка на подкомпонентите, които той съдържа. Това са трите инстанции на класа Layer2, наречени Layer2\_1, Layer2\_2 и Layer2\_3.

В началото трите пласта се инициализират идентично. Те са идентични инстанции на един компонентен клас.

Последователно всички променливи във всяка инстанция могат да получат индивидуални стойности при инициализация. Това се извършва от компонентата от високо равнище, в случаят компонентата Lake2.

Като допълнение трябва да декларираме променливите в Lake2. В компонентите от по-високо равнище, които са сами по себе си подкомпоненти и могат също да съдържат подкомпоненти, променливите не могат да бъдат променяни.

# Компоненти от високо равнище в модела CedarBog

Те само могат да преминат от едно равнище в йерархията към друго.

Декларацията на моделните величини, които се разпространяват отгоре надолу, се извършва чрез входна връзка. Моделните величини, които са деклариирани във входна връзка отговарят на сензорни променливи. В този случай връзка може да се създаде при следващото по-високо равнище.

Декларацията на входна връзка трябва да декларира към коя моделна величина в коя компонента ще премине съответната променлива.

Затова входната връзка има функции от две части. Тя декларира моделна величина в компонентата и определя с какво ще се свърже тази моделна величина.

# Компоненти от високо равнище в модела CedarBog

Изходните еквиваленти се използват в компонентата от високо равнище, за да декларират моделна величина, която има същата функция в компонентата от по-високо равнище, която има в основната компонента. Тогава те могат да получат достъп отвън чрез връзки съответни на концепцията на стъклена кутия.

По същото време изходните еквиваленти (като входни връзки) описват коя моделна величина от подчинената компонента ще се използва.

# **Компоненти от високо равнище в модела CedarBog**

## **HIGH LEVEL COMPONENT Lake2**

### **SUBCOMPONENTS**

Layer2\_1 OF CLASS Layer2,

Layer2\_2 OF CLASS Layer2,

Layer2\_3 OF CLASS Layer2

### **INPUT CONNECTIONS**

sun --> (Layer2\_1.sun, Layer2\_2.sun,  
Layer2\_3.sun);

# Компоненти от високо равнище в модела CedarBog

## OUTPUT EQUIVALENCES

o1 := Layer2\_1.o;

e1 := Layer2\_1.e;

o2 := Layer2\_2.o;

e2 := Layer2\_2.e;

o3 := Layer2\_3.o;

e3 := Layer2\_3.e;

## INITIALIZE

Layer2\_1.f := 1;

Layer2\_2.f := 0.8;

Layer2\_3.f := 0.6;

END OF Lake2

# Компоненти от високо равнище в модела CedarBog

Компонентите Sun2, Organic2 и Environ2 са на същото равнище от йерархията като компонентата Lake2. Те са основни компоненти.

Четирите компоненти Sun1, Lake2, Organic2 и Environ2 трябва да са свързани на най-високото равнище на абстракция в компонентата CedarBog\_High2.

# Компоненти от високо равнище в модела CedarBog

BASIC COMPONENT Sun2

USE OF UNITS

TIMEUNIT = [a]

DECLARATION OF ELEMENTS

CONSTANTS

Pi(REAL) := 3.14

# Компоненти от високо равнище в модела CedarBog

DEPENDENT VARIABLES

CONTINUOUS

suns (REAL[kJ/m^2]) := 0 [kJ/m^2]

DYNAMIC BEHAVIOUR

suns := 95.9[kJ/m^2] \* (1+0.635 \* SIN (2[1/a] \*  
Pi \*T));

END OF Sun2

# Компоненти от високо равнище в модела CedarBog

BASIC COMPONENT Organic2

USE OF UNITS

TIMEUNIT = [a]

DECLARATION OF ELEMENTS

DEPENDENT VARIABLES

CONTINUOUS

o (REAL[t]) := 0 [t]

# Компоненти от високо равнище в модела CedarBog

SENSOR VARIABLES

CONTINUOUS

o1 (REAL[t]),

o2 (REAL[t]),

o3 (REAL[t])

