Софийски университет „Св. Кл. Охридски“
Факултет по математика и информатика
катедра „Компютърна информатика“

ХАБИЛИТАЦИОНЕН ТРУД

на тема

Подходи, програмни среди и езици за проверка на коректността на програми и прилагането им при подготовката на софтуерни специалисти

на доц. д-р Магдалина Василева Тодорова

по конкурс за професор
в професионално направление: 4.6 „Информатика и компютърни науки“,
научна специалност: 01.01.12 „Информатика“
(програмиране)

2013, София

Съдържание

Списък на фигурите и таблиците 4
Увод 5
1. Обзор на областта и мотивация на изследването 7
1.1. Верификация на програмно осигуряване – дефиниция, подходи 7
1.2. Формални методи за верификация на програми 10
1.2.1. Формални модели за анализ на ПО 11
1.2.1.1. Модели, основани на свойства 11
1.2.1.2. Изпълними модели 11
1.2.1.3. Модели, интегриращи черти на модели, основани на свойства и изпълними модели 12
1.2.2. Класификация на методите и средствата за формална верификация на ПО 12
1.2.2.1. Методи и средства за дедуктивен анализ 13
1.2.2.1.1. Средства, основани на разширени съждителни логики или логики от първи ред 13
1.2.2.1.2. Средства, основани на логики от по-висок ред 14
1.2.2.2. Методи и средства за проверка на модели 14
1.2.2.3. Методи и средства за проверка на съгласуваност 15
1.3. Заключителни бележки 15
2. Формални методи и средства за дедуктивен анализ и прилагането им в обучението на софтуерни специалисти 16
2.1. Използван апарат 16
2.1.1. Логика на Хоар 16
2.1.2. Метод на преобразуващите предикати 17
2.1.3. Моделът „дизайн чрез договор“ 20
2.1.4. Система за доказване на теореми HOL 21
2.2. Основни резултати 23
2.2.1. Подход за формална верификация на процедурни и обектно-ориентирани програми 23
2.2.2. Среда за верификация на процедурни и обектно-ориентирани програми 25
2.2.2.1. Генератор на Hoare изрази 25
2.2.2.2. VL - език за верификация на програми 26
2.2.3. Грид среда за електронно обучение по тематиката 33
2.2.4. Прилагане на методи и средства за дедуктивен анализ при обучението по програмиране на студентите от направление „Информатика и компютърни“ науки и ученици в профилираното обучение по информатика 36
2.2.4.1. Верификация по време на изпълнение 37
2.2.4.2. Синтезиране на програми 38
2.3. Заключителни бележки 40
3. Подходи за верификация от тип проверка на съгласуваност 41

3.1. Използван апарат 41
3.1.1. Обобщени мрежи 41
3.1.2. Други 43
3.2. Основни резултати 43
3.2.1. Верификация на обектно-ориентирани програми 43
3.2.1.1. Формален обобщеномрежов проект на клас на ООП 44
3.2.1.2. Дефиниране на подход за проверка на коректността на ООП 47
3.2.1.3. Реализация на подхода 53
3.2.1.4. Образователна среда за проверка на коректността на ООП 60
3.2.1.5. Методологически аспекти на подхода за построяване на коректни ООП 62
3.2.2. Верификация на процедурни програми 62
3.2.2.1. Формален обобщеномрежов проект на връзките между функциите на процедурна програма 65
3.2.2.2. Подход за верификация на процедурни програми 68
3.2.2.3. Реализация на подхода за верификация на процедурни програми 70
3.3. Заключителни бележки 74
Заключение 75
Литература 77
Списък на публикациите за участие в конкурса 84
Приложение 88

Списък на фигурите и таблиците

Фигура 1. Среда за верификация на процедурни и обектно-ориентирани програми
Фигура 2. GridEdu архитектура
Фигура 3. Education Service Grid
Фигура 4. Education Service LMS
Фигура 5. Преход в ОМ с m входни и n изходни позиции
Фигура 6. Дефиниция на преходи на обобщена мрежа, представяща формален ОМ проект на клас
Фигура 7. Архитектура на Model Checker
Фигура 8. Алгоритъм за проверка за консистентност в случая когато функцията дефинира и използва само един обект на клас
Фигура 9. Алгоритъм за проверка за консистентност в случая когато функцията дефинира и използва повече от един обект на клас
Фигура 10. Преход на формален ОМ проект на връзките между функциите на процедурна програма
Фигура 11. Алгоритъм за проверка за консистентност в случая на процедурни програми

