ИНСТИТУТ ПО ИНФОРМАЦИОННИ И КОМУНИКАЦИОННИ ТЕХНОЛОГИИ – БЪЛГАРСКА АКАДЕМИЯ НА НАУКИТЕ

Контекстно-ориентирано управление на електронни услуги

Дисертационен труд

за присъждане на образователна и научна степен "доктор"
в област *4. Природни науки, математика и информатика*,
професионално направление *4.6 Информатика и компютърни науки*,
докторска програма *Информатика*

Докторант: Владимир Николаев Вълканов

Научни ръководители: акад. проф. дтн. Иван Попчев

София

2013

Съдържание

Индекс на фигурите

Увод

Средите за доставка на електронни образователни услуги стават все по-налагаща се интегрална част на съвременното обучение. В отговор на потребността за подпомагане на обучението посредством използване на съвременни информационни и комуникационни технологии, във ФМИ на Пловдивския университет се разработва инфраструктура, наречена Разпределен център за електронно обучение, познат като DeLC [117]. Предназначението на центъра е да доставя по един контекстно-зависим, адаптивен и персонализиран начин електронни образователни услуги и електронно учебно съдържание, разположени върху физически разделени сървъри [126]. Концептуално DeLC е динамична мрежова структура, състояща се от [127,128,129,48]:

- *Възли* - върху тях могат да бъдат разполагани електронни образователни услуги и хранилища за електронно съдържание;
- *Релации* – специфицират определени зависимости между възлите.

Спрямо предназначението си възлите могат да бъдат от различен тип:

- *Образователни възли* – предоставящи електронни образователни услуги и електронно учебно съдържание посредством подходящ потребителски интерфейс;
- *Помощни възли* – прозрачни за потребителя възли, задачата на които е да подпомагат функционирането на образователните възли;
- *Специализирани възли* – предоставят специфични услуги за потребители със специален статус.

Спрямо достъпа до предлаганите информационни ресурси в модела на DeLC различаваме два вида образователни възли:

- *Възли с фиксиран достъп* – предназначени за доставка на услуги и учебно съдържание предимно през фиксирани комуникационни мрежи, без специална поддръжка на мобилен достъп.

Потребителският интерфейс на тези възли се разработва под формата на образователен сайт или портал;

- *Възли с мобилен достъп* – предназначени за мобилна доставка на услуги и съдържание през InfoStation-базирани комуникационни мрежи посредством посредничеството на специализиран софтуер (мидълуер) [47,46,125].

Възлите могат да оперират самостоятелно или динамично да се свързват помежду си, образувайки комплексни виртуални структури, наречени **образователни клъстери**. В актуалната версия на DeLC са изградени два два образователни клъстера.

Първият клъстер, наречен MyDeLC, се използва за организация и провеждане на електронно обучение във ФМИ, като доставя фиксиран достъп до услугите и електронното съдържание посредство специализиран образователен портал [164]. Клъстерът се състои от четири възела:

- *Образователен портал DeLC* – ядрото на клъстера, доставящо достъп на потребителите до електронните услуги и учебното съдържание;

- *eLSE (e-Learning in Software Engineering)* [171] - специализиран възел, в който са интегрирани SREPTICS [170], rLE [119,130] и eLSEBuilder – средства, предназначени за подпомагане на обучението по *Софтуерни технологии* [118,120,52];

- *Agent* Village *(AV)* [172] - помощен възел, който доставя специализирана помощ на услугите, предоставяни от образователния портал на DeLC, под формата на „асистенти” [121] [131]. Асистентите са реализирани като интелигентни агенти [122]. Разширяване на инфраструктурата на Центъра с този възел цели усилване нейната проактивност, т.е. действие от „от името на потребителя”, „поемане на самоинициатива” и „самоактивиране”, когато асистентите „преценят”, че е необходима тяхната намеса. Агентите не са интегрирани директно в портална архитектура, а „населяват” агентно-ориентирания възел AV.

- *Interface* Node – възел за интеграция на клъстера MyDeLC с информационната система на университета.

Вторият клъстер, наречен InfoStation клъстер [124], предоставя трислойна архитектура, осигуряваща мобилен достъп до услуги и информационни ресурси посредством интелигентни безжични точки на достъп (наричани Information Stations - ISs), разположени около сградата на университета [116,123,124,49]. Като комуникационна мрежа за този клъстер се използва InfoStation архитектура, състояща се от три нива – InfoStation Center, InfoStations и мобилни устройства. Върху InfoStation Center са разположени образователни услуги и необходимите за тяхната обработка информационни ресурси. Различен брой InfoStations се използват като посредници между InfoStation Center и мобилните устройства на потребителите. В концепцията на DeLC ролята на InfoStations е разширена, като върху тях се разполагат също образователни услуги, които се изпълняват и управляват локално (за разлика от тях, разположените върху InfoStation Center стават глобални). Софтуерното осигуряване на мобилния клъстер се разработва като контестно-зависим и адаптивен агентно-ориентиран мидълуер, който е в състояние да открива промените в средата и в зависимост от тях да се адаптира за ефективно изпълнение заявките на потребителите. Разширената концепция за използване на InfoStation мрежата поражда нови проблеми, свързани с идентификация и локализиране на събитията, предизвикващи промени в средата. Така напр., случващите се локални събития върху (вече) активните и интелигентни отделни InfoStations влияят върху средата, в която се изпълняват и управляват потребителските заявки. В тази ситуация особено съществени стават идентификацията на събитията и синхронизацията на необходимите адаптивни активности във времето, т.е. отчитане и управление на темпоралните аспекти на случващото се в мобилния клъстер.

Освен двата клъстера, използвайки модела и технологията на DeLC, са изградени следните среди и компоненти, подпомагащи електронно обучение извън ФМИ на Пловдивския университет:

- *DeLC Test Center (DeTC)* – клъстер с отворена инфраструктура, която предлага услуги за електронно тестване, локализирани върху

различни възли, с възможност за частична и автоматично контролирана интеграция, в рамките на предварително дефинирани виртуални структури [168,94]. DeTC може да се разглежда като един нов подход за мрежово базирано електронно тестване, където взаимодействат физически разпределени интерактивни и кооперативни единици, помощни средства, обучаеми, обучаващи и администратори [93,95].

- *CADeLC (Creativity Assitant)* - специализиран възел за изучаване и идентифициране на креативното мислене и действие на обучаемите. Възелът е адаптация на средата Creativity Assistant, разработена в Де Монтфорт университет, Лестър [160]. CADeLC [162] е програмна среда за събиране и съхраняване информация за настроенията и действията на обучаемите в процеса на усвояване и прилагане на знания. Събраните данни се представят в структури, наречени „карти на креативност" (Creativity Maps). Разработва се концепция за откриване и използване на подходящи анализатори на картите на креативността;

- *Education* Portal *for Secondary School in Brezovo (EP_B)* - възел с фиксиран достъп, в който е реализиран образователен портал на СОУ "Хр. Смирненски", гр. Брезово [163].

Цялостната концепция за DeLC, като контекстно-зависима и адаптивна инфраструктура за електронно обучение е представена в [115].

С реализирането на двата образователни клъстера е изградена стабилна версия на средата за електронно обучение DeLC, която се използва в реалния образователен процес във ФМИ на Пловдивския университет. В съответствие с концепцията за развитие на центъра, следващ етап е изграждане на Виртуално Образователно Пространство (ВОП). В пространството, контекстно-зависимостта на образователните услуги ще се поддържа от автономни интелигентни компоненти с реактивно, интерактивно и проактивно поведение. Съществуващите образователни клъстера поетапно ще се трансформират в такива компоненти. Архитектурата на пространството ще бъде напълно децентрализирана. Трансформацията на DeLC във ВОП се

предприема поради въздействието върху начина на опериране на Интернет-базирани приложения (каквото е DeLC), което ще имат в недалечното бъдеще две глобални тенденции:

- *Интернет*, като мрежа от компютърни мрежи, постепенно се трансформира в *мрежа на нещата* (Internet of Things) [5,38];
- В сегашния, *синтактичен уеб* възниква следващия *семантичен уеб* [11,1].

Първите идеи за създаване на пространството са дадени в [161]. Възможностите за трансформацията на DeLC във ВОП са разгледани в [164]. Изследвания за изграждане и използване на ВОП в средното училище са описани в [161].

Инфраструктурата с която ще оперира ВОП се състои от три функционални слоя [166]:

- Самото *ВОП*;
- *Интегрирана Технологична Платформа* (ИТП) – предоставя интегрирана платформа върху която ще се изгражда ВОП;
- *Компютърна и Комуникационна Инфраструктура* (ККИ) – доставя необходимата за опериране на ВОП хардуерна инфраструктура.

Основната цел на дисертацията е да се ***разработи контекснтно-ориентирано управление на електронните услуги, доставяни във ВОП***. Съществено изискване е реализиращите такова управление софтуерни компоненти да могат лесно да се интегрират в архитектурата на пространството ВОП и същевременно да запазват нейната хомогенност.

За постигане на целта, в дисертацията се следва и адаптира **методологията** за изграждане на контекстно-зависима архитектура, представена в [115]. Основна роля играе понятието „*контекст*”, което съгласно Дей [34,37] е всяка информация, използваема за характеризиране на ситуацията в една идентичност. Идентичност може да бъде човек, местоположение или обект, които са съществени (не могат да бъдат

пренебрегвани) при взаимодействието между потребител и приложение (включително самият потребител и самото приложение). Контекстът може да бъде от различен тип, напр., освен горепосочените също действие, време и т.н. В този смисъл, една система е контекстно-зависима, ако идентифицира и използва контексти при доставка на желана информация и услуги на потребителя, като степента на използваемост зависи от действията на потребителите. В [115], *адаптивността* и *персонализацията* са дефинирани като атрибути на *контекстно-зависимостта*. Целта на адаптивността е да се осигури безпроблемно, прозрачно и адекватно изпълнение на потребителските заявки за услуги и електронно съдържание, като се вземат предвид различните аспекти на контекста. Целта на персонализацията е, софтуерът да е в състояние динамично да се адаптира в зависимост от индивидуалните особености на конкретния потребител. Потребителите трябва да имат усещането, че той (софтуерът) е разработен специално за изпълнение на техните заявки. Така, когато едно приложение идентифицира определени промени в околната среда, то трябва да е в състояние да извърши определени, зависещи от вида на промяната, компенсаторни действия, такива като напр., персонализация на услуги и съдържание, преформатиране на съдържание, превключване към подходящи хранилища на съдържание.

Съществен аспект на контексно-зависимото ВОП е осигуряване на контекстно-ориентирано управление на предлаганите електронни услуги. В разпределената архитектура на пространството промените могат да бъдат идентифицирани само локално, т.е. в обхвата (околната среда) на позиционирания там автономен интелигентен агент (който има само частичен контрол върху околната среда, т.е. само в своя обхват). Така, състоянието на пространството е **подредено** множество от локални състояния. Решаваща за управлението в такава ситуация е възможността за синхронизация между отделните агенти. За да могат да се синхронизират, те се нуждаят от **изчислителен механизъм** за идентифициране и подредба на промените във ВОП. Тук не ни интересува реалното физическо време, а по-скоро моментите на промяна на локалните състояния на пространството. Добро решение е ако този механизъм е **формален**.

От направения предварителен анализ за търсене на възможности за реализиране на поставената цел беше избрана системата Tempura (подробно представена във втората глава на дисертацията). Въпросът е как тази система да се направи използваема във ВОП. Съществено изискване е запазване хомогенността на архитектурата [115].

ВОП се изгражда от отделни интелигентни автономни компоненти, където електронните услуги са снабдени със съответни обслужващи агенти. Архитектурата на пространството е реализирана върху виртуалната машина на Java. Решението, предложено в дисертацията, е съществуващата архитектура да се разшири с *Tempura-базиран изчислителен механизъм*, който да идентифицира локално случващите се във времето промени в пространството и да генерира съответна **наредба**. Реализацията на този механизъм трябва да удовлетворява следните изисквания:

- Да запазва съществуващата функционалност на ВОП;
- Да запазва съществуващата хомогенност на ВОП;
- Да се вгражда безпрепятствено в съществуващата архитектура на ВОП.

Или изводът е, че се нуждаем се от агентно-ориентирана версия на Tempura, имплементирана на езика за програмиране Java.

Съществуват три възможности за реализацията на Tempura:

- Нова реализация, използвайки спецификацията на Tempura;
- Разработване на подходяща „обвивка" около съществуващата C/C++ реализация на Tempura, която да позволи интегрирането й в архитектурата на ВОП;
- Реинженеринг на съществуващата реализация за получаване на агентно-ориентирана версия на Tempura, реализирана на Java.

В дисертацията е избрана третата възможност.

В съответствие с методологията, за постигане целта на дисертацията са дефинирани следните основни задачи на дисертацията:

1) Разработване на подход за реинженеринг на оригиналния интерпретатор на Tempura при спазване на дадените по-горе изисквания;

2) Разработване на модели, поддържащи предложения реинженерингов подход;

3) Изграждане на целевата агентно-ориентирана архитектура;

4) Създаване на нова програмна версия на оригиналния интерпретатор на Tempura (реализиран на C/C++) в съответствие с подхода, предложен в дисертацията.

Дисертацията е структурирана както следва:

- Увод, който прави въведение в изследваната тематика. Тук се прави преглед на базовата архитектурата на разпределения център за електронно обучение, наречен DeLC. Посочени са неговите основни компоненти и тяхното предназначение. Представен е следващия етап на развитие, които е изграждане на Виртуално Образователно Пространство (ВОП) и нуждата от управление на настъпилите в него събития. Коментирани са основни понятия като контекст и контекстно-зависими системи. В увода е дефинирана целта на дисертацията, като са посочени средства и начин за постигането й.

- В първа глава е направен преглед на състоянието на научно-изследователски области, имащи отношение към изследването.

- Във втора глава се прави преглед на базови теоретични модели и системи, които служат за основа на нашата разбработка. Представени са накратко основните концепции на Interval Temporal Logic (ITL), нейния синтаксис и базови оператори. Разгледан е съществуващия интерпретиращ механизъм на ITL наречен Tempura. Представен е начин, по който интерпретаторът Tempura се ползва в реални условия, чрез специално създадена за него околна среда наречена AnaTempura. Тази глава също така представя подход, чрез който съществуващите интерпретатор и околна среда (Tempura и AnaTempura), могат да бъдат

преобразувани в система с подходяща архитектура, удовлетворяща нашите цели и желания.

- Трета глава прави подробен преглед на използвания подход за смяна на архиктетурата на ITL интерпретатора Tempura, с цел постигане на желаната от нас обектно-ориентирана версия написана на програмния език Java. Разгледани са различни аспекти при избора на подходящ подход, като са посочени конкретни предимства и недостатъци. В тази глава е синтезиран основния жизнен цикъл на интерпретатора Tempura, направен е анализ на програмния код съществуващата версия и са описани стъпките за постигането на желаната от нас обектно-ориентирана архитектура. В края на главата е описан резултата от направените стъпки – първа обектно-ориентирана версия на интерпретатора, написан на програмния език Java и наречен от нас jTempura.

- Глава четвърта е посветена на следващата стъпка от разработката, а именно постигането на агетно-ориентирана версия на jTempura. С тази цел е направено описание на базовите характеристики както на обектно-ориентираните (ОО) архитектури, така и на агентно-ориентираните (АО) такива. Представен е сравнителен анализ и са разгледани възможностите за трасформиране на ОО архитектурата в АО. В тази глава също така е описан модел на контекстно-зависима архитектура на Виртуално Образователно Пространство (ВОП). Представен е подход за извършване на трансформацията и избрани от нас трансформатори. Като заключение на четвърта глава е представено подробно описание на постигнатата програмна реализация на агентно-ориентираната версия на jTempura, която ще бъде наричана от нас AjTempura. Тук е описан основния жизнен цикъл на новата система, комуникацията между различните агенти и са предоставени снимки от реалните тестове на системата.

- Заключението обобщава резултатите от дисертацията и дава някои периспективни насоки за продължаване на изследванията по темата на дисертационния труд.

Глава 1 Състояние на изследвания проблем

В тази глава е направен кратък преглед на състоянието на научно-изследователските области, свързани с проведеното в дисертацията изследване.

1.1 Реинженеринг и рефакторинг на наследен софтуер

Съществуването на голям брой наследени софтуерни системи, функционалността на които е използваема в нови условия, поражда необходимостта те да бъдат подобрявани посредством специфичен инженерингов процес, наречен реинженеринг [73]. Реинженеринг се свързва обикновено с еволюционно развитие на софтуера, когато е необходимо напр. разделяне на монолитни системи на отделни модули с цел по-лесното им управление, подобряване на производителността, прехвърляне в нова платформа, подобряване или смяна на архитектурата на софтуерната система, пренаписване на кода на нов език за програмиране при липсваща системна документация, използване на нови технологии, промяна на изискванията на клиентите.

В съвременните условия софтуерът се адаптира и модифицира непрекъснато, неговият код става все по-сложен и първоначалната му архитектура и структура с течение на времето се размива (губи). Това е една от причините основната част от цената за разработка на софтуер да се формира от неговата поддръжка [29,56,67,135].

Въпреки съществуването на добре познати и ефективни подходи за разработка на софтуер, все още не е намерено задоволително решение на проблема за справяне със сложността на кода. Една от причините е съсредоточаване на усилията и изследванията върху разработването на нови изисквания и същевременно подценяване поддръжката на софтуера [135]. За решаване на този проблем е необходимо използване на техники за опростяване на сложността (обикновено посредством декомпозиране, прегрупиране и др.) и същевременно подобряване на вътрешна структура на приложенията. Примери за такива техники са преструктуриране (restructuring) [28] и рефакторинг (refactoring) [45]. Преструктурирането [54,53] е реинженерингова техника за подобряване на кода на софтуера, написан на някои от процедурните езици за програмиране [28,74]. Рефакторингът е реинженерингова техника за подобряване на обектно-ориентиран код [45,86]. И двете техники имат за цел подобряване на кода и структурата при запазване външното поведение на приложенията.

Жизненият цикъл на рефакторинга е последователност от малки трансформации на кода (оттук и името ‘refactoring’ – запазващи поведението на кода трансформации), които в края на процеса могат да предизвикат значително преструктуриране на кода [45]. През целия процес системата се запазва напълно работеща. Рефакторингът се реализира чрез следните стъпки, като всяка дейност обикновено се поддържа от различни средства, техники или формализми [9,99,151,138]:

- Локализиране на местата в кода, където може да бъде приложен рефакторинг [64,7,32];
- Избиране на подходящ рефакторинг от списъка с шаблони за конкретно място от кода;
- Гарантиране, че приложеният рефакторинг запазва поведението на софтуера [86] – създава се тестови случай, който се прилага върху кода преди прилагането на рефакторинг;
- Приложение на рефакторинг;
- Приложение на тестовите случаи, създадени преди прилагането на рефакторинг, след приключване на трансформацията (рефакторинг);
- Определяне на влиянието на рефакторинг върху характеристиките за качество на софтуера (сложност, разбираемост, възможност за поддръжка) или на процеса (производителност, цена, усилия) [32];
- Поддръжка на съгласуваност между кода, на който се прилага рефакторинг и другите софтуерни артефакти (проектни документи, спецификация на изискванията, тестове и др.) [18] [144] [96].

Рефакторингът е специфичен вид реинженеринг на софтуер [33]. Важен проблем е определяне компонентите от наследения софтуер, които трябва да бъдат преработени, вземайки предвид ограниченията на разработчиците и потенциалното влияние на предложените промени. Проблемите възникват още при опитите за разбиране функционалността на наследения софтуер. За разлика от разработването на нов

софтуер, поддържано от различни модели на софтуерни процеси (спирален, водопаден, итеративен), при реинженеринга няма утвърдени работещи модели. Обикновено се използват шаблони [33], които представят добри практики за актуализация на наследен софтуер. В много случаи такива шаблони липсват.

1.2 Интелигентни агенти

В специализираната литература съществуват различни дефиниции за агенти. Ще направим кратък преглед на най-популярните от тях. Според [100] „Един агент е идентичност, която може да възприема *околната среда* посредством *сензори* и въздейства върху тази среда посредством *ефектори*“ – подчертава се наличието на околна среда, с която агентът взаимодейства посредством сензори и ефектори. В [71] „Автономните агенти са изчислителни системи, които обитават някаква комплексна динамична среда, възприемат и действат *автономно* в тази среда и действайки така достигат едно множество от цели или действия, за които са създадени“ - подчертава се автономността на агента, едно особено важно свойство, наличието на което дава възможност на агента да взема решения и предприема действия на базата на своите убеждения и ментални състояния. „Интелигентните агенти непрекъснато извършват три функции: *възприемане* на динамични условия в околната среда, д*ействие* за въздействие на условията в средата и *разсъждения* за интерпретация на възприятията, решаване на проблемите, правене на заключения и определяне на въздействия върху средата“ [60] - съществени свойства на агентите, дефиниращи възможностите им за взаимодействие със средата и разсъждения, което предполага тяхната интелигентност. В съответствие с [51] „Интелигентните агенти са софтуерни компоненти, които извършват някакво множество от операции *от името на един потребител* или друга програма с определена степен на независимост или автономност и действайки така използва някакви знания или представяния на потребителски цели или желания“ – изтъква изискването агентите, като интелигентни софтуерни компоненти, да предприемат действия от името на потребител или друг софтуер. В [156] “Един агент е компютърна система, която е разположена в някаква околна среда и която има възможност за автономно действие в тази среда за удовлетворяване на целите на разработването им”.

Обобщавайки дефиниции от специализираната литература в дисертацията използваме следното определение за интелигентен агент: *компютърна система, която*

може да оперира гъвкаво и автономно в някаква околна среда за постигане на набелязаните си цели. Под „гъвкавост“ разбираме следните свойства на агента:

- Реактивност - възприемане на околната среда и реакция с цел промяната ѝ за постигане на целите;
- Проактивност - целево-ориентирано поведение и поемане на инициатива за постигане на целите;
- Социалност - способност за взаимодействие с други агенти (или хора) за постигане на целите;

В различни случаи агентите могат да бъдат разширявани с допълнителни свойства, като напр.:

- Мобилност - способност на агентите да се преместват в една компютърна мрежа [90,77];
- Правдивост - агентите няма да разменят съзнателно грешна информация [31];
- Доброжелателност - агентите не трябва да имат конфликтни цели, като всеки агент се опитва да извършва това, което се изисква от него [85];
- Рационалност - агентите ще оперират винаги така, че да достигнат целите си и няма да действат така, че да пречат достигането на целите (колкото позволява вярата на агентите) [55];
- Обучение (адаптивност) - агентите подобряват производителността си в течение на времето .

Създаването на системи, съставени от интелигентни агенти (мулти-агентни системи) е комплексна задача, за решаването на която съществуват множеството концепции и същевременно сравнително малко (в сравнение с концепциите) реално функциониращи реализации. В системите за електронно обучение софтуерни агенти могат да бъдат използвани в различни приложения. В такива системи ролята на обучаващ и оценяващ преподавател може частично или дори изцяло да бъде поета от

софтуерен агент. Това би разкрило нови възможности за подобряване на учебния процес, включително от гледна точка на ефективно използване на човешки и преподавателски ресурси, адаптивност и персонализация на обучението, ефективно използване на времето (софтуерът може да работи и извън работното време). Според [158], почти е невъзможно преподавателят да се справи с отговорностите, свързани с обучението на хора с различно културно и образователно ниво, всеки с индивидуално ниво на знания и възприемане. Много е трудно за всеки обучаван да бъдат подбрани подходящите материали за неговото обучение с оглед на споменатите характеристики от индивидуалния му профил. Преподавателят също така трябва регулярно да подобрява учебните материали, поддържайки постоянен контакт с обучаваните. Затова в публикацията се предлага нов модел на обучение, в който да има три типа агенти: агент-преподавател, агент предоставящ учебните материали и студент. Предлага се първите два типа да бъдат автономни софтуерни компоненти, изградени с изкуствен интелект, разположени в глобалното пространство така, че да могат да се кооперират и да обслужват исканията на студентите. Отново според тях подобни системи неминуемо ще се развият вследствие на еволюцията на човечеството. В системата AIMS софтуерни агенти се използват за събиране информация за предпочитаната информация от конкретните потребители [4]. В [98] е описана многослойна архитектура на системи за автоматично управление на образователни курсове. Състои се от два слоя – презентационен и слой за услуги. Презентационният слой отговаря за комуникацията с потребителите чрез визуализация на информация и обработване на данни, въведени от потребителите. Слоят с услугите съдържа базовата функционалност. Той е разделен на 3 подслоя: организационен (реализира бизнес логиката), приложен (предоставя образователни услуги) и агентен (агенти, обслужващи цялостната система). В [66] е предложено използване на агенти като web услуги в мулти-агентна автономна среда. В системата, представена в [88] е реализирано автоматично управление на задачите, възлагани на студентите. В публикациите [153,154,136] са представени идеи и архитектура на система, като се подчертава огромния потенциал на агентно-базираните системи, конструирани така, че да бъдат отворени за външния свят в една разпределена хетерогенна мрежа от агентни платформи, в които работят агенти и услуги. Основната цел на тази система е да направи възможно динамичното, интелигентно и автономно комбиниране на услуга за постигане на потребителска или бизнес цел, и по този начин създавайки съставни услуги, които да отговарят на променящите се и растящи нужди и изисквания. В прототипната агентната система, описана в [152], агенти се използват за

подпомагане на студентската работа в екипи по проекти. В [107] е представена мулти-агентна среда за подпомагане на обучението на студенти по компютърни науки. Използва се хибриден модел, състоящ се от базирано на проблеми и базирано на случаи (problem-based и case-based) обучение и е реализиран в Java платформа. За всеки студент се поддържа персонален агент, който управлява профила му, включващ знания и умения, предпочитания, интереси, записани курсове и т.н. Персоналният агент комуникира с другите агенти и действа от името на студента в търсенето на информация, подпомагайки го при вземането на решения. Той може още да наблюдава процеса на обучение на студента и автоматично да извлича информация, за да изгради профила му. Тази информация се използва за улеснение на преподаването, ориентирано към индивидуалния студент. Преподавателският агент общува със студента и служи като интелигентен наставник по време на курса.

1.3 Виртуални интелигентни пространства

Всеобхватното използване на Интернет и нейната постепенна трансформация в мрежа на предмети (Internet of things), както и глобализирането на киберпространството, са предпоставка за бързото развитие на CPSS (Cyber-Physical-Social Systems). Нарастващото търсене и разработки на негови приложения, предизвиква значими технологични, икономически и социални последици през последните години [140]. Понятието кибер-физически системи (CPS) се използва за специфициране на все по-тясното интегриране и координация между изчислителни ресурси (кибернетично пространство) и материални ресурси (реално пространство) [150]. Тези системи се отличават с тясна интеграция между изчисление, комуникация и контрол, както и с взаимодействие със средата, в която те са разположени.

За много приложни области е целесъобразно отчитане присъствието в CPS на човешкото и социалното измерения. Това се дължи главно на безпрецедентното въздействие на киберпространството върху начина, по който взаимодействат и общуват хората помежду си. Достигнали сме точката, където социалната и човешката динамика става неразделна част от CPS, така че включване на понятието „социално” е напълно оправдано [68].

Като логическо следствие на CPSS възниква понятието за проникващи интелигентни пространства (Pervasive Intelligent Spaces или съкратено PIS), където

хората и обектите взаимодействат интелигентно помежду си по начин, познат като „отвсякъде, по всяко време и по всякакъв начин” (anywhere-anytime-anyhow). Основни технологични подходи, лежащи в основата на такъв тип взаимодействие, са всепроникващия компютинг, комуникацията и контрола. Пространствата стават интелигентни, когато те са в състояние да наблюдават какво се случва вътре в тях, могат да моделират поведението си и да оперират въз основа на собствените си решения, както и да общуват с населяващите ги общности . Очевидно, освен жители, тези пространства изискват изграждане на подходяща информационна инфраструктура.

Разработването на интелигентни пространства изисква значителни усилия за системна интеграция. Съществуват специфични проблеми, които са обект на интензивни научни изследвания в областта на интелигентните пространства, като напр. следните [68]:

- Интелигентни устройства;
- Сензори и събиране на значими данни от физическия свят;
- Динамични комуникационни инфраструктури, свързващи пространствено разпределени устройства;
- Системна архитектура и мидълуер;
- Разбиране на информацията и интерфейси;
- Вземане на решения и планиране на действия.

Интелигентните пространства имат широк спектър от приложения (текущи и потенциални), като напр. те могат да включват здравни грижи [111], контрол на трафика и безопасността [69], роботика [30], контрол на процеси, спестяване на енергия, контрол на околната среда [50], защита на критични инфраструктури [70]. Този вид пространства могат да въведат нови подходи и сценарии за решаване на комплексни проблеми и в областта на електронното обучение.

1.4 Контекстно-зависими системи

Предизвикателство е да се предложи задоволително и пълно определение на понятието „контекст”. Едно от първите определения описва контекста като място, идентичности от хора и заобикалящите ги обекти [104]. В [12] контекстът се определя като активен процес по разпознаване на заобикалящата среда от хората, като се изхожда от техните познавателни способности за нея. Според [101] контекстът се определя като идентичност, местоположение, заобикаляща среда и време. Контекст са знанията на компютъра за елементите на заобикалящата потребителя среда в [20]. Според [105] контекст е абстрактното ниво на текущото състояние, което се констатира от физическите и логически сензори. В [103] се твърди, че важните характеристики на контекста са къде си, с кого и какви са наличните ресурси. Според [89] контекстът е физическото и концептуално състояние (на интересите) на конкретен човек, обект или място. Според [35] контекстът е просто информация за околната среда. Може би най-пълната дефиниция е на Dey [34], където контекст е всяка информация. Която може да се използва за категоризиране на състоянието на една идентичност. Идентичност може да бъде човек, място или обект, който се смята за свързан с взаимодействието между потребител и приложение, като включва самите потребител и приложение.

Най-общо един контекст може да бъде описан с помощта на два атрибута:

- Контекстен тип: време, място, идентификатор;
- Контекстна стойност: необработена информация, събрана от сензорите.

В някои контекстно-зависими системи тези два атрибута не са достатъчни и по тази причина се добавят и други атрибути, като напр. [8]:

- Времеви маркер (time stamp): съдържа датата и часа, когато атрибутът е възприет;
- Източник: съдържа идентификатор на източника (физическия сензор);
- Достоверност: описва точността на разпознатите данни в зависимост от източника.

Една система е контекстно-зависима, когато тя е наясно с околната среда, в която функционира и използва информацията от тази среда, за да се адаптира, така че да осигурява информация или услуга на потребителя. Придобиване на информация (разбирането на контекста) е основен проблем на контекстната зависимост [89]. Контекстът се улавя с помощта на сензори. Сензорът не е понятие, което трябва да се свързва само със сензор в хардуерен смисъл, но и да се свързва с всеки източник на данни, който може да осигури полезна информация за системата. Сензорите могат да се класифицират в три различни категории [8]:

- Физически сензори: известни още като хардуерни сензори;
- Виртуални сензори: това е друг начин за улавяне на данни посредством софтуерни услуги или приложения. Местоположението на даден студент в университета може да се определи не само чрез физически сензор (например GPS), но също така и чрез процесите по „вписване“ и „отписване“ на студента.
- Логически сензори: този тип сензори комбинират информацията за контекста, която е събрана от физическите и виртуалните сензори с цел решаване на по-големи проблеми.

Контекстът е необработена информация, която трябва да бъде обработена и съхранена. За да се постигне това е необходимо контекстът да се дефинира и съхранява в подходяща форма. Основните подходи за моделиране на контексти са следните [133]:

- Модел на ключовата стойност (key-value model): това е най-простата форма на контекстно моделиране. Този модел се използва често, защото е лесен за манипулиране, но пък не дава възможност за прецизно структуриране на значими алгоритми за извличане на контекст.
- Модел на маркиращите тагове (markup scheme model): това са йерархична структура от данни, която се състои от маркиращи тагове с атрибути и съдържания. Съдържанията на маркиращите тагове са рекурсивно свързани с други маркиращи тагове.

- Графичен модел: UML (унифициран език за моделиране) е основно средство за моделиране, което има ефективни графични възможности. Общата структура на UML го прави подходящ за моделиране на контексти. Този тип моделиране е приложимо за реализиране на ERM (Entity Relationship Model), което го прави полезен инструмент при структурирането на релационни бази от данни в архитектури за управление на контекстни системи.

- Обектно-ориентиран модел: динамичната природа на контекста е проблем за контекстно-зависимите системи. Използвайки този подход се постига капсулация и възможност за повторна употреба на данните. В този модел на ниво обект се капсулират детайлите за обработката на контекста и достъпа до контекстната информация се осъществява само чрез специфицирани интерфейси.

- Логически модел: в този модел контекстът се дефинира като факти, изрази и правила, за които се правят заключения базирайки се на набор от други изрази. Характерно за логическите модели е високото ниво на формализация.

- Модел базиран на онтологии: най-общо онтологиите са способни да специфицират концепции и взаимовръзки, т.е. контексти. Добър пример за моделиране на контекст е използването на системата CoBrA [27], която осигурява набор от онтологически концепции за характеризиране на ентитита (като личности, места и устройства).

Контекстно-зависими са тези системи, които имат способността да адаптират своята работа в зависимост от текущия контекст, като по този начин имат за цел постигането на по-голяма производителност и ефективност чрез отчитане на заобикалящия ги контекст [8,102]. Schilit и Threimer [104] първи дефинират контекстно-зависимия компютинг през 1994 като способност на мобилно приложение да взаимодейства с информацията, която съществува в заобикалящата го среда. Също така контекстно-зависимата система се определя като система, която има свойството да усеща, открива, интерпретира и отговаря на заобикалящия я контекст [89,101,21] и да

използва информацията от заобикалящата я среда, за да осигури информация и/или услуга на потребителите й в съответствие с техните нужди [34].