DYNAMIC BEHAVIOUR

$o := o1 + o2 + o3;$

END OF Organic2

# **Компоненти от високо равнище в модела CedarBog**

**BASIC COMPONENT Environ2**

**USE OF UNITS**

**TIMEUNIT = [a]**

**DECLARATION OF ELEMENTS**

**DEPENDENT VARIABLES**

**CONTINUOUS**

**e (REAL[t]) := 0 [t]**

# Компоненти от високо равнище в модела CedarBog

SENSOR VARIABLES

CONTINUOUS

e1 (REAL[t]) := 0 [t],

e2 (REAL[t]) := 0 [t],

e3 (REAL[t]) := 0 [t]

DYNAMIC BEHAVIOUR

e := e1 + e2 + e3;

END OF Environ2

# **Компоненти от високо равнище в модела CedarBog**

**HIGH LEVEL COMPONENT CedarBog\_High2**

## **SUBCOMPONENTS**

Environ2,

Organic2,

Lake2,

Sun2

# Компоненти от високо равнище в модела CedarBog

## COMPONENT CONNECTIONS

Lake2.e1 --> Environ2.e1;

Lake2.e2 --> Environ2.e2;

Lake2.e3 --> Environ2.e3;

Sun2.suns --> Lake2.sun;

Lake2.o1 --> Organic2.o1;

Lake2.o2 --> Organic2.o2;

Lake2.o3 --> Organic2.o3;

END OF CedarBog\_High2

# Компоненти от високо равнище в модела CedarBog

Пълният синтаксис за компоненти от високо равнище има вида:

high\_level\_component ::= HIGH LEVEL COMPONENT  
identifier

[ dimension\_constant\_definition\_part ]  
subcomponent\_declaration  
[ input\_connection\_part ]  
[ output\_equivalence\_part ]  
[ component\_connection\_part ]  
[ initialization\_part ]  
END OF identifier

# Компоненти от високо равнище в модела CedarBog

Описанието на езика дава следното за

input\_connection\_part:

input\_connection\_part ::= INPUT CONNECTION [S]

input\_connection { input\_connection }

Всички input\_connections са последвани от ключовите думи INPUT CONNECTION или алтернативно INPUT CONNECTIONS.

Input\_connections имат следната форма:

input\_connection ::= identifier '-->' input\_ident ';'

| identifier '-->'

‘(‘ input \_ident { ‘,’ input \_ident }’ )’ ‘;’

# Компоненти от високо равнище в модела CedarBog

Тя свързва името на моделната величина с input\_ident чрез символа '-->'.

input\_ident се дава чрез:

input\_ident ::= indexed\_identifier '.' identifier

Може да се види, че са позволени само пътища с две равнища.

Същото е вярно и за изходните еквиваленти и инициализацията.

В частност е възможно директно да се инициализира само от едно равнище на йерархията към следващо. Следният израз не е позначен:

INITIALIZE

Lake2.Layer2\_1.f := 1;

# Компоненти от високо равнище в модела CedarBog

Ако константата f в компонентата CedarBog\_High2 се инициализира, тя първо трябва да се направи достъпна чрез изходен еквивалентен израз в компонента Lake2.

## OUTPUT EQUIVALENCE

f1 := Layer2\_1.f;

f1 е нормална моделна величина в компонента Lake2, на която може да се присвои стойност в следващата по-висока компонента CedarBog\_High2 чрез следния израз:

## INITIALIZE

Lake.f1 := 1;

# Компоненти от високо равнище в модела CedarBog

За да бъде възможен достъп до моделна величина от подчинени компоненти, в наблюдателя са позволени по-дълги пътища в прозореца за параметрите. Ако например променливата на състоянието  $p$  от компонента Layer2\_3 трябва да се запише от наблюдател OLayer2\_3 (наблюдател за променливата на състоянието Layer2\_3), верният път е:

CedarBog\_High2/Lake2/Layer2\_3/p

# Компоненти от високо равнище в модела CedarBog

Забележки:

- Компонентите се обработват от стартиращи контроли в реда, в който са деклариирани в компонента от високо равнище. Наредбата няма влияние върху симулационния резултат.
- Когато се компилира йерархичен модел, потребителят трябва да се уверим, че сме избрали правилната текуща версия за всички подчинени компоненти.

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

Моделите Queue и Transport се състоят от индивидуални секции, които се комбинират от основна компонента. Тези секции могат да се изнесат в независими компоненти, които се свързват, за да формират цялостен модел.

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

Разделяне на модел Queue.