Таблица 1. Класификация на методите и средствата за формална верификация на ПО

Увод

Хабилитационният труд е разработен в съответствие с изискванията на чл. 5 (3) от Правилника за условията и реда за придобиване на научни степени и за заемане на научни длъжности във Факултета по математика и информатика на СУ „Св. Кл. Охридски“. Целта му е да представи основните научни и научно-приложни резултати, получени от доц. д-р Магдалина Тодорова, в областта на проектирането, реализирането и прилагането на среди, езици, методи и подходи за проверка на коректността на програми. Написан е във връзка с участието на авторката му в конкурс за професор по професионално направление 4.6. „Информатика и компютърни науки“ (програмиране), обявен в Държавен вестник бр. 14/12.02.2013 г.

Описаните в хабилитационния труд резултати на автора му са публикувани в 20 статии [№№ II.1-II.8; II.12, II.14-II.20, II.23-II.26] и 1 книга [№ II.27] от приложените за участие в конкурса 35 научни публикации. Резултатите, отразени в статиите са от последните 5 години.
Тези 21 публикации към 30.03.2013 г. имат повече от 60 забелязани цитирания, описани подробно в приложението към хабилитационния труд.

Хабилитационният труд е структуриран в три глави, заключение, използвана литература, списък на публикациите за участие в конкурса и приложение.

В глава 1 е направен кратък обзор на предметната област и са мотивирани изследванията. Предложена е класификация на методите за формална верификация в зависимост от вида на използваните модели на спецификацията и на реализацията на програмното осигуряване.

В глава 2 са представени накратко техниките за дедуктивен анализ: логика на Хоар (Hoare), преобразуващи предикати, дизайн чрез договор, системи за доказване на теореми, които са използвани за научните и научно-приложните изследвания на автора, описани в главата. Описани са резултати на автора, които могат да бъдат причислени към формалните методи и средства за дедуктивен анализ: подход и среда за верификация на процедурни и обектно-ориентирани програми; прилагане на методи и средства за дедуктивен анализ при обучението по програмиране на студенти от направление „Информатика и компютърни науки“.

В глава 3 е направено кратко представяне на апарата за моделиране, който е използван. Описан е нов подход за проверка на коректността на програми относно зададена спецификация, разработен от автора на хабилитационния труд. Подходът е от тип проверка на съгласуваност на моделите на програмите и спецификациите и е представен

в случаите на процедурни и на обектно-ориентирани програми. Освен подробни описания на подхода в главата са дадени и фрагменти от реализацията му. В случая на обектно-ориентирани програми е направено кратко описание на образователна среда за проверка на коректността на програмите. Построяването на коректни обектно-ориентирани програми е представено и от методологически аспект.

В заключението са обобщени основните приноси на автора в двете основни направления – методи и средства за верификация от тип дедуктивен анализ и от тип проверка за съгласуваност на модели.
Приложението съдържа сведения за публикациите, на базата на които е написан хабилитационният труд, и справка, съдържаща забелязани цитирания на публикациите в работи на други автори.

Глава 1
Обзор на областта и мотивация на изследванията

1.1. Верификация на програмно осигуряване – дефиниция, подходи

Увеличаването на сложността на програмното осигуряване (ПО) води до нарастване на количеството на грешките в него. Последното е свързано с увеличаване на загубите от тези грешки. Например, загубите на икономиката на САЩ от некачествено програмно осигуряване са от порядъка на 60 милиарда долара за година [I.1]. Могат да се приведат множество примери за грешки в ПО, които са довели до нарушаване на работата на инфраструктурната мрежа, до разрушаване на космически апарати, до човешки жертви и др. Някои примери са дадени и коментирани в [I.2]. Това мотивира големите разходи на софтуерните компании за проверка на коректността (верификацията) на разработвания софтуер. В една типична търговска организация за разработка на софтуер разходите за верификация са от 50% до 70% от общите разходи за цялостната разработка на ПО [I.1].

Стандартът за софтуерна верификация и валидация IEEE 1012-2004 [I.3] определя *верификацията на ПО* като техника, която проверява дали артефактите (техническо задание, модел на предметната област, описание на архитектурата, програмен код, потребителска документация и др.), създадени в хода на разработката на ПО, съответстват на други артефакти, определени като начални (входни) данни, а също дали тези артефакти съответстват на правилата и стандартите.