Контекстно-зависимите приложения предоставят информация и/или услуги на потребителите. Според [65] процесът по предоставяне на информация и услуги може да бъде следните три класа:

- Представяне на информация и услуги: или се предоставя информация на потребителя, или се извършва набор от няколко действия за предоставяне на услуги.
- Автоматично изпълняване на услуга: тук, системата предприема действие и извършва дадена услуга от името на потребителя.
- Свързване на информацията за контекста за последващо възстановяване: отнася се за приложение, което свързва разпознатите данни със свързаната с тях информация за контекста.

В контекстно-зависимите системи разпознатият контекст може да се категоризира в четири основни категории: *идентичност (identity)*, *местоположение (location)*, *активност (activity)* [157] и *време (time)* [36]. Тези четири категории се смятат за основни контексти (primary context), които отговарят само на въпросите кой, какво, къде и кога. Но тези контексти играят ролята и на индекси за друга информация (т.е. за второстепенния контекст) [37]. Напр., от идентичността на един потребител може да се осигури много друга информация, като телефонен номер, адрес и списък с приятели. Също така със събирането на информацията за заобикалящата потребителя среда ние може да определим какви други обекти и личности са близо до потребителя и какви събития се случват наоколо. Това е второстепенен контекст.

Една контекстно-зависима система трябва да се справя с информацията за контекста по следния начин:

- Да възприема (усеща) контекста (sense);
- Да разсъждава върху контекстната информация (reason);
- Да действа в зависимост от контекстната информация (act).

1.5 Формализми за контекстно-зависими системи

Calculus of Context-aware Ambients (CCA) [108,3] е създаден с цел моделиране на контекстно-зависими системи. CCA е изграден на основата на Calculus of Mobile Ambients (MA) [24]. Формализмът въвежда нови конструкции за моделиране на *амбиенти (ambients)* и процеси, които са в състояние да опознават околната среда, където се изпълняват. Този изчислителен модел е базиран на нотацията на амбиенти. Понятието „ambient“ означава околен, заобикалящ, обкръжаващ. Един амбиент е ентити, което се използва, за да опише някой обект или компонент (например личност, процес, устройство, местоположение и т.н) . Един амбиент може да се определи като ограничено място, където се извършват изчисленията. Амбиентът има име, граници и може да съдържа други амбиенти в себе си. Амбиентът може да бъде мобилен и има способността да комуникира с други амбиенти.

Контекстът играе съществена роля в оперативното поведение на съвременните работни потоци. Съществуващите техники за моделиране на работни потоци предполагат централизирана оперативна инфраструктура. Следователно, едно от големите предизвикателства на разработването на системи за управление на работни потоци е тяхното опериране в контекстно-зависими среди. Това изисква доставка на изчислим модел, в основата на който да лежи концепцията за контекст и всеки един контекст поддържа собствена политика за сигурност, която ограничава неговото поведение – от контрола за достъп до осигуряване на усамотение и конфиденциалност. Такъв изчислителен модел трябва да третира две ограничения – сигурност и контекст – по еднакъв и интегриран начин. За формално представяне на работните потоци във ВОП е избрана формалната нотация CS-Flow [159,2], поради следните причини:

- Основно предназначение на нотацията е формално представяне на работни потоци в разпределена среда;
- Изпълнението на работните потоци се извършва в конкретен контекст, който може да бъде променян;
- За формализма е разработен специален език, който може да се използва за описание на сценариите;

- Разработва се интерпретатор на CS-Flow езика, което улеснява значително разработването на програмна среда за поддръжка на пространството.

Всичко това прави нотацията SC-Flow адекватно средство за моделиране управлението на процесите във виртуалното образователно пространство.

В съвременните информационни технологии съществува необходимост от придобиване и разбиране на различни типове знания. Разполагайки с достатъчно знания и дефинирани правила за разбирането им, имаме възможност да вземаме определени решения. За да постигнем по-добро представяне и използване на непрекъснато нарастващите познавателни ресурси и да имаме възможност да вземаме решения, въз основа на тях, ние непрекъснато подобряваме начините за представяне на знанията и възможностите си за моделиране на реални събития. Езикът на класическата логика от първи ред е удачен инструмент за формално описание на статистически инцидентни събития. Събития, случващи се във времето, се нуждаят от нов подход за формално представяне. За да имаме силата да описваме ситуации, които съдържат в себе си времеви компоненти, ние се нуждаем от езика на темпоралната логика. В съвременните мобилни и разпределени информационни системи видът и редът на случващите се събития играят съществена роля. Понятието за времето, в което се случва дадено събитие, е основен компонент за смисъла на това събитие. Доставянето на определено количество информация в подходящия вид и време е основна характеристика на коректно работещите системи .

Нуждата от анализиране и изследване на темпоралните аспекти е значима в много области на науката, като напр. географски информационни системи, философия, психология и други. От гледна точка на информатиката разбирането и моделирането на времево-зависими процеси е обект на изследване при създаването на информационни системи, верифицирането на софтуер, изкуствения интелект, както и в редица области на науката, свързани с моделирането на процеси.

В много приложения информацията, съдържаща времеви компоненти, е от голяма важност. Тя често служи като основа за предвиждане на бъдещото поведение на системата, съставяне на планове и сценарии за развитие или разбиране на вече случили се събития, за да анализираме работата на дадена система.

Темпоралната логика, като апарат за формално описание, би била полезна, когато правим описание на знания от гледна точка на тяхната промяна във времето. В този смисъл времето и промяната са две тясно свързани понятия. Взаимовръзката между време и промяна са в основата на два подхода за тяхното изследване:

- Подход, базиран на концепцията за промяна;
- Подход, базиран на концепцията за време.

Подходът базиран на концепцията за промяна, има редица ограничения. Недостатъците на моделите, базирани на концепцията са: непосредствени действия (те не бива да бъдат разтягани във времето) или непосредствени резултати (идващите резултати в продължение на времето не са възможни за описание с такива модели). Още повече, използването на такива модели не позволява показването на непрекъснати процеси, едновременно случващи се събития или събития, препокриващи се едно друго.

При подхода, базиран на концепцията за време, то бива представено като реално множество или едномерно пространство. Времето може да има точкова структура, където точки във времето са основната концепция или интервална структура. В общия случай описанието се базира на линейно и дискретно време. От тази гледна точка трябва да се отбележи, че когато се използва времевата точка като основна концепция е допустимо времето да се разглежда като мрежа от времеви точки. При такова описание на времето се допуска да се въведат понятия като неограничено време или реална времева линия. От гледна точка на избягване на технически трудности, при моделите използващи интервали времето се приема за дискретно понятие. Това определя множество разлики при моделиране с двата подхода. Разглеждането на времето като дискретно понятие дава възможност за въвеждане на предикатното смятане при тези модели.

Важна характеристика на езиците, представящи темпоралната логика е възможността за имплементирането им в спецификациите на широк спектър от информационни системи. От гледна точка на информационните системи, разглеждането на времето като дискретно понятие е от голяма важност - това доближава логическият модел за описание на времето до основните принципи на програмните езици. Работата с понятието време, като крайно понятие дава възможност

за реално моделиране на времево-зависими процеси, които протичат в информационните системи.

Времето е неизменна част от нашия живот, използването и осъзнаването на времето не предизвиква особени трудности за хората, но нещата не са така прости, когато искаме да вградим фактора време в информационните системи. Имплементирането на времево-зависима информация предизвиква редица проблеми като различното интерпретиране на понятията за дължина на времето (една година се възприема като малък период от време за историците, а една секунда е много за инженери, работещи с микропроцесори). При това положение става трудно моделирането на процеси, свързани с едновременното или последователното протичане на процеси или различни точки за начало и край на процесите. Всичко това може да бъде избегнато, ако времето се приеме като дискретното понятие, което е разделено на краен брой времеви интервали, които са с еднаква продължителност.

Темпоралната логика е теория, изградена на основата на логиката от първи ред, но разширена и допълнена с времеви оператори. Тези нейни допълнения са превърнали Темпоралната логика в силен инструмент за създаване на формални модели за различни системи, включващи конкурентно и паралено програмиране, дигитални кръгове и различни видове хардуер. За да се възползваме от предимствата, които имат моделите, представящи времето като дискретно понятие, разделено на времеви интервали, ние ще се занимаем с един вариант на темпоралната логика, наречен Интервална Темпорална Логика. Полезното допълнение на ITL е пълно имплементиране на теоретичния модел за поделяне на равни времеви интервали на понятието време. Механизмът за боравене с формулите и операторите на ITL включва последователно обработване на изразите, следвайки хронологично разглежданите интервали от времевата линия. Този формализъм притежава всички стандартни оператори, познати ни от логиката от първи ред. Теоретичният модел е допълнително обогатен с времеви оператори, като „винаги” , „понякога” и др.

За да бъдат знанията значими и смислени, ние се нуждаем не само от формализъм, с който те да бъдат описани, но и от интерпретиращ механизъм. Подходящ интерпретатор на ITL е софтуерната система Темпура. Темпура представлява една изпълнима рамка и интерпретатор на едно подмножество на ITL. Първоначално интерпретаторът е бил разработен от Роджър Хейл в Кембридж на програмния език

Prolog. В периода 1984-1985г. Бен Московски защитава докторски тезис в Станфордския университет на тема „Executing Temporal Logic programs”, като част от неговата разработка е нова версия на интерпретатора Темпура, който в тази си версия е програмиран на езика C. В следващите години интерпретаторът бива развиван, добавяни са нови оператори и структури. В момента поддръжката на Темпура се извършва от Бен Московски и Антонио Кау, а текущата версия, върху която ние спираме нашето внимание е Темпура 2.16. Подробен преглед на основните характеристики на Интервална Темпорална Логика и нейния интерпретатор Темпура ще бъдат направени в следващите глави.

Обобщавайки, можем да мотивираме избора на ITL за решаване на задачите на дисертацията както следва:

- ITL доставя формализъм за работа с времето под формата на състояния и промяна на тези състояния;
- Съществува интерпретиращ алгоритъм за работа със състояния.

1.6 Български принос към електронното обучение

В последните години във водещите световни университети все по интензивно се въвеждат системи за електронно и дистанционно обучение. Тези тенденции се наблюдават и в българските университети, където такива приложения са адаптация на системи с отворен код (предимно Moodle [78]) или използват фирмени образователни сървъри (напр. Class Server на Microsoft [75]). Същевременно много български университети разработиха собствени системи за електронно и дистанционно обучение (напр. ARCADE [14,13] в Софийския университет, eSchool [6] във Варненския свободен университет, PeU в Пловдивския университет [169], Flame [165] в Националната военна академия).

В българските висши учебни заведения се изградиха и немалко системи за провеждане на електронни тестове. Едни от първите публикации, посветени на тази тема са [41,42,76]. Някои проблеми на изграждане на адаптивни системи за електронно обучение се разглеждат в [146,147,16,15]. Изграждане на архитектури за електронно

обучение, използващи услуги и интеграцията на учебно съдържание в Интернет среда се разглеждат в [114,113,112]. Различни аспекти на нововъзникващото игрово-базирано обучение са представени в [167,19,145].

Глава 2 Теоретични основи

2.1 Интервална темпорална логика (Interval Temporal Logic)

Interval Temporal Logic (ITL) е гъвкава нотация, както за съждителната логика, така и за логиката от първи ред. С нейна помощ се дефинират и моделират времево-зависими процеси; често се използва в описанията на хардуерни и софтуерни системи.

За разлика от повечето темпорални логики, ITL е подходяща за представяне и моделиране както на последователни, така също и на паралелни процеси и композиции. Тя предлага мощна и разширима спецификация и техники за верификация чрез аргументиране на свойства, като напр. сигурност, жизненост, планиране на времето. Темпоралните ограничения са лесно изразими. Голям брой императивни програмни конструкции могат да бъдат представени като формули в слабо променена версия на ITL. Едно подмножество на ITL, наречено Tempura, предоставя езикови конструкции, удобни за спецификация поведението на софтуерни системи, които могат да бъдат автоматично интерпретирани. В допълнение, ITL и Tempura се използват широко за специфициране свойствата на системи, работещи в реално време, където жизнения цикъл може да бъде представен директно чрез множество от прости темпорални формули. Различни изследователски проекти използват Tempura за симулации, където времето е критичен фактор.

Темпоралната логика е полезен инструмент за аргументиране на процесите в конкурентни програми и хардуер [72]. В рамките на темпоралната логика могат да се изразят логически оператори обвързани с времезависещи концепции като „*винаги*” и „*понякога*”. Нека разгледаме следното примерно твърдение:

„*Ако съжденията P и Q са винаги верни, то P е винаги вярно.*”

В темпоралната логика това може да бъде представено чрез формулата

$$\Box\left(P \wedge Q\right) \supset \Box P.$$

Тук операторът $\Box$ отговаря на нотацията „*винаги*”. И така подформулата $\Box\left(P \wedge Q\right)$ може да бъде разбрана като „*P и Q са винаги верни*”.

Темпоралната логика е причислявана като инструмент за специфициране и доказване свойства на програми използващи паралени процеси [62,63]. Различията между темпоралните логики и езиците за програмиране създават проблеми, когато трябва по формален начин да се изрази стойността променлива, която се променя динамично по време на изпълнение на дадена програма. Програмни формализми, като напр. логиката на Hoare [61], динамична логика [58,92] или процесна логика [26,59] също отразяват това противоречие. Така напр. следната клауза на Hoare дава определение, че ако I е първоначално равна на 3, то след нарастване с 1 нейната стойност е 4:

$$\{I = 3\}\, I := I + 1\, \{I = 4\}.$$

Тук имаме формулите $I = 3$ и $I = 4$ и твърдението $I := I + 1.$

Подобни твърдения в познатата ни логика от първи ред биха били приети като невярни и биха били равносилни на базовото твърдение за неистина – “false”. От гледна точка на софтуерните системи и тяхното програмиране е напълно естествено явление една променлива да променя стойността си по време на жизнения цикъл на приложението. Един от начините да се преодолее пропастта между логика и програма е като се намерят начини за използване на самата темпорална логика като инструмент за програмиране и моделиране. С тази цел е създадена Tempura – императивен език за програмиране, използващ едно подмножество на темпоралната логика. Всяко твърдение в Tempura е формула от темпоралната логика.

- Основни свойства на ITL

Преди да представим накратко Tempura е необходимо да дадем някои обяснения за основната темпорална логика. Някои от конструкциите, които се описват тук, са изпозвани по-късно в програми на Tempura, а други просто улесняват аргументирането относно поведението на програмата. Нека първо разгледаме някои основни оператори.

Произход

Най-напред се нуждаем от мотивация за изполването на темпоралната логика за определяне и дефиниране на динамичното поведение. Предикатното смятане [39] е гъвкава и точна нотация за формалното описване на ситуациите. Например изречението

„*I* е равно на 2 и *J* е равно на *I плюс* 1” може да бъде лесно изразено чрез следната формула:

$$(I = 2) \wedge (J = I + 1).$$

Динамичното поведение обаче е по-проблематично. Например, твърдението „Променливата *I* в даден момент е равна на 1, а в друг по-късен момент е равна на 2”, не се удовлетворява напълно от формула като

$$(I = 1) \vee (I = 2).$$

Тя само описва постоянна ситуация, при която *I* може да е равна на 1 или 2. Формулата

$$(I = 1) \wedge (I = 2)$$

определено е неподходяща тук, тъй като е логически еквивалент на *false*!

За да се заобиколи статичното естество на логиката от първи ред е нужно да се моделират времезависещи променливи като явно описани времеви функции. Бихме могли например, да специфицираме променящата се стойност на *I* като използваме следната формула:

$$\exists t, t' : ([t \le t'] \wedge [I(t) = 1] \wedge [I(t') = 2]).$$

Тук променливите t и t' индикират две времеви точки, за които са посочени стойностите на *I*. Това е една много добра техника за представяне на динамично поведение, но страда от пренасищане с допълнителни времеви променливи и идентификатори.

В този и в следващя раздел се разглежда алтернативен подход за дефиниране на периодите от време, базиращ се на темпоралната логика. Той представлява формализъм, включващ както общоприетите логически оператори като ∨ и =, така и времезависещите □ (чете се „*винаги*”) и ◇(чете се „*понякога*”). Въпреки че първоначално темпоралната логика се създава за приложение във философията, тя бива предложена от Burstall [23], Pnueli [91] и други като полезен инструмент за справяне с компютърни програми и дигитален хардуер.

В рамките на темпоралната логика е възможно да се опише динамично поведение с прост и елегантен стил. Например твърдението *„I е винаги по-голямо от 3 и понякога по-малко от 6”* може да бъде изразено чрез формулата:

$$\Box(I > 3) \wedge \Diamond(I < 6).$$

Формулата

$$\Diamond\left[(I = 1) \wedge \Diamond(I = 2)\right]$$

описва интервал от време, при който променливата *I* в даден момент е равна на 1 и в някой следващ момент е равна на 2. Свойствата на времето могат също да бъдат изразени. Например, ако *I* е винаги равна на 1 и *J* понякога е равна на 3, то може да се заключи, че сумата $I + J$ понякога е равна на 4:

$$\left[\Box(I = 1) \wedge \Diamond(J = 3)\right] \supset \Diamond(I + J = 4)$$

Тези примери дават само бегла представа за ползата и удобството от употребата на темпорална логика. Както ще се види по-нататък, темпоралната логика осигурява естествен начин за описване на такива динамични нотации като стабилност, прекратяване и дължина на интервала. Сега ще разгледаме на кратко основния синтаксис и семантика на формализма.

Синтаксис на ITL

Основното множество изграждащи елементи включва общоприетите логически оператори като = (*равенство*), $\wedge$ (*логическо „И”*), $\vee$ (*логическо „ИЛИ”*) и т.н., като са добавени и темпоралните оператори $\Diamond$ (*понякога*), $\Box$ (*винаги*) и др.

Синтаксис на изрази

Изразите се изграждат индуктивно както следва:

- Променливи: A, B, C,…
- Функции: f(e1,…,ek), където $k \geq 0$ и e1,...,ek са изрази.
- Next: $\bigcirc$e, където e е израз.

Примери за синтактично верни изрази:

$$I + (\bigcirc J) + 1, \qquad (\bigcirc I) + J - \bigcirc\bigcirc(I + \bigcirc J).$$

Синтаксис на формули

Формулите се изграждат индуктивно както следва:

Предикатите: $p(e_1,...,e_k)$, където $k \geq 0$ и $e_1,...,e_k$ са изрази. Предикатите включват $\geq$ и други основни връзки.

- Равенство: $e_1 = e_2$, където e1 и e2 са изрази.
- Логически връзки: ¬w и w1 ∧ w2, където w, w1 и w2 са формули.
- Next: ○w, където w е формула.
- Винаги: □w, където w е формула.

Примери за синтактично верни ITL формули:

$$(J = 2) \wedge \bigcirc(I = 3),$$
$$(\bigcirc \Box[I = 3]) \wedge \neg([\bigcirc J] = 4)$$
$$\bigcirc(\Box[I = 3] \wedge \bigcirc\bigcirc[J = 4]).$$

Трябва да се отбележи, че операторът ○ може да бъде използван както за изрази (○J), така и за формули (○(I = 3)).

Модели

Модел се нарича наредена тройка (D, ∑, M), съдържаща област от стойности D, множество от състояния ∑ и интерпретиращ механизъм M, придаващ смисъл на всяка функция и предикатен символ. За по-лесно боравене, без да ограничаваме общовалидността, приемаме, че множеството D приема само целочислени стойности. Едно състояние е функция, която съпоставя всяка променлива на стойност от областта D. Нека ∑ е множеството от всички такива функции. За едно състояние *s* от ∑ и променлива *A*, нека $s\left[\left[A\right]\right]$ означава стойностите на *A* в състоянието *s*. За всеки *k*-ти функционален символ има интерпретиращ механизъм $\mathrm{M}\left[\left[f\right]\right]$, който е функция съпоставяща *k* елементи от D в една единствена стойност:

$$\mathcal{M}\left[\left[f\right]\right] \in \left(\mathcal{D}^k \to \mathcal{D}\right).$$

Интерпретациите на предикатни символи са подобни, но сочат към истинни стойности:

$$\mathcal{M}\left[\left[p\right]\right] \in \left(\mathcal{D}^k \to \{true, false\}\right).$$

Приемаме, че M дава стандартни интерпретации на оператори като + и <.

Дадената семантика държи интерпретациите на функции и предикатни символи независими от интервали. Те могат обаче да бъдат въведени в употреба за функции и предикати, които се съобразяват с динамичното поведение на параметрите.

Използвайки състоянията в ∑ се изграждат времеви *интервали* от Σ^+ - множеството от всички непразни, крайни последователности на състояния. Ако *s*, *t* и *u* са състояния от ∑, то са възможни следните интервали:

$$\langle s \rangle, \ \langle sttsus \rangle, \ \langle tttt \rangle.$$

Отбелязваме, че един интервал винаги съдържа поне едно състояние.

Ще представим някои основни нотации за работа с интервали. Нека с *I* означим множеството от всички интервали. Засега приемаме *I* да бъде множеството Σ^+. По-

нататък *I* ще бъде ограничено до известна степен. При даден интервал σ от *I*, нека $|\sigma|$ да бъде дължината на σ. В темпоралната логика има установено правило, което определя дължина на интервал като броя на състоянията в интервала *минус единица*. Следователно дължините на интервалите по-горе са съответно 0, 5 и 3. Отделните състояния на интервал σ се означават както следва: $\sigma_0,\sigma_1,...,\sigma_{|\sigma|}$. За допълнителна яснота ще разгледаме следното равенство, а то е *вярно* тогава и само тогава, когато променливата *A* има стойност 5 в поседното състояние на интервала σ:

$$\sigma_{|\sigma|}\left[\left[A\right]\right]=5.$$

Моделът описан тук разглежда времето като дискретна величина и от него не се очаква да дава реалистично представяне на заобикалящия ни свят. Въпреки това, той ни осигурява солидна основа за моделиране на много интересни динамични явления, включващи поведение, зависещо от времето и функционалното поведение. Нещо повече, дискретното описание на света около нас, много повече се доближава до моделите, използвани за изграждане на дигитални системи и компютърни програми. Във всеки случай, винаги можем да дадем една относително добра детайлност на времето.

Интерпретация на изрази и формули

Понятието интерпретиращ механизъм M се разширява, за да може да работи с изрази и формули в интервали. Конструкцията $\mathcal{M}_\sigma\left[\left[e\right]\right]$, се дефинира така, че да бъде равна на стойността от D на израза *e* в интервала σ. Аналогично, $\mathcal{M}_\sigma\left[\left[w\right]\right]$ става равна на истинната стойност на формулата *w* в σ.

$\mathcal{M}_\sigma[\![V]\!] = \sigma_0[\![V]\!]$, където *V* е променлива.

От тук следва, че стойността на една променлива за един интервал е равна на стойността на променливата в началното състояние на интервала.

$$\mathcal{M}_\sigma[\![f(e_1,\ldots,e_k)]\!] = \mathcal{M}[\![f]\!](\mathcal{M}_\sigma[\![e_1]\!],\ldots,\mathcal{M}_\sigma[\![e_k]\!])$$

На интерпретацията на функцията *f* е присвоена интерпретацията на $e_1,...,e_k$.

$$\mathcal{M}_\sigma[\![\bigcirc e]\!] = \mathcal{M}_{\langle\sigma_1 \ldots \sigma_{|\sigma|}\rangle}[\![e]\!], \quad \text{if } |\sigma| \geq 1.$$

Оставяме стойността на $\bigcirc e$ неопределена за интервал с дължина 0.

- $\mathcal{M}_\sigma[\![p(e_1, \ldots, e_k)]\!] = \mathcal{M}[\![p]\!](\mathcal{M}_\sigma[\![e_1]\!], \ldots, \mathcal{M}_\sigma[\![e_k]\!])$
- $\mathcal{M}_\sigma[\![e_1 = e_2]\!] = true \quad \text{iff} \quad \mathcal{M}_\sigma[\![e_1]\!] = \mathcal{M}_\sigma[\![e_2]\!].$
- $\mathcal{M}_\sigma[\![\neg w]\!] = true \quad \text{iff} \quad \mathcal{M}_\sigma[\![w]\!] = false.$
- $\mathcal{M}_\sigma[\![\Box w]\!] = true \quad \text{iff}$
  $\text{for all } i \leq |\sigma|, \ \mathcal{M}_{\langle\sigma_i \ldots \sigma_{|\sigma|}\rangle}[\![w]\!] = true.$
  $\mathcal{M}_\sigma[\![w_1 \wedge w_2]\!] = true \quad \text{iff}$
  $\mathcal{M}_\sigma[\![w_1]\!] = true \text{ and } \mathcal{M}_\sigma[\![w_2]\!] = true.$

и

Примери:

Видяхме употребата на M според семантиката на тези прости темпорални формули. Нека *s*, *t* и *u* са състояния, при които променливите *I* и *J* имат следните стойности:

	I	*J*
s	1	2
t	3	4
u	3	1

Формулата

$$(J = 2) \wedge \bigcirc(I = 3)$$

е вярна за един интервал σ, тогава и само тогава, когато σ има дължина ≥ 1, стойността на *J* в състоянито σ_0 е 2 и стойността на *I* в състоянието σ_1 е 3. Така формулата е вярна за интервала $\langle stu \rangle$. От друга страна формулата е грешна за интервала $\langle ttu \rangle$, тъй като началната стойност на *J* в този интервал е 4 вместо 2.

Формулата

$$(\bigcirc\Box[I=3])\wedge\neg([\bigcirc J]=4)$$

е вярна за всеки интервал σ с дължина ≥ 1, в който I е равна на 3 в състоянията $\sigma_1,...,\sigma_{|\sigma|}$ и J не е равно на 4 в σ_1. Така формулата е вярна за интервала $\langle sutut\rangle$, но е грешна за $\langle t\rangle$ и $\langle stutu\rangle$.

Формулата

$$\bigcirc(\Box[I=3]\wedge\bigcirc\bigcirc[J=4])$$

е вярна за интервал σ с дължина ≥ 3, в който променливата I е равна на 3 в състоянията $\sigma_1,...,\sigma_{|\sigma|}$ и променливата J е равна на 4 в състояние σ_3. Така тази формула е вярна за интервала $\langle suutu\rangle$, но е грешна за $\langle s\rangle$ и за $\langle sutuu\rangle$.

Удовлетвореност и валидност

Една формула w е *удовлетворима*, ако е възможно да се направи интерпретация, която прави формулата истинна:

$$\mathcal{M}_\sigma\left[\left[w\right]\right]=true.$$

Това се означава както следва:

$$\sigma\vDash w.$$

Ако всички иинтервали удовлетворяват w, то w е *валидна*, пише се: $\vDash w$.

Пример (Валидност):

Следната формула е истинна за един интервал σ тогава и само тогава, когато $|\sigma|\geq 1$, променливата I е винаги равна на 1, а в състоянието σ_1 тя е равна на 2:

$$\Box(I=1)\wedge\bigcirc(I=2).$$

Никой интервал не може да има всички тези характеристики. Поради това формулата е невярна за всички интервали и нейното отрицание е винаги вярно, а следователно и валидно:

$$\models \quad \neg[\Box(I=1) \wedge \bigcirc(I=2)].$$

Производни оператори

Представените дотук изграждащи елементи на видовете поведения на интервали може да изглеждат доста ограничени, но всъщност това изобщо не е всичко, тъй като е възможно да се създаде едно доста голямо разнообразие от *производни* оператори. Сега ще разгледаме няколко, които се доказват като полезни в работата с прости изчисления.

Операторът $\Diamond$

Конструкцията $\Diamond w$ е вярна за интервал σ, ако съществува някакъв допълнителен подинтервал, за който формулата е вярна:

$$\mathcal{M}_\sigma[\![\Diamond w]\!] = true \quad \text{iff} \quad \text{за някое } i \leq |\sigma|,\ \mathcal{M}_{\langle \sigma_i \ldots \sigma_{|\sigma|} \rangle}[\![w]\!]$$

Това поведение може да бъде описано според правилата на операторите ¬ и □:

$$\Diamond w \quad \equiv_{\text{def}} \quad \neg \Box \neg w.$$

Пример (Настояще и бъдеще):

Следната формула показва важни разлики между различните темпорални конструкции:

$$(I=1) \wedge \bigcirc(I=2) \wedge \Diamond(I=3)$$

Това е вярно за интервал σ с дължина поне 2, в който променливата *I* има стойност 1 в началното състояние σ_0, стойност 2 в следващото състояние σ_1 и след това е равно на 3 в някое следващо състояние.

Операторите empty и more

Формулата *empty* е вярна за един интервал тогава и само тогава, когато интервалът има дължина 0:

$$\sigma \models empty \quad \text{iff} \quad |\sigma| = 0.$$

Можем да дефинираме *empty* както следва:

$$empty \quad \equiv_{\text{def}} \quad \neg \bigcirc true.$$

Формулата *more* за един интервал е вярна тогава и само тогава, когато интервалът има ненулева дължина. Можем да изразим *more* както следва:

$$more \quad \equiv_{\text{def}} \quad \bigcirc true.$$

От тези дефиниции ясно следва, че *more* е противоположен на *empty*:

$$\models \quad more \equiv \neg empty.$$

Пример (Измерване дължината на интервал):

Можем да използваме конструктивните елементи ○ и *empty*, за да определим дължината на интервал. Например, формулата

$$\bigcirc\bigcirc\bigcirc empty$$

е вярна за интервал σ тогава и само тогава, когато σ има дължина 3.

Оператор halt

Можем да укажем, че една формула *w* е вярна само в края на интервал σ, като използваме формулата *halt w*:

$$halt\, w \quad \equiv_{\text{def}} \quad \Box\left(w \equiv empty\right).$$

По този начин *w* трябва да е *false* до последното състояние, чак когато *w* е вярна. Например формулата:

$$halt\left(I > 100\right)$$

е вярна за σ тогава и само тогава, когато стойността на променливата *I* превишава 100 точно в последното състояние на σ.

Пример (Многократно удвояване на стойност):

От това, което разгледахме до сега, можем да преценим, че формулата:

$$\left(I = 1\right) \wedge halt\left(I > 100\right) \wedge \left(I \; gets \; 2I\right)$$

е изпълнена за интервал, при който променливата *I* е с начална стойност 1 и се удвоява многократно докато не надвиши 100. Следните валидни импликационни състояния показват, че тези интервали, за които формулата е вярна ще се прекратят при *I* равно на 128:

$$\models \Big[\big(I=1\big)\wedge halt\big(I>100\big)\wedge\big(I\ gets\ 2I\big)\Big]\supset halt\big(I=128\big).$$

Темпорално Равенство

Конструкцията $e_1 \approx e_2$ се нарича *темпорално равенство* и е вярна тогава и само тогава, когато изразите e_1 и e_2 са винаги равни:

$$e_1 \approx e_2 \ \equiv_{\text{def}} \ \Box\big(e_1 = e_2\big).$$

Операторът ⓦ

За да може конструкцията ○*w* да бъде вярна за един интервал σ, дължината на σ трябва да е поне 1. Поради това тази конструкция назоваваме *strong next*. Сродната ѝ конструкция ⓦ*w* се нарича *weak next* и тя е вярна за един интервал σ, ако σ има дължина 0, или ако подформулата *w* е вярна за $\langle \sigma_1 ... \sigma_{|\sigma|} \rangle$. *Weak next* можем да изразим чрез правилата на *strong next*:

$$ⓦw \ \equiv_{\text{def}} \ empty \wedge \bigcirc w.$$

Операторът *weak next* ни осигурява кратък и естествен начин, за изразяване на един конструктивен елемент като конюнкция от неговия моментен ефект и бъдещ такъв.

$$\begin{aligned}
\models \quad & \Box w \quad \equiv \quad w \wedge ⓦ\Box w, \\
\models \quad & \bigcirc w \quad \equiv \quad more \wedge ⓦ w, \\
\models \quad & e_1\ gets\ e_2 \quad \equiv \quad \big[more \supset \big([\bigcirc e_1] = e_2\big)\big] \wedge ⓦ(e_1\ gets\ e_2) \\
\models \quad & halt\, w \quad \equiv \quad (empty \equiv w) \wedge ⓦ(halt\, w), \\
\models \quad & e_1 \approx e_2 \quad \equiv \quad (e_1 = e_2) \wedge ⓦ(e_1 \approx e_2).
\end{aligned}$$

Тези видове еквивалентности се оказват доста полезни в конструирането на интерпретатори.

Програмиране в ITL

Разглеждаме формулата

$$(M=4)\wedge(N=1)\\ \wedge halt(M=0)\wedge(M\ gets\ M-1)\wedge(N\ gets\ 2N). \quad (1\text{-}1)$$

Тя е вярна за интервали с дължина 4, където M последователно преминава през стойностите 4, 3, 2, 1 и 0, като едновременно с това N минава през съответно 1, 2, 4, 8 и 16. Нека сега да видим как ще се автоматизира процесът за използване на такава формула и за намиране на удовлетворяващия я интервал. Един от начините да направим това е като създадем процес, който описва формулата и определя поведението на всички свободни променливи във всяко състояние. Самият резултат може да бъде изразен като темпорална формула. Например ето един начин да представим резултата за формула **Error! Reference source not found.**:

$$\begin{aligned}&([M=4]\wedge[N=1])\\ &\quad\wedge\bigcirc([M=3]\wedge[N=2])\\ &\quad\wedge\bigcirc\bigcirc([M=2]\wedge[N=4])\\ &\quad\wedge\bigcirc\bigcirc\bigcirc([M=1]\wedge[N=8])\\ &\quad\wedge\bigcirc\bigcirc\bigcirc\bigcirc([M=0]\wedge[N=16]\wedge empty)\end{aligned}$$

Забелязва се, че новата формула е логически еквивалент на оригиналната формула **Error! Reference source not found.**. Можем да я разглеждаме като своеобразна нормална форма, описваща поведението на всички свободни променливи стъпка по стъпка. Процесът на установяване на такава нормална форма може да бъде компютъризиран. По същество това е форма за изпълнение на програма, където оригиналната формула представлява програмата, а получената нормална форма отговаря на фактическото изчисление.