Моделът Queue се състои от източник, опашка, сървър и контейнер. Източникът, сървърът с опашка и контейнера ще бъдат конвертираны в основни компоненти.

# **РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ**

HIGH LEVEL COMPONENT Queue5

SUBCOMPONENTS

Source5,

Station5,

Sink5

COMPONENT CONNECTION

Source5.OutputP --> Station5.WaitP;

Station5.OutputP --> Sink5.WaitP;

END OF Queue5

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

В компонентата Source5 новосъздадените работи се събират в регион OutputP. В компонентата от високо равнище Queue5 този регион се свързва с региона WaitP в компонента Station5. Региона WaitP се декларира като сензорна променлива в компонента Station5. Работите се изтриват оттук както се изисква и се преместват в региона Station.

Няма разлика между GET и SEND командите в основна компонента.

Могат да се променят региони, но не и сензорни региони. Сензорните региони не могат да стоят отляво на ADD, GET, SEND, REMOVE или PLACE команда.

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

BASIC COMPONENT Source5

MOBILE SUBCOMPONENTS OF CLASS Customer5

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

TArrive (REAL) := 0,

Protocol (LOGICAL) := FALSE

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

## RANDOM VARIABLES

Arrive (REAL) : EXPO

(Mean:=15,LowLimit:=0.5,UpLimit:=75)

## LOCATIONS

OutputP (Customer5) := 0 Customer5

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

## DYNAMIC BEHAVIOUR

```
# Създаване на задача
WHENEVER T >= TArrive
DO
    OutputP^ : ADD 1 NEW Customer5;
    TArrive^ := T + Arrive;
    IF Protocol
        DO DISPLAY ("T= %f New customer \n",T); END
    END
END OF Source5
```

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

Station е деклариран като регион. На него е присвоена подвижна компонента от клас Customer5. На следващия цикъл работата ще се намира в Station. От дясната страна, мястото, от което е преместена подвижната компонента е сензорен регион.

GET и SEND трябва да се изберат подходящо, за да осигурят, че лявата страна на присвояването съдържа региони.

За да се постигне това временния регион WaitP се въвежда в спад. Това се използва само за прехвърляне на подвижни компоненти в региона Sink. Тогава те могат да се унищожат.

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

BASIC COMPONENT Station5

MOBILE SUBCOMPONENTS OF CLASS Customer5

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

TWork (REAL) := 0,

Protocol (LOGICAL) := FALSE

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

## RANDOM VARIABLES

Work (REAL) : EXPO

(Mean:=10,LowLimit:=0.32,UpLimit:=50)

## LOCATIONS

Station (Customer5) := 0 Customer5,

OutputP (Customer5) := 0 Customer5

## SENSOR LOCATION

WaitP (Customer5)

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

DYNAMIC BEHAVIOUR

# Начало на обработката

WHENEVER (NUMBER(Station)=0) AND  
(NUMBER(WaitP)>0)

DO Station<sup>^</sup> : FROM WaitP GET Customer5[1];

TWork<sup>^</sup> := T + Work;

IF Protocol

DO DISPLAY ("T= %f Cust. enters Station \n",T);

END

END

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

# Край на обработката

WHENEVER (T >= TWork) AND  
(NUMBER(Station)=1)

DO

Station<sup>^</sup> : TO OutputP SEND Customer5[1];

IF Protocol

DO DISPLAY ("T= %f Cust. leaves Station \n",T);  
END

END

END OF Station5

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

BASIC COMPONENT Sink5

MOBILE SUBCOMPONENTS OF CLASS  
Customer5

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

Protocol (LOGICAL) := FALSE

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

## LOCATIONS

Sink (Customer5) := 0 Customer5

## SENSOR LOCATION

WaitP (Customer5)

## DYNAMIC BEHAVIOUR

```
# Преместване на задачи към контейнера
WHENEVER NUMBER(WaitP) >= 1
```

```
DO
```

```
    Sink^ : FROM WaitP GET Customer5[1];
END
```

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

# Унищожаване на задачи

WHENEVER NUMBER(Sink) >= 4

DO

Sink<sup>^</sup> : REMOVE Customer5{ALL};

IF Protocol

DO DISPLAY("T= %f 4 Customers destroyed \n",

T); END

END

END OF Sink5

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

Всички преходи на състоянията са начално в една основна компонента. Добра идея е да се направят индивидуални сегменти като WaitP или Station1 в отделни компоненти, които са свързвани заедно в компонента от високо равнище Trans2.