В литературата са известни следните подходи за верификация на ПО:

- софтуерна експертиза (review, inspection);
- статичен анализ (static code analysis);
- формални методи (formal methods);
- динамични методи (dynamic methods);
- синтетични методи (synthetic methods).

Стандартът за софтуерна експертиза IEEE 1028-1997 [I.4] определя софтуерната експертиза като „процес или заседание, по време на което софтуерният продукт се проверява от персонала на проекта, от мениджъри, потребители, клиенти, представители на потребителите или други заинтересовани страни за обсъждане или одобрение“. В практиката се използват методите: *експертиза на кода* (code review, peer review), *програмиране на две лица* (pair programming), *инспекция* (inspection), *проиграване* (walkthrough), *техническа експертиза* (technical review), *формална експертиза на кода* (formal code review), *лек преглед на кода* (lightweight code review) [I.5, I.6] и др. Методите на софтуерната експертиза могат да се приложат към произволно свойство на ПО, към произволен артефакт на жизнения цикъл на ПО на произволен етап от разработката му.

Недостатък на този вид верификация е, че не може да бъде автоматизиран и изисква активно участие на разработчиците на проекта, на мениджъри, потребители, клиенти, представители на потребителите и др.
Основно предимство на този вид методи за верификация е високата им ефективност – по-голяма от тази на другите подходи за верификация. Тези методи намират от 50% до 90% от всички съществуващи грешки [I.7]. Ефективността им зависи от опита и мотивацията на реализиращите верификацията.

Статичният анализ [I.8, I.9] е анализ на ПО, който се провежда без реално изпълнение на изследваните програми. Методите от този вид се използват за проверка на правилността на формализациите на проверяваните свойства и за търсене на често срещащи се грешки чрез използване на шаблони. Известни са голямо количество средства за статичен анализ. Примери за такива са [I.10]:

- *многоезикови* – Visual Studio Team System, Understand, Axivion Bauhaus Suite, Veracode, Black Duck Suite, BugScout, CAST Application Intelligence Platform, ChecKing, Coverity, DMS Software Reengineering Toolkit, Compuware DevEnterprise, HP Fortify Source Code Analyzer, JustCode – Visual Studio, LDRA Testbed, GrammaTech, MALPAS, Parasoft, SofCheck Inspector, Rational Software Analyzer, Protecode, Yasca и др.
- *.NET ориентирани* – FxCop, Parasoft dotTEST, JustCode, NDepend, ReSharper, StyleCop, CodeIt.Right, CodeRush, Kalistick и др.
- *C/C++ ориентирани* – BLAST, Astrée, Eclipse software, Frama-C, Lint, cpplint, Green Hills Software, Parasoft C/C++test, PC-Lint, PVS-Studio, QA-C, LDRA Testbed, Monoidics INFER, Red Lizard's Goanna и др.
- *Java ориентирани* – LDRA Testbed, Jtest, FindBugs, AgileJ StructureViews, Hammurapi, SemmleCode, Checkstyle, Soot, PMD и др.

Възможностите на анализа, извършван от реализираните средства варира от анализ само на отделни оператори и декларации, до анализ, който включва пълния изходен код на програмата.
Недостатъци на тези методи са, че са приложими само към кода или към определен формат за представяне на артефактите на ПО, както и че чрез тях се откриват ограничен вид грешки.
Предимството им е, че могат напълно да се автоматизират, макар че понякога се налага ръчно да се определят някои техни компоненти.

Формалните методи за верификация се прилагат за верифициране на свойства, които могат да се изразят формално в рамките на някои математически модели, а също на тези артефакти, за които могат да се построят съответни формални модели.

Недостатъци на този вид методи са, че създаването на формалните модели изисква значителни усилия. Построяването на формалните модели и осъществяването на анализа на моделите може да се реализира от висококвалифицирани специалисти по формални модели. Наличието на такива специалисти не е голямо и цената на този вид верификация е висока. Построяването на формалните модели не може да се автоматизира, то винаги се извършва от човек. Съществуват инструменти, чрез които се автоматизира в

значителна степен процесът на анализ на някои формални модели със сложност на промишлено равнище.

Предимства на тези методи са, че чрез тях се откриват сложни грешки от логическо естество, практически неоткриваеми от методите на другите подходи, а също и че могат да се прилагат за верификация на апаратно осигуряване.