Основният проблем за намиране на универсална нормална форма за всяка случайна темпорална формула остава неразрешим. Въпреки това съществуват отделни подмножества на темпоралната логика, за които тази цел е постижима. Tempura е език за програмиране, базиран на такова подмножество, което ефективно се поддава на имплементиране и на употреба при описване на занимателни и практически

изчисления. Например формула (1-1) е действителна програма на Tempura, която при изпълнение извежда резултата показан на следната фигура:

```
State  0: M= 4  N= 1
State  1: M= 3  N= 2
State  2: M= 2  N= 4
State  3: M= 1  N= 8
State  4: M= 0  N=16
```

Фиг. 1: Изпълнение на формула (1-1)

$$(M=4)\wedge(N=1)\wedge halt(M=0) \\ \wedge(M\ gets\ M-1)\wedge(N\ gets\ 2N)\wedge\Box display(M,N). \quad (1\text{-}2)$$

Така многократно се отпечатват стойностите на M и N с помощта на конструктивният елемент *display*. Трябва да отбележим и че поведението на програмата остава незасегнато дори да се промени редът на конюнктивните операнди. При следващия вариант на формулата те са разместени:

$$\Box display(M,N)\wedge(N\ gets\ 2N)\wedge(M\ gets\ M-1) \\ \wedge halt(M=0)\wedge(N=1)\wedge(M=4)$$

По време на изпълнението на следната програма, потребителят бива подканван да даде нови стойности за *I* с помощта на конструктивният елемент *request*:

$$\Box request(I)\wedge halt(I=0)\wedge(J=0) \\ \wedge(J\ gets\ J+I)\wedge\Box display(J). \quad (1\text{-}3)$$

Сумата на тези стойности се прибавят към *J* и се показва самата *J*. Интервалът прекратява работа при *I* = 0.

Синтаксис на Tempura

Нека разгледаме основния синтаксис на Tempura. Най-важните синтактични категории в Tempura са локациите, изразите и твърденията.

```
State  0: I = 6
State  0: J = 0
State  1: I = 2
State  1: J = 6
State  2: I = 5
State  2: J = 8
State  3: I = 0
State  3: J = 13
```

Фиг. 2: Изпълнение на формула (1-3)

Локации

Локация е мястото, където се съхраняват и изследват стойностите, участващи в програмата. Променлви като *I*, *J* и *K* са допустими локации. Ако *l* е локация, такава е и темпоралната конструкция $\bigcirc l$.

Изрази

Изразите могат да бъдат както аритметични, така и булеви. Всички числови константи и променливи са валидни аритметични изрази. Ако e_1 и e_2 са аритметични изрази, то такива са и следните операции:

$$e_1 + e_2,\ \ e_1 - e_2,\ \ e_1 \cdot e_2,\ \ e_1 \div e_2,\ \ e_1 \, mod \, e_2.$$

Връзки от типа на $e_1 = e_2$ и $e_1 \geq e_2$ са булеви изрази. Ако *b*, b_1 и b_2 са булеви изрази, то такива са и следните:

$$\neg b,\ \ b_1 \wedge b_2,\ \ b_1 \vee b_2,\ \ b_1 \supset b_2,\ \ b_1 \equiv b_2.$$

Булеви изрази са също така константите *true* и *false* и темпоралните конструктивни елементи *empty* и *more*. Условният израз

$$if\ b\ then\ e_1\ else\ e_2$$

е разрешен. Тук e_1 и e_2 могат да бъдат както булеви, така и аритметични изрази.

Твърдения

Някои темпорални формула са валидни твърдения в Tempura. Едно твърдение може да бъде *просто* или *сложно*. *Простите* твърдения са изградени от конструктивните елементи дадени по-долу:

true	(никаква операция)
false	(отменя)
$l = e$	(просто присвояване)
empty	(прекрати)
more	(не прекратявай)

Твърдението $l = e$ запазва стойността от аритметичния израз *e* в локацията *l*. В допълнение към тези твърдения може да бъде използвано за извикване и изобразяване на стойности следното:

request($l_1, \ldots, l_n$) (*извиква стойности от локациите*)

display($e_1, \ldots, e_n$) (*показва стойности от изразите*)

Сложни твърдения са изградени с помощта на конструкцията показана по-долу. Тук *w*, w_1 и w_2 са сами по себе си твърдения и *b* е булев израз:

$w_1 \wedge w_2$	(*паралелно съединяване*)
$b \supset w$	(*импликация*)
Ⓦ w	(*weak next*)
$\Box w$	(*always*)

Отбелязваме, че някои темпорални формули могат да бъдат използвани както като булеви изрази, така и като булеви твърдения. Например:

$$I = 3, \;\; (J = 2) \wedge (K = J + 3), \;\; (I = 0) \supset empty.$$

От друга страна, следните валидни булеви изрази не са твърдения в Tempura, макар и да са семантично еквивалентни на съответните формули дадени по-горе:

$$3 = I,\ \ \left(2 = J\right) \wedge \left(J + 3 = K\right),\ \ \neg\left(I = 0\right) \vee empty.$$

Детерминизъм

Както се споменава по-горе, твърденията в Tempura са сведени до подмножество от темпорални формули, но дори и синтактично верни програми биха могли да бъдат неизпълними. Това е така, защото интерпретаторът очаква потребителя да специфицира напълно поведението на променливите в програмата и да посочи кога трябва да настъпи прекратяване. Например за формулата:

$$I\ gets\ (I + 1)$$

липсва информация относно началната стойност на *I* и не дава спецификация кога да спре. По този начин формулата сама по себе си не може да бъде причислена към никой конкретен изчислителен цикъл на *I* и поради това не се счита за завършена програма. За да може интерпретаторът да работи както трябва е нужно да бъдат добавени други детайли. Аналогично на това сложни твърдения изградени с операторите $\vee$ и $\diamond$ не са разрешени. Разбира се може да се използва търсене с връщане назад (backtracking) или подобни техники, за да се избегне всяка двусмисленост. Но целейки простота и ефективност на работата на интерпретатора, на този етап изглежда благоразумно да се изисква ясно формулирана информация във всички аспекти на поведението на променливите.

Операторът chop

Темпоралната логика съдържа редица конструкции, каквито са *chop* и *while*, които са доста подобни на определени видове твърдения в Алгол и сродни на него езици за програмиране. За да се включи *chop* в темпоралната логика първо трябва да се разшири синтаксиса и семантиката . Произтичащият от това формализъм наричаме Interval Temporal Logic (ITL). В него са дефинирани редица интервалозависещи оператори и в последствие е разширена Tempura, за да могат те да бъдат включени.

Синтаксис и семантика на chop

Разрешаваме формули в следната форма:

$w_1; w_2$, където w_1 и w_2 са формули.

Операторът „ **;** ”е известен като *chop*. Една формула $w_1;w_2$ е вярна за интервал σ тогава и само тогава, когато има поне един начин да разделим σ на два подинтервала такива, че формулата w_1 е вярна за левия подинтервал и формулата w_2 е вярна за десния подинтервал:

$$\mathcal{M}_\sigma\left[\left[w_1;w_2\right]\right]=true \;\Leftrightarrow$$

за някое $i \le |\sigma|$,

$$\mathcal{M}_{\langle\sigma_0\ldots\sigma_i\rangle}\left[\left[w_1\right]\right]=true \text{ и } \mathcal{M}_{\langle\sigma_i\ldots\sigma_{|\sigma|}\rangle}\left[\left[w_2\right]\right]=true.$$

Отбелязваме, че и двата подинтервала $\langle\sigma_0\ldots\sigma_i\rangle, \langle\sigma_i\ldots\sigma_{|\sigma|}\rangle$ притежават състоянието σ_i.

Пример (Цикличен характер на присвояването):

Формулата

$$\left(K+1\to K\right);\left(K+2\to K\right)$$

е вярна за един интервал σ тогава и само тогава, когато има такова $i \le |\sigma|$, че подформулата $K+1\to K$ е вярна за подинтервала $\langle\sigma_0\ldots\sigma_i\rangle$ и подформулата $K+2\to K$ е вярна за оставащия подинтервал $\langle\sigma_i\ldots\sigma_{|\sigma|}\rangle$. Общият резултат е K увеличена с 3. Това се изразява чрез следното свойство:

$$\vDash \quad \left[\left(K+2\to K\right);\left(K+1\to K\right)\right]\supset\left(K+3\to K\right).$$

Пример (Използване на chop за изразяване на ◇ и □):

Чрез разнообразните операнди на *chop*, можем селективно да изследваме различни видове подинтервали. Например една формула от формата:

$$true;w$$

е вярна за един интервал, ако формулата *w* е вярна за един суфиксен (краен) подинтервал. Това ни дава възможност да изразим оператора ◇, а като негов дуален следователно и □. По същия начин една формула от формата:

$$true; w; true$$

е вярна за един интервал, ако *w* е вярна за някой негов случаен подинтервал. Тъй като *chop* е асоциативен оператор, можем да пропуснем скобите без опасения за двусмисленост.

Коментар към оператора chop

Конструкцията *chop* е доста различна от традиционните темпорални оператори □ и ○. Те изследват определени крайни подинтервали на един интервал, докато *chop* разделя интервала и изследва и двете части. Това улеснява разглеждането на случайни времеви подинтервали.

В [59] се разглежда за първи път оператора *chop* като темпорална конструкция. Разгледан е по-детайлно в [26]. В [57,82] е използван *chop*, за да се улесни аргументирането на времезависещ дигитален хардуер. Функционалността на *chop* в [83,80] той използва, за да се дадат спецификации и свойсва на прости алгоритми и системи за предаване на съобщения. В следващите Глави ще разучим *chop*, както и други ITL-изграждащи конструкции, за да разширим познанията си за Tempura като усвоим някои от тях.

2.2 Tempura и AnaTempura

Съществува дебат до колко са приложими формалните модели при конструирането на съвременните компютърни системи, било то хардуерни или софтуерни. Някои твърдят, че използването на формални модели решава изцяло най-често срещаните проблеми при проектирането на тези системи, докато други са на мнението, че моделите на практика са слабо или трудно приложими при реални условия. Би било твърде смело да твърдим, че подходите за проектиране, използващи формални модели са универсално решение за софтуерните системи. От една страна, съществуват доста аспекти, които трудно биха били описани формално. От друга

страна, използването на подходящи формални модели дават голямо предимство при проектирането и доказване коректността на софтуерните системи.

Един от основните проблеми при избора и прилагането на формализъм при моделирането на съвременните софтуерни системи е възможността за реализация на подходящ интерпретиращ формалната спецификация механизъм. Съществуването на такъв механизъм би направил възможно директно прилагане на формалните модели като част от системата, а не използването им само като теоретична постановка.

Interval Temporal Logic (описана в 2.1) е подходящ формализъм за описание на времево-зависими процеси, за който съществува интерпретиращ механизъм – софтуерната системата, наречена Tempura. Интерпретаторът Tempura представлява едно изпълнимо подмножество на ITL, което използва синтаксиса на ITL и се базира на неговата основна философия да възприема времето като крайна последователност от състояния, като на всяко състояние се съпоставя значещи (и налични) променливи, представящи интересуващите ни атрибути (свойства). На всяка променлива могат да бъдат присвоявани целочислени стойности. Подобна конструкция силно напомня поведението на софтуерна система по време на нейната работа и изменението на стойностите на съставящите я променливи.

Първата версия на интерпретатора Tempura е написан на езика Prolog от Бен Московски в началото на декември 1983 година. Съвременната версия на интерпретатора е написана на програмния език C през 1985 година от Роджър Хейл в университета в Кембридж, като негов докторантски тезис. През следващите години интерпретаторът е многократно дописван и разширяван с различни оператори и функционалности, като текущата версия Tempura 2.16 се поддържа от Антонио Кау, университет Де Монфорт, Лестър.

Основният принцип на работа на интерпретатора Tempura е обработването на текстови низове, които по своята същност представляват ITL твърдения или формули. Този принцип на работа, макар и коректно работещ, на база математическия модел на ITL, е труден за директно прилагане в реални системи. Поради тази причина е разработен съпътстващ модул, наречена AnaTempura. Той играе ролята на околна среда на интерпретатора, като събира и доставя информация, която да бъде интерпретирана.

Принципния механизъм на взаймодействие между Tempura и AnaTempura може да бъде описан по следния начин:

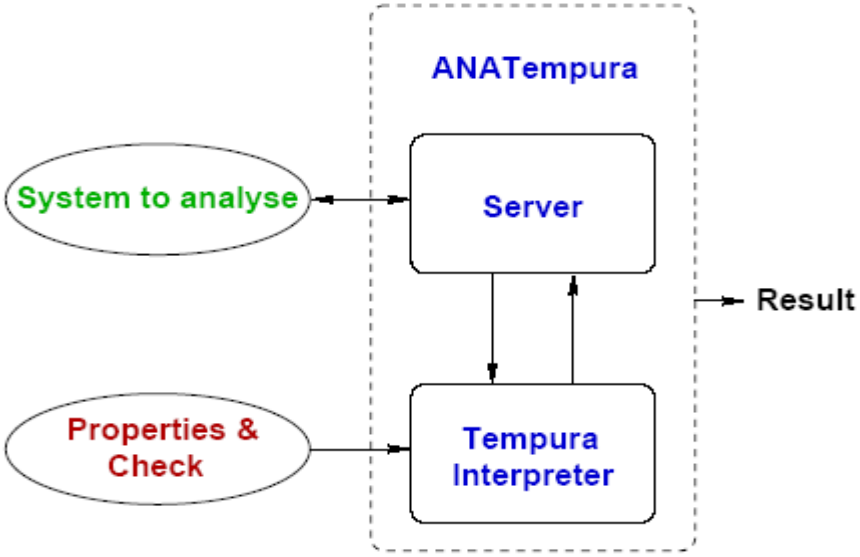
- Въз основа на анализа, който искаме да направим на дадена софтуерна или хардуерна система, се дефинира набор от променливи. Изменението на стойностите на тези променливи се следи по време на изпълнението на програмата с помощта на предварително дефинирани ITL изрази;

- AnaTempura служи като интерфейс между Tempura и системата, която се анализира. Нейната задача е да събира стойностите на следените променливи и да реализира присвоявания на променливите в предварително дефинираните ITL изрази, които впоследствие ще бъдат обработени от Tempura;

- AnaTempura подава като текстови низ съставените изрази, който Tempura интерпретира на база теоретичния модел на ITL и връща получения резултат;

- AnaTempura анализира интерпретирания резултат и прави съответните изводи.

Архитектурата на този механизъм за анализ и верификация на различни системи може да бъде онагледена с принципната схема:



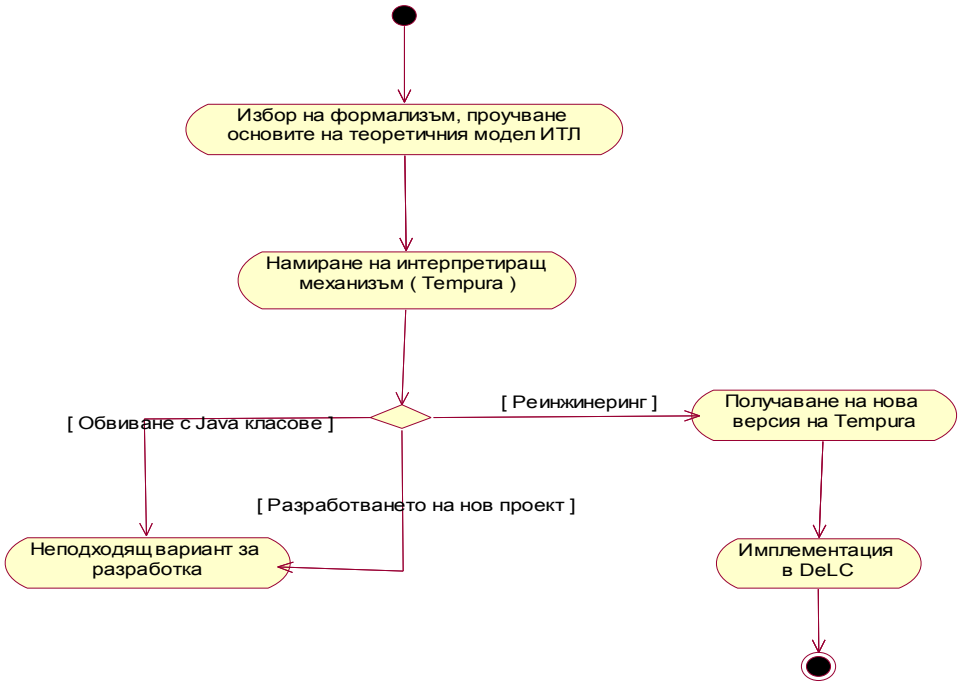
Фиг. 3: Принципна схема на взаимодействието Tempura - AnaTempura

2.3 Подход за разработване на архитектурата

Голямата сложност и комплексност на проблема изискват ясно дефиниран систематичен подход за решаването му. За създаване на механизъм за управление на времево-зависими процеси и вграждането му във вече изграждащата се архитектура на ВОП, ние предлагаме подход, включващ следните основни стъпки:

- Избор на формализъм за описание на времево-зависими процеси;
- Избор на интерпретиращ механизъм;
- Реинженеринг на съществуваща система;
- Интеграция в архитектурата на ВОП.

Стъпките са строго последователни и са представени визуално на диаграмата на Фиг. 4



Фиг. 4: Подход за разработване на архитектура

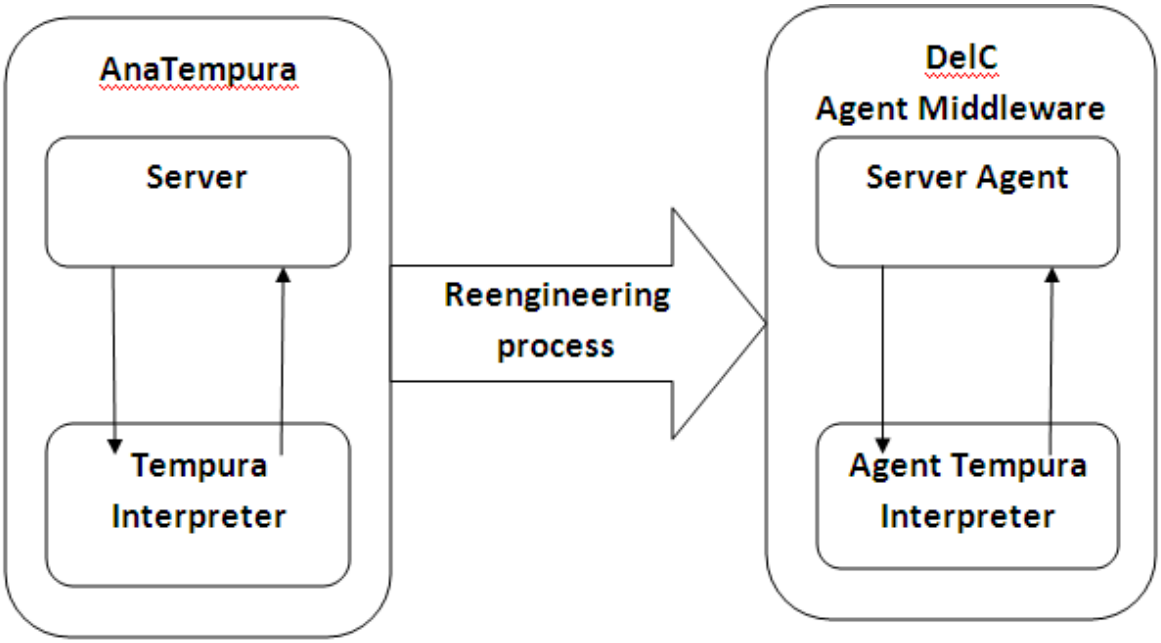
Някои от съществуващите механизми за дефиниране и описание на времево-зависими процеси бяха разгледани в Глава 2. Избраният от нас формализъм е специализирана темпорална логика, наречена Interval Temporal Logic (ITL). Този формализъм разглежда времето като крайна последователност от времеви интервали, което много се доближава до начина на обработка на данни в софтуерните системи и би довело до по-лесната му програмна реализация.

За постигането на нашите цели ние ще преминем през няколко последователни стъпки. Ще бъде извършен реинжинеринг на интерпретатора Tempura за преминаване към езика Java, с цел да се запази хомогенността на системата. Освен това разработването на новата версия ще премине през няколко ключови етапа, които да направят процеса по лесно проследим и коректен:

- Първо ще бъде направен директен превод от програмния език C на Java, за да се запази оригиналния синтаксис на една вече изпитана система, което ще спомогне и за тестването на новата версия.

- Следва промяна на архитектурата на Java интерпретатора, за да се получи обектно ориентирана версия. Това би направило системата по-лесна за използване и в други бъдещи проекти.

- Последната стъпка е създаване на агентно-ориентирана версия на Tempura, която да бъде вградена в съществуващата архитектура на DeLC. По този начин околната среда AnaTempura, която работи с оригиналния C интерпретатор, ще бъде заменена с околната среда изградена изцяло от агенти, а именно вече съществуващия агентен мидълуер на DeLC.



Фиг. 5: Трансформация на AnaTempura в арх. на DeLC

Глава 3 Обектно-ориентирана jTempura

3.1 Въведение

За да можем да използваме възможностите на ITL и нейния интерпретатор Tempura, трябва да намерим подход за неговата интеграция в архитектурата на ВОП, като от една страна запазваме хомогенността на пространството, а от друга – съхраняваме пълната функционалност на интерпретатора. За целта са разгледани и анализирани три основни подхода за трансформация на интерпретатора и неговата интеграция във ВОП [142]:

- Разработване на подходяща „обвивка" на съществуващия интерпретатор, съдържаща изпълними входно-изходни Java класове;
- Реализиране на изцяло нов проект, използващ теоретичния модел на ITL и програмиран на езика за програмиране Java;
- Реинжинеринг на оригиналния интерпретатор, програмиран на езика за програмиране C/C++, при съхраняване на пълната функционалност на доказано работещия и широко използван в различни приложения интерпретатор.

Обвиването на съществуващия интерпретатор във входно-изходни Java класове бе сметнат за неподходящ метод. Като основната причина за това е очакването за неефективна run-time система - комуникацията между компоненти, написани на различни програмни езици, може да доведе до забавяне на изпълнението на кода. Освен това обвиването на оригиналния интерпретатор ще попречи да бъде създадена една съвременна многократно използваема обектно и агентно-ориентирана версия на компилатора, реализирана на езика за програмиране Java, която в сравнение с оригиналната, да бъде по-лесна за развитие и внедряване и в други проекти.

Създаването на изцяло нов ITL интерпретатор на езика Java би отнел много време и ресурси. Основавайки се само на теоретичния модел на ITL бихме имали нужда от сложен и подробен анализ, за да бъде проектирано подобно приложение. Създаването

на тестови случаи, за доказване коректността на новия интерпретатор също не би била лесна задача. Евентуалното използване на вече създадени тестови случаи би ни накарало да напишем интерпретатор, спазващ вече реализирания в Tempura синтаксис, което като техника се доближава до идеята за реинжинеринг на съществуващата система. Имайки предвид това смятаме, че проектирането и създаването на изцяло нов ITL интерпретатор не е неподходящо за целите на нашия проект.

В дисертацията е избран третият подход, т.е. прилагане на реинжинеринг върху C/C++ версията на интерпретатора и пренаписването му на езика за програмиране Java като най-подходящ. Нашето решение може да бъде обосновано със следните причини:

- Всички софтуерни модули на ВОП са реализирани на езика за програмиране Java и от тук, най-подходящо е интерпретаторът да бъде написан на същия език – по този начин ще бъде осигурена необходимата хомогенност, което от своя старна дава допълнителни възможности за бъдещо разширяване и развитие на системата при добра ефективност на run-time;

- Също така една Java версия би дала възможност за по-лесно създаване на агентно-ориентирана версия на интерпретатора, която да бъде лесно вградена в мулти-агентната архитектура на ВОП;

- От гледна точка на развитието на самия интерпретатор, версия, реализирана на езика за програмиране Java, може да се използва в бъдещи проекти, използващи съвременни обектно и агентно-ориентирани архитектури;

- При този подход изцяло се използва проверената и доказалата своята ефективност в много приложения функционалност на Tempura. Оригиналната версия е използвана в реални системи и разполага с много тестови примери, които ще бъдат добра основа за първоначални тестове на новата версия ***jTempura***.

Обобщавайки, целта ни на този етап е пренасяне на пълната функционалност на Tempura в нова обектно-ориентирана архитектура, реализирана на езика за програмиране Java, която да може:

- Да съществува самостоятелно и да се интегрира в обектно-ориентирани приложения, реализирани на Java;
- В следваща стъпка, да се трансформира в агентно-ориентирана версия, която ще бъде интегрирана в изграждащата се мулти-агентна архитектура на ВОП.

При реализиране на реинжинерингговия подход сме изправени пред различни предизвикателства, като напр. следните:

- Оригиналната Tempura е реализирана на C/C++ - език, чиято програмна парадигма е различна от целевия език Java;
- В годините от създаването на интерпретатора са правени многократни допълнения и разширения на първичния код, в много случаи несистематично и без създаване на подробна системна документация – в крайна сметка актуалената Tempura е с много неясна архитектура;
- Липса на съпровождаща документация и модели – необходими са допълнителни усилия за детайлен анализ на съществуващия първичен код;
- Няма възможност за комуникация и допълнителни разяснения от оригиналните разработчици на интерпретатора.

След писмено мотивиране и запитване към собственика (Software Technology Research Laboratory, De Monfort University, Leicester, UK) на Tempura е получено официално разрешение за използване на оригиналния първичен код на интерпретатора.

3.1 Общ преглед на реинженерингговия процес за Tempura

Взаимствайки от добрите практики за разработване на обекно-ориентиран софуер, реинжинерингът на интерпретатора Tempura бе извършен чрез итеративен подход, по

подобие на стандартните модели, като бяха реализирани две ясно различими итерации. Итерациите бяха предхождани от стъпка, при която беше възпроизведена (от първичния код) и описана архитектурата на оригиналната Tempura.

Всяка итерация се състои от три основни фази:

- Проучване;
- Разработване;
- Тестване.

Проучващата фаза се състои от серия експерименти с ревърс-инжинерингови техники, приложени върху части от програмния код на интерпретатора. Резултатите от тези експерименти показаха използваемостта на съответната техника и приложимостта й върху целия код. Също така, тази фаза включва проучване на различни софтуерни продукти за реинжинеринг [139,79,25]. При реинженеринговия процес не използваме автоматизирани средства, поради следните причини:

- Необходимост от задълбочено и детайлно познаване и анализиране на първичния код, вкл. неговата структура;
- Избягване на неефективни автоматични трансформации.

Като резултат от разработващата фаза получихме аналогична на оригиналната C-Tempura версия, написана на програмния език Java. Тестовете в третата фаза на процеса бяха използвани за проверка коректността на Java-версията на интерпретатора и запазване функционалността на оригиналния. При реинжинеринговия процес искахме да запазим оригиналния синтаксис на интерпретатора, така че тестовете се проведоха, чрез изпълняване на скриптове от оригиналната C-Tempura. Паралелното изпълнение на тестови примери и сравняването на резултатите ни даде възможност да подтвърдим коректността на новата версия и съответсвието и като формат на входно-изходните данни с оригиналната Tempura.

Крайната цел е създаване на агентно-ориентирана версия на интерпретатора, която да бъде лесно вградена в съществуващата архитектура на ВОП. Поради това се нуждаем от междинен, коректно работещ Java-базиран прототип на интерпретатора.

Целта на първата итерация е чрез директен превод на оригиналния C код на Java да получим коректно работеща версията. При тази фаза бяха пренебрегнати някои от основните обектно-ориентирани характеристики и конвенции на програмния език Java, като целта е максимално запазване на структурата и синтаксиса на оригиналния интерпретатор. Така новата Java-базираната версия, макар и некоректна от гледна точка на добрите стилове на програмиране, е много лесно сравнима с оригиналния интерпретатор, имайки предвид именуването на променливи, процедури и т.н. Този начален вид на новия интерпретатор осигурява предимство при началното тестване и откриване на синтактични и логически грешки. Също така при тази първа версия не са вземани под внимание проблемите, свързани с оптималното изпълнение от гледна точка на паметта или времето за изпълнение.

Тази стъпка сама по себе си не е лесна и праволинейна поради факта, че се реализа превод между езици, поддържащи близки, но все пак различаващи се програмни парадигми. Някои от основните характеристики на езика за програмиране C не могат директно да се пренесат в Java, като следните примери, многократно срещани в оригиналния код на Tempura:

- Работа с паметта на ниско ниво – указатели към променливи, указатели към функции и адресна аритметика;
- Множествено присвояване - напр. x = y = 5;
- Предаване на параметри към функциите по стойност - достъпът до външна променлива чрез параметър може да стане само индиректно чрез подаване на указател;
- Изобилстваща употреба на макро-дефиниции – необходимост от препроцесор за включване на файлове и условно компилиране.

3.2 Реализация на реинженеринга на Tempura

3.2.1 Архитектура на оригиналния Tempura

В този раздел ще представим структурата на първичния код на оригиналния интерпретатор, получена като резултат от предхождащата реинженерингови итерации

стъпка [143]. Програмният код на интерпретатора (C-код) се състои от деветнадесет файла първичен код и пет библиотечни (хедър) файла, представени на Фиг. 6

Име на файл	Описание
Error.c	Прихваща различни типове грешки и прекъсвания на изпълнението
Init.c	Инициализира глобалните променливи и даннови структури
IO.c	Вход/Изход
Lexer.c	Лексикален скенер
Mem.c	Тук са сложени процедурите за достъп до паметта на Tempura
Parcer.c	Парсер
r-assign.c	Процедури за редуциране на „assignment" оператори
r-expr.c	Процедури за редуциране на изрази
r-interval.c	Процедури за редуциране на интервални оператори
r-io.c	Процедури за редуциране на входно/издходни оператори
r-linear.c	Процедури за редуциране на линейно-времеви оператори
r-lists.c	Процедури за редуциране на оператори обработващи списъци
r-symbol.c	Процедури за редуциране на символи, константи и идентификатори
r-system.c	Процедури за редуциране на системни команди
store.c	Осигурява място за съхранение на интервални структури
system.c	Системно зависещи параметри
top-level.c	Основният команден цикъл на Темпура, реализира се базовия алгоритъм
val.c	Процедури за обработване на стойности в интервална форма
version.c	Информация за версиите на Темпура

Фиг. 6: Списък от файлове на първичния код на Tempura

Отчитайки теоретичния алгоритъм за интерпретиране на ITL формули и твърдения, от резултатите на извършения анализ на програмния код могат да се направят следните изводи:

- Основният програмен цикъл на интерпретатора е реализиран във файл top-level.c;
- Базовите модели и правила за редуциране на темпорални формули и твърдения, според типа на използваните оператори, са реализирани в осем „r" файла. Всяка редуцираща итерация ще се

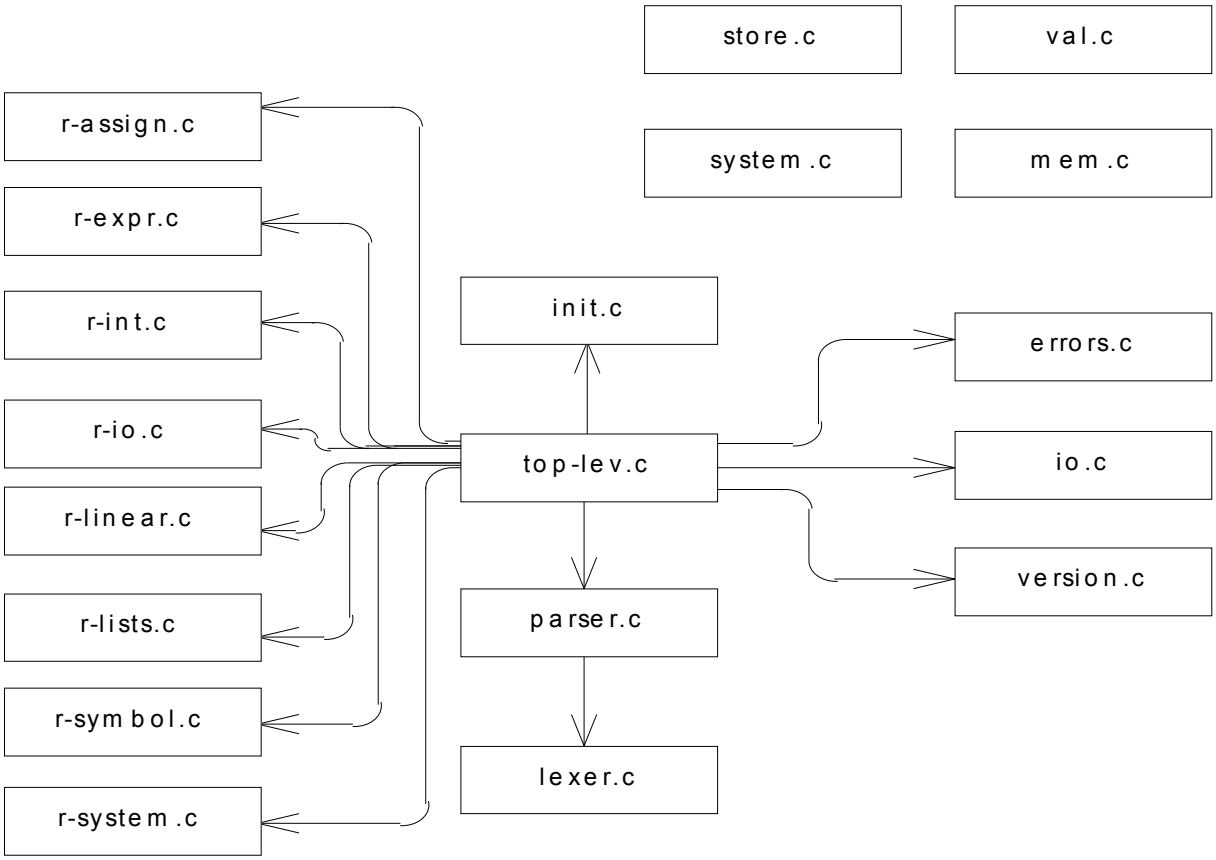
обръща към някои от тези файлове, което изисква специално внимание при превода и тестването на тази част от кода, която реално реализира теоретичните основи на ITL;

- Парсерът, модулът за лексикален анализ и входно/изходни операции са включени в три отделна файла. Тези компоненти се явяват ключови за бъдещата интеграция на Tempura в архитектурата на ВОП (реализират комуникация на интерпретиращия механизъм с външни системи);

- Формулите, които ще бъдат интерпретирани, се приемат като текстови низ от входно/изходните процедури. Това би улеснило дефинирането на асинхронни съобщения за взаимодействие в мулти-агентната архитектура на ВОП.