Поради тяхната подобност, всяка от компонентите QTaxi1, QTaxi2 и Road1, Road2 се разглежда като членове на класове съответно QTaxi\_2 и Road\_2.

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

## HIGH LEVEL COMPONENT Trans2

### SUBCOMPONENTS

WaitP2,

QTaxi1 OF CLASS QTaxi\_2,

QTaxi2 OF CLASS QTaxi\_2,

Road1 OF CLASS Road\_2,

Road2 OF CLASS Road\_2,

Station1\_2,

Station2\_2

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

## COMPONENT CONNECTIONS

# Региони

QTaxi1.LocQTaxi --> Station1\_2.LocQTaxi;

WaitP2.LocWaitP --> Station1\_2.LocWaitP;

Station1\_2.LocSta1 --> Road1.LocSta;

Road1.LocRoad --> QTaxi2.LocRoad;

QTaxi2.LocQTaxi --> Station2\_2.LocQTaxi;

Station2\_2.LocSta2 --> Road2.LocSta;

Road2.LocRoad --> QTaxi1.LocRoad;

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

# Променливи на състоянието

Station1\_2.TLoad --> Road1.TLoad;

Station2\_2.TUnload --> Road2.TLoad;

INITIALIZE

Road1.RoadN := 1;

Road2.RoadN := 2;

QTaxi1.QTaxiN := 1;

QTaxi2.QTaxiN := 2;

END OF Trans2

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

Аналогично се извършва инициализацията с 10 таксита на Trans1 в компонента QTaxi1.

Инициализацията се извършва след създаването на експеримента и първото изпълнение чрез избирането на Break0. Тогава може да се използва команда Model Parameters в контекстното меню.

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

В прозореца на измененията QTaxi1 може да се разшири и тогава да се избере региона LocQTaxi. След натискане на бутона Change се появява входно поле, в което на броя на подвижните компоненти може да се даде стойност 10. В дясната част на прозореца на измененията се показва команда

AddMobile/Trans2/QTaxi1/LocQTaxi 10

Промяната може да се провери чрез проверка на стойностите за следващия сегмент на следващото изпълнение. Това се постига чрез избиране на Break0 и изпълняване на команда Show Initial State.

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

BASIC COMPONENT QTAXI\_2

MOBILE SUBCOMPONENTS OF CLASS TAXI2  
DECLARATION OF ELEMENTS

STATE VARIABLES

QTAXIN(INTEGER) := 1, # Номер на QTAXI  
Protocol(LOGICAL) := FALSE, # Контрол на  
протокола

TaxiOut(INTEGER) := 0 # Такси излиза от  
маршрут

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

TRANSITION INDICATORS

Print

LOCATIONS

LocQTaxi (Taxi2) := 0 Taxi2

SENSOR LOCATIONS

LocRoad(Taxi2)

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

DYNAMIC BEHAVIOUR

ON START

DO

IF QTaxiN = 1

DO

LocQTaxi<sup>^</sup>: FROM LocQTaxi GET Taxi2{ALL i}

CHANGING

TaxiN<sup>^</sup> :=i;

END

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

IF Protocol

DO

    DISPLAY(" T= %f Component QTaxi%ld \n", T,  
    QTaxiN);

    DISPLAY(" Numbering of Taxis \n\n");

END

END

END

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

```
IF NUMBER(LocRoad) > 0
DO
  WHENEVER T >= LocRoad:Taxi2[1].TTravel
  DO
    LocQTaxi^ : FROM LocRoad GET Taxi2[1];
    TaxiOu^ := LocRoad:Taxi2[1].TaxiN;
    SIGNAL Print;
  END
END
```

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

```
# Изходна функция
ON Print
DO
  IF Protocol
    DO
      DISPLAY(" T= %f Component QTaxi%ld \n", T,
QTaxiN);
      DISPLAY("  Taxi Number %ld enters QTaxi%ld \n\n",
TaxiOu, QTaxiN);
    END
  END
END OF QTaxi_2
```

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

BASIC COMPONENT WaitP2

MOBILE SUBCOMPONENTS OF CLASS  
Customer2

DECLARATION OF ELEMENTS

STATE VARIABLES

TNext(REAL) := 0

Protocol(LOGICAL) := FALSE

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

## RANDOM VARIABLES

ZArrive(REAL) :