Тъй като този вид верификация ще бъде предмет на разглеждане в хабилитационния труд, класификация на методите и обзор на програмни средства, предлагащи този вид верификация ще бъдат дадени в следващата част на тази глава.

Динамичните методи за верификация анализират и оценяват свойства на програмното осигуряване по резултатите от реалната работа на ПО или от работата на неговите модели и прототипи. Пример за такъв вид верификация е тестването [I.11-I.19]. В зависимост от областта на теста тестването се категоризира като: тестване на единична функция или клас (unit test); тестване на група програмни модули и/или класове (module test, integration test, system test); крайно тестване, тестване за приемане на ПО (functional test; performance, stress test).

В литературата са известни следните подходи за динамична верификация:

- тестване от тип „черна кутия“ (quivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, traceability matrix, exploratory testing и specification-based testing);
- тестване от тип „бяла кутия“ (API testing, code coverage, fault injection methods, mutation testing, static testing);
- тестване от тип „сива кутия“ (matrix testing, regression testing, orthogonal array testing);
- визуално тестване (visual testing).

Основните недостатъци на тези методи са: позволяват да се намерят само грешки, които възникват по време на работата на ПО; прилагането им изисква допълнителна подготовка – създаване на тестове, разработка на системи за тестване или мониторинг, разработка на средства за визуализация, анимация, симулация, което понякога е доста трудоемка задача.

Основни предимства [I.18] на динамичните методи са, че чрез тях могат да се открият сериозни грешки, най-малко ресурсоемки са и са лесни и предпочитани. Считат се за най-евтините техники за верификация и на практика са най-често използвани.

Синтетичните методи за верификация интегрират инструменти от подходите, описани по-горе. Най-често обекти на интегриране са методите за статичен анализ, формален анализ и тестване. Целта е да се засилят предимствата и да се намалят недостатъците на тези методи за верификация. Най-разпространени методи от този вид са *верификация по време на изпълнение* (runtime verification) [I.20 - I.23] и *тестване, основано на модели* (model based testing) [I.24, I.25]. Съществуват и методи, които интегрират формална верификация със статичен анализ [I.26]. Верификацията по време на изпълнение най-често комбинира формална верификация с изпълнение на програмата. Проверяваните свойства се описват чрез формален модел и се вграждат в системата за тестване.

Методите, реализиращи тестване, базиращо се на модели, използват формални модели за получаване на тестовете.

Анализирайки съществуващите методи за верификация на програми, софтуерните инженери на конкретното програмно осигуряване избират един или друг метод за верификация.

Обект на изследване в този труд са методи за формална верификация, както и такива, които могат да се причислят към синтетичните, в основата на които е формалната верификация.
Причините за направения избор са:
- Формалните методи за верификация на програми се явяват най-надеждното средство, осигуряващо правилното функциониране на програмното осигуряване.
- Формалните методи за верификация изискват от прилагащите ги да мислят по-различен начин – начин, който се формира в резултат от интегрирано обучение на компютърните науки с математика.
- Формалните методи за верификация позволяват изчерпателно и точно да се формулира целта. В резултат на това софтуерният инженер трябва да се фокусира единствено върху целта.
- Формалните методи за верификация позволяват драстично подобрение на дизайна на проекта и продуктивността на верификацията.

Въпреки трудностите при прилагането им, формалните методи и синтетичните методи, в основата на които е формалната верификация са обект на активни изследвания през последните години. Причината е повишаването на сложността на създавания софтуер и необходимостта от изграждането на математически модели за него. В основата на тези методи е изграждането на формални модели за верификация и анализ на свойства на ПО.

1.2. Формални методи за верификация на програми

В тази част е дадена класификация на известните методи за формална верификация на ПО в зависимост от вида на използваните формални модели на спецификацията и на реализацията на програмното осигуряване (Таблица 1) [I.27]. Според нея методите и средствата за формална верификация се разделят на методи и средства за:
- дедуктивен анализ,
- проверка на моделите,
- проверка за съгласуваност.

В раздел 1.2.2 е направен кратък анализ на тези формални методи и средства за верификация. Тъй като предложената класификация се основава на вида на използваните формални модели, в раздел 1.2.1 е направен преглед на тези модели.
Резултатите в главата са публикувани в статия [II.12].