Структурата на кода и изводите бяха полезни за определяне последователността, в която да се извърши превода на кода от C на Java, както и при провеждане на последващи тестове и отстраняване на грешки, т.е. за определяне съдържанието на двете последващи итерации. Направеният анализ ни дава предимството да превеждаме и тестваме поетапно отделни части от кода. Предимството на този подход е, че тестваме по-малки функционалности, реализиращи определени логически структури, като по този начин по-лесно се отстраняват получените грешки и несъответствия. Още повече тестовите пакети, които получихме от екипа поддържащ текущата версия на интерпретатора, са групирани и тестват различни ITL структури, като списъци, твърдения, времеви оператори и т.н. Едва след успешното преминаване на тази базова група тестове можеше да поискаме по-сложна група от тестови случаи, с чието покриване да претендираме за първата коректна jTempura версия.

Принципната структура на първичния код на оригиналния интерпретатор и компонентите му, включени в отделните итерации е представена на схемата на Фиг. 7



Фиг. 7: Принципна схема на оригиналния първичен код

3.2.2 Реинженерингови итерации

Първата итерация включва ядрото на интерпретатора - Фиг. 7. Целта на *проучването* в първата итерация на реинжинерингговия процес е, използвайки знанията за теоретичните основи и концепции на ITL, както и анализа на първичния код, на основния програмен цикъл, на структурите и на променливите на оригинала, да се подготвят описание и модел на съществуващата архитектура на Tempura. Същевременно бяха проучени възможности и средства за автоматична трансформация на C-код в Java-код .

По време на *разработването* оригиналният C-код беше трансформиран в кореспондиращ Java-код. Имайки предвид сложността и обема на интерпретатора, извършването на пълна трансформация на оригиналния код, при последващо тестване би довело до трудно откриване на евентуално възникналите грешки. За преодоляването на този проблем беше адаптирана стандартна техника, позната като „*декомпозиция на проблем*”. В нашия случай това означава следното:

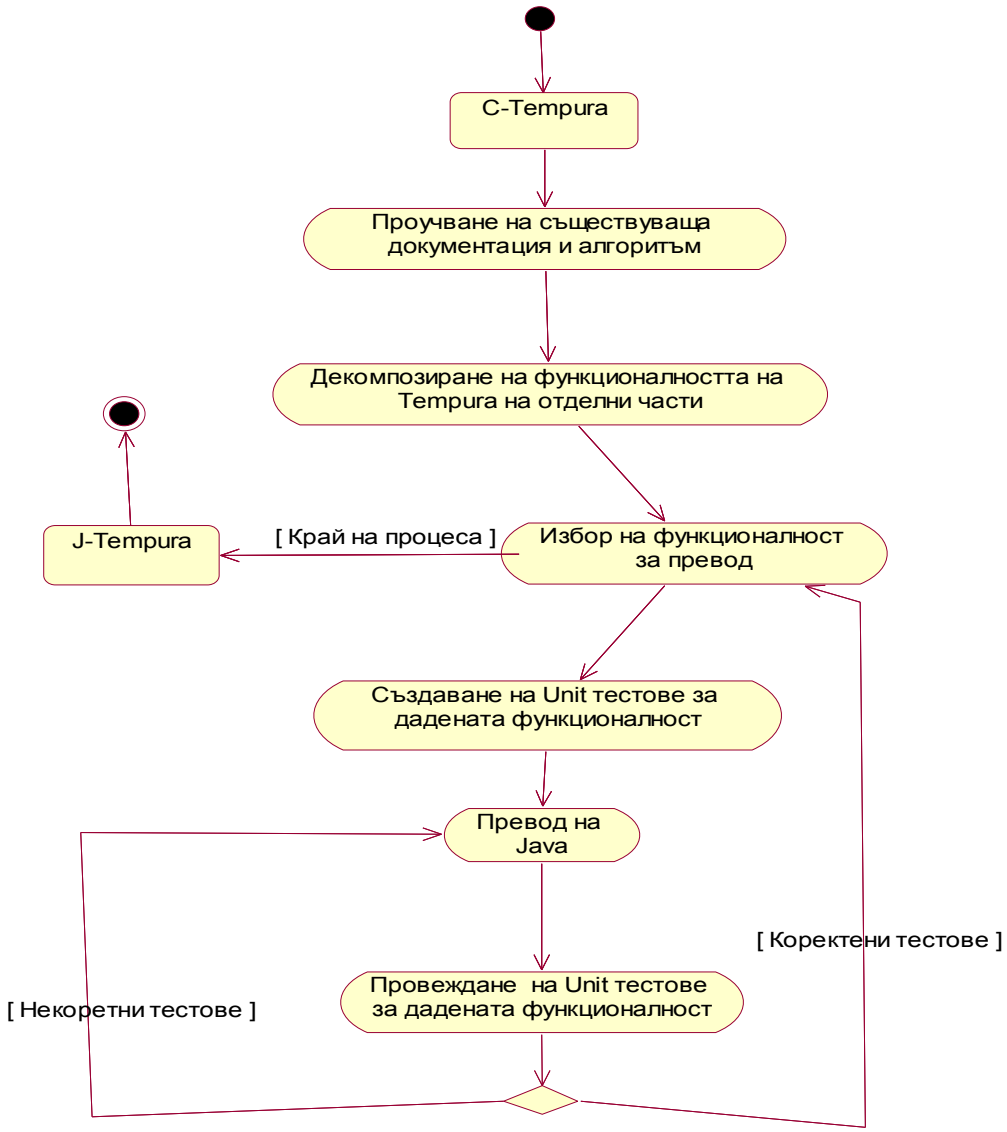
- Декомпозиране оригиналния код на малки части, които наричаме базова функционалност;
- Последователно пренаписване (трансформираме) всяка отделна базова функционалност;
- Новата версия на базовата функционалност се подлага на самостоятелно (unit) тестване;
- Едва когато пренаписаната базова функционалност покрие всички тестови случаи се започва с превода (пренаписването) на следваща базова функционалност.

Така в рамките на един цикличен процес поетапно беше пренаписан и тестван пълният код на Tempura. За успешното реализиране на разработващата фаза и преодалявайки някои базови различия на C и Java бяха създадени специални трансформационни правила. Някои примери за такива правила са следните:

- Създаване на Java клас за всеки C файл и неговия съответен хедър файл;
- Създаване на статичен метод в Java класа за всяка съответна функция в C файла със същото име, като конвенциите на Java за именуване на класове беше игнорирана;
- Създаване на статична променлива в Java класа за всяка глобална променлива в хедър файла;
- Създаване на статичен метод в Java класа за всяка макро-функция в хедър файла (където беше необходимо);
- Превръщане на всички булеви изрази и конструкции в съответните такива в Java;
- Заменяне на всички указатели към функции с инстанции на специален интерфейс, който ще извиква съответни Java-методи.

Задачата на *тестването* е да бъдат проведени интегрирани тестове на ядрото на jTempura, използвайки Tempura Standard Test Suite, доставена от разработчиците заедно с оригиналния код на интерпретатора. Тестовите примери бяха разделени на групи с нарастваща сложност и обем. Целта е да се тества поотделно всяка преведена функционалност, за да се открият по-лесно технически и логически грешки.

Жизненият цикъл на реализиране на първата итерация е представен на UML acticity-диаграмата на Фиг. 8



Фиг. 8: Първа реинженерингова итерация

Втората итерация стартира след създаването на работещо ядро на jTempura. Итерацията включва следните модулите от оригиналната структура на интерпретатора:

- Lexer.c;
- Parcer.c;
- Init.c;
- IO.c;
- Error.c;
- Version.c;

Жизненият цикъл на втората итерация е аналогичен на този, представен на Фиг. 8.

3.3 Резултати от реинженеринга на Tempura

В тази точка, накратко ще бъдат обобщени основните резултати от реинженеринга на Tempura. Основният резултат е създаването на jTempura. По време на етапа, предхождащ същинските реинженерингови дейности проучването бе съсредоточено в запознаване с основните принципи и идеи на ITL. С помощта на теоретичните постановки, описани в труда на Бен Московски [81], успяхме да извлечем семантиката на различните типове оператори и структури, поддържани в Temura.

3.3.1 Алгоритъм на жизнения цикъл на интерпретатора

Преди да започне практическото пренаписване на интерпретатора беше синтезиран алгоритъма на жизнения цикъл на интерпретатора, управляващ обработването на ITL формули и ITL твърдения. В този раздел ще представим накратко функционирането на алгоритъма, като за демонстарция ще използваме следната интервално темпорална формула:

$$(\bigcirc\bigcirc empty) \wedge (I = 0) \wedge (I \, gets \, I + 1) \wedge \Box(J = 2 \cdot I)$$

Формулата е вярна за интервали с дължина 2, при които променливата *I* приема последователно стойностите 0, 1 и 2, а променливата *J* – съответно стойностите 0, 2 и 4.

Идеята на алгоритъма е ITL формулите да се обработват посредством *последователни преобразуващи стъпки*, като се отчитат състоянията на формулата във всеки един времеви интервал. По време на първата стъпка примерната формула се преобразува в логически еквивалентна конюнкция на двете формули *present_state* и ⓦ*what_remains*:

$$present\_state \wedge ⓦ what\_remains.$$

Тук формулата *present_state* се състои от присвояване на стойности на променливите и също индикира, дали интервалът не е последен от гледна точка на валидността на формулата. Формулата *what_remains* показва какво би трябвало да се изпълни в следващото състояние, ако формулата е валидна за последващ времеви интервал. По този начин обработката на една ITL формула може да се разглежда като вид конюнкция на две формули.

За разглежданата формула *present_state* има следната стойност:

$$\left(I = 0\right) \wedge \left(J = 0\right) \wedge more.$$

Стойността на *what_remains* е формулата:

$$(\bigcirc empty) \wedge (I = 1) \wedge (I\, gets\, I + 1) \wedge \Box (J = 2 \cdot I)$$

По време на времевите интервали, за които гореспоменатата формула е валидна тя претърпява следните преобразования съгласно теоретичния модел на ITL:

Преобразуване на формула

Преди състояние 0:

$$(\bigcirc\bigcirc empty) \wedge (I = 0) \wedge (I\, gets\, I + 1) \wedge \Box (J = 2 \cdot I)$$

След състояние 0:

$$\begin{aligned}&\left[(I = 0) \wedge (J = 0) \wedge more\right] \\ &\quad \wedge ⓦ\left[(\bigcirc empty) \wedge (I = 1) \wedge (I\, gets\, I + 1) \wedge \Box (J = 2 \cdot I)\right]\end{aligned}$$

Преди състояние 1:

$$(\bigcirc empty) \wedge (I = 1) \wedge (I\,gets\,I + 1) \wedge \Box(J = 2 \cdot I)$$

След състояние 1:

$$\begin{aligned}&\big[(I = 1) \wedge (J = 2) \wedge more\big] \\ &\quad \wedge \textcircled{w}\big[empty \wedge (I = 2) \wedge (I\,gets\,I + 1) \wedge \Box(J = 2 \cdot I)\big]\end{aligned}$$

Преди състояние 2:

$$empty \wedge (I = 2) \wedge (I\,gets\,I + 1) \wedge \Box(J = 2 \cdot I)$$

След състояние 2:

$$\begin{aligned}&\big[(I = 2) \wedge (J = 4) \wedge empty\big] \\ &\quad \wedge \textcircled{w}\big[false \wedge (I\,gets\,I + 1) \wedge \Box(J = 2 \cdot I)\big]\end{aligned}$$

Естествено, формулата може да бъде разглеждана и за по дълги времеви интервали, но при всеки следващ интервал прочитането на оператора *empty* ще показва, че формулата вече е невалидна.

За да изпълни теоретичния модел на ITL, интерпретаторът Tempura реализира техника, използваща описаните по-горе преобразования. От гледна точка на програмната реализация на интерпретатора, за представяне състоянието на обработваната програма, интерпретаторът използва четири системи променливи:

- ***Program*** - това е променлива, която първоначално съдържа в себе си цялата Tempura програма. След изпълнението на всяко състояние, тя се трансформира, чрез описания по-горе алгоритъм, във формула от вида Ⓦ*w*, където *w* описва какво трябва да се изпълни в следващото състояние. Ако отнесем това към теоретичните основи, то *Program***,** във всяко едно състояние след началното, съответства на елемента *what_remains* от теоретичния модел за обработка на формули. Променливата *Program* всъщност по време на изпълнение на програмата приема различни стойности, които съответстват на състоянието на обработваната

формула или твърдение във всеки един от времевите интервали, за които тя е валидна;

- ***Memory*** - това е индексиран списък от клетки в паметта. Всяка клетка може да бъде празна или да съдържа целочислена стойност или списъчен дескриптор. В началото на всяко състояние, на всяка клетка се присвоява празен списък $\langle\ \rangle$, по този начин се показва, че не е била съхранена никаква стойност. Когато стойността *c* бъде поставена в клетката, то на конкретните съдържания на клетката се присвоява единичното множество $\langle c \rangle$.;

- ***Current_Env*** - в една Tempura програма това е среда, която представлява списък с отделни входове за всяка променлива. Всеки вход е наредена двойка, съставена от име на променлива и индекс на клетката в паметта, където се съхранява стойността на променливата. За демонстрационния пример, стойността на *Current_Env* би могла да бъде следната:

$$\langle\langle "I", 0\rangle\rangle, \langle\langle "J", 1\rangle\rangle.$$

- ***Current_Done_Cell -*** представлява индекс на клетка от паметта, която се нарича *done-flag*. Интерпретаторът оценява с *true* или *false* всяко едно нейно състояние, в зависимост от това дали състоянието е последно, от гледна точка на времевите интервали, в които е валидна дадената ITL формула. Например, в състояние, където твърдението *empty* се среща, интерпретаторът слага стойността *true* в *done-flag* клетката. Ако твърдението *more* се среща, то се използва стойността *false*. Ако потребителят не присвои стойност за конкретно състояние на *done-flag*, тогава интерпретаторът дава съобщение за грешка.

Базовият алгоритъм, използван от интерпретатора, може да бъде синтезиран и представен в процедурната форма, дадена на Фиг. 9.

```
begin
    local Program, Memory, Current_Env, Current_Dane_Cell;
    prepare_execution_of_program;
    loop
        execute_single_state
    exit when Memory [Current_Done_Cell] = ⟨true⟩
    otherwise
        advance_to_next_state
end.
```

Фиг. 9: Базов алгоритъм на Tempura - процедурна форма

Накратко ще опишем отделните стъпки на алгоритъма:

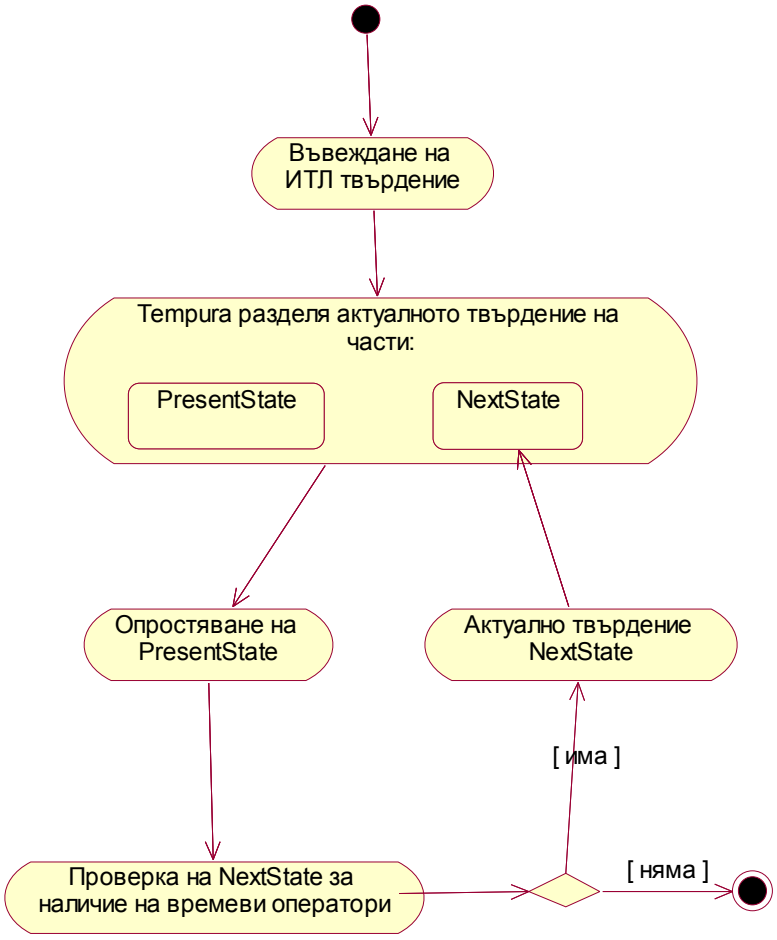
- *prepare_execution_of_program*: променливата *Program* на интерпретатора маркира синтактичното дърво на програмата, а променливата *Current_Env* е инициализирана да посочва и създава връзки с клетка от паметта. От своя страна паметта е разпределена да има една клетка за всяка променлива на програмата плюс една допълнителна клетка, където да бъде записана стойността на *done_flag* от променливата *Current_Done_Cell*. Всички клетки от паметта първоначално са празни (присвоява се към ⟨ ⟩);

- *execute_single_state*: стойността на променливата *Program* се трансформира докато е във формата ⓦ*w*. Тук се прави и проверка, за да се установи дали *done_flag* е получил стойност *true* или *false*, също така това се отразява върху клетките от паметта, който съхраняват стойностите на *Memory* променливата;

- *advance_to_next_state*: ако текущото състояние не е последно, се прави подготовка за отбработка на следващото състояние. Това се осъществява посредством изчистване на клетките от паметта, съхраняващи *Memory* и редуциране на променливата *Program* според точно определени логически правила базирани на теорията на ITL. Вече редуцираната форма бива отново разглеждана в познатия вид:

$$present\_state \wedge \textcircled{w} what\_remains.$$

Така итеративно се редуцира въведената формула или твърдение до премахването на всички времеви оператори, т.е. до преминаване през всички времеви интервали, за които е дефинирано въведеното твърдение или формула. Принципната схема на действие на интерпретатора може да бъде онагледена със следната диаграма:



Фиг. 10: Принципна схема на базовия алгоритъм

Следвайки плана за фаза „Разработване" на двете итерации, поетапно беше извършен превод на първичния код, като всички типични C конструкции бяха заменени с еквивалентни на Java. Най-често срещаните различия и трудности при превода бяха свързани с подмяната на указателите, използвани от оригиналните разработчици, с контрукции на Java. Естеството на проблемите при превода е породено от относително различните типове езици, с които се опитваме да боравим. Езикът за програмиране C е от по-ниско ниво на абстракция; той е предназначен за програмиране на системи с голямо бързодействие и прецизно управление на паметта, поради което разполага с механизми, като указателите, за взаимодействие и директно манипулиране на клетки от паметта. Несъмнено това дава много предимства и е характерно за стила на програмиране, по времето, когато е програмирана първата версия на Tempura (C-Tempura).

Езикът за програмиране Java (основното средство за разработване на ВОП) е от по-високо ниво на абстракция и е изцяло обектно-ориентиран език за програмиране. Освен това Java разполага с възможности за компонентно програмиране, което дава редица преимущества, готови компоненти и възможности за боравене с контейнери, с които имаме възможност да създадем и управляваме така важните за ВОП агенти.

Важно е да се отбележи, че при така направения първоначален превод са пренебрегнати някои от добрите практики и конвенции за писане на езика Java. Това е направено с цел максимално да се запазят именуването на променливите и структурите от оригиналния код, както и за по-бързо постигане на работещ Java прототип. Запазването на оригиналните структури би било от полза при началното тестване и верифициране на нашата Java версия.

След завършването на първоначалния превод започнахме тестове на jTempura. За да сме сигурни в правдивостта на тестовите примери и тяхните резултати бяха използвани тестове и скритове, предоставени заедно с програмния код от оригиналните разработчици. Самите тестове се извършват чрез паралелно стартиране на тестови скриптове на двете версии на интерпретатора и сравняване на върнатите резултати. Провеждането на тестовете и отстраняването на грешките следваше добре познатия итеративен модел за разработване на софтуерни приложения. Трябва да се отбележи, че на този етап от разработката не се взема предвид бързодействието на jTempura. Вероятно в бъдеще, поради естеството на ВОП (предоставя електронни услуги в реално време и бързодействието не е от маловажно значение за реалното опериране на пространството), ще бъде наложителна оптимизация на новия интерпретатор.

След успешното приключване на първоначалния набор от тестове (Tempura Standard Test Suite) беше инициирана кореспонденция с поддържащите официалната C-версия на интерпретатора. Целта на кореспонденцията беше да получим по-комплесни тестови случаи (Tempura Extended Test Suite), с които окончателно да се потвърди коректната работа на jTempura. Преминаването на голям брой допълнителни тестове, съпроводено със съответните корекции в кода на системата, доведе до създаването на стабилен прототип на интерпретатора, който беше официално признат за първа Java версия на ITL интерпретатора Tempura.

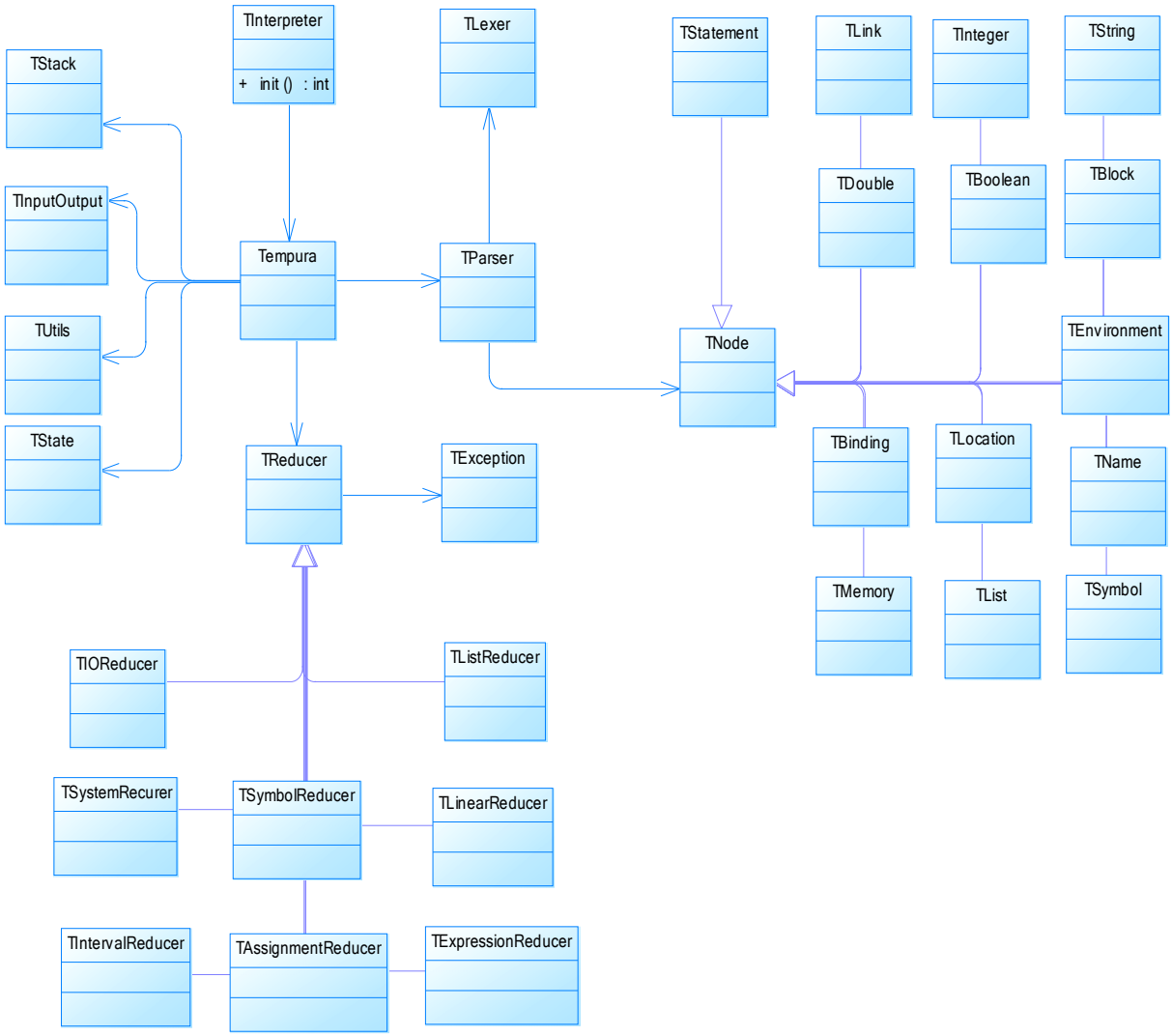
Въпреки директения превод на кода бяха направени някои промени по структурата, като в Java версията се опитахме да обособим логически близки групи от класове. Това ще ни даде възможност в следващите етапи на разработка да постигнем така важната и характерна за Java пакетна структура на интерпретатора. Основните предимства от това са следните:

- Постигане на по-добре структуриран код, което ще даде възможност за бъдещо развитие на интерпретатора, лесното му използване в други проекти и от други работни групи;

- Създаването на пакети може да позволи части от кода да бъдат дефинирани като библиотеки, които да се вградят във външни системи, които се нуждаят от контрол на времево-зависими процеси – начин, по който се използва оригиналния интерпретатор в практиката;

- От гледна точка на създаването на агентно-ориентирана версия, пакетите са много по-лесни за трансформация. Теорията казва, че агентите трябва да имат проста функционалност и не бива да изпълняват много и разнородни задачи. От тази гледна точка е логично да се търси архитектура, която преобразува всеки логически обособен пакет в агент.

Архитектурата на новия интерпретатор jTempura е представена на диаграмата на Фиг. 11.



Фиг. 11: Архитектура на jTempura

Глава 4 Агентно-ориентирана AjTempura

При изграждане на агентно-ориентираната версия на Tempura (AjTempura), която ще бъде интегрирана в архитектурата на ВОП, трябва да бъдат спазвани две съществени изисквания:

- Запазване хомогенността на пространството;
- Осигуряване на лесна интеграция на нови компоненти в съществуващите архитектури.

С реализиране на jTempura първото условие е изпълнено, т.е. новата версия на Tempura може да работи върху Интегрираната Технологична Платформа (ИТП) [166] на пространството, използваща виртуална машина на Java. Това свойство ще бъде автоматично пренесено в агентно-ориентираната версия, понеже при реинженеринга на jTempura не се предвижда промяна на езика за имплементация.

По дефиниция ВОП се състои от автономни контекстно-зависими компоненти, което предполага агентно-ориентирана архитектура на пространството. Така jTempura, със своята обектно-ориентирана структура, не удовлетворява второто изискване. По тази причина е необходима нова реинженерингова стъпка, където обектно-ориентираната структура на интерпретатора ще бъде трансформирана в агентно-ориентирана, която ще наричаме AjTempura, при запазване пълната функционалност на jTempura.

При търсене на решение за успешно провеждане на трансформацията възникват различни въпроси, като напр. следните:

- Как може да бъде пренесена функционалността на класовете във функционалност на кореспондиращи агенти, отчитайки съществуващите релации между класовете в изходната (обектно-ориентираната) архитектура?
- Как ще бъде трансформирана вътрешната структура на класовете във вътрешна архитектура на кореспондиращите агенти?

- Как ще се изградят елементи в агентно-ориентираната архитектура, които нямат аналог в обектно-ориентираните архитектури, като напр. ментални състояния и околна среда?

За целите на дисертацията от съществено значение е отговорът на първия въпрос. Решения на останалите два проблема могат да се търсят в рамките на конкретна работеща агентно-ориентирана архитектура. Така напр., околната среда на AjTempura е ВОП, като за връзка с тази околна среда ще бъдат разработени специализирани интерфейсни агенти, както е обяснено по-долу в главата.

4.1 Обектно-ориентирани архитектури

Основно предназначение на обектно-ориентираните парадигми е справяне с комплексността на решаваните с помощта на компютри задачи. За целта се използват два принципни подхода – декомпозиция и абстракция. Базови характеристики на добрите обекно-ориентирани архитектури са „слабо свързаност" и „силна съгласуваност" [17]. *Силната съгласуваност* изисква всеки компонент в архитектурата да бъдат проектиран така, че да реализира точно определена задача от общата функционалност на приложението. *Слабата свързаност* изисква в архитектурата на приложенията да се поддържа минимален брой семантични връзки между компонентите.

От синтактична гледна точка, в обектно-ориентирания език за програмиране Java съществуват два вида компоненти: класове и интерфейси (специален вид абстрактен клас, който се реализира посредством класове). Концепцията за класове в Java (също: C++ и др.) отразява недостатъчно значимите видове софтуерни компоненти. От семантична гледна точка (функционалност, предназначение, използване), класовете могат да бъдат различни видове компоненти. В този контекст различаваме две големи групи:

- Императивни компоненти;
- Обектно-ориентирани компоненти.

Императивните компоненти обикновено се използват за реализиране на определена функционалност или структури данни, при приложението на които създаването на инстанции е безсмислено. Императивните компоненти могат да бъдат:

- *Множество функции* – това са множество от свързани функции без общи данни. Данните не могат да служат за обмен на информация между функциите, като така те могат да се използват за реализиране на независими алгоритми;

- *Абстракция данни* - множество от свързани функции с общи скрити данни, без възможност за създаване на инстанции

- *Множество константи* – това е вид обобщение на видими константи за поделено използване.

Обектно-ориентирани компоненти са предназначени за реализиране на функционалност, която може да се използва за решаване на проблеми посредством генериране на техни инстанции. Основни обекно-ориентирани компоненти са следните:

- *Абстрактни типове данни (ADT)* - множество от свързани функции с общи скрити данни. Могат да се използват за създаване на нови типове данни, дефинирани от потребителя;

- *Класове данни* - обобщение на видими данни в един тип, които се използват за предоставяне на инстанции на различни видове структури от данни.

По отношение на обектно-ориентираните архитектури слабата свързаност означава компонентите (класове) да имат възможно най-малко връзки помежду си. Видовете връзки в обектно-ориентираните архитектури могат да бъдат следните:

- *Асоциация* – семантична връзка между два класа, където обектите на един клас имат достъп до публичните методи и атрибути на друг клас;

- *Агрегация* – семантична връзка между два класа, която показва, че един обект от един клас има указател към обект от друг клас или че съдържа в себе си обект от друг клас;

- *Наследяване* – семантична връзка между два класа, при която един клас наследява атрибутите и методите на друг клас;

- *Зависимост* – семантична връзка, при която обектите на един клас зависят от изпълнение на методи или дефиниране на атрибути в другия клас.

4.2 Агентно-ориентирани архитектури

Съществено за агентите е, че те оперират в някаква, различима и частично контролируема от тях околна среда. Теорията за агентите и съответно за тяхната архитектура се формира около понятието за „*рационално поведение*". В съответствие с поставената им задача рационалните агенти трябва да могат да оценяват поведението си и на основата на тези оценки да измерват ефективността на действията си. При агентите със сравнително опростена архитектура (обикновено се наричат реактивни агенти) програмистите задават „рационални" действия директно в програмния код, реализиращ тяхното поведение. При агентите със сложна архитектура (интелигентни агенти) програмистите реализират модел, при който агентите са в състояние самостоятелно да подобряват поведението си (да се обучават).

Изискване към рационалните агенти е те да бъдат „успешни". За определяне доколко един агент е „успешен" съществуват различни гледни точки и оценки, които могат да бъдат и конфликтни. В такива случаи е необходимо намиране на удовлетворителен "trade-off". Съществен проблем за „успешните" агенти е възможност за реагиране на неочаквани събития и промени в околната среда. За тази цел са необходими допълнителни разходи за предвиждаща калкулация.