EXPO(Mean:=10,LowLimit:=0.32,UpLimit:=50)

## TRANSITION INDICATORS

Print

## LOCATIONS

LocWaitP(Customer2) := 0 Customer2

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

## DYNAMIC BEHAVIOUR

```
WHENEVER T >= TNext
DO
    LocWaitP^ : ADD 1 NEW Customer2;
    TNext^ := T + ZArrive;
    SIGNAL Print;
END
```

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

ON Print

DO IF Protocol

DO

DISPLAY(" T= %f Component WaitP \n",T);

DISPLAY(" Cust. enters WaitP \n");

DISPLAY(" Next generation TNext = %f \n",TNext);

DISPLAY(" Number of cust. in WaitP2 = %ld  
\n",NUMBER(LocWaitP));

END

END      END OF WaitP2

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

BASIC COMPONENT Station1\_2

MOBILE SUBCOMPONENTS OF CLASS Taxi2,Customer2

DECLARATION OF ELEMENTS

CONSTANTS

Load(REAL) := 3

STATE VARIABLES

TLoad (REAL) := 0,

Protocol(LOGICAL) := FALSE

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

TRANSITION INDICATORS

Print1,

Print2

LOCATIONS

LocSta1(Taxi2) := 0 Taxi2

SENSOR LOCATIONS

LocWaitP(Customer2),

LocQTaxi(Taxi2)

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

DYNAMIC BEHAVIOUR

WHENEVER (NUMBER(LocWaitP) > 0) AND  
(NUMBER(LocQTaxi) > 0) AND  
(NUMBER(LocSta1) = 0)

DO

LocSta1<sup>^</sup> : FROM LocQTaxi GET Taxi2[1];  
SIGNAL Print1;

END

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

```
IF NUMBER(LocSta1) = 1
DO
  WHENEVER NUMBER(LocSta1:Taxi2[1].Seat) = 0
    DO
      LocSta1:Taxi2[1].Seat^ : FROM LocWaitP GET
        Customer2{ALL i | i <= 4 };
      TLoad^ := T + Load;
      SIGNAL Print2;
    END
  END
```

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

ON Print1

DO

IF Protocol

DO

DISPLAY(" T= %f Component Station1 \n",T);

DISPLAY(" Taxi Number %ld enters Station1 \n\n",  
LocSta1:Taxi2[1].TaxiN);

END

END

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

ON Print2

DO

IF Protocol

DO

DISPLAY(" T= %f Component Station1 \n",T);

DISPLAY(" Taxi Number %ld, Num. cust. %ld \n",

LocSta1:Taxi2[1].TaxiN,

NUMBER(LocSta1:Taxi2[1].Seat));

DISPLAY(" End loading in T= %f \n\n",TLoad);

END

END

END OF Station1\_2

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

BASIC COMPONENT Station2\_2

MOBILE SUBCOMPONENTS OF CLASS Taxi2,Customer2

DECLARATION OF ELEMENTS

CONSTANT

Unload(REAL) := 2

STATE VARIABLES

TUnload (REAL) := 0,

Protocol(LOGICAL) := FALSE,

CustomN (INTEGER) := 0

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

TRANSITION INDICATORS

Print1,

Print2

LOCATIONS

LocSta2(Taxi2) := 0 Taxi2

SENSOR LOCATIONS

LocQTaxi(Taxi2)

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

DYNAMIC BEHAVIOUR

WHENEVER (NUMBER(LocSta2) = 0) AND  
(NUMBER(LocQTaxi) >0)

DO

LocSta2<sup>^</sup> : FROM LocQTaxi GET Taxi2[1];

SIGNAL Print1;

END

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

```
IF NUMBER(LocSta2) =1
DO
  WHENEVER NUMBER(LocSta2:Taxi2[1].Seat) >0
    DO
      LocSta2:Taxi2[1].Seat^ : REMOVE Customer2{ALL};
      TUnload^ := T + Unload;
      CustomN^ := NUMBER(LocSta2:Taxi2[1].Seat);
      SIGNAL Print2;
    END
  END
```

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

ON Print1

DO

IF Protocol

DO

DISPLAY(" T= %f Component Station2 \n",T);

DISPLAY(" Taxi number %ld enters Station2 \n\n",

LocSta2:Taxi2[1].TaxiN);