1.2.1. Формални модели за анализ на ПО

Чрез средствата за изграждане на модели могат да се създават модели на спецификацията (проверяваното свойство) и реализацията (проверявания артефакт). Известните в литературата формални модели за анализ на програмно осигуряване могат да се разделят на:

- *модели, основани на свойства* (property-based models);
- *изпълними модели* (executable models);
- *модели, интегриращи модели, основани на свойства и изпълними модели*.

1.2.1.1. Модели, основани на свойства

Известни са *логическите* и *алгебричните* модели.

Примери за средства за изграждане на логически модели:

- *Съждително смятане* (propositional calculus) [I.28, I.29];
- *Предикатно смятане* (predicate calculus) [I.28, I.29];
- *Предикатно смятане от по-висок ред* (higher-order predicate calculus) [I.28];
- *Ламбда смятане* (lambda calculus) [I.30];
- *Ламбда смятане от по-висок ред* (higher-order lambda calculus) [I.30] ;
- *Модални логики* (modal logics) [I.31];
- *Темпорални логики* (temporal logics) [I.32];
- *Линейни темпорални логики* (linear temporal logic, LTL) [I.32, I.33];
- *μ-смятане* (или смятане с неподвижни точки, μ-calculus) [I.34];
- *Логики с явно време* (timed temporal logics) [I.32].

Примери за средства за изграждане на алгебрични модели:

- *Релационни алгебри* [I.35], лежащи в основата на релационните системи за управление на бази от данни;
- *Алгебрични модели на абстрактните типове данни* [I.36];
- *Алгебри на процесите* (process algebras, process calculus) [I.37].

1.2.1.2. Изпълними модели

Известни са следните средства за изграждане на изпълними модели:

- *Крайни автомати* (finite state machine) [I.38];
- *Системи за преход* (labeled transition systems) [I.39];
- *Взаимодействащи автомати* (communicating finite state machines) [I.38];
- *Йерархични автомати* (hierarchical state machines) [I.40];
- *Времеви автомати* (timed automata) [I.41];
- *Хибридни автомати* (hybrid automata) [I.42];
- *Мрежи на Петри* (Petri nets) [I.43];
- *Цветни мрежи на Петри* (coloured Petri nets) [I.44];
- *Предикатно преходни мрежи* (predicate/transition nets) [I.45];
- *ω-автомати* [I.46];
- *Абстрактни автомати* (abstract state machines) [I.47];

- *Обобщени мрежи* (generalized nets) [I.48, I.49];
- *Цветни обобщени мрежи* (coloured generalized nets) [I.48, I.49].

1.2.1.3. Модели, интегриращи черти на модели, основани на свойства и изпълними модели

- *Логика на Хоар* (Hoare logic) [I.50];
- *Обобщения на логиката на Хоар*, *динамични* или *програмни логики* (dynamic logics, program logics) [I.51];
- *Програмиране с договори* (design by contracts) [I.52].

1.2.2. Класификация на методите и средствата за формална верификация на ПО

За да се верифицира формално някакво свойство е необходимо формално да се провери определено съответствие между моделите на *спецификацията* и *реализацията*. Възможни са:

	модел на спецификацията	*модел на реализацията*	*метод за формална верификация*
1	модел, основан на свойства	модел, основан на свойства	дедуктивен анализ (доказване на теореми)
2	модел, основан на свойства	изпълним модел	проверка на моделите
3	изпълним модел	модел, основан на свойства	не съществуват
4	изпълним модел	изпълним модел	проверка за съгласуваност

Таблица 1. Класификация на методите и средствата за формална верификация на ПО

В първия случай верификацията на спецификацията *S* в реализацията *P* се свежда до доказване на изводимост, която най-често се записва чрез $P \vdash S$. Последната се осъществява чрез методи, наречени дедуктивен анализ или доказателство на теореми (theorem proving).
Във втория случай верификацията на спецификацията *S* в реализацията *P* се свежда до проверка на изпълнимост и се записва чрез $P \models S$. За целта се използват методи за проверка на модела (model checking).
Третият случай практически не се реализира. Причината е, че така реализацията става по-абстрактна от спецификацията.
В четвъртия случай верификацията на спецификацията *S* в реализацията *P* се осъществява чрез редукция или симулация. В литературата тези методи са известни като методи за проверка за съгласуваност.

Следва кратък обзор на формалните методи за верификация, определени в Таблица 1.