Обикновено, рационалните агенти оперират с ограничен капацитет (ресурси) относно планиране, предвиждане, избор и извършване на действията. Тогава говорим за „ограничена рационалност", т.е. намиране на смислени решения при целесъобразно използване на наличните ресурси. Така мащабът на действие на един рационален агент може да се определи като оптимален успех в съответствие с неговите:

- Актуални знания и способности;
- Ограничени ресурси;
- Ограничено време.

Основно за опериране на рационалните агенти е така нареченият. "*sense-think-act*" жизнен цикъл, спрямо който агентите се структурират относно:

- Възприемане – подготовка на входа или входните сигнали;
- Избор на решения за действие;
- Самото действие.

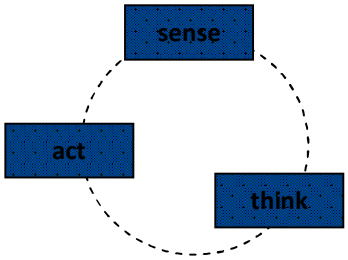
Стандартният жизнен цикъл на рационалните агенти може да бъде представен формално както е дадено на Фиг. 12. Различните видове рационални агенти имат различни модификации на стандартния жизнен цикъл [22].

repeat

p := sense(x);

s_{new} := thought(s_{old}, p);

a := act(s_{new});



forever

x ∈SensoryInputs, множество на възприетите сигнали (входни данни);

p ∈Perceptions, множеството на възприятията;

s ∈InternalStates, множество на вътрешните състояния;

a ∈Actions, множество на действията;

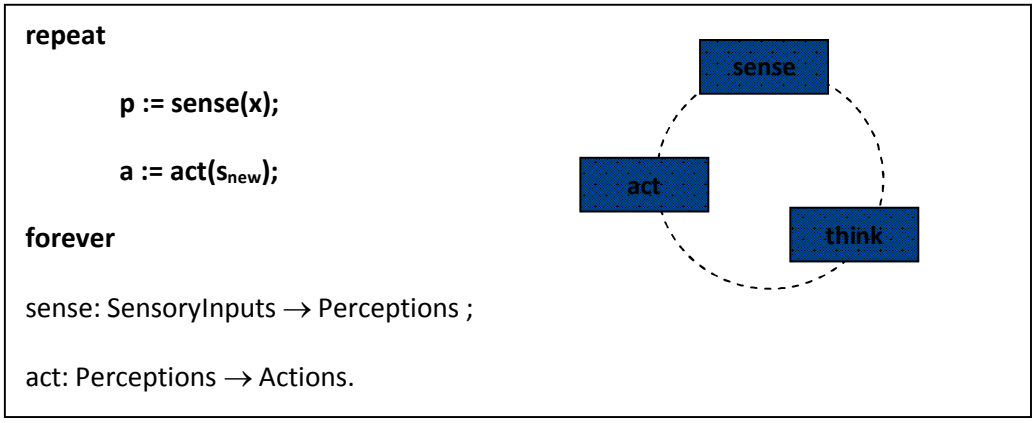
sense: SensoryInputs → Perceptions ;

thouhgt: InternalStates x Perceptions → InternalStates ;

act: InternalStates → Actions.

Фиг. 12: Sense-Think-Act жизнен цикъл

Най-опростените архитектури са тези на реактивните агенти Фиг. 13. Базирайки се на възприятията си реактивните агенти определят действието, което ще извършат.



Фиг. 13: Sense-Think-Act жизнен цикъл на реактивен агент

Те могат да притежават „памет". „Паметта" на агентите може да бъде отнесена към различни неща, като напр. ситуацията в околната среда (минало и настояще), ангажименти и планове (бъдеще). При този тип агенти вътрешното състояние се състои от два компонента:

- Вяра (beliefs) – това са допускания (предположения) на агента за околната среда;
- Ангажименти (commitments) – агентите трябва да могат да поемат ангажименти, като стабилността в тяхното поведение се постига посредством съблюдаване на стари ангажименти.

repeat

p := sense(x);

b_{new} := update(b_{old}, p);

c_{new} := commit(b_{new}, c_{old});

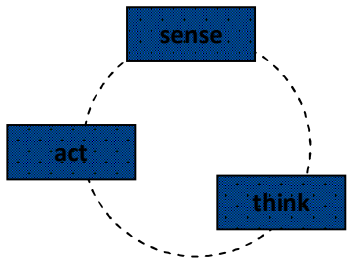
a := act(c_{new});

forever

update: Beliefs x Perceptions → Beliefs ;

commit: Beliefs x Commitments → Commitments .

act: Commitments → Actions.



Фиг. 14: Sense-Think-Act жизнен цикъл на агент със състояния

Третият тип са така наречените stimulus-response агенти Фиг. 15, които разполагат с модел на околната среда. По принцип това са по-сложни реактивни агенти, които могат да вземат комплексни решения без използване на „памет”. При тези агенти обикновено съществуват две възможности за избор на следващо действие:

- Оценяване на актуално възможните действия и избор на най-подходящото;
- Оценяване на възможните последователности от действия, избор на най-подходяща, изпълнение на първата дейност и повторение на същата процедура.

repeat

p := sense(x);

b_{new} := update(b_{old}, p);

c := commit(b_{new});

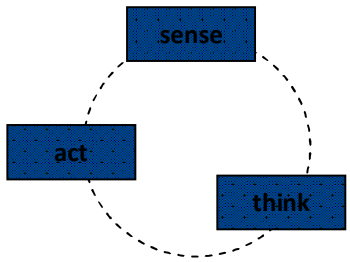
a := act(c);

forever

update: Beliefs x Perceptions → Beliefs ;

commit: Beliefs → Commitments .

act: Commitments → Actions.



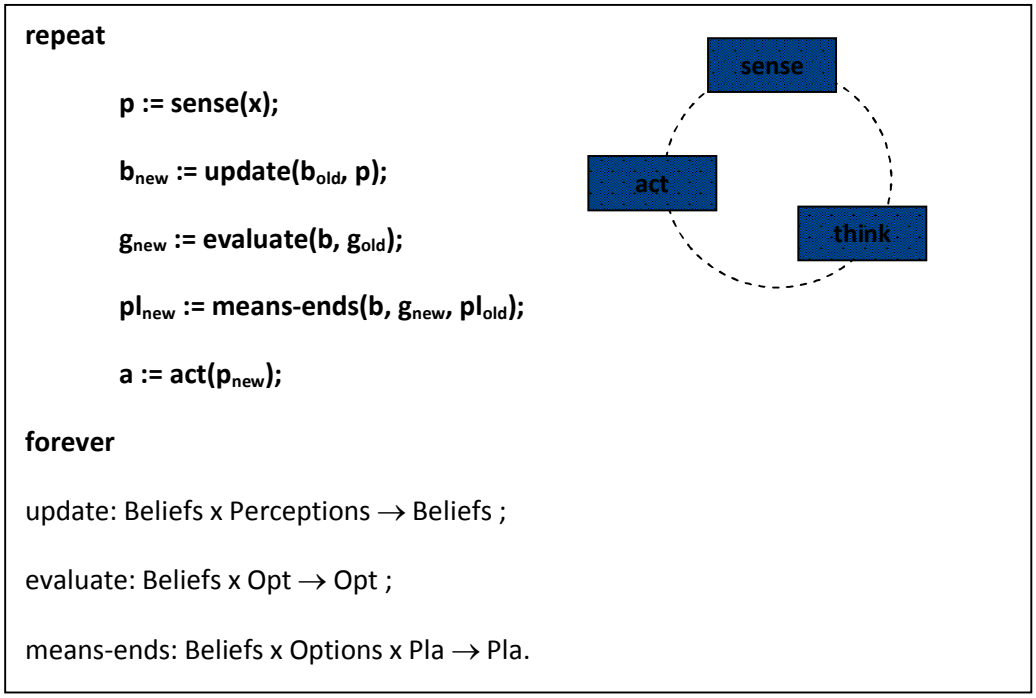
Фиг. 15: Sense-Think-Act жизнен цикъл на реактивен агент с модел на околната среда

Една от най-сложните архитектури е тази на агенти с цели Фиг. 16. Целите са основа за проактивност на агентите. При тях процесът за вземане на решение се извършва на два етапа:

- Обмисляне (deliberation) – на този етап се определя една (желана, полезна, ...) цел (съотв. поемане на задача);
- Търсене на решение (means-ends-reasoning) – избор или генериране на (полезни, приложими, ...) планове за достигане на една цел.

При този тип агенти един ангажимент *c* се декомпозира на *c = [g, pl]*, където:

- *g* е цел – елемент от едно множество на опции (g ∈ Opt);
- pl е план – елемент от едно множество от планове *(pl ∈ Pla)*.



Фиг. 16: Sense-Think-Act жизнен цикъл на агент с цели

За интелигентните агенти, класическата архитектура се нарича Belief-Desire-Intention (BDI) [97]. Тази архитектура използва модел на човешката дейност за представяне на „ограничената рационалност", където:

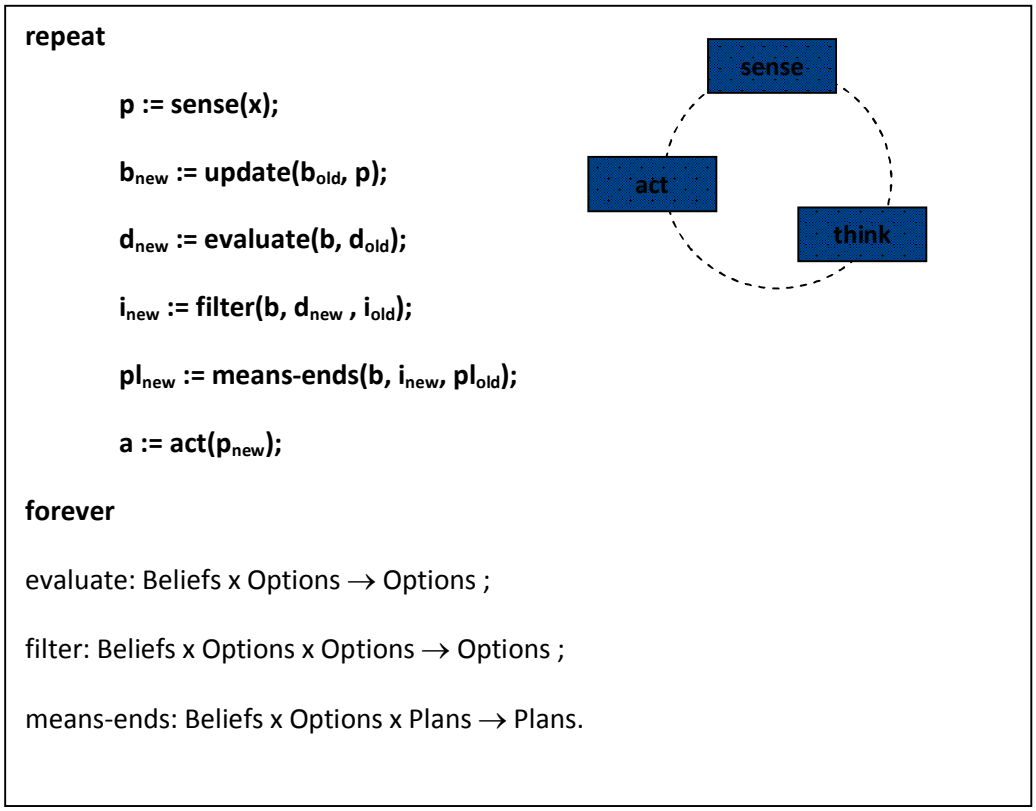
- Вяра (belief) – както и при агентите с цели представя приеманията на агента за околната среда (модел на околната среда);
- Желания (desire) – общи желания или задачи на агента, които още не са трансформирани в конкртени намерения;
- Намерения (intention) – намеренията на агента са еквивалентни на съществуващите му ангажименти, вкл. и към самия себе си.

Изборът на желанията се извършва от множество на съществуващи опции. Изборът на намерения се извършва от досегашни намерения и нови желания. Така ангажиментите в този архитектурен модел се декомпозират на $c = [d, i, pl]$, $c \in Commitments$, $Commitments = Options\ x\ Options\ x\ Plans$, където:

- d, i са множества на опции;

- Options – тук е множество от множества на опции, т.е. Options $\subseteq$ 2Opt ;

- Plans – тук е множество от множества на планове, т.е. $Plans \subseteq 2^{Pla}$.



Фиг. 17: BDI агент

При мулти-агентните системи, където всеки агент получава индивидуална задача или подзадача, комункацията и кооперирането са основни проблеми. Комуникацията между агентите е асинхронна и предполага наличието на посредническа система, реализираща обмен на съобщения (брокерска система). Обикновено за реализиране на комуникацията и кооперирането се използват многослойни протоколи, поддържащи както техническите аспекти (за целта се използват общоприложимите комуникационни протоколи), така също и съдържателни аспекти (протоколи на по-високо семантично ниво). Съдържателните протоколи са вид координационни правила на „социално ниво”, които могат да бъдат моделирани като напр. договорни мрежи (contract-net-protocol) [109], механизми от тип „черна дъска” (blackboard) [40], социални норми, правила на игри, пазарни механизми [155]. В стандарта за агентни архитектури Foundation for Intelligent Physical Agents (FIPA) [44] е специфициран език за взаимодействие между

агенти Agent Communication Language (ACL) [43], поддържащ комуникация и кооперация между агенти на високо семантично ниво. Езикът се базира на Speech Act теорията [106], като основните структури са перформативи, доставящи съобщения между агентите.

4.3 Сравнение между двете архитектури

Сравнение на някои от основните характеристики на обектно-оринетираните и агентно-ориентираните архитектури са обобщени в Фиг. 18. Съществуват съществени различия между двата типа архитектури по отношение на управлението, организацията и структурирането на приложенията, както и на комуникацията между отделните компоненти. Определени компоненти нямат аналози, като напр. понятията от агентно-ориентираната парадигма „околна среда" и „ментални свойства", както и различните релации в обектно-ориентираните архитектури.

Характеристика	**Обектно-ориентирана архитектура**	**Агентно-ориентирана архитектура**
Управление	Централизирано	Децентрализирано
Организация и структуриране	Йерархична, със строго специфицирани релации между отделните класове	Нейерархична, състояща се от автономни агенти, които сами могат да решават в какви релации може да участват
Комуникация	Синхронна	Асинхронна
Околна среда	Не	Съществен елемент в архитектурата
Ментални свойства	Не	Съществени за разработване на интелигентни агенти
Видове релации	Асоциация, агрегация, наследяване, зависимост	Не се поддържат експлицитно, могат да се симулират различни релационни отношения

Фиг. 18: Основни сравнителни характеристики

4.4 Модел на контекстно-зависима архитектура на ВОП

По дефиницията ВОП се състои от автономни контекстно-зависими компоненти, които могат да оперират в променяща се околна среда. Тези компоненти ще бъдат реализирани като интелигентни агенти с „ограничена рационалност". Предполагайки използването на такива агенти приемаме следните допускания:

- Функционалността на пространството в даден момент е променливо множество от информационни ресурси (агенти, услуги, структури данни, ...);

- Без ограничаване на общовалидността върху пространство, приемаме, че функционалността се доставя от агенти;
- При настъпване на промяна в околната среда определени компонентите от пространството трябва да извършат определени компенсаторни действия, отразяващи тази промяна;
- Съществува минимална функционалност, без която пространството не може да оперира.

За реализация на пространството предлагаме абстрактен модел за контекстно-зависима агентна архитектура, наречен C3A (Context-Aware Agent Architecture). C3A оперира по следния начин:

- С *Agents* означаваме множеството на всички потенциално опериращи в простарнството агенти;
- $Agents = PA \cup OA$, така че $PA \cap OA = \varnothing$ – в множеството на всички агенти различаваме две дизюнктивни групи агенти;
- *PA* – наречени *персистентни агенти*, представят минималната функционалност на пространството;
- *OA* – наречени оперативни (компенсиращи) агенти, предоставят определена специфична условна функционалност;
- С *Activities* означаваме всички възможни действия на агентите;
- $Activities = Fuct \cup Gen$, така че $PA \cap OA = \varnothing$ – различаваме две дизюнктивни групи действия. В първата група се включват дейности, които реализират функционалността на пространството. Във втората група са включени така наречените генератори, с помощта на които динамично могат да се генерират и премахват оперативни агенти;
- $\forall\ oa \in OA\ (\exists\ Act_{oa} \subseteq 2^{Funct})$ - всеки оперативен агент реализира определена специфична условна функционалност, която е

необходима като реакция на идентифицираната промяна в околната среда. Този вид агенти се генерират динамично, при необходимост, и обикновено са с кратък жизнен цикъл;

- С *Events* означаваме всички възможни събития в пространството;

- $Events = Ev(pa_1) \cup Ev(pa_2) \cup ... \cup Ev(pa_k)$, където $Ev(pa_i)$ е множеството на локални събития, случващи се в околната среда (обсега) на персистентния агент pa_i , $i = 1, ..., k$, (k е броят на персистентните агенти);

- $\forall\ pa \in PA\ (\exists\ Act_{pa} \subseteq 2^{Funct} \times \{\ Gen(e_i)\ |\ e_i \in Events\ \}$ - персистентните агенти имат две различни предназначения. Първо, те реализират сравнително стабилна малка функционалност (принадлежаща към минималната), която трябва да се изпълнява безусловно. Второ, използвайки локалния си контрол върху околната среда (обикновено агентите нямат пълен контрол върху околната среда, а само ограничен) те могат да установяват определен вид промени в околната среда. В зависимост от вида промяна те генерират динамично определен вид оперативен агент, който от своя страна извършва необходимото компенсаторно действие. Впоследствие оперативният агент се премахва. Персистентните агенти винаги са налични.

```
repeat
        running(pa1), running(pa2), …, running(pan), pan ∈ PA, i = 1, …, n;
        anytime ∀ pai {
                ∃ ej , ej ∈ Ev(pai) ⇒
                        { GENERATE (pai , oai );
                          REQUEST (pai , oai );
                          if INFORM (oai , pai , 'done' )
                          then REMOVE( pai , oai );
                        }
         }
        endanytime
forever
```

Фиг. 19: Жизнен цикъл на С3А

На Фиг. 19 е даден (като псевдокод) принципен жизнен цикъл на С3А. Представена във времето една контекстно-зависима агентно-ориентирана архитектура изглежда като пулсиращо ядро (реализиращо минималната функционалност на пространството), което периодично се „разширява" в различни посоки (в зависимост от промяната в околната среда) и отново се „свива" до обичайните си размери (минималната функционалност).

4.5 Подход за реализиране на трансформацията

Вземайки предвид направения по-горе анализ на двата типа архитектури са възможни следните подходи за трансформация на jTempura (в AjTempura):

- Всеки отделен клас се трансформира в отделен агент – предимство на този подход е, че агентите ще бъдат със сравнително малка функционалност. Недостатък при трансформацията е игнорирането на връзки между класовете в jTempura, което ще доведе до необходимост от значително усложняване на

комуникацията между агентите в AjTempura (нетрансформираните връзки трябва да се симулират с възможностите на асинхронно взаимодействие между агентите);

- Всички класове се обединяват в един единствен агент – предимство на подхода е, че при трансформацията напълно се съблюдават връзките между класовете в изходната архитектура, което от своя страна значително облекчава комуникацията между агентите в целевата архитектура. Основен недостатък тук е значително нарастналата функционалност на агента;

- Един клас се декомпозира в много агенти – подходът е приложим при класове, предоставящи голяма по обем функционалност. Така, агентите в целевата архитектура ще бъдат със сравнително малка функционалност. Недостатък е, че освен игнориране на връзките между класовете в jTempura, възниква необходимост от допълнителни нови взаимодействия, породени от декомпозирането на класа;

- Избрано множество класове да се групират и трансформират в един агент – този подход предлага компромисен вариант между разгледанатите вече възможности. Съществото тук е намирането на сполучлив избор за групиране на класовете в обектно-ориентираната архитектура.

В дисертацията е приет четвъртият подход. Нашите очаквания са, че той може да бъде ефективно реализиран, при положение, че пакетният модел в Java се използва като изборен механизъм.

Поради липсата на ясно дефиниран модел или шаблон за преминаване от обектно-ориентирана към агентно ориентирана архитектура, приехме подхода класовете с близка функционалност да преминат в един агент. Позовавайки се на теорията за пакетиране на класовете, а именно, че класове с близка функционалност се разполагат в един пакет, то ние можем да приемем следния подход – всеки пакет от нашата архитектура да бъде реализиран като агент с еквивалентна функционалност.

Предлагаме реинженерингов подход, включващ следните две стъпки:

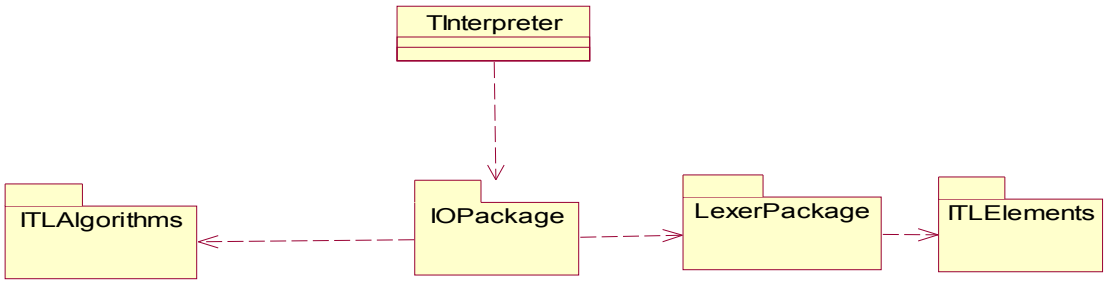
- Трансформация на оригиналната обектно-ориентирана архитектура в архитектура с пакетна организация – в езика за програмиране Java пакетите са вид библиотечни модули, обединяващи множества от класове с подобни задачи;

- Всеки пакет се трансформира във функционалност на съответен агент, т.е. в множество от операции, които може да извършва този агент.

4.6 Трансформационен модел на AjTempura

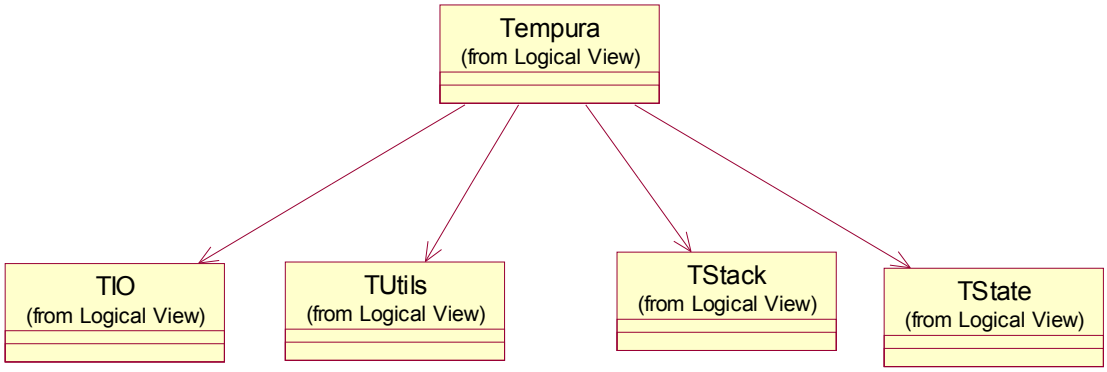
В този раздел представяме модел, наречен *трансформационен*, който служи като технологична рамка за програмната реализация на AjTempura [141]. Съществото за *трансформационния модел* е разделянето на агентите на два типа – *абстрактни* и *реални*. Едно от предимствата на това разделяне е, че впоследствие моделът (абстрактните агенти) може да бъде реализиран в различни развойни среди (в дисертацията е реализиран в JADE). За спецификацията на абстрактните агенти използваме пакетния механизъм на Java. За да подчертаем разликата между абстрактните агенти в трансформационния модел и реалните агенти в реализацията, използваме различни имена. В реализацията е дадена кореспондираща таблица.

При първата стъпка на реинженеринга, обектно-ориентираният код на jTempura може да се групира в няколко основни пакета според функционалните зависимости на класовете. Създадени са четири отделни пакета и самостоятелно е оставен един базов клас (TInterpreter) (Фиг. 20). При интегриране на AjTempura в разпределената агентно-ориентирана архитектура на ВОП, този клас ще бъде заменен с подходящи интерфейсни агенти.



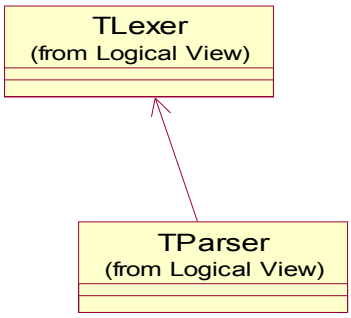
Фиг. 20: jTempura пакети

IOPackage пакетът (Фиг. 21) съдържа пет класа от обектно-ориентираната версия на интерпретатора. Функционалността, концентрирана в тези структури е да обслужват входно-изходните операции. В тези класове е реализиран и контролиран основния жизнен цикъл на интерпретатора, т.е. как протича обработката на една темпорална формула или твърдение. Връзките между класовете в този пакет са от тип „асоциация".



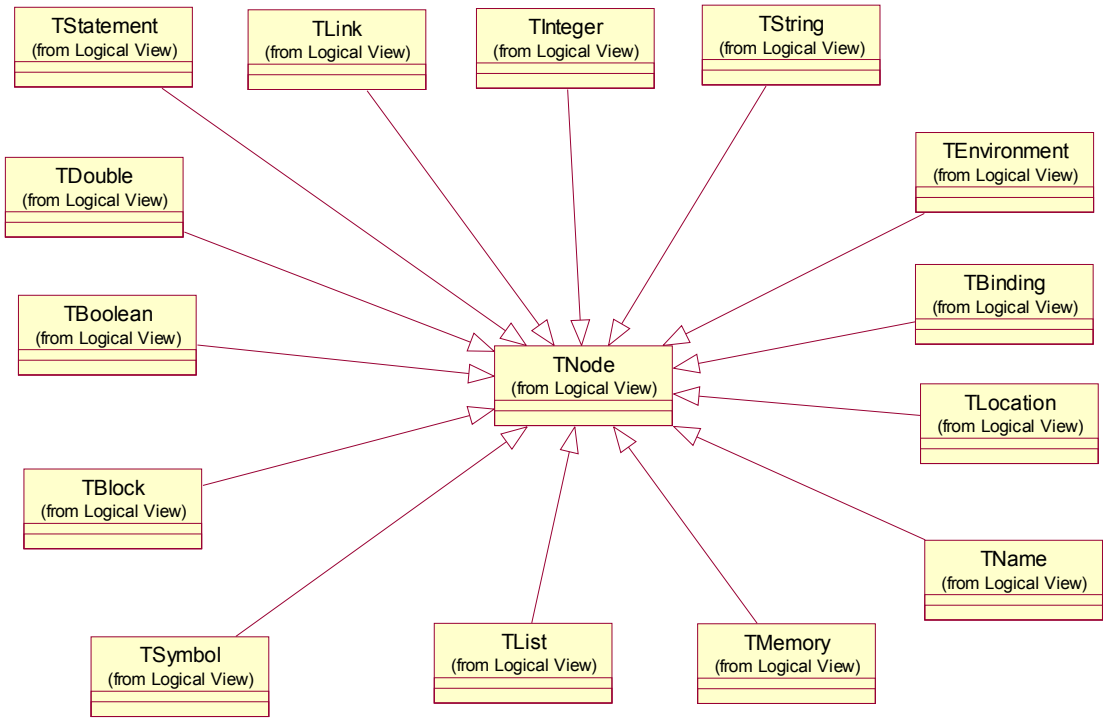
Фиг. 21: IOPackage

В LexerPackage пакета (Фиг. 22) са поместени два класа, реализиращи лексикалния анализ и парсера. Функционалността на тези класове е да извършват анализ на постъпилите твърдения и да бъдат разпознати техните характерни елементи, базирайки се на математическия модел на ITL. Коректното разпознаване на характерните елементи е ключов момент при обработката на ITL твърденията. Логическите връзки между класовете в пакета са от тип „асоциация".



Фиг. 22: LexerPackage

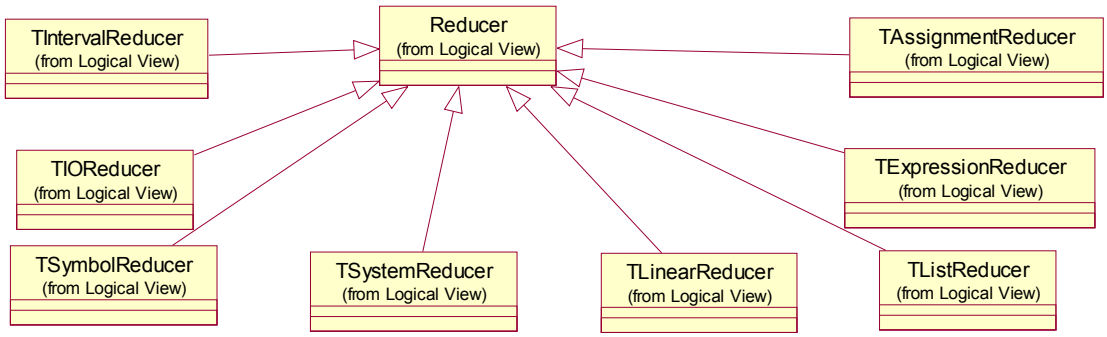
ITLElementsPackage пакетът (Фиг. 23) съдържа само един базов клас (TNode) и четиринадесет класа, реализирани като наследници на базовия. Тези класове описват основни елементи на теоретичния модел на ITL. По време на парсирането те се ползват като шаблони за разпознаване на получените от околната среда (AnaTempura) твърдения или формули. Връзките между класовете в пакета (както се вижда от диаграмата) са от тип „ наследяване”.



Фиг. 23: ITLElementsPackage

ITLAlgorithmsPackage (Фиг. 24) пакетът съдържа базовия клас Reducer и осем негови наследници. Класове реализират алгоритми за редуциране на структури (формули, твърдения, изрази) съгласно теоретичния модел на ITL. Всеки клас описва

отделен метод за обработка на определени типове структури (типовете структури са описани в пакета ITLElementsPackage). Връзките между класовете в този пакета са от тип „наследяване”.



Фиг. 24: ITLAlgorithmsPackage

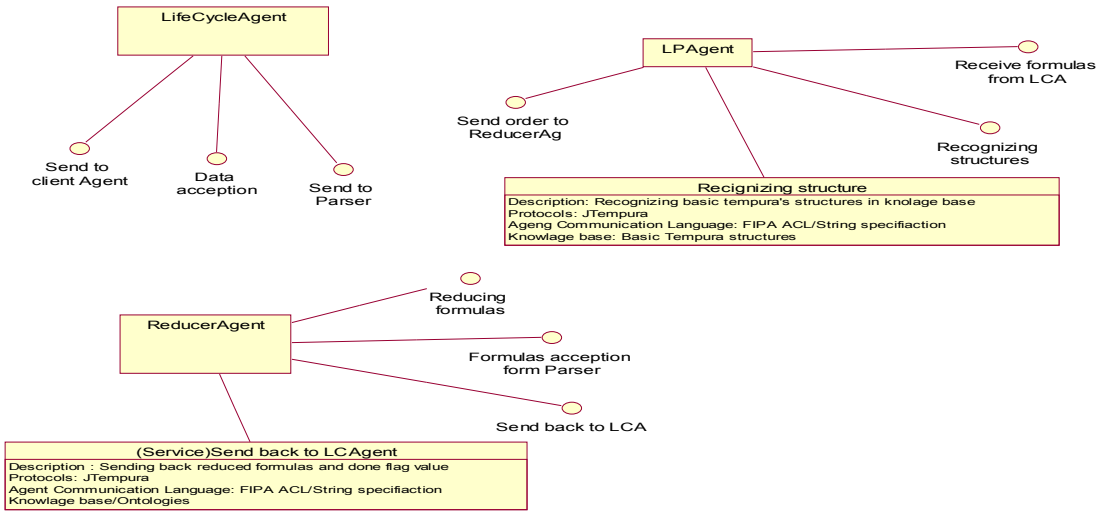
В съответствие с *трансформационния модел* от пакетната структура на jTempura, са дефинирани три абстрактни агенти (Фиг. 25.) LPAgent агентът възниква при трансформацията на два пакета – LexerPackage и ITLElementsPackage. Първият пакет (LexerPackage) доставя възможните операции на агента, които са следните:

- Парсиране на възприетите посредством сензорите структури;
- Разпознаване (по зададени шаблони) типа на структурите;
- Предаване на структурите (посредством ефекторите) за допълнителна обработка.

Елементите на втория пакет (ITLElementsPackage) се трансформират в шаблони, които представляват вградените знания (built-in knowledge), агенти (реализирани като различни ментални свойства). Тези шаблони се използват за разпознаване типовете на обработваните структури.

Вторият агент (ReducerAgent) предоставя функционалността на ITLAlgorithmsPackage пакета, т.е. реализира редуциращите алгоритми върху предадените му от LPAgent структури. Един от вариантите за реазлизация на този агент е всеки един редуциращ алгоритъм, описан в пакета, да бъде реализиран като различна операция (в използваната от нас среда JADE операциите се реализират като поведения на агентите) на ReducerAgent. Така сензорът на агента, при получаване на определена структура, ще активира подходящото поведение.