END

END

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

```
ON Print2
DO
  IF Protocol
    DO
      DISPLAY(" T= %f Component Station2 \n",T);
      DISPLAY("  Taxi number %ld num. cust. %ld \n",
              LocSta2:Taxi2[1].TaxiN,CustN);
      DISPLAY("  Unloading ends in %f \n\n",TUnload);
    END
  END  END OF Station2_2
```

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

BASIC COMPONENT Road\_2

MOBILE SUBCOMPONENTS OF CLASS Taxi2

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

RoadN (INTEGER) := 1,

Protocol(LOGICAL) := FALSE,

TaxiIn (INTEGER) := 0

CONTINUOUS

TaxiTT(REAL) := 0

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

SENSOR VARIABLES

TLoad(REAL) := 0

RANDOM VARIABLES

ZTravel(REAL)  
:GAUSS(Mean:=36,Sigma:=10,LowLimit:=0,UpLimit:=72)

TRANSITION INDICATORS

Print

LOCATIONS

LocRoad(Taxi2 ORDERED BY INC TTravel) := 0 Taxi2

SENSOR LOCATIONS

LocSta(Taxi2)

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

DYNAMIC BEHAVIOUR

ON ^T >= TLoad^

DO LocRoad^ : FROM LocSta GET Taxi2[1]

CHANGING

TTravel^ := T + ZTravel;

END

TaxiIn^ := LocSta:Taxi2[1].TaxiN;

TaxiTT^ := T + ZTravel;

SIGNAL Print;

END

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

ON Print

DO

IF Protocol

DO

DISPLAY(" T= %f Component Road%ld \n", T, RoadN);

DISPLAY(" Taxi number %ld enters Road%ld \n",  
TaxiIn, RoadN);

DISPLAY(" End travelling in T= %f \n\n", TaxiTT);

END

END            END OF Road\_2

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

Масиви от компоненти.

Разглеждаме модела Queue5.

Вместо само един сървър сега моделът ще използва последователност от три сървъра.

Начално можем да използваме концепцията на класа и да декларираме различни инстанции на клас Station. Компонентата Station представя описанието на класа.

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

Моделът може да се подобри от един от тези три сървъра чрез заместване на компонента от високо равнище Queue5 с компонента от високо равнище Queue6.

Тъй като компонентата Station се среща няколко пъти в модела, извежданията не могат да се асоциират уникално с всеки един сървър, когато протоколът е включен (Protocol = TRUE).

Информацията може да се идентифицира чрез даване на всеки сървър на уникален номер StationN. Инициализацията се осъществява в компонента от високо равнище Queue6.

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

HIGH LEVEL COMPONENT Queue6

SUBCOMPONENTS

Source6,

Station\_1 OF CLASS Station6,

Station\_2 OF CLASS Station6,

Station\_3 OF CLASS Station6,

Sink6

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

## COMPONENT CONNECTION

Source6.OutputP --> Station\_1.WaitP;

Station\_1.OutputP --> Station\_2.WaitP;

Station\_2.OutputP --> Station\_3.WaitP;

Station\_3.OutputP --> Sink6.WaitP;

## INITIALIZE

Station\_1.StationN := 1;

Station\_2.StationN := 2;

Station\_3.StationN := 3;

## END OF Queue6

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

BASIC COMPONENT Station6

MOBILE SUBCOMPONENTS OF CLASS

Customer6

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

TWork (REAL) := 0,

StationN (INTEGER) := 0,

Protocol (LOGICAL) := FALSE

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

## RANDOM VARIABLES

Work (REAL) : EXPO (Mean :=  
10,LowLimit:=0.32,UpLimit:=50)

## LOCATIONS

Station (Customer6) := 0 Customer6,  
OutputP (Customer6) := 0 Customer6

## SENSOR LOCATION

WaitP (Customer6)

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

DYNAMIC BEHAVIOUR

WHENEVER (NUMBER(Station)=0) AND  
(NUMBER(WaitP)>0)

DO

Station<sup>^</sup> : FROM WaitP GET Customer6[1];

TWork<sup>^</sup> := T + Work;

IF Protocol

DO DISPLAY ("T= %f Cust. enters Station%ld \n",T,  
StationN); END

END

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

```
WHENEVER (T >= TWork) AND
  (NUMBER(Station)=1)
DO
  Station^ : TO OutputP SEND Customer6[1];
  IF Protocol
    DO DISPLAY ("T= %f Cust. leaves Station%ld
      \n",T, StationN); END
  END
END OF Station6
```

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

По-нататъшно опростяване може да се направи чрез използване на възможностите SIMPLEX MDL, за да се декларират масиви от компоненти.