1.2.2.1. Методи и средства за дедуктивен анализ

При тези методи програмата се разглежда като формално твърдение, за което трябва да се докажат изразени чрез предикати свойства. Исторически първите методи за дедуктивен анализ на програми са предложени от Флойд [I.53] и Хоар [I.50] в края на 60-те години на миналия век. В основата им лежат логиката на Хоар и предложената от Флойд и усъвършенствана от Манна и Пнюели техника за доказване завършването на изпълнението на операторите за цикъл. Последната се основава на инварианти на цикъл и монотонността на функцията за завършване на изпълнението на оператора за цикъл (ограничаваща функция, свързана с цикъла). С цел опростяване на техниката, предложена от Флойд, Дейкстра [I.54] предлага техниката на преобразуващите предикати. Следват аналогични методи за верификация на програми, съдържащи масиви, указатели, обръщения към процедури и функции, а също и за верификация на паралелни програми [I.55]. Прилагането на методите на дедуктивния анализ за верификация на практически значими програмни системи води до изграждането на специализирани средства за автоматично построяване на доказателства (provers, proof assistants).

Тези средства могат да се разделят в две категотии, описани накратко по-долу.

1.2.2.1.1. Средства, основани на разширени съждителни логики или логики от първи ред

Най-известните средства в тази категория са:

- ***Theorem Prover Vampire*** [I.56] – среда за автоматично доказване на теореми за класическата логика от първи ред;
- ***KeY*** [I.57] – формално средство, което интегрира дизайн, реализация, формална спецификация и формална верификация на обектно-ориентиран софтуер; притежава средство за доказване на теореми, изразени чрез динамичната логика от първи ред;
- ***ACL2*** [I.58] – език за програмиране, чрез който могат да се моделират компютърни системи и инструмент за доказване на свойства на модели;
- ***E theorem prover*** [I.59] – модерно средство за доказване на теореми за логики от първи ред с равенства;
- ***Minlog*** [I.60, I.61] – интерактивна система за доказване на теореми, базираща се на логика от първи ред, с проста математическа семантика и предоставяща средства за извличане на коректен програмен код;
- ***Waldmeister*** [I.62] – високоефективно средство за доказване на теореми, изразени чрез равенствената логика;
- ***Darwin*** [I.63] – ефективно средство за доказване на теореми, изразени чрез моделно-еволюционното смятане (Model Evolution Calculus).

1.2.2.1.2. Средства, основани на логики от по-висок ред

Най-широко известни и използвани такива средства са:

- ***HOL*** (Higher-Order Logic) [I.64] – среда за интерактивно доказване на теореми в логика от по-висок ред;
- ***Isabelle*** [I.65] – интерактивна среда за доказване на теореми, наследник на HOL;
- ***Coq*** [I.66] – среда за доказване на теореми, изразени чрез смятането на индуктивни конструкции (Calculus of Inductive Constructions);
- ***PVS*** (*Prototype Verification System*) [I.67] – система за верификация, базираща се на спецификационен език, интегриран с допълнителни инструменти и средства за доказване на теореми.

При извършване на доказателства с тези системи често се налага намесата на висококвалифициран специалист, който да формулира помощни леми или да промени схемата на доказателството.

1.2.2.2. Методи и средства за проверка на модели

Същината на тези методи се описва чрез следните три стъпки. На първата стъпка се конструира модел на програмата (програмното осигуряване), най-често основан на система от преходи. На втората стъпка моделът на програмата се допълва със спецификациите на свойствата, които трябва да притежава програмата (програмното осигуряване). На третата стъпка се осъществява верификацията на свойствата на програмата. В процеса на работа на алгоритъма за проверка на модела, се построява множество от състояния на модела, в които се изпълняват спецификациите. В случай, че не е намерено състояние на модела, което не удовлетворява спецификацията, алгоритъмът връща „истина“. В противен случай, алгоритъмът връща „лъжа“ и дава информация защо не е в сила спецификацията.
Първите методи за проверка на моделите са предложени в началото на 80-те години на миналия век. Реализират пълно изследване на модела на Крипке с помощта на автоматични средства за проверка на формули от логиката CTL (Computation Tree Logic) [I.68]. След тези методи са разработени символните методи [I.69, I.70]. При тях проверката се осъществява чрез обработка на Ordered Binary Decision Diagrams (OBDD) автомат, съответстващ на модела. Последното позволява да се проверяват модели с количество на състоянията до 10 000 – 10 500.
Недостатъци на тези методи са, че не всяка система може да се представи в достатъчно компактен вид чрез OBDD, както и че за някои видове времеви логики алгоритмите за проверка на моделите са неефективни.