Трансформаторът IOPackage се преобразува в LifeCycleAgent. Ролята на този агент е да получава от околната среда структури във вид на ITL формули или твърдения (като съобщения от някой от интерфейсните агенти – заместниците на TInterpreter класа) и да ги предава на LPAgenta за обработка. Също така той трябва да получава „редуцирани" структури от ReducerAgent, да определя дали те имат нужда от допълнителна обработка (за да ги подаде отново на LPAgent). След обработване на структурата агентът се използва за връщане на резултата.



Фиг. 25: AjTempura агенти

При реинженеринга на обектно-ориентирана архитектура към агентно-ориентирана трябва да се реши един съществен проблем, възникващ от различния тип взаимодействие между компоненти в двете архитектури. Взаимодействието между класове в обектно-ориентираните архитектури е синхронна – за разлика от агентно-ориентираните архитектури, където е асинхронна. За да запазим оригиналната функционалност на интерпретатора, синхронна комуникация между класовете трябва да бъде симулирана посредством асинхронна размяна на съобщения между агентите в AjTempura.

Един начин за симулация е агентите да бъдат дефинирани с мастър-слейв архитектура, т.е. да различаваме главни и подчинени агенти. По този начин съобщенията от главните към подчинените агенти могат да бъдат специфицирани като перформативи (използваме стандарта за взаимодействие между агентите ACL) от тип „заповед", което ще доведе до поведение, аналогично на синхронен режим на комуникация.

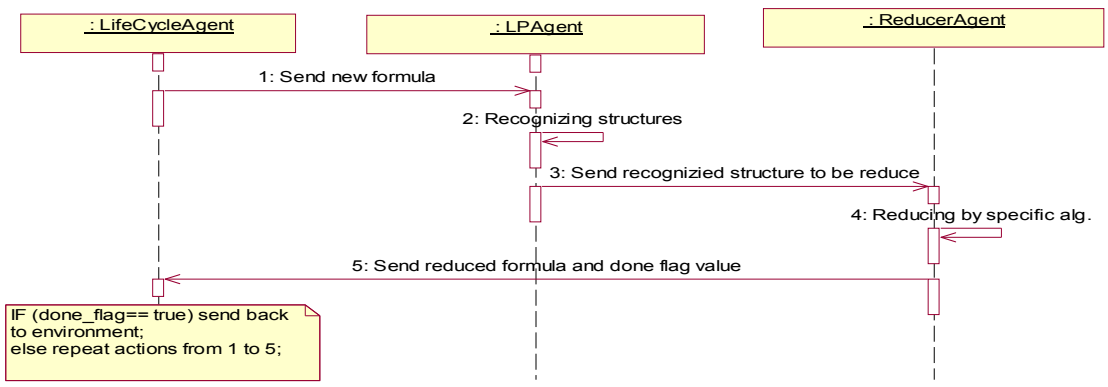
В дисертацията се предлага следният жизнен цикъл Фиг. 26 за обработване на една ITL формула или твърдение в условията на симулирано (посредством асинхронен механизъм) синхронно взаимодействие:

- LifeCycleAgent получава ITL формула от околната среда (някой интерфейсен агент) и изпраща съобщение тип „заповед" със съдържание „*Парсирай тази формула*" към LPAgent;

- LPAgent, специфициран като подчинен спрямо LifeCycleAgent, е длъжен да парсира получената формула в съответствие с built-in знанията си (наличните си шаблони) и да изпрати съобщение тип „заповед" на ReducerAgent (от своя страна той е специфициран като подчинен на LPAgent) със съдържание „*Редуцирай разпознатата от мен структура*";

- ReducerAgent активира подходящо поведение за редуциране на получената структура, като след обработка задава флаг или стойност на променлива, която индикира дали е възможно последващо редуциране. След това изпраща съобщение на LifeCycleAgent, съдържащо редуцираната структура;

- LifeCycleAgent проверява стойността на флага (променливата) за да прецени дали обратботката е приключила и трябва да върне резултата към околната среда или трябва отново да изпрати заповед към ReducerAgent за започване на нов цикъл за разпознаване и последващо редуциране.



Фиг. 26: Жизнен цикъл на обработка на ITL формули

4.7 Реализация на AjTempura

4.7.1 Развойна среда

Представените в предишните точки модел, подход и трансформатори са реализирани с помощта на развойната среда JADE [10,137]. JADE (Java Agent DEvelopment Framework) е софтуерна платформа, разработена от Tilab, изцяло реализирана на езика за програмиране Java. Средата се използва за разработване на разпределени агентно-ориентирани приложения, базирани на комуникационната архитектура от точка до точка (peer-to-peer), като са застъпени следните основни принципи :

- Съвместимост с FIPA (Foundation for Intelligent Physical Agents, стандартизираща организация за агенти и мултиагентни системи, част от IEEE) спецификацията. Така JADE агентите могат да взаимодействат с други агенти, които удовлетворяват стандарта, но не са реализирани с JADE;
- Прост и интуитивен набор от програмни интерфейси;
- Среда за изпълнение, осигуряваща еднакви програмни интерфейси за J2EE, J2SE и J2ME.

Агентната платформа може да се разполага на различни компютри, на които функционират различни операционни системи. Конфигурацията на JADE може да се контролира отдалечено чрез графичен потребителски интерфейс. За стартиране на JADE върху компютър се изисква да има инсталирана Java виртуална машина (JRE).

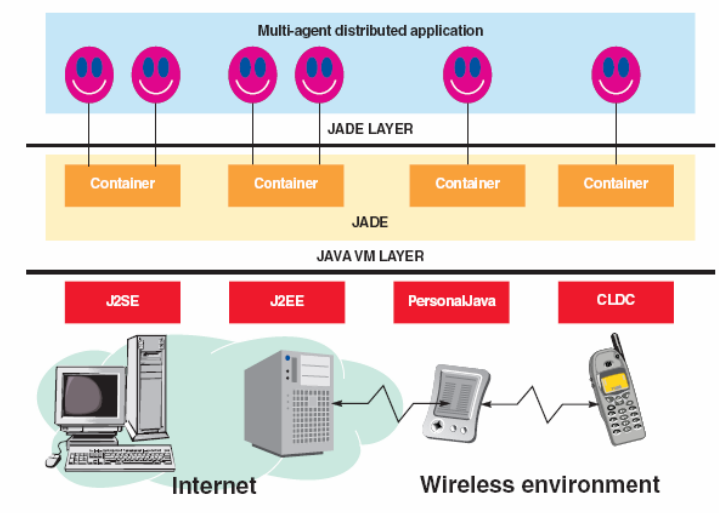
Комуникационната архитектура пренася гъвкаво и ефикасно съобщения, където JADE създава и управлява отделна за всеки агент опашка от пристигащи ACL съобщения. Съответно агентите имат достъп до тази опашка. Изцяло е реализиран комуникационният модел на FIPA и неговите компоненти са ясно представени и напълно интегрирани:

- Протоколи за взаимодействие (interaction protocols),
- ACL,

- Език на съдържанието, схеми, онтологии и транспортни протоколи.

Транспортният механизъм се адаптира към всяка ситуация чрез прозрачен избор на най-добрия за конкретната ситуация разрешен протокол. Използват се протоколи като: RMI (Remote Method Invocation) [87], събитийно известяване и IIOP (Internet Inter-ORB Protocol) [134]. Лесно могат да се добавят и интегрират и други протоколи. Повечето от протоколите за взаимодойствие са дефинирани от FIPA. Те могат да се използват след дефиниране на всяка стъпка от поведението на протокола. Управлението на онтологии чрез агенти има готова реализация. Осигурена е поддръжката на дефинирани от потребителя език на съдържанието и онтологии. Те могат да се регистрират чрез агенти и автоматично да се използват от работната среда.

JADE използва библиотеки, чрез които се разработват програмни агенти и се осигурява среда за изпълнение. Средата за изпълнение се грижи за основните действия и трябва да бъде активна на устройството, преди агентът да се стартира. Всяка инстанция на средата за изпълнение в JADE е наречена контейнер Фиг. 27. Наборът от всички контейнери е наречен платформа и осигурява хомогенен слой, който скрива комплексността и сложността на лежащите под тях слоеве (хардуер, операционна система, вид на мрежата, JVM) от агентите.



Фиг. 27: Архитектура на JADE

От функционална гледна точка JADE предоставя базови услуги, необходими за разпределени peer-to-peer приложения във фиксирани и мобилни среди. Тя позволява

всеки агент динамично да намира други агенти и да комуникира с тях, ръководейки се от peer-to-peer парадигмата. От гледна точка на приложението, всеки агент се идентифицира с уникално име и предоставя набор от услуги. Той може да регистрира и модифицира своите услуги и да комуникира с останалите потребители.

Агентите комуникират чрез размяна на асинхронни съобщения, като комуникационният модел приема двойка хетерогенни обекти (агенти), които не знаят нищо един за друг. В комуникацията един агент просто изпраща съобщение до дестинацията. Агентите се идентифицират чрез име и няма ограничение във времето за комуникация - изпращачът и получателят могат да не бъдат достъпни в едно и също време. Получателят дори може да не съществува или да не бъде директно познат на изпращача. Поради това, че агентите се идентифицират един на друг чрез име, промените в техните обектни връзки са прозрачни за приложението.

Въпреки този тип комуникация, сигурността е контролирана от приложението, което е подало заявка, защото JADE предоставя механизъм за намиране на правилния агент. Когато е необходимо, приложението може да идентифицира изпращача на съобщението и да избегне неразрешени действия (например на агента може да е разрешено да получи съобщение от агент, представляващ шефа, но да не може да изпраща съобщения до него). Структурата на съобщението се придържа към езика ACL, дефиниран от FIPA и включва полета като променливи, които показват:

- Контекста на съобщението;
- За кого е предназначено съобщението;
- Времето за изчакване на отговор на съобщението.

За да улесни създаването на съобщение, JADE автоматично конвертира между възможните формати за обмяна на съдържание, включващи XML [148], RDF [149] и формат, подходящ за манипулация на съдържанието - Java обекти. Тази поддръжка е интегрирана с някои редактори на онтологии като PROTÉGE [110], който позволява на потребителя графично да създава онтологии.

В J2SE и Personal Java средите JADE предоставя мобилност на кода, дори той да е в състояние на изпълнение. Това означава, че агентът може да бъде активен на даден

контейнер, след което може да мигрира на друг, отдалечен контейнер и да рестартира изпълнението си от точката на прекъсване. Тази функционалност позволява управление на натоварването на системата (load balancing) чрез местене на агенти на по-малко натоварени машини, без това да влияе на приложението. Платформата също включва именувани услуги и услуги „жълти страници", които могат да бъдат разпределяни върху няколко хоста. Друго много важно качество на JADE платформата се състои в богатия комплект от графични инструменти, поддържащи отстраняване на грешки, управление и контролиране на фази от жизнения цикъл на приложението. Чрез тези инструменти е възможно отдалечено контролиране на жизнения цикъл на агентите и техните задачи.

Описаните части от функционалността и възможността за отдалечено активиране (от кода и от конзолата) на задачи, "разговори" и нови участници, правят JADE много подходящ за поддържане и изпълнение на интелигентни и проактивни приложения. В контекста на предвидената разработка, това означава, че с избора на JADE като технология се полагат основите за успешна трансформация на функционалността на jTempura и последващата интеграция във ВОП.

Съществен недостатък на базовата версия на JADE е, че не поддържа директно разработване на агенти с BDI архитектури [97]. Създаване на агенти в JADE, поддържащи работа с ментални състояния, е много трудоемка задача. По тази причина се предлага използване на разширена развойна среда, която се състои от два конпонента:

- JADE;
- BDI4JADE.

BDI4JADE [84] е програмна библиотека, с която може да бъде надстроена средата JADE. С помощта на разширената развойна среда в Tempura-агентите ще бъде възможно реализирането на ментални свойства. BDI4JADE поддържа директно управлението и използването на следните три ментални свойства:

- *Beliefs* – представят съществени за работата на агента характеристики на околната среда. Тези характеристики могат да бъдат актуализирани с помощта на възприятията на агента,

получавани от сензорната му система. Принадлежат към информационното състояние на агентите.

- *Desires* – в тези структури се съхранява информация за целите, които възнамеряват да постигнат агентите. Поддържане на целево ориентирано поведение е основа за проактивността на агентите. Принадлежат към мотивационното състояние на агентите.

- *Intentions* – представят актуалния план за действие на агентите. Обхващат съвещателния компонент на агентите.

В момента се провеждат изследвания за търсене на възможности за представяне на други ментални състояния с помощта на BDI4JADE.

4.7.2 Архитектура на AjTempura

Програмната реализация на AjTempura е извършена на основата на *трансформационния модел* и *C3A модела*. AjTempura се състои от следните два основни модула:

- Клиентски модул - означен като bg.uniplovdiv.fmi.delc.aot.client, реализиращ LifeCycleAgent абстрактния агент и базовия клас TInterpreter;

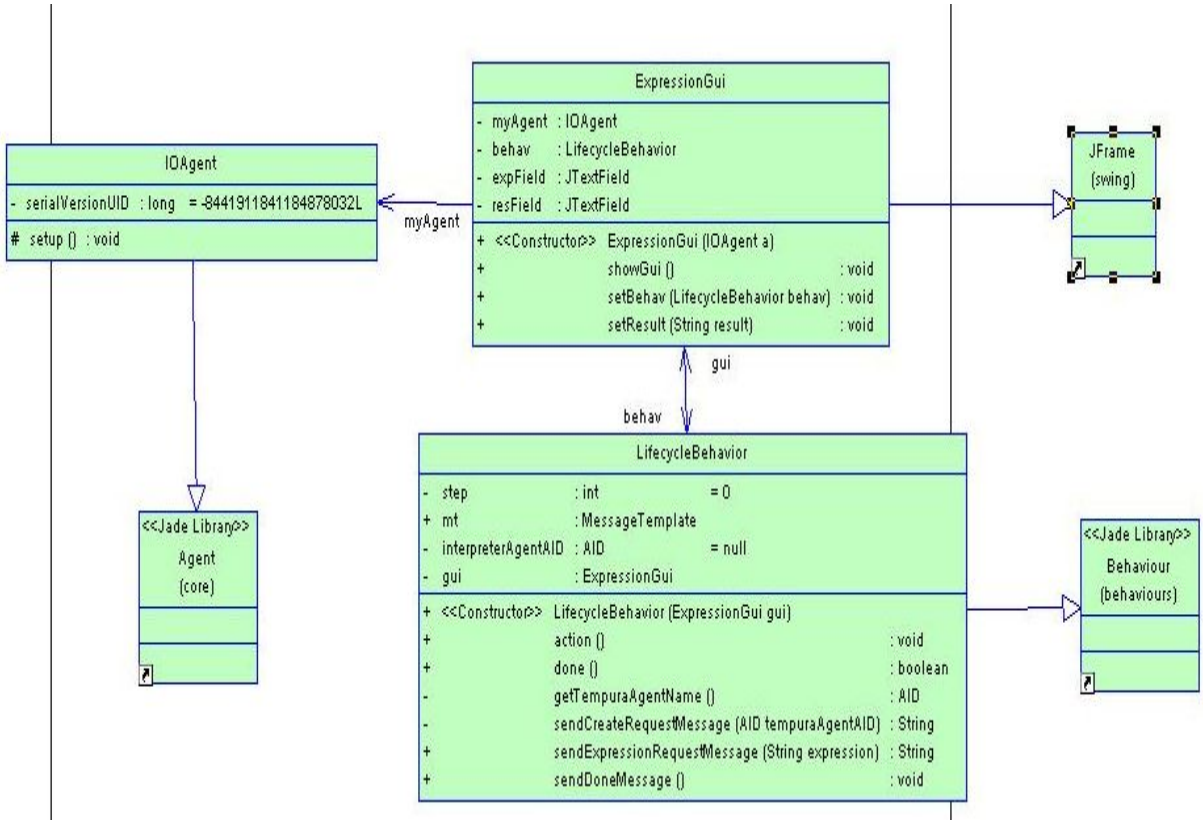
- Сървърен модул - означен като bg.uniplovidv.fmi.delc.aot.server, реализиращ абстрактните агенти LPAgent и ReducerAgent.

В клиентския модул е реализиран агент, който служи като посредник между клиентското устройство и агентите от сървърния модел, реализиращи интерпретиращия механизъм. В съответствие със C3A модела, този агент (наречен *IOAgent*) е специфициран като *персистентен*.

В сървърния модул са реализирани два агента (наречени *TempuraAgent* и *InterpreterAgent*), чиято работа е пряко свързана с обработката на ITL твърдения и формули. В съответствие със C3A модела, *TempuraAgent* е *персистентен*, а *InterpreterAgent* е *оперативен*, т.е. генерира се динамично при необходимост.

Клиентски модул

Агентът *IOAgent* събира данни и конструира ITL твърдение или формула, отразяваща моментното състояние на околната среда (ВОП), която се изпраща за анализ на сървърните агенти. В съответствие с теорията за агентни архитектури и следвайки технологичните особености на JADE, в реалния агент IOAgent се изгражда от два базови Java класа, реализиращи съответно вътрешната архитектура и поведение (функционалността) на агента. С цел улесняване тестването на прототипната версия на AjTempura беше допълнително реализирана тестова среда, симулираща ВОП (като графична среда), в която автоматично се генерират тестови ITL формули и твърдения. Общата архитектура на клиентския модул е представена като UML клас-диаграма на Фиг. 28.



Фиг. 28: Клас диаграма на клиентски модул

Класът, реализиращ агента, IOAgent, наследява класа Agent - предоставен от развойната среда JADE. Логиката на класа е сравнително проста; реално той служи като рамка, в която се изпълнява поведението на агента. Тежестта на комуникацията и вземането на решения е реализирана в класа, реализращ поведението на агента.

Поведението на IOAgent е реализирано в класа LifeCycleBehavior. Този клас е създаден като наследник на класа Behaviour. В JADE се поддържат различни типове поведения. Класът Behaviour е от най-високо абстрактно ниво; той реализира *най-общия тип* поведение, което се изпълнява *еднократно* - за сметка на това предоставя различни механизми за блокиране или рестартиране на поведението. В JADE е възможно специфициране на повече от десет различни типа поведения, като всички те се реализират като класове, наследници на базовия клас Behaviour. Някои от останалите типове се използват в реализацията на сървърните агенти на AjTempura.

Поведението е съставено от няколко базови метода, които се изпълняват в определена последователност, в зависимост от входните данни и комуникацията с другите AjTempura агенти. Основните методи, използвани в поведението на IOAgent са следните:

- getTempuraAgentName - това е метод, чрез който IOAgent се опитва да получи името на агент, към който да се обръща за обработката на ITL формула или твърдение;
- sendCreateRequestMessage - чрез този метод се изпраща съобщение, за генериране на оперативен агент, който да отговаря за обработката на конкретно твърдение/формула;
- sendExpressionRequestMessage - методът изпраща съобщение към вече генериран оперативен агент за обработка на твърдение или фомула, дадени като част от съдържанието на съобщението;
- sendDoneMessage - метод, чрез който се изпраща съобщение за приключена работа. Това предизвиква унищожаването на оперативния агент.

Поведението на агентите в мулти-агенти системи (като напр. AjTempura) зависи пряко от комуникацията с останалите агенти в системата. В съответствие с ACL [43] съществуват различни типове съобщения, дефинирани чрез така наречените *перформативи*. Тъй като това е съществен момент от разработката на системата, по нататък ще бъде отделено специално внимание на съобщенията и комуникацията между агентите.

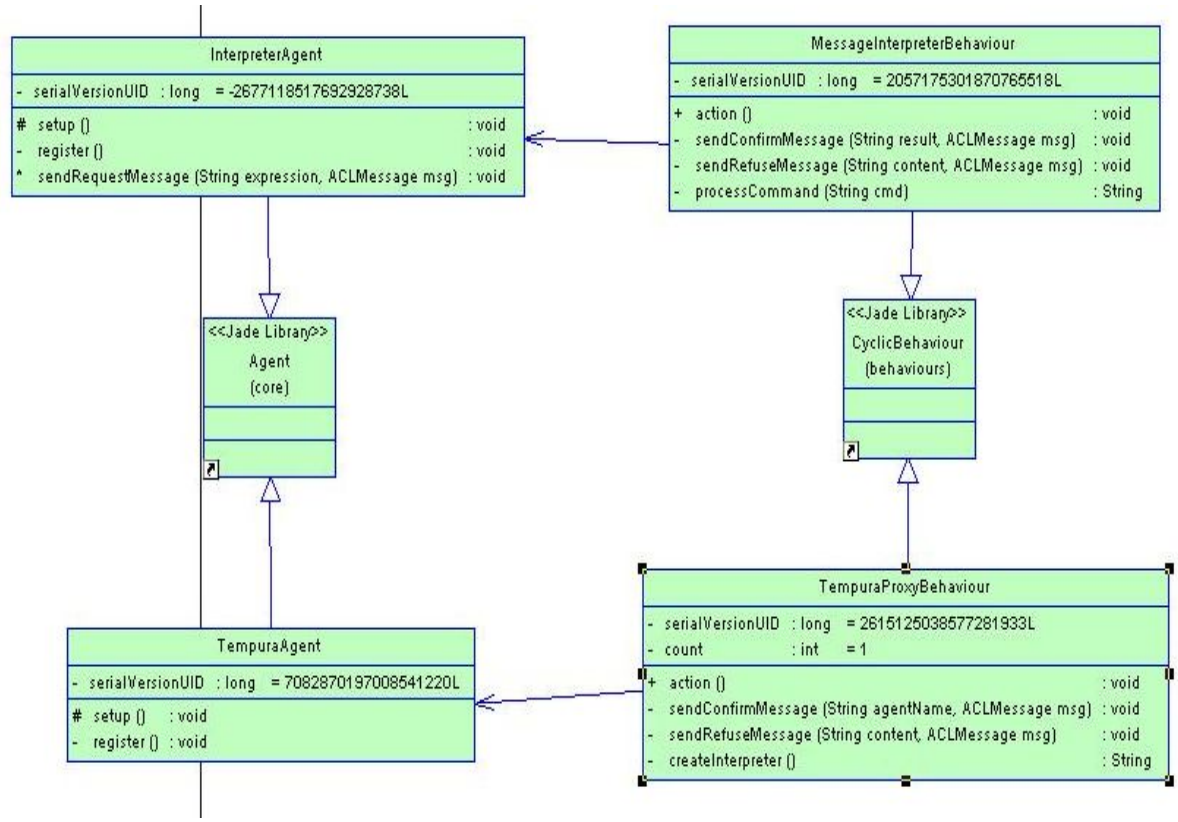
Последният клас от клиентския пакет е *ExpressionGui* - той представлява малка графична форма, която ни дава възможност да проведем тестове със системата преди да бъде интегрирана във ВОП. Формата предоставя текстово поле, където тестващият може да въвежда различни ITL изрази. По този начин можем да използваме оригиналните Tempura Test Suites (вече използвани за тестване на jTempura). Формата предоставя два бутона, като всеки от тях задейства определена част от поведението на агента. Първият бутон изпраща съобщение с ITL израз, който предизвиква изпълнение на метода *sendExpressionRequestMessage*. Вторият бутон служи като изход, т.е. предизвиква обработка на описания по-горе s*endDoneMessage* метод, който от своя страна изпраща съобщение за край на комуникацията.

Друг интересен ефект, реализиран чрез този елементарен графичен интерфейс, е че той реално замества околната среда (ВОП), от която се събират данни и генерират трърденията за интерпретация. От гледна точка на оригиналния интерпретатор, тази графична форма симулира AnaTempura. От гледна точка на бъдещата интеграция на AjTempura, формата временно замества околната среда от агенти, изграждана като ВОП.

Сървърен модул

В сървърния модул са реализирани два агента – като вътрешни архитектури и вградени в тях поведения. Първият агент, наречен *TempuraAgent*, е *персистентен* (в съответствие с C3A модела) и е първата комуникационна цел на *IOAgent*. Задачата на *TempuraAgent* е да отговаря на заявките, получени от IOAgent, като генерира интерпретиращ оперативен агент, който от своя страна да интерпретира ITL израза, представящ актуалното състояние на околната среда. Вторият сървърен агент, наречен *InterpreterAgent*, по своята същност е *оперативен* агент, т.е. генерира се динамично, разширявайки базовата архитектура. Веднъж генериран, този агент отговаря за обработката на конкретно твърдение и след приключване на работата си се самоунищожава. Това е интересна ситуация, произтичаща от C3A модела, където обстоятелствата налагат определен агент да може да се самоунищожи (една от демонстрациите на огромните възможности, предоставяни от C3A модела, за изграждане на гъвкави и мощни софтуерни архитектури).

И двата агента, реализирани в сървърния модул, следват базовата архитектура и предоставените от средата JADE средства и са реализирани чрез наследяване на базови конструкции, като напр. класовете *Agent* и *Behaviour*. Структурата на сървърния модул е онагледена посредством UML клас-диаграмата на Фиг. 29.



Фиг. 29: Клас диаграма на сървърен пакет

Подобно на *IOAgent*, архитектурата на всеки от агентите се състои от два класа. Единият е наследник на базови клас *Agent*, реализиран в библиотеките на JADE. Вторият клас е реализиран като наследник на един от подкласовете на общия клас за поведение *Behaviour*. Типът на поведението на агента е *CyclicBehaviour*. Характерно за този тип поведения е цикличното изпълнение, което означава повторение на едно и също действие, докато агентът не бъде унищожен. Тази логика се налага от спецификата на комуникацията между агентите, която ще бъде коментирана по-подробно впоследствие.

Архитектурата на агентите в този модул не се различава особено от тази на *IOAgent*, с едно малко изключение - тъй като в нашата архитектура *IOAgent* е инициатор на комуникацията, той е агентът, който търси връзка с останалите агенти (за да изпрати израз за интерпретиране) и е по-лесен за идентификация. От своя страна

сървърните агенти са коренно различни един от друг – единият (*TempuraAgent*) е *персистентен* и осъществява началната връзка с *IOAgent*. Другият (*InterpreterAgent*) е *оперативен* агент и отговаря за обработката на конкретен ITL израз. Това налага, при декларацията на агентите, да дефинира типа на агента. Типизирането е стандартна процедура, залегнала в спецификацията на JADE. Типовете служат за разпознаване на логически еднотипни агентни в една мулти-агентна система. Много често в реална ситуация съобщенията не се изпращат до конкретен агент, а до определен тип агенти и първия свободен отговаря на заявката. Реално типът е символен низ, който играе роля на идентификатор. Агентите, които се генерират като инстанции на класа *TempuraAgent* са дефинирани с тип „tempura", а тези, производни от класа *InterpreterAgent* са с тип „tempura-interpreter".

Поведението на *TempuraAgent* е реализирано в клас TempuraProxyBehaviour, който е наследник на CyclicBehaviour, което означава, че това поведение ще се изпълнява многократно. Логиката на поведението се поддържа от следните методи:

- sendConfirmMessage - метод, чрез който *TempuraAgent* изпраща потвърждение към *IOAgent*, че, в отговор на искането за интерпретатор, е генерирал съответния *InterpreterAgent*. В съдържанието на това съобщение стои името на конкретния *InterpreterAgent*, за да знае *IOAgent* къде да бъде изпратен актуалният ITL израз;

- sendRefuseMessage - методът се активира, ако към агента бъде подадено съобщение, което е с грешен перформатив. Както е известно всяко съобщение в JADE се конструира като перформатив (стриктура на езика ACL) и притежава определен семантичен тип;

- createInterpreter - методът се извиква, когато *TempuraAgent* трябва да генерира нов агент от тип *TempuraInterpreter*. Методът не само генерира нов агент, но генерира и уникален идентификатор на агент, отличаващ го от другите инстанции на този тип агент. Този идентификатор се изпраща на IOAgent за комуникация с интерпретиращия агент. Този метод е конструиран така, че чрез

него всеки *TempuraAgent* може да генерира до хиляда копия (напълно достатъчни за нашата цел) на *InterpreterAgent.*

- Класът *MessageInterpreterBehaviour* реализира поведението на *InterpreterAgent* и всички негови копия. Поведението е реализирано отново като наследник на CyclicBehavioura. Базовите функционалности, които изпълнява поведението, са описани чрез методите на класа.

- sendRefuseMessage - методът е аналогичен на този, използван при *TempuraAgent*. При получаване на съобщение с грешен перформатив или такъв, който не се разпознава от сензорите на агента, се изпраща съответното съобщение;

- sendConfirmMessage - този метод се използва, за да се изпрати обратно към *IOAgent* резултата от вече интерпретиран ITL израз. В съдържанието на съобщението влизат, както обработената информация, така също и индикация в отговор на коя заявка е този резултат;

- processCommand - този метод е вътрешен за класа, реализиращ поведението. Той не е свързан с междуагентната комуникация, както другите методи. Неговата задача е да предаде постъпилото за обработка ITL твърдение към класовете, извършващи интерпретацията. Тези класове са пакетирани като библиотека под формата на .jar файл. Те бяха описани по-рано в тази глава, като част от ITLAlgorithms пакета на jTempura;

4.7.3 Между-агентна комуникация и жизнен цикъл на AjTempura

Преди да представим и коментираме основния жизнен цикъл на AjTempura се нуждаем от пояснение на някой основни моменти, свързани с комуникацията между агенти, които сме използвали при нашата реализация.

Основна характеристика на мулти-агентните системи е, че отделните агенти комуникират и си взаимодействат. Това се постига чрез размяна на съобщения, като от

съществено значение е агентите да ползват съобщения с един и същи формат (за да могат да се разбират помежду си). Развойната среда JADE следва стандартите на FIPA и базира между-агентната комуникация на езика ACL (Agent Communication Language). На теория това дава възможност всеки агент, написан с помощта на JADE, да комуникира с агенти, реализирани в други платформи.

Като всеки стандарт ACL има дефиниран формат на съобщенията със задължителни и незадължителни атрибути. Един от задължителните атрибути за ACL съобщенията е да се дефинира перформатив. Перформативите представляват набор от предварително дефинирани типове съобщения – заявка, информиране, предложение и т.н. Перформативите използвани при реализирането на AjTempura с три вида:

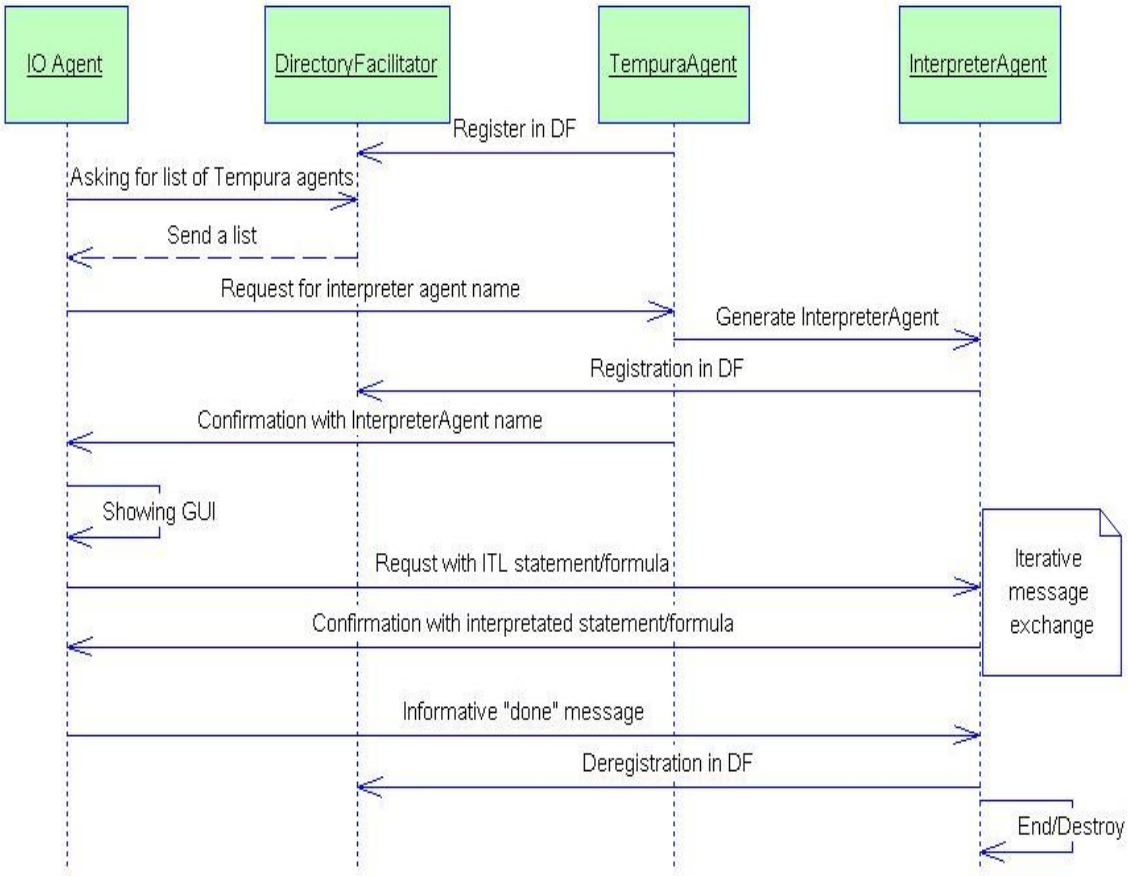
- REQUEST - това е съобщение от тип заявка; съобщенията от този вид са с нисък приоритет и не задължават получателя да ги изпълнява;
- CONFIRM - съобщение от тип потвърждение; обикновено такъв тип съобщение се ползва при отговор на заявка;
- INFORM - тип уведомление; съобщение, което информира за настъпването на дадено събитие.