Позволени са всички операции. Това дава възможност за много ясна и сбита бележка за големи модели с много връзки между компонентите на един и същ клас.

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

HIGH LEVEL COMPONENT Queue7

SUBCOMPONENTS

Source7,

ARRAY [3] Station7,

Sink7

# РАЗДЕЛЯНЕ НА МОДЕЛИ С РЕГИОНИ

## COMPONENT CONNECTION

Source7.OutputP --> Station7[1].WaitP;

Station7{ $i$  OF 1..2}.OutputP -->  
Station7[ $i+1$ ].WaitP;

Station7[3].OutputP --> Sink7.WaitP;

## INITIALIZE

Station7{ALL  $i$ }.StationN :=  $i$ ;

## END OF Queue7

# Състояния и преходи между състоянията в йерархични модели

С добавянето на подкомпоненти трябва да се прибавят по-нататъшните състояния.

Разглеждаме йерархичен модел с подкомпоненти и компоненти от високо равнище, които ще бъдат използвани, за да илюстрират състояния и преходите между състоянията. На най-ниското равнище са основните компоненти A1, A2, B1, B2 и C. Компонентите B1, B2 и C са част от компонента D на следващото по-високо равнище. Накрая компонентите A1, A2 и D се свързват, за да формират компонента E.

Компонента D се състои от подкомпоненти B1, B2 и C. Сама по себе си D е подкомпонент на E. От гледна точка на D, E е подкомпонента.

# Състояния и преходи между състоянията в йерархични модели

Състояния, използвани в управлението на моделната библиотека.

- Непроверена.

Компонента има такова състояние веднага след като се създаде. Това може да се случи или чрез редактиране (текстово или графично) или чрез операция за копиране. Също в това състояние влизат версиите, чийто MDL код се променя и чийто синтаксис и интерфейси все още не са проверени за вярност.

- Синтаксис Наред.

В това средно състояние синтаксиса на компонента е бил променен, но не и съвместимостта на интерфейсите към подкомпонентите.

# Състояния и преходи между състоянията в йерархични модели

- Проверена.

MDL компилаторът е превел компонента в C код успешно.

- Подготвена.

Тук се компилира С кода в обектен код.

# **Състояния и преходи между състоянията в йерархични модели**

Компонента влиза в състояние Синтаксис Наред, когато е сигурно, че описанието на модела в SIMPLEX MDL е вярно. Проверката дали подчинени компоненти са съвместими с компоненти от високо равнище вече не се изпълнява.

# Състояния и преходи между състоянията в йерархични модели

Пример:

Компонентите от високо равнище могат например да описват две променливи в различни компоненти, които ще се свързват. Ако описанието е синтактично вярно, компонентата може да влезе в състояние Синтаксис наред. В това състояние не е сигурно, че и двете променливи са декларириани в компоненти от ниско равнище, тъй като няма извършена проверка за съвместимост.

Само компоненти от високо равнище могат да бъдат в състояние Синтаксис наред. Основните компоненти на най-ниското равнище на йерархията не могат да бъдат в това състояние.

# **Състояния и преходи между състоянията в йерархични модели**

1. Компонента в състояние Непроверена се премества в състояние Синтаксис Наред чрез MDL компилатор, когато се определи, че синтаксиса му е правилен.
2. Преходът от състояние Синтаксис Наред към Проверена се извършва, когато MDL компилатора успешно провери съвместимостта на компонентата.
3. Компонента е в състояние подготвена, когато С кода е компилиран успешно.
4. Когато MDL кода на всяка компонента се промени, тя се връща в състояние Непроверена.

# Състояния и преходи между състоянията в юерархични модели

5. Ако се направят по-нататъшни промени надолу в юерархията на симулационния модел вместо на MDL кода на самата компонента, тогава всички компоненти от по-високо равнище, които са били поне в състояние Проверена, се връщат към състояние Синтаксис наред. Това е следствие от факта, че въпреки че, MDL кода на компонента от високо равнище е все още синтактично коректен, измененията на подчинените компоненти могат да причинят интерфейсите вече да не са верни. Например променлива, която се нуждае от компоненти от високо равнище може да се преименува или изтрива в подчинена компонента.