Някои от най-известните в практиката инструменти за проверка на модели са:

- **BLAST** (Berkeley Lazy Abstraction Software Verification Tool) е средство за проверка на модели за C програми [I.71];
- **CADP** (Construction and Analysis of Distributed Processes) е средство за проектиране на протоколи и разпределени системи [I.72]. Инструменти за проверка на модели за различни темпорални логики и μ-смятане в CADP са модулите му EVALUATOR и XTL;

- **GNTicker** е инструмент за симулация на обобщеномрежови модели [I.73, I.74];
- **UPPAAL** е средство за моделиране, валидация и верификация на вградени системи и системи в реално време [I.75];
- **NuSMV** реализира символна проверка на модели [I.76];
- **HyTech** е автоматизирано средство за верификация на хибридни модели на вградени системи [I.77];
- **Design/CPN** е пакет от средства, поддържащ използването на цветни мрежи на Петри (CP-nets или CPN) [I.78].

Още методи и средства за проверка на модели могат да се намерят в [I.79].

1.2.2.3. Методи и средства за проверка на съгласуваност

Методите за проверка на съгласуваност анализират съответствието между двата изпълними модела – на проверяваните свойства и на проверявания артефакт. Повечето използват тестване и затова понякога се отнасят и към методите тестване на базата на модели. Съществуват и методи, които използват аналитични методи за проверка на съгласуваност.
Сред най-известните среди за проверка на съгласуваност са ***Verity-Check*** [I.80] и модулите BISIMULATOR [I.81] и REDUCTOR [I.82] на средата ***CADP*** [I.72] за проектиране на комуникационни протоколи и разпределени системи.

1.3. Заключителни бележки

Формалната верификация, като използва формални методи, доказва коректността или некоректността на програмното осигуряване. В сравнение със софтуерната експертиза, статичния анализ и динамичните методи за верификация на ПО тя е най-ефективният и най-надеждният метод за верификация, който има недостатъка, че прилагането му изисква значителни усилия и наличие на висококвалифицирани специалисти, които да го реализират.
Пътят за преодоляването на този недостатък е интегрирането на методите на формална верификация с други методи за верификация и използване на инструменти за автоматично доказване на теореми и за автоматична проверка на модели.

В следващите части на хабилитационния труд са представени резултати на автора му, свързани с формални методи и средстава за верификация на ПО, които могат да се причислят към методите и средствата за дедуктивен анализ (глава 2) и за проверка на съгласуваност на модели (глава 3). Важно място в изложението заема прилагането на получените резултати при подготовката на софтуерни специалисти.

Глава 2

Формални методи и средства за дедуктивен анализ и прилагането им в обучението на софтуерни специалисти

В тази глава са представени резултати на автора на хабилитационния труд, свързани със създаването и прилагането на средства, езици, методи и подходи за проверка коректноста на програми, които интегрират черти на модели, основани на свойства и на изпълними модели. Тъй като техниките за дедуктивен анализ са преобладаващи ги причисляваме към формалните методи и средства за дедуктивен анализ.
Резултатите в нея са публикувани в [II.1-II.8, II.19, II.24, II.26]. Изложението започва с кратко описание на използваните формални средства и системи: логика на Хоар [I.50, I.83], метод на преобразуващите предикати [I.85, I.84], моделът дизайн чрез договор [I.86, I.52], както и на системата за доказване на теореми HOL [I.64].

2.1. Използван апарат

2.1.1. Логика на Хоар

Основен елемент на логиката на Хоар е тройката на Хоар

$$\{Q\}\ S\ \{R\}$$

където *Q* и *R* са твърдения (формули в предикатната логика), а *S* е оператор (програма).
Q се нарича предпоставка (предусловие), а *R* – постусловие.

В стандартната логика на Хоар тройката на Хоар изразява частична коректност. След разширяване на правилото за извод за оператора за цикъл *while* с условия за завършване на изпълнението, тройката на Хоар изразява тотална коректност, т.е. ако изпълнението на *S* започва в състояние, което удовлетворява предиката *Q*, то то ще завърши след крайно време в състояние, което удовлетворява *R*. Разширяването на това правило е направено от Дейкстра [I.54] и Манна-Пнюели [I.87].
Логиката на Хоар предоставя аксиоми и правила за извод за основните оператори на императивните езици за програмиране.