Освен перформативите, за съобщенията е задължително да се дефинира изпращач (sender) и получател (receiver) на съобщението. Друг задължителен атрибут е идентификатор на разговора; той се генерира при изпращане на всяко съобщение - за да не се наруши последователността на разговора. За да се постигне това, в JADE има реализиран специален клас *MessageTemplate*, чрез който се създават шаблони за идентификатор на разговора. В поведението на нашите агенти също има реализирани такива шаблони, с цел запазване цялостта на разговора.

При стартирането на една мулти-агентна система, базирана на JADE, винаги има набор от служебни агенти, които работят във фонов режим и доставят жизнено важно услуги за работата на системата. Един от тези агенти е *DirectoryFacilitator* или за кратко наричан DF. Той играе ролята на централизиран списък, в които се записват идентификаторите на всички съществуващи (налични) агенти в дадената мулти-агентна система. По този начин, когато се нуждаем от даден агент, имаме възможност да дадем

цялостно или частично описание, като критерий за търсене в DF. Като резултат се връща масив от идентификаторите на всички агенти, отговарящи на описанието. При появата на нов агент или генерирането на оперативен такъв, той първо се регистрира в DF и след това активира своето поведение. Съответно при унищожението на оперативен агент той се дерегистрира от DF.

Имайки предвид тези особености на комуникацията между агенти в една мулти-агентна система и познавайки жизнения цикъл на интерпретатора Tempura (виж глава 2) нека проследим протичането на жизнения цикъл при AjTempura. Последователността от действия и размяна на съобщения може да бъде онагледена с диаграмата на Фиг. 30.



Фиг. 30: Диаграма на жизнен цикъл на AjTempura

Жизненият цикъл е съставен от последователност от действия, които привидно изглеждат повече от колкото при оригиналната версия на интерпретатора. Това се дължи на известен брой служебни операции, налагани от JADE и сложната междуагентна комуникация.

При първоначалното установяване на системата присъства единствено служебният агент *DirectoryFacilitator* (DF), изпълняващ ролята на централизиран списък, където трябва да се регистрират всички новопоявили се в системата агенти. Агентът TempuraAgent се регистрира веднага след стартирането си чрез стъпка „register in DF"; същото се случва и с IOAgent (тези два агента са дефинирани като *персистентни* в нашата архитектура).

При постъпване на израз за интерпретация в IOAgent, той изпраща съобщение с перформатив REQUEST (заяка) до DF. Това съобщение съдържа филтър, който показва, че IOAgent се интересува само от агент от тип „tempura". Причината за това е, че агентите от този тип могат да генерират интерпретиращи агенти. След получаване на съобщението, DF извършва търсене в регистъра по посочения критерий, като връща съобщение с перформатив INFORM и съдържание - масив от всички налични агенти от този тип. След тези две стъпки агентът IOAgent вече има информация за наличност на агент, към който да прати своята заявка. Ако масивът е празен и няма налични агенти, тогава IOAgent продължава, като праща (на всяка секунда) нови съобщения до DF. Този интервал е дефиниран от нас и подлежи на промяна според средата, в която ще се ползва системата.

Когато IOAgent има списък с налични агенти от тип „tempura", тогава той изпраща съобщение с перформатив REQUEST до наличния агент TempuraAgent, като иска да му бъде дадено име на агент от тип "tempura-interpreter", към когото да отнесе за интерпретация ITL израза. От своя страна TempuraAgent създава оперативен агент InterpreterAgent, който е с уникално име (спрямо създадените до момента агенти от същия тип). Едва след създаването на оперативния агент, всичките четири агента от схемата на Фиг. 30, съществуват реално в средата JADE. Новосъздаденият агент автоматично се регистрира в централизирания списък на DF. Когато InterpreterAgent е регистриран, тогава TempuraAgent връща отговор с перформатив CONFIRM на заявката на IOAgent - със съдържание името на интерпретиращия агент, към който да се обърне.

Щом IOAgent научи името на интерпретиращия агент, тогава той изпраща заявка към него, съдържаща ITL израз. В прототипната версия, тук е моментът, в който се визуализира графичния интерфейс (за да подадем тестово твърдение). Когато интерпретиращият агент е готов, той връща съобщение от тим CONFIRM, със съдържание, обработеното ITL твърдение или формула. Последните две стъпки по

изпращане/получаване на израз за обработка може да се повторят неограничен брой пъти, според моментното състояние и изискванията на системата.

Когато IOAgent вземе решение, че няма нужда от обработката на повече твърдения, той изпраща съобщение с перформатив INFORM и специално съдържание, което информира интерпретиращия агент, че няма нужда повече от него. При тази ситуация се активира поведение на InterpreterAgent, което го кара да се дерегистрира от DF и да се самоунищожи.

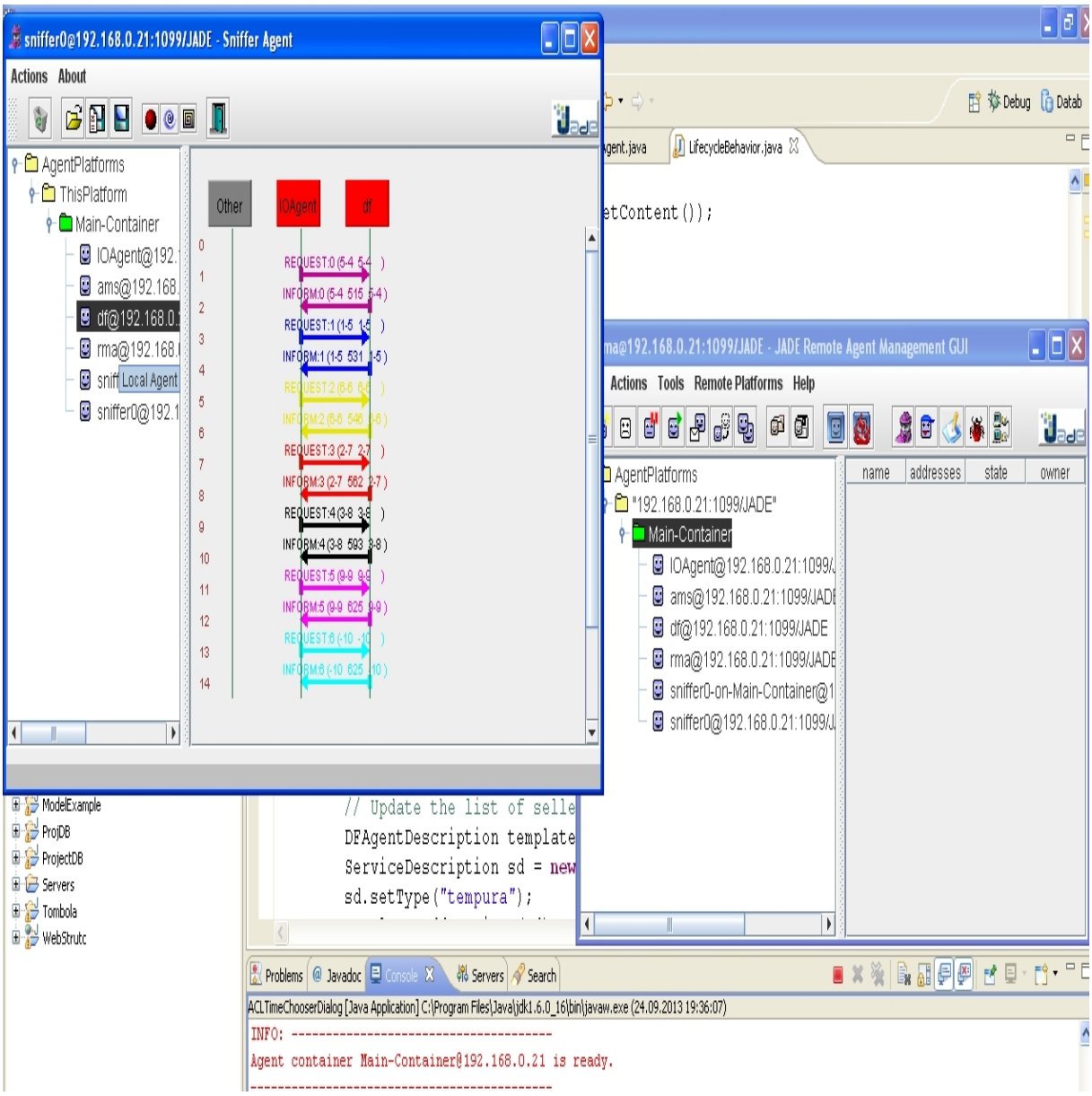
При тази комуникация проличава най-ясно гъвкавостта на контекстно-зависимата архитектура. В зависимост от натовареността на системата тя е в състояние да генерира толкова интерпретиращи агенти, колкото са й необходими. Още повече системата не търпи излишни агенти, а своевременно се освобождава от излишните такива. Персистентните агенти са представители на IOAgent, чийто брой ще зависи от броя на клиентите на системата. Същото важи и за броя на TempuraAgent, но той от своя страна може да генерира до хиляда оперативни интерпретиращи агенти и по този начин, при необходимост и достатъчни ресурси, да поеме огромно натоварване.

4.7.4 Първи стъпки с AjTempura

В тази точка накратко ще бъде демонстрирана практическата работа с първата прототипна версия на AjTempura. Тестовете на системата бяха извършени чрез програмната среда на Eclipse и използването на специалните библитеки на JADE за реализация на мулти-агентни системи.

При стартирането на системата в нея присъстват набор от системни агенти, които реализират базови функционалности на средата JADE и спомагат за комуникацията и анализа на системата. Някои от тях вече бяха споменати по-рано. Освен него в набора със служебни агенти има друг, който е от съществено значение за проследяване на комуникацията при агентите; той се нарича Sniffer. Неговата роля е да предоставя просто визуално средство, чрез което да се онагледи последователната размяна на съобщения между агентите в една система. По своята същност, Sniffer агентът изчертава диаграми, подобни на Sequence диаграми от езика UML. Началното състояние на тази диаграма, при стартиране на AjTempura, може да се види на Фиг. 31. В лявата част на фигурата е даден списък със съществуващите агенти, сред които са един представител на персистентния IOAgent и служебни DF агент. Съгласно жизнения

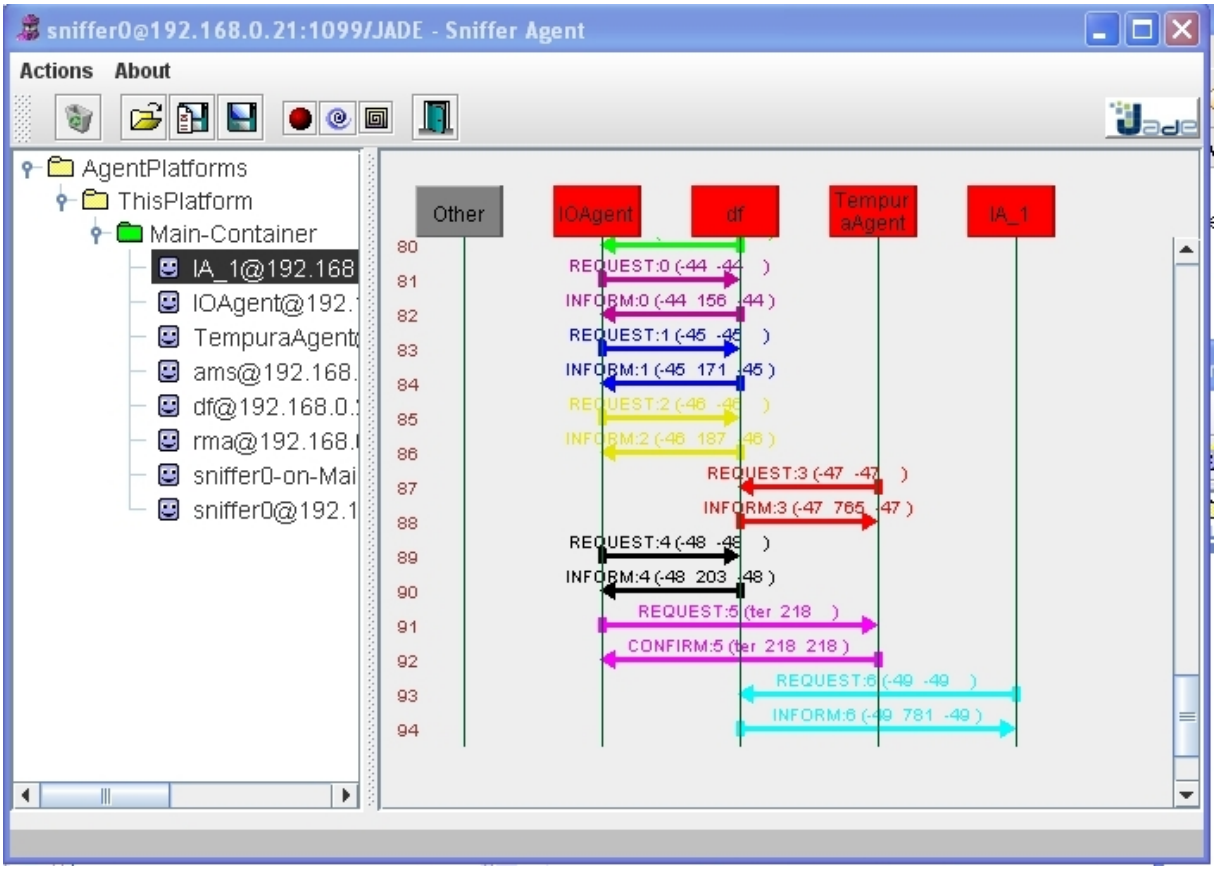
цикъл на AjTempura, IOAgent започва периодично да изпраща запитвания до DF за наличието на агенти от тип TempuraAgent. Съобщенията се изпращат на всяка една секунда до получаване на отговор, които съдържа списък с наличните агенти от търсения тип.



Фиг. 31: Начален изглед от Sniffer агента

При появата на агент от тип TempuraAgent в списъка на DF веднага протича размяна на съобщения между него и IOAgent, която води до генерирането на оперативен интерпретиращ агент. Появата на искания интерпретиращ агент поставя нячалото на размяна на ITL изрази между него и IOAgent до постигане целите на

IOAgent. Комуникацията на пълния набор от агенти може да се види от последващото състояние на диаграмата, генерирана от Sniffer агента (Фиг. 32).



Фиг. 32: Втори изглед от Sniffer агента

Както се вижда от диаграмата, като резултат от появата на TempuraAgent и неговата комуникация с IOAgent, се е появил нов агент IA_1. Този агент е оперативния интерпретиращ агент, които обработва ITL изрази. Неговото име е уникално, всеки последващ такъв ще бъде генериран с пореден номер. Тези агенти съществуват докато IOAgent не им изпрати съобщение със съдържание „done”. Това съобщение индикира, че няма повече ITL изрази за обработка и агента IA_1 е излишен. Това води до неговата дерегистрация и самоунищожението му. На диаграмата на Sniffer агента, това се визуализира чрез маркирането на унищожения агент в жълт цвят и изтриването му от списъка с действащи агенти. Това може да се види от следващата диаграма, генерирана от Sniffer агента Фиг. 33.

Фиг. 33: Трети изглед от Sniffer агента

Заключение – резюме на получените резултати

В дисертацията са изследвани проблеми, свързани с контекснтно-ориентирано управление на електронните услуги, доставяни във виртуално образователно пространство. Особено внимание е обърнато на времевите аспекти на това управление и по-специално на идентифициране и подредба във времето на събития и промени, случващи се в разпределената инфраструктура на пространството. За реализиране на това управление е избран формализъм, базиращ се на интервална модална логика, познат като ITL (Interval Temporal Logics) и поддържащата го софтуерна среда Tempura.

Контекстно-зависима доставка на образователни електронни услуги във ВОП се поддържа от интелигентни агенти, които са в състояние да разпознават и реагират на динамично настъпилите промени, при изпълнение на потребителските заявки и предприема компенсаторни действия, за да подсигури ефективното и безпроблемно изпълнение на заявките. Контекстът може да бъде от различен тип, напр. локация, идентичност или активност.

Като пример за използване на AjTempura ще разгледаме контекстно-зависимо доставяне на образователни услуги във ВОП. По отношение на мобилността, следните основни типове промени могат да възникнат в средата:

- Локация на мобилното устройство (мобилност на устройството) – в някои случаи тази мобилност води до промяна на обслужващия InfoStation. Това е особено важно от гледна точка на движението на потребителите (напр. движението на студенти в университестския кампус) и промяната на тяхната локация в системата. Това движение има отношение към локалното разположение на услуги в различните възли на системата.

- Потребителско устройство – тази мобилност има отношение към начина, по който трябва да бъде предоставено съдържанието, като отговор на определена потребителска заявка. Важното при този аспект е да може да се разпознае типа на устройството и според неговите технически възможности да бъде предоставено подходящо по формат електронно съдържание (адаптивност);

- Тип на комуникацията – в зависимост от местоположението и различни ограничения на средата потребителят може да използва различни комуникационни протоколи (WiFi, Bluetooth и др.);

- Потребителски профили – персонализирането на услугите трябва да бъде извършвано в съответствие с промяната в потребителските профили;

Целта на адаптивността е да се гарантира безпроблемно, прозрачно и адекватно изпълнение на потребителските заявки за услуги, като се вземат предвид различните аспекти на контекста споменати по-горе. С други думи, след определяне на конкретна промяна в околната среда на услугата, мидълуерът трябва да може да предпрееме компенсаторни действия (мерки за противодействие), като например препредаването на потребителски сесии на услуги от един InfoStation към друг, преформатиране на съдържанието на услугата, според промяна на мобилното устройство (според специфичните технически възможности), персонализация на услугата според потребителския профил и т.н.

За да осигурим адекватна поддръжка от гледната точка на мобилността на потребителите и устройствата им, бяха дефинирани следните четири базови сценария:

- „Без промяна” – поисканата електронна услуга се изпълнява в обхвата на един InfoStation и без промяна на потребителското мобилно устройство;

- „Промяна на потребителското устройство” – по време на изпълнение на дадена услуга, потребителят сменя първоначално използваното мобилно устройство напр. преминава към устройство с по-големи технически възможности;

- „Промяна на InfoStation” – в рамките на InfoStation парадигмата, връзката между InfoStations и потребителските устройства е географски неограничен (в рамките на InfoStation мрежата). Това позволява на потребителя да се движи из мрежата, като по време на изпълнението на дадена заявка многократно излиза от обхванат на един InfoStation и влиза в друг. Тази промяна на InfoStations трябва да бъде абсолютно прозрачна за потребителите, без те да губят връзка със системата и предоставяните услуги;

- „Промяна на мобилното устройство и InfoStation” – този сценарий е най-сложния и е комбинация от едновременното настъпване на по-горните два сценария.

Използвайки AjTempura ние можем да дефинираме едно множество от променливи, които да служат като идентификатори на мобилни устройстава(idMD) и InfoStation (idIS) във ВОП. В определен времеви момент комбинацията от тези променливи може да специфицира къде се намира потребителя (в рамките на нашата комуникационна мрежа) и какво мобилно устройство използва. Проследявайки настъпващите промени в стойностите на набора от идентификационни променливи, ние ще сме в състояние да разберем кой от базовите сценарии се изпълнява.

В съответствие с архитектурата на конкретна InfoStation мрежа ние можем да дефинираме ITL твърдение, съставено от множество променливи от тип idMD и idIS, което да описва „поведението” на потребителя (напр. тип използвано устройство и локация според използвания InfoStation). По време на изпълнение на потребителските заявки, ние ще използваме подходяща версия на Tempura (за ВОП това ще е AjTempura), която да следи промяната в стойностите на идентифициращите променливи и да разпознава настъпилите сценарии.

Приносите на автора, представени в дисертационния труд, са научно-приложни и приложни.

Научно-приложните могат да бъдат обобщени както следва:

1) Предложен е подход за реинженеринг на оригиналния интерпретатор на Tempura;

2) Разработен е теоретичен модел, наречен C3A, за контекстно-зависими софтуерни архитектури, който се използва за реализиране на контекстно-зависима мулти-агентна версия на Tempura интерпретатора, наречен AjTempura. В C3A се различават два типа агенти – *персистентни* и *оперативни*. Персистентните агенти предоставят минималната функционалност на едно приложение, която включва също възможности за идентифициране на случващи се промени в околната среда. В

съответствие с класическата теория за агенти, тези агенти са винаги налични. В отклонение от класическата теория, оперативните агенти не са винаги налични – в зависимост от вида на идентифицираните промени в околната среда (в нашия случай ВОП) те се генерират динамично (от кореспондиращи персистентни агенти). Тяхната задача е да извършват компенсиращи действия, чиято необходимост е породена от случващите се промени. След окомплектоване на съответните компенсации, тези агенти се унищожават (или самоунищожават). Моделът е независим от конкретната приложна област и може да има различни приложения;

3) За реализиране на AjTempura е предложен също втори модел, наречен *трансформационен*, който представя възможен начин за реинженеринг на обектно-ориентираната версия jTempura в AjTempura. Съществото за трансформационния модел е използването на пакетния модел на Java за спецификация на *абстрактни агенти*. Едно от предимствата на това разделяне – абстрактни и реални агенти - е, че впоследствие моделът (абстрактните агенти) може да бъде реализиран в различни развойни среди (в дисертацията са реализирани в JADE).

Приложните приноси на автора са следните:

4) Разработена е нова обектно-ориентирана версия на интерпретатора, наречена jTempura. В съответствие с реинженерингов подход, jTempura е междинна стъпка към крайната цел – създаване на AjTempura. jTempura може да се използва също като самостоятелен софтуерен продукт и да се вгражда в обектно-ориентирани архитектури, използващи JVM;

5) Реализиран е контекстно-зависим мулти-агентен интерпретатор (AjTempura). AjTempura се интегрира в изграждащото се виртуално образователно пространство (ВОП) за усилване неговата контекстно-зависимост;

6) Създадена е програмна реализация на С3А модела, вградена в AjTempura. Първите тестове на AjTempura демонстрират големия потенциал на С3А модела за създаване на гъвкави и мощни софтуерни архитектури.

Междинни резултати и отделни части от работата, освен в цитираните публикации, са представени и в отчетите на международни и национални научно-изследователски проекти, в които е участвал авторът (виж Приложения).

За бъдещото развитие на получените в дисертацията резултати предвиждаме следните насоки:

- *jTempura* – усъвършенстване на обектно-ориентираната структура и стандартизиране на интерфейсните класове, с цел по-лесно интегриране в различни обектно-ориентирани приложения;

- *AjTempura* – продължава работата по интегриране във ВОП. Разработване на различни типове интерфейсни агенти, които напълно да заменят AnaTempura и да разширят възможностите за интеграция на интерпретатора;

- *Усъвършенстване на двата модела* - провеждане на експерименти, анализ и оценка на резултатите, които ще се използват за усъвършенстване на СЗА модела и трансформационния модел;

- *Формални средства за моделиране на ВОП* – за моделиране и изследване на виртуалното пространство предвиждаме изграждане на моделираща среда, в която ще бъде възможно комбиниране на AjTempura с други формални средства, като напр. CCA [132] и CS-Flow. Така ще могат да се изследват различни характеристики на пространството.

Декларация за оригиналност на резултатите

Декларирам, че настоящата дисертация съдържа оригинални резултати получени при проведени от мен научни изследвания (с подкрепата на научния ми ръководител). Резултатите, които са получени, описани и/или публикувани от други автори/учени са надлежно цитирани в библиографията.

Настоящата дисертация не е прилагана за придобиване на научна степен в друго висше училище, университет или научен институт.

Подпис:

Вл.Вълканов

1. Цитирани публикации на автора

1) Stoyanov, S., I. Popchev, E. Doychev, D. Mitev, V. Valkanov, A. Stoyanova-Doycheva, V. Valkanova , I. Minov. DeLC Educational Portal. *// Cybernetics and Information Technologies (CIT),* vol. 10, No 3, 2010, pp. 49-69. <http://www.cit.iit.bas.bg/CIT_2010/CIT_10-3.html>
2) Valkanov, V., D. Mitev. Tempura Reengineering. // *REMIA 2010* : *Anniversary International Conference,* 2010, pp. 311-320.
3) Stoyanov, Stanimir, Ivan Ganchev, Damyan Mitev, Vladimir Valkanov, Martin O`Droma. Service-oriented and Agent-based Architecture Supporting Adaptable, Scenario-based and Context-aware Provision of Mobile E-learning Services. //*IJCISIM,* vol. 3, 2011, pp. 771-779. ISSN 2150-7988
4) Stoyanov, S., H. Zedan, E. Doychev, V. Valkanov, I. Popchev, G. Cholakov, and M. Sandalski. Intelligent Distributed eLearning Architecture. // *Intelligent Systems, InTech March*, *V.M.Koleshko (Ed.)*, Hand Cover, 2012, 366 pages, pp. 185-218, ISBN 978-953-51-0054-6
5) S. Stoyanov, E. Doychev, A. Stoyanova-Doycheva, V. Valkanova, V. Valkanov, Education Cluster Supporting eTesting and eLearning in Software Engineering, 2nd Annual International Conference on Web Technologies & Internet Applications (WebTech 2012), 7- 8 May 2012, Bali, Indonesia, 23-28.

6) Владимир Вълканов, „Темпура реинжинеринг", Докторантски семинар по проект "Изграждане на висококвалифицирани млади изследователи по съвременни информационни технологии за оптимизация, разпознаване на образи и подпомагане вземането на решения" - BG051PO001-3.3.04/40 , 16 февруари 2010, ИИТ-БАН, София;

7) Владимир Вълканов, „Context-aware management of e-services (Tempura reengineering)", 13th Workshop on "Software Engineering, Education and Reverse Engineering", DAAD, 24th August – 31th August 2013, Банско, България

2. Пълен списък с публикации на автора

1) Mitev, D., I. Minov, V. Valkanov, T.Velev, M. Kalinov, G.Kraev. Multi-agent architecture for InfoStation infrastructure. // *Creativity and Innovation in Software Engineering (CISE'09 in association with IEEE and IFIP) : The Second International Conference,* Ravda (Nessebar), Bulgaria. [CD-ROM], 2009.
2) Mitev, D., V. Valkanov. Tempura Reengineering. *// Creativity and Innovation in Software Engineering (CISE'09 in association with IEEE and IFIP) : The Second International Conferenc,* Ravda (Nessebar), Bulgaria : [CD-ROM], 2009.
3) Stoyanov, S., I. Ganchev, V.Valkanov, A. Murdjeva. Distributed Agent-Oriented Development Environment for BECC catalogue. //*Creativity and Innovation in Software Engineering (CISE'09 in association with IEEE and IFIP) : The Second International Conference*, Ravda (Nessebar), Bulgaria : [CD-ROM], 2009.
4) Valkanov, V., G.Kraev, Deploying Web Services in an InfoStation system. // *Creativity and Innovation in Software Engineering (CISE'09 in association with IEEE and IFIP) : The Second International Conference*, Ravda (Nessebar), Bulgaria : [CD-ROM], 2009.
5) Stoyanov, S., I. Popchev, E. Doychev, D. Mitev, V. Valkanov, A. Stoyanova-Doycheva, V. Valkanova , I. Minov. DeLC Educational Portal. *// Cybernetics and Information Technologies (CIT),* vol. 10, No 3, 2010, pp. 49-69. <http://www.cit.iit.bas.bg/CIT_2010/CIT_10-3.html>
6) Valkanov, V., D. Mitev. Tempura Reengineering. // *REMIA 2010* : *Anniversary International Conference,* 2010, pp. 311-320.
7) Stoyanov, Stanimir, Ivan Ganchev, Damyan Mitev, Vladimir Valkanov, Martin O`Droma. Service-oriented and Agent-based Architecture Supporting Adaptable, Scenario-based and Context-aware Provision of Mobile E-learning Services. //*IJCISIM,* vol. 3, 2011, pp. 771-779. ISSN 2150-7988
8) Stoyanov, S., H. Zedan, E. Doychev, V. Valkanov, I. Popchev, G. Cholakov, and M. Sandalski. Intelligent Distributed eLearning Architecture. // *Intelligent Systems, InTech March*, *V.M.Koleshko (Ed.)*, Hand Cover, 2012, 366 pages, pp. 185-218, ISBN 978-953-51-0054-6
9) S. Stoyanov, E. Doychev, A. Stoyanova-Doycheva, V. Valkanova, V. Valkanov, Education Cluster Supporting eTesting and eLearning in Software

Engineering, 2nd Annual International Conference on Web Technologies & Internet Applications (WebTech 2012), 7- 8 May 2012, Bali, Indonesia, 23-28.

3. Пълен списък с доклади от работни срещи

1) Дамян Митев, Владимир Вълканов, Interval Temporal Logic, SEDILia Kick-off Meeting , 1-3 април 2009, Боровец;

2) Владимир Вълканов, Дамян Митев, Tempura Interpreter, SEDILia Kick-off Meeting, 1-3 април 2009, Боровец;

3) Владимир Вълканов, Дамян Митев, Tempura reengineering, SEDILia Kick-off Meeting, 1-3 април 2009, Боровец;

4) Дамян Митев, Владимир Вълканов, Tempura retargeting, 9th Workshop "Software Engineering Education and Reverse Engineering", по проект JCSE – DAAD, 31 август – 5 септември 2009г., Неум, Босна и Херцеговина;

5) Владимир Вълканов, Дамян Митев, Runtime verification of Java programs using ITL, 9th Workshop "Software Engineering Education and Reverse Engineering", по проект JCSE – DAAD, 31 август – 5 септември 2009г., Неум, Босна и Херцеговина,;

6) Владимир Вълканов, „Темпура реинжинеринг", Докторантски семинар по проект "Изграждане на висококвалифицирани млади изследователи по съвременни информационни технологии за оптимизация, разпознаване на образи и подпомагане вземането на решения" - BG051PO001-3.3.04/40 , 16 февруари 2010, ИИТ-БАН, София;

7) Владимир Вълканов, Дамян Митев, „JTempura", работна среща по проект JCSE, DAAD, 6-12 септември 2010, Сърбия, Иваница;

8) Владимир Вълканов, Минко Сандалски, „Event management in DeLC", работна среща по проект JCSE, DAAD, 22-27 август 2011, Охрид, Македония;

9) Емил Дойчев, Владимир Вълканов, „MSc Education Supporting Infrastructure", 13th Workshop on "Software Engineering, Education and Reverse Engineering", DAAD, 24th August – 31th August 2013, Банско, България

10) Владимир Вълканов, „Context-aware management of e-services (Tempura reengineering)", 13th Workshop on "Software Engineering, Education and Reverse Engineering", DAAD, 24th August – 31th August 2013, Банско, България

4. Списък с цитирания

S.Stoyanov, I. Ganchev, D. Mitev, V. Valkanov, and M. O'Droma, "Service-oriented and Agent-based Architecture Supporting Adaptable, Scenario-based and Context-aware Provision of Mobile e-Learning Services," International Journal of Computer Information Systems and Industrial Management Applications, vol. 3, 2011, USA, pp. 771-779

Цитирана брой пъти: 4
Цитирана в:

- А. Стоянова-Дойчева, Дефиниране на процес и средства за рефакторинг в обучението по софтуерни технологии, Пловдивски университет, 2011, дисертация
- А. Стоянова-Дойчева, Дефиниране на процес и средства за рефакторинг в обучението по софтуерни технологии, Пловдивски университет, 2011, автореферат
- Е.Дойчев, **СРЕДА ЗА ЕЛЕКТРОННИ ОБРАЗОВАТЕЛНИ УСЛУГИ,** Пловдивски университет, 2013, дисертация
- Е.Дойчев, **СРЕДА ЗА ЕЛЕКТРОННИ ОБРАЗОВАТЕЛНИ УСЛУГИ,** Пловдивски университет, 2013, автореферат

Stoyanov, S.; Popchev, I.; Doychev, E.; Mitev, D.; Valkanov, V.; Stoyanova-Doycheva, A.; Valkanova, V.; Minov, I., "DeLC Educational Portal," Cybernetics and Information Technologies (CIT), vol. 10, no. 3, pp. 49-69, 2010

Цитирана брой пъти: 6
Цитирана в:

- А. Стоянова-Дойчева, Дефиниране на процес и средства за рефакторинг в обучението по софтуерни технологии, Пловдивски университет, 2011, дисертация
- А. Стоянова-Дойчева, Дефиниране на процес и средства за рефакторинг в обучението по софтуерни технологии, Пловдивски университет, 2011, автореферат

- Даниела Орозова, Обобщеномрежови модели на интелигентни среди за обучение, Академично издателство „Проф. Марин Дринов“, София, 2011, ISBN 978-954-322-481-4

- Даниела Орозова, Анализ и проектиране на бази от данни и знания, Бургаски свободен университет , Бургас, 2011, ISBN 978-954-9370-86-7

- Е. Дойчев, Среда за електронни образователни услуги, Пловдивски университет, 2013, дисертация

- Е. Дойчев, Среда за електронни образователни услуги, Пловдивски университет, 2013, автореферат

Stoyanov, S.; H. Zedan, E. Doychev, V. Valkanov, I. Popchev, G. Cholakov and M. Sandalski, *Intelligent Distributed eLearning Architecture*, V. M. Koleshko (Ed.), Intelligent Systems, InTech, March, 2012, ISBN: 978-953-51-0054-6, Hard cover, 366 pages, pp. 185-218.