# Състояния и преходи между състоянията в юерархични модели

Забележки:

- Системата на моделна библиотека съхранява и управлява файлове, отговарящи на всички предишни състояния на компонента в по-напреднало състояние.
- MDL версията на основна компонента, която е например в състояние Проверена, все още съществува. Командата Edit version... получава достъп до тази MDL версия. Тази команда става налична, когато се избере компонентна версия.
- Когато се измени компонента чрез командата Edit version..., компонентата влиза в състояние Неопределена. Файлът, съдържащ С версията на компонента (в състояние Проверена) се изтрива.

# Състояния и преходи между състоянията в йерархични модели

Процесът на превод, и на преходи между състоянията са засегват от потребителските команди New component, Copy, Edit version..., Check version и Current version.

- New component – нова компонента

Създава нова компонента. Компонентата е в състояние Непроверена.

- Copy – копиране

Компонентата е в състояние Непроверена.

- Edit version... - редактиране на версия...

Сменя компонента.

Изменената компонента влиза в състояние Непроверена. Ако модела е йерархичен, управлението на моделната библиотека осигурява, че състоянието на всички компоненти от високо, равнище се променя на Синтаксис Наред. Всички компоненти, които са подкомпоненти на променената компонента запазват състоянието си.

# Състояния и преходи между състоянията в йерархични модели

- Check version

Синтактична проверка и проверка за съвместимост .

Превеждане на С.

Командата Check version премества компонента от състояние Неопределена в състояние Проверена.

Тогава основните компоненти имат проверен синтаксис. Ако има синтактични грешки, се извежда съобщение за грешка. Грешката може да се поправи с команда Edit version... Когато синтаксисът е правилен се извършва превод в С. Тогава компонентата е в състояние Проверена.

# Състояния и преходи между състоянията в йерархични модели

Ако команда Prepare version е приложена на компоненти от високо равнище, се извършват следните действия:

1. Синтактична проверка на компоненти от по-високо равнище и преход в C.

Състояние на компонентата Синтаксис Наред.

2. Превод на всички подчинени компоненти.

Тези компоненти преминават в състояние Проверени.

3. Проверка за съвместимост.

Премества компонентите от по-високо равнище в състояние Проверени. Ако проверката за съвместимост открива грешка, компонентата от по-високо равнище остава в състояние Синтаксис Наред.

# Състояния и преходи между състоянията в юерархични модели

- Prepare version – подготви версия

Компилира до обектен код.

Командата Prepare version привежда компонентата от състояние Checked в състояние Prepared. Компонента и всичките му подкомпоненти се компилират в обектен код и преминават в състояние Подгответа.

Командата Prepare version съдържа извикване на командата Check version за всички компоненти, които все още не са в състояние Проверени. Така команда Prepare version прави възможен прехода от състояние Непроверена в състояние Подгответа. Извикването (ръчно) на Check version в такъв случай не е необходимо.

# Състояния и преходи между състоянията в йерархични модели

- Current version – текуща версия
- Една версия се избира като текуща версия, която се използва в йерархичния модел.
- При йерархичните модели е важно да се осигури, че когато се представят няколко версии на подчинени компоненти, върната версия се избира за текуща.
- Истинското преимущество на концепцията на версията се изяснява, когато се използват йерархично структурирани модели. Управлението на моделната библиотека автоматично осигурява, че от всички възможни версии винаги се използва текущата. Не са необходими промени за описанието на модела, когато се промени версия.

# Състояния и преходи между състоянията в йерархични модели

Следващата фигура показва компонента С в 2 различни версии. Командата Current version може да се използва, за да се специфицира текущата версия. Текущата версия е тази, която се интегрира в модела.

Фигурата показва също, че версиите могат да имат различни нива на сложност. Всички версии трябва да имат идентичен интерфейс към външния свят. Може да се види, че входната и изходната точка е една и съща, въпреки различната вътрешна структура.

# Състояния и преходи между състоянията в йерархични модели

Концепцията на версията има две приложения:

- Могат да се изучат изменения на модела.
- Първо се изгражда приста версия (бърз прототип), която може да се замести по-късно от по-подробна версия.