Цитирана брой пъти: 2
Цитирана в:

- Г. Чолаков, Хибридна архитектура за изграждане на Разпределен център за електронно обучение (DeLC), Пловдивски университет, 2013, дисертация

- Г. Чолаков, Хибридна архитектура за изграждане на Разпределен център за електронно обучение (DeLC), Пловдивски университет, 2013, автореферат

5. Списък на проекти с участие на автора

1) “Изграждане на висококвалифицирани млади изследователи по съвременни информационни технологии за оптимизация, разпознаване на образи и подпомагане вземането на решения” - BG051PO001-3.3.04/40
2) Дигитална библиотека „софтуерни инжинерство” (SEDILIA)– DVU01-0414, финансиран от Фонд Научни Изследвания.
3) „Интелигентен Образователен Портал” – ИО-0101, финансиран от Фонд Научни Изследвания.
4) „Разработка и приложение на иновативни ИКТ за провеждане на качествени конкурентноспособни научни изследвания и цялостно осъвременяване процеса на обучение във ФМИ” - НИ11-ФМИ-004 към НПД на ПУ
5) Международен проект Joint Course in Software Engineering (JCSE), Координатор Хумболдтовия университет в Берлин, финансиран от DAAD, Германия;

Библиография

[1] Allemang, D. и J. Hendler, *Semantic Web for the Working Ontologist*.: Elsevier, 2011, ISBN 978-0-12-385965-5.

[2] Alotaibi, H M, *Ph.D. Thesis: Context-aware and Secure Workflow Systems*. Leicester, UK: STRL, De Montfort University, 2012.

[3] Al-Sammarraie, M H, *Ph.D. Thesis: Policy-based Approach For Context-aware Systems*. Leicester, UK: STRL, De Montfort University, 2011.

[4] Aroyo, и de Bra Lora and Paul. (2013, June) Agent-oriented Architecture for Task-based Information Search System. <http://wwwis.win.tue.nl/~debra/infwet99/aroyo.html>

[5] Ashton, K. , "That 'Internet of Things' Thing", *RFID Journal*, 2009.

[6] Bakardjieva, T , k Kosev и P Porayzov, "Course Development Module in Open Source eLearning platform ESCHOOL" в *Proc. Of the International Conference Informatics in the Scientific Knowledge*, Varna, 2008, pp. 315-328.

[7] Balazinska, , Merlo, Dagenais, Lague и Kontogiannis, "Advanced clone-analysis to support object-oriented system refactoring" в *Conf. Reverse Engineering, IEEE Computer Society*, 2000, pp. 98-107.

[8] Baldauf, M , S Dustdar и F Rosenberg, *A survey on context-aware systems*.: Int. J. Ad Hoc Ubiquitous Computing, 2007.

[9] Banerjee, J. и W. Kim, "Semantics and implementation of schema evolution in object-oriented databases" в *SIGMOD Conf, ACM*, 1987.

[10] Bellifemine, F. L., G. Caire и D. Greenwood, *Developing Multi-Agent Systems with JADE*.: Wiley, 2007.

[11] Berners-Lee, T. , J. Handler и O. Lassila, "The Semantic Web", *Scientific American* , no. 284, pp. 34-43, May 2001.

[12] Bolchini, C , C A Curino, E Quintarelli, F A Schreiber и L Tanca, *A data-oriented of context models*.: SIGMOD, 2007.

[13] Bonchev, B. и A. Aleksieva-Petrova, "The ARCADE e-learning platform according the IEEE LTSA standard" в *Proc. of 2nd Int. Conf. Computer Science*, ISBN 954 438 526 6 , 2005, pp. 283-287.

[14] Bonchev, B и T. Iliev, "ARCADE – Web-based Authoring and Delivery Platform for Distance Education" в *Proceedings of the 1st Balkan Conference in Informatics*, Thessaloniki, Greece, 2003, pp. 293-306.

[15] Bonchev, B. и D. Vassileva, "Rule-driven approach for adaptable e-learning content delivery" в *Proc. of 7th Int. Conf. on Education and Inf. Systems, Technologies and Applications (EISTA 2009)*, ISBN-10: 1-934272-74-4 , 2009, pp. 143-148.

[16] Bonchev, B. , D. Vassileva, B. Chavkova и V. Mitev, "Architectural Design of a Software Engine for Adaptation Control for the ADOPTA E-learning Platform" в *Proc. of International Conference on Computer Systems and Technologies (CompSysTech' 09)*, Ruse, Bulgaria, ISSN: 1313-8936 , 2009, pp. 263-268.

[17] Bothe, K. , "Software Engineering", Humboldt Universitaet, Berlin, Lecture Notes 2011.

[18] Bottoni, P. , F. Parisi-Presicce и G. Taentzer, "Coordinated distributed diagram transformation for software evolution", *Electronic Notes in Theoretical Computer Science*, vol. vo.72, no. 4, 2002.

[19] Boychev, P. , B. Bonchev, R. Nikolov, M. Olivera и I. Koychev, "Designing a Toolset for the PRIME Virtual Business Environment" в *Proc. of 10th Int. IFIP Workshop on Experimental Interactive Learning in Industrial Management*, Trondheim, Norway, ISBN 82-7706-224-9 , 2006, pp. 41-49.

[20] Brown, P. J., "The stick-e document: a framework for creating context-aware applications", 1996.

[21] Brown, P K, "Triggering information by context", *Personal and Ubiquitous Computing*, 1998.

[22] Burkhard, H-D. , *Moderne Methoden der KI*. Berlin: Humboldt Universitaet, 2007.

[23] Burstall, R. , "Program proving as hand simulation with a little induction" в *Proceedings of IFIP Congress 74*, Valbonne,France, 1974, pp. 308-312.

[24] Cardekku, K и A Gordon, "Mobile ambients", *Theoretical Computer Science*, pp. 177-213, 2000.

[25] Casais, E. , "Automatic reorganization of object-oriented hierarchies: a case study", *Object Oriented Systems*, vol. vol. 1, pp. 95-115, 1994.

[26] Chandra, A. , J Halpern и A. Meyer, "Equation between regular terms and an application to process logic" в *Thirteenth Annual ACM Symposium on Theory of computing* , Milwaukee, Wisconsin,May, 1981, pp. 384-390.

[27] Chen, H , T Finin и A Joshi, *Using owl in a pervasive computing broker*., 2003.

[28] Chikofsky, E. J. и J. H. Cross, "Reverse engineering and design recovery: A taxonomy", *IEEE Software*, vol. vol. 7, no. 1, pp. 13-77, 1990.

[29] Coleman, D. , D. Ash, B. Lowther и P. Oman, "Using metrics to evaluate software system maintability", *IEEE Computer*, vol. no. 27(8), pp. 44-49, 1994.

[30] Coradeschi, S. и A. Saffiotti, "Symbolic Robotic Systems: Humans, Robots and Smart Environments", *IEEE Intelligent Systems*, vol. vol 21, no 3, pp. 82-84, 2006.

[31] de Jong, Steven , "Fairness in multi-agent systems" в *AAMAS `08 Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: doctoral mentoring program*, 2008, pp. 1742-1743.

[32] Demeyer, S. , "Maintainability versus performance: What's the effect of indtroducing polymorphism", Lab on Reengineering, Universiteit Antwerpen, Tech. Rep. , 2002.

[33] Demeyer, S. , S. Ducasse и O. Nierstransz, *Object-Oriented Reengineering Patterns*.: Morgan Kaufmann and DPunkt, 2002.

[34] Dey, A. K., "Understanding and Using Context", *Personal and Ubiquitous Computing Journal*, vol. 5, no. 1, pp. 4-7, 2001.

[35] Dey, A K и G D Abowd, *The context toolkit: Aiding the development of context-aware applications*., 1999.

[36] Dey, A K и G D Abowd, "Towards a better understanding of context and context-awareness" в *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, 1999.

[37] Dey, A.K. и G.D. Abowd, "Towards a better understanding of context and context-awareness" в *Proceedings of the Workshop on the What, Who, Where, When and How of Context-Awareness*, New York, ACM Press, 2000.

[38] Dieter, U. , H. Mark и M. Florian, "Architecting the Internet of Things", *Springer*, 2011, ISBN 978-3-642-19156-5.

[39] Enderton, H. B., *A Mathematical Indtroduction to Logic*. New York: Academic Press, 1973.

[40] Erman, L. D., B. Hayes-Roth, V. R. Lesser и D. R. Reddy, "Understanding System: Itegrating, Computing Survey", *The Hearsay-II*, vol. Vol 12, no. 2, pp. 213-253, June 1980.

[41] Eskenasi, A. и R Radev, "Quality Evaluation of Authoring Systems", *Education and Computing*, vol. vol. 5, no. no. 199, pp. 43-47, 1989.

[42] Eskenasi, A. и T. Vladimirova, "Incorporating Student Models in Adaptive Testing Systems", *Educ. and Training Technol.*, vol. vol. 3, no. no. 2, pp. 135-142, 1993.

[43] FIPA. (2013, May) FIPA ACL. <http://www.fipa.org/repository/aclspecs>

[44] FIPA. (2013, May) FIPA Official. <http://www.fipa.org/about/index.html>

[45] Fowler, M. , *Refactoring: Improving the Design of Existing Programs*.: Addsion-Wesly, 1999.

[46] Ganchev, I. , S. Stojanov, D. Meere и M. O'Droma, "InfoStation-based mLearning System

Architectures: Some Development Aspects" в *The 8th IEEE International Conference on Advanced Learning Technologies (ICALT'08)*, Santander, Spain, 2008, pp. 504-505.

[47] Ganchev, I. , S. Stojanov, M. O'Droma и D. Meere, "An InfoStation-Based University Campus System Supporting Intelligent Mobile Services", *Journal of Computers*, vol. 2, no. 3, pp. 21-33, 2007.

[48] Ganchev, I. , S. Stoyanov и M. O'Droma, "Consumer-oriented DeLC Service Architecture" в *In Proc. of the 3rd International Conference on Education and Information Systems, Technologies and Applications (EISTA 2005)*, Orlando, Florida, USA, ISBN 980-6560-34-5 , 2005, pp. 213-218.

[49] Ganchev, I. , S. Stoyanov, V. Valkanova и M. O'Droma, "Service-oriented and Agent-based Architecture Supporting Adaptable Context-Aware Provision of Mobile e-Learning Services" в *IADIS e-Learning 2010 Conference*, Freiburg, Germany, 26 - 29 July 2010, pp. 97-104.

[50] Gehrke, J. и L Liu, "Sensor Network Application", *IEEE Internet Computing*, vol. vol 10, no. 2, pp. 16-17, 2006.

[51] Gilbert, D. , *IBM Intelligent Agent Strategy*.: White Paper, 1995.

[52] Glushkova, T. и A. Stoyanova, "Interaction and adaptation to the specificity of the subject domains in the system for e-Learning and distance training DeLC" в *International Scientific Conference "Informatics in scientific knowledge"*, June 2008, Varna Free University "Chernorizets Hrabar".

[53] Griswold, G. и D. Notkin, "Automated assistance for program restructuring", *Trans. Software Engineering and Methodology*, vol. vol. 2, no. 3, pp. 228-269, 1993.

[54] Grundy, J. , J. Hosking и W. Mugridge, "Inconsistency management for multiple-view software development environments", *Trans. Software Engineering*, vol. vol. 24 no.11, pp. 960-981, 1998.

[55] Guessoum, Z. , "Adaptive agents and multiagent systems", *Distributed Systems Online, IEEE* , vol. vol 5, no. 7, 2004.

[56] Guimaraes, T. , "Managing application program maintanance expenditure", *Comm ACM*, vol. no. 26(10), pp. 739-746, 1983.

[57] Halpern, J. , Z. Manna и B. Moszkowski, "A hardware semantics based on temporal intervals" в *Proceedings of the 10th International Colloquium on Automata, Languages and Programming*, vol. Number 154 in the series Lecture Notes in Computer Science, Berlin, 1983, pp. 278-291.

[58] Harel, D. , "First-order dynamic logic", *Lecture notes in computer science*, vol. Number 68, Berlin, 1979.

[59] Harel, D. и D Kozen, "Process logic: Expressiveness, decidability, completeness", *Journal of*

computer and System science, vol. vol 25, pp. 144-170, October 1982.

[60] Hayes-Roth, и Barbara, "An Architecture for Adaptive Intelligent Systems", Department of Computer Science, Stanford University, Scientific report STAN-CS-TR-93-1496, 1993.

[61] Hoare, C. A.R., "An axiomatic basis of computer programming.", *Communications of the ACM* , pp. 576-580, Octomber 1969.

[62] Hoare, C. A.R., *Communicating sequential processes*. London: Printise Hall International, 1985.

[63] Hoare, C. A.R., *Towards a theory of parallel programming in C*, C.A.R. Hoare and R.H. Perrot, Ed. London: Academin Press London , 1972.

[64] Kataoka, Y. , M. D. Ernst, W. G. Griswold и D. Notkin, "Automated support for program refactoring using invariants" в *Int'l Conf. Software Maintenance, IEEE Computer Society*, 2001, pp. 736-743.

[65] Khedo, K K, "Context-aware systems for mobile and ubiquitous networks" в *Proceedings of the International Conference on Mobile Communications and Learning Technologies, IEEE Compuer Society*, 2006.

[66] Kumar, , Ankit, Tayal Ankur, Kumar Senthil и B. S. Bindhumadhava, "Multi-Agent Autonomic Architecture based Agent-Web Services" в *Advanced Computing and Communications*, Bangalor, 2008, pp. 329-333.

[67] Lientz, B. P. и E. B. Swanson, *Software maintanance management: a study of the maintanance of computer application software in 487 data processing organizations*.: Addison-Wesley, 1980.

[68] Liu, B и др., "Inelligent Spaces: An Overview", *IEEE* , 2007.

[69] Li, L и F. Y. Wang, "Cooperative driving at blind crossings using intervehicle communication", *IEEE Transaction on Vehicular Technology*, vol. vol 55, no 6, pp. 1712-1724, 2006.

[70] Lorincz, K , D. J. Malan и Fuford-Jones, "Sensor Networks for Emergency Response: Challenges and Opportunities", *IEEE Pervasive Computing*, vol. vol 3, no. 4, pp. 16-23, 2004.

[71] Maes, и Pattie, "Artificial life meets entertainment: life like autonomous agents", *Communications of the ACM*, vol. vol. 38, no. 11, pp. 108-114.

[72] Manna, Z. и A. Pnueli, "Verification of concurent programs: Temporal framework", *The Correctnes Problem in Computer Science*, pp. 215-273, 1981.

[73] Mens, Tom и Serge Demeyer, *Software evolution*.: Springer-Verlag, 2008.

[74] Mens, T. и A. Van Deursen, *Refactoring: Emerging Trends and Open Problems*., 2003.

[75] Microsoft. (2013, June) Microsoft Class Server.

<http://www.microsoft.com/education/products/>

[76] Mitkov, R. , R. Pavlov и A. Eskenasi, "Testing in Bulgaria with microcomputers", *Revista de Informatica y Automatica*, vol. vol. 19, no. no. 3, pp. 20-23, 1986.

[77] Mitrovich, Dejan , Mirjana Ivanovic, Zoran Budimac и Milan Vidakovic, "An overview of agent mobility in heterogeneous environments" в *Proceedings of the Workshop on Applications of Software Agents*, 2011, pp. 52-58.

[78] Moodle. (2013, July) Moodle. <http://moodle.org>

[79] Moore, , "Automatic inheritance hierarchy restructuring and method refactoring" в *in Proc. Int'l Conf. OOPSLA, ACM SIGPLAN Notices*, 1996, pp. 235-250.

[80] Moszkowski, B. , *A temporal analysis of some concurrent systems*. Berlin: Springer Verlag, Appear in Analysis of Concurrent Systems, in the series Lecture Notes in Computer Science.

[81] Moszkowski, B. , *Executing Temporal Logic Programs*. Cambridge, England: De Montford University, 1985.

[82] Moszkowski, B. , *Ph.D. Thesis: Reasoning about Digital Circuits*.: Department of Compuer Science, Stanford University, 1983, Avaible as technical report STAN-CS-83-970.

[83] Moszkowski, B. и Z. Manna, "Reasoning in interval temporal logic" в *Proceedings of the ACM/NSF/ONR Workshop on Logic of Programs*, 1984, pp. 371-383.

[84] Nunes, I. и C.J. P. Lucena, *BDI4JADE: a BDI layer on top of JADE*.: Monogra_as em Ciência da Computação, 2010, November, vol. NO. 15/10.

[85] Nwana, H. S., L. Lee и N. R. Jennings, "Co-ordination in software agent systems", *BT Technology Journal*, vol. vol 14, no. 4, pp. 79-88, 1996.

[86] Opdyke, W. F., *Ph.D. Thesis: Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks.*: University of Illinois at Urbana-Champaign, 1992.

[87] Oracle. (2013, June) <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html.>

[88] Pardo, Abelardo , "A Multi-Agent Platform for Automatic Assignment Management" в *ITiCSE `02 Proceedings of the 7th annual conference on Innovation and technology in computer science education*, 2002, pp. 60-64.

[89] Pascoe, J , *Adding generic contextual capabilities to wearable computers.*, 1998.

[90] Pham, , Anh Vu и A. Karmouch, "Mobile software agents: an overview", *Communications Magazine, IEEE*, vol. vol 36, no.7, pp. 26-37, 1998.

[91] Pnueli, A. , "The temporal semantics of concurrent programs", *Theoretical Computer Science*, vol. vol 13, pp. 45-60, 1981.

[92] Pratt, V. , "Semantical considerations on Floyd-Hoare logic" в *Proceedings of the Seventeenth Annual IEEE symposium on Foundations of computer science*, Houston,Texas, Octomber, 1976, pp. 109-121.

[93] Rahneva, O. , A. Rahnev и N. Pavlov, "Functional Workflow and Electronic Services In a Distributed Electronic Testing Cluster – DeTC" в *Proceedings 2nd International Workshop on eServices and eLearning*, Otto-von-Guericke Universitaet Magdeburd, ISBN 3-929757-76-1 , 2004, pp. 147-157.

[94] Rahnev, A. , N. Pavlov и O. Rahneva, "Architecture & Design of Distributed Electronic Testing Cluster (DeTC) based on Microsoft.NET Framework" в *IMAPS CS International Conference*, Brno, Czech Republic, ISBN 80-214-2990-9 , 2005, pp. 417-422.

[95] Rahnev, A. и O. Rahneva, "Algorithms for Generation of Circuitries and Drafts in Distributed e-Testing Cluster (DeTC)", *Scientific Works*, vol. 35, no. Book 3, 2007 – Mathematics, pp. 85-100, 2007, ISSN 0204–5249.

[96] Rajlich, V. , "A model for change propagation based on graph rewriting" в *Int'l Cong. Software Maintanance, IEEE Computer Society*, 1997, pp. 84-91.

[97] Rao, A. S. и M. P. Georgeff, "BDIagents: from theory to practice" в *ICMAS*, 1995.

[98] Riad, A. M. и H. A. El-Ghareeb, "A Service Oriented Architecture to integrate Web services and Agents in Course Management Systems", *Egyptian informatics Journal, Cairo University*, vol. vol 8, no. 1, 2007.

[99] ROberts, Don , *Ph.D. Thesis: Practical Anlysis for Refactoring*.: University of Illinois, Urbana-Champaign, 1999.

[100] Russel, , J. Stuart и Peter Norvig, *Artificial Intelligence: A Modern Approach*.: Prentice Hall, 2009.

[101] Ryan, N. S., J. Pascoe и D. R. Morse, "Enhanced Reality Fieldwork: the Context-aware Archaeological Assistant", *Computer Applications in Archaeology*, vol. Tempus Reparatum, 1998.

[102] Salberg, D , A K Dey и G D Abowd, "Ubiquitous computing: Defining an hci research: Agenda for an emerging intercation paradigm", Technical report 1998.

[103] Schilit, B. , N Adams и R Want, "Context-aware computing applications" в *Proceedings of the Workshop on Mobile Computing Systems and Applications, IEEE Computer Society*, 1994.

[104] Schilit, B N и M M Theimer, *Disseminating active map information to mobile hosts*., 1994.

[105] Schmidt, A и K V Laerhoven, *How to build smart appliances*.: IEEE Personal Communications, 2001.

[106] Searle, J. , *Expression and Meaning: Studies in the Theory of Speech Acts*.: Cambridge University Press, 1985.

[107] Shi, Hongchi , Shang Yi и Chen Su-Shing, "A multi-agent system for computer science education" в *ITiCSE `00 Proceedings of the 5th annual SIGCE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, New York, 2000, pp. 1-4.

[108] Siewe, F , H Zedan и A Cau, "The calculus of context-aware ambients", *Journal of Computer and System Sciences*, 2010.

[109] Smith, R. G., "The Contract Net Protocol: High-Level", *IEEE Transactions on Computers*, vol. C-29, no. 12, pp. 1104-1113, December 1980.

[110] Stanford University. (2013, April) Protege. <http://protege.stanford.edu>

[111] Stanford, V. , "Using Pervasive Computing to Deliver Elder Care", *IEEE Pervasive Computing*, vol. vol. 1, no. 1, pp. 10-13, 2002.

[112] Stoilov, T. и K. Stoilova, "Technologies for integration of e-learning content" в *Proceedings of "E-learning conference'06: Computer science education"*, Coimbra, Portugal, 2006, pp. pp.2.11-1, 2.11-6.

[113] Stoilov, T. и K Stoilova, "Web service paradighm", *Challenges in Higher Education&Research*, vol. vol. 6, pp. 125-128, 2008, ISBN: 978-954-580-247-8.

[114] Stoilov, K. , T. Stoilov, V Valchev и O. Farhi, "Virtual learning platforms and good practice", *Scientific work of University of Rousse*, vol. vo. 48, no. ISSN 1311-3321., pp. 149-154, Nov. 2009.

[115] Stoyanov, S. , "Context-Aware and Adaptable eLearning Systems", STRL, De Montfort University, Leicester, UK, PhD Thesis 2012.

[116] Stoyanov, S.; Ganchev, I.; O'Droma, M.; Zedan, H.; Meere, D.; Valkanova, V., "Semantic Multi-Agent mLearning System" in *Semantic Agent Systems: Foundations and Applications*, A. Elci, M. T. Kone, and M. A. Orgun, Eds.: Springer Verlag, 2011, vol. 344, ISBN: 978-3-642-18307-2.

[117] Stoyanov, S.; Popchev, I.; Doychev, E.; Mitev, D.; Valkanov, V.; Stoyanova-Doycheva, A.; Valkanova, V.; Minov, I., "DeLC Educational Portal", *Cybernetics and Information Technologies (CIT)*, vol. 10, no. 3, pp. 49-69, 2010.

[118] Stoyanova-Doycheva, A. , "A Software Refactoring Process and the Supported Tools" в *Fourth International Conference for Informatics and Information Technology*, Bitola, Macedonia, 11-14 Dec 2003, pp. 70-77, ISBN 9989-668-45-0.

[119] Stoyanova-Doycheva, A. , "Development of Refactoring Learning Environment" в *Research and Education in Mathematics, Informatics and their Applications*, Plovdiv, Bulgaria, December 2010.

[120] Stoyanova, A. и T. Glushkova, "Model for implementation of interactive distance learning" в *International Scientific Conference "Informatics in scientific knowledge"*, Varna, June 2010, Varna Free University "Chernorizets Hrabar".

[121] Stoyanov, S. , G. Cholakov, V. Valkanova и M. Sandalski, "Personalized, Reactive and Proactive Providing of e-Learning Services" в *EdiLib Conference , AACE E-Learn 2011 – World Conference on E-Learning in Corporate, Government*, Honolulu, Hawaii, USA, 17-21 October, 2011.

[122] Stoyanov, S. , E. Doychev, A. Stoyanova-Doycheva, V. Valkanova и V. Valkanov, "Education Cluster Supporting eTesting and eLearning in Software Engineering" в *2nd Annual International Conference on Web Technologies & Internet Applications*, Bali, Indonesia, 2012, pp. 23-28.

[123] Stoyanov, S. , I. Ganchev, I.Popchev, M.O'Droma и V. Valkanova, "Agent-Oriented Middleware for InfoStation-based mLearning Intelligent Systems" в *5th IEEE International Conference on Intelligent Systems IS'10*, London, IEEE Catalog Number: CFP10802-CDR, ISBN:978-1-4244-5164-7, Library of Congress:2009934065 , 07.07 – 09.07.2010, pp. 91-95.

[124] Stoyanov, S. , I. Ganchev, D. Mitev, V. Valkanov и M. O'Droma, "Service-oriented and Agent-based Architecture Supporting Adaptable, Scenario-based and Context-aware Provision of Mobile e-Learning Services", *International Journal of Computer Information Systems and Industrial Management Applications*, vol. 3, pp. 771-779, 2011, ISSN 2150-7988.

[125] Stoyanov, S. , I. Ganchev, M. O'Droma, D. Mitev и I. Minov, "Multi-Agent Architecture for Context-Aware mLearning Provision via InfoStations" в *The 5th International Conference on Soft Computing as Transdisciplinary Science and Technology*, Paris, 2008, pp. 549-552.

[126] Stoyanov, S. , I. Ganchev, I. Popchev и M. O'Droma, "An Approach for the Development of a Context-Aware and Adaptive eLearning Middleware" in *Intelligent Systems: From Theory to Practice*, V. Sgurev et al., Ed. Berlin Heidelberg: Springer-Verlag, 2010, pp. 519-535, ISBN: 978-3-642-13427-2.

[127] Stoyanov, S. , I. Ganchev, I. Popchev и M. O'Droma, "An Approach for the Development of InfoStation-Based eLearning Architectures", vol. 61, no. 9, pp. 1189-1198, 2008.

[128] Stoyanov, S. , I. Ganchev, I. Popchev, M. O'Droma и R. Venkov, "DeLC – Distributed eLearning Center" в *1st Balkan Conference in Informatics BCI'2003*, Thessaloniki, Greece, ISBN 960-287-045-1 , 2003, pp. 327-336.

[129] Stoyanov, S. и I. Popchev, "Evolutionary Development of an Infrastructure Supporting the Transition from CBT to e-Learning", *Cyberbetics and Information Technologies (CIT)*, vol. 2, pp. 101-114, 2006, Bulgarian Academy of Sciences.

[130] Stoyanov, Stanimir , Asya Stoyanova-Doycheva и Mihail Chigrichenko, "Improving the creativity in software engineering education with Refactoring Learning Environment" в *Second International Conference Creativity and Inovation in Software engineering*, Ravda, Bulgaria, 2009.

[131] Stoyanov, S. , V. Valkanova, G. Cholakov и M. Sandalski, "Education Portal for Reactive and Proactive Service Provision" в *The Third International Conference on Advanced Cognitive Technologies and Applications (COGNITIVE 2011)*, Rome, Italy, September 25-30, 2011.

[132] Stoyanov, S. и др., "Intelligent Distributed eLearning Architecture" in *Intelligent Systems*, V. M. Koleshko, Ed.: ИнТецх, 2012, pp. 185-218, Hard cover, 978-953-51-0054-6.

[133] Strang, T и C Linnhoff-Popien, "A context modeling survey" в *The Sixth International Conferene on Ubiquitous Computing*, Nottingham, 2004.

[134] Sun. (2013, May) <http://java.sun.com/products/rmi-iiop/>

[135] Tahvildari, L. и K. Kontogiannis, "A methodology for developing transformations using the maintainability soft-goal grpah" в *Working Cong. Reverse Engineering, IEEE Computer Society*, 2002, pp. 77-86.

[136] (2013, June) The Agentcities Network. <http://www.agentcities.net>

[137] TILAB. (2013, June) TILAB. <http://jade.tilab.com>

[138] Tip, F. , A. Kiezun и D. Baumer, "Refactoring for generalization using type constraints" в *SIGPLAN Conf. Object-oriented programming systems, languages, and applications*, 2003, pp. 13-26.

[139] Tokuda, L. и D. S. Batory, "Evolving object-oriented designs with refactoring", *Automated Software Engineering*, vol. vol. 8, no. 1, pp. 89-120, 2001.

[140] US National Science Foundation. (2008) Cyber-Physical Systems (CPS) "NSF 08-611". <http://www.nsf.gov/pub/2008/nsf08611/nsf08611.htm>

[141] Valkanov, V. , "Context-aware management of e-services (Tempura reengineering)", 13th Workshop on "SOftware Engineering, Education and Reverse Engineering", Банско, България, 2013.

[142] Valkanov, V. , "Темпура реинжинеринг", ИИТ-БАН, София, Вътрешен доклад Изграждане на висококвалифицирани млади изследователи по съвременни информационни технологии за оптимизация, разпознаване на образи и подпомагане вземането на решения" - BG051PO001-3.3.04/40, 16 ФЕВРУАРИ, 2010.

[143] Valkanov, V. и D. Mitev, "Tempura Reengineering" в *REMIA 2010: Anniversary International COnference*, Plovdiv, 2010, pp. 311-320.

[144] Van Der Straeten, R. , J. Simmonds, T. Mens и V. Jonckers, "Using description logic to maintain consistency between UML models", *UML, Lectures Notes in Computer Science,Springer-Verlag*, vol. vol. 2863, pp. 326-340, 2003.

[145] Varbanov, S. , S. Stoykov и B. Bonchev, "Software Agents in a Serious Business Game - the PRIME Project Case" в *Proc. of LG'2007 – Learning with Games*, ophia Antipolis, France, ISBN 978-88-901168-0-3 , 2007, pp. 301-306.

[146] Vassileva, D. , "ADAPTIVE E-LEARNING CONTENT DESIGN AND DELIVERY BASED ON LEARNING STYLES AND KNOWLEDGE LEVEL", *Serdica J. Computing*, vol. vol. 6, pp. 207-252, 2012.

[147] Vassileva, D. и B. Bonchev, "Management of storyboard graphs for adaptive courseware construction", *WSEAS TRANSACTIONS on INFORMATION SCIENCE and APPLICATIONS*, vol. vol. 3, no. ISSN: 1790-0832, pp. 321-330, 2010.

[148] W3. (2013, May) <http://www.w3.org/XML>

[149] W3. (2013, May) <http://www.w3.org/RDF>

[150] Wang, F. Y., "The Emergence of Intelligent Enterprises", *From CPS to CPSS, IEEE Intelligent Systems*, pp. 85-88, July/August 2010.

[151] Ward, M. P. и K. H. Bennett, "Formal methods to aid the evolution of software", *Int'l Journal of Software Engineering and Knowledge Engineering*, vol. vol.5, no.1, pp. 25-47, 1995.

[152] Whatley, Janive , "An Agent System to Support Student Teams Working Online", *Journal of Information Technology Education*, vol. vol. 3, 2004.

[153] Willmott, Steven. (2013, June) The Agent cities Network Architecture (Online). <http://liawww.epfl.ch/Publications/Archive/Willmott2002NA.pdf>

[154] Willmott, Steven , Dale Jonathan, Burg Bernard, Charlton Patricia и O`Brien Paul. (2013, June) Agent citie: A Worldwided Open Agent Network (Online). <http://eprints.agentlink.org/4784/>

[155] Wooldridge, M. , *An Introduction to MultiAgent Systems*.: John Wiley & Sons, 2002.

[156] Wooldridge, M. и N. R. Jennings, *Intelligent agents: theory and practice*.: The Knowledge Engineering Review, 1995.

[157] Xiaosheng, T , S Qinghua и Z Ping, *A distributed context-aware model for pervasive service environment*.: IEEE Compuer Society, 2006.

[158] Yoshida, Hiroshi , "Agent-Oriented Global Education System for Chemistry ", Virtual Chemical Education (Online), Scientific report 2000, Feb.

[159] Zedan, H. , "CS-Flow. Computational Model – Lunguistic Support", STRL, De Montfort

University, Internal Report 2012.

[160] Zedan, H. и др., "Mapping Human Creativity" in *STRL Internal Monograph (STRL-2008-09)*. Leicester, UK: De Montfort University, 2008.

[161] Вълканов, Веселина , "Изследване за ВОП в средното училище", ИИКТ, София, 2013.

[162] Вълканова, В. , С. Стоянов, Х. Зедан и И. Попчев, "Модел за изследване на креативното мислене и действие на ученици" в *39-та пролетна конференция на СМБ*, Албена, 2-6 Април, 2010, pp. 274-280.

[163] Глушкова, Т. , "Адаптивна среда за електронно обучение в средните училища", ФМИ на ПУ "П. Хилендарски", Дисертационен труд 2011.

[164] Дойчев, Емил , *Среда за електронни образователни услуги*. Пловдив: Пловдивски университет "П. Хилендарски", 2013.

[165] Ненова, С. , *Моделиране на уеб-базирана система за дистанционно обучение*. София, 2004, Дисертационен труд.

[166] Орозова, Д. , С. Стоянов и И. Попчев, "Виртуално образователно пространство" в *Научна конференция с международно участие „Знанието – традиции, иновации, перспективи”*, Бургас, 2013, приета за печат.

[167] Паунова, Е. и К. Стоилова, "Информационните технологии в помощ на изграждането на знания", *Електронно списание на Варненския свободен университет „Ч. Храбър”*, vol. бр 6, pp. с.1-27, 2013, ISSN 1313-7514.

[168] Рахнева, О. , "Разпределен клъстер за електронно тестване", Пловдивски университет, дисертация 2006.

[169] Сомова, Е. и Г. Тотков, "PeU 2.0 платформа за виртуално обучение на пловдивските висши училища" в *Научно-практическа конференция Новите технологии в образованието и професионалното обучение*, София, 2003, pp. 114-120.

[170] Стоянова-Дойчева, А. , "Автоматизирано средство за навигация на процеси за refactoring" в *Национална научна конференция “Информатиката в научното познание”*, юни 2004, Варненски Свободен Университет "Черноризец Храбър".

[171] Стоянова-Дойчева, А. , "Дефиниране на процес и средства за рефакторинг в обучението по софтуерни технологии", Пловдивски университет, дисертационен труд 2011.

[172] Чолаков, Г. , "Хибридна архитектура за изграждане на Разпределен център за електронно обучение (DeLC)", Пловдивски университет, дисертация 2013.