***Аксиоми и правила за извод*:**

Аксиома за празния оператор	*Аксиома за оператора за присвояване*
$\dfrac{}{\{Q\}\ \text{празен оператор}\ \{Q\}}$	$\dfrac{}{\{P(x \leftarrow E)\}\ x := E\ \{P\}}$

Правило за конкатенацията

$$\frac{\{P\}\,S\,\{Q\},\,\{Q\}\,T\,\{R\}}{\{P\}\,S;T\,\{R\}}$$

Правило за условния оператор

$$\frac{\{B \wedge P\}\,S\,\{Q\},\,\{\neg B \wedge P\}\,T\,\{Q\}}{\{P\}\text{ if } B \text{ then } S \text{ else } T\,\{Q\}}$$

Правило за следствието

$$\frac{P \Rightarrow P',\,\{P'\}\;S\;\{Q'\},\,Q' \Rightarrow Q}{\{P\}\;S\;\{Q\}}$$

Правило за оператора while (частична коректност)

$$\frac{\{P \wedge B\}\;S\;\{P\}}{\{P\}\;\text{while } B \text{ do } S\;\{\neg B \wedge P\}}$$, P е инварианта на цикъла.

В аксиомата за оператора за присвояване записът *P(x ← E)* означава израза *P*, в който всяко свободно участие на променливата *x* е заместено с израза *E*.

Формулировка на правилото за тоталната коректност на оператора *while* може да се намери в [I.28, стр 203]. По-долу е предложена модификация на това правило, при която добре-ограниченото множество, което е използвано, е множетвото от естествените числа с обичайната наредба <.

Правило за оператора while (тотална коректност)

$$\frac{\begin{array}{l}\{P \wedge B\}\;S\;\{P\}\\ P \wedge B \Rightarrow t > 0\\ \{P \wedge B\}\;z := t; S\;\{t < z\}\end{array}}{\{P\}\;\text{while } B \text{ do } S\;\{\neg B \wedge P\}}$$

Това правило допълва правилото за частичната коректност на *while* с още две предпоставки, изразяващи завършване на изпълнението на оператора за цикъл. За целта е използвана помощна функция *t*, която е целочислена, ограничена отдолу от 0 и строго намаляваща при всяко изпълнение на цикъла. Функцията *t* се нарича *функция за завършване на изпълнението на оператора за цикъл* или *ограничаваща функция за цикъла*.
В раздел 2.2, съдържащ резултати на автора на хабилитационния труд, тройката на Хоар означава тотална коректност.

2.1.2. Метод на преобразуващите предикати

За целите на провежданите изследвания е използвано подмножество на императивен (процедурен) език за програмиране, състоящо се от: празен оператор, блок, оператор за присвояване, условен оператор (пълна и кратка форма) и оператор за цикъл *while*.
За верифицирането на програми се използва специален предикат, наречен *преобразуващ* [I.85, I.84].

Дефиниция 1. Нека *S* е произволен оператор, а *R* е предикат, който описва очаквания от изпълнението на оператора *S* резултат. *Преобразуващ предикат* за *S* и *R* е предикатът *Wp(S, R)*, представящ множеството от всички състояния, за които изпълнението на *S*, започващо от такова състояние, задължително завършва след крайно време в състояние, за което е в сила изходният предикат *R*.

Предикатът Wp(S, R) се нарича още *най-слабо предусловие* [I.84, част 1.6] *за S относно R*, тъй като то определя множеството от всички състояния, такива че изпълнение на *S*, започващо от произволно от тях завършва при изпълнение на постусловието *R*.

Означението {Q}S{R}, изразяващо тотална коректност, е друга форма на Q => Wp(S, R).

Някои свойства на преобразуващия предикат [I.84]

- *Дистрибутивност на конюнкцията*

  $Wp(S, Q) \wedge Wp(S, R) = Wp(S, Q \wedge R)$
- *Закон за монотонността*

  Ако Q => R, то Wp(S, Q) => Wp(S, R)
- *Дистрибутивност на дизюнкцията*

  $Wp(S, Q) \vee Wp(S, R) \Rightarrow Wp(S, Q \vee R)$

Следващите дефиниции определят преобразуващия предикат за операторите от подмножеството, определено по-горе. Те следват от съответните правила, описани в [I.84]. За операторите е използван синтаксисът им в езика *C++*.
Нека *R* е произволен предикат.

Дефиниция 2. Wp(празен_оператор, R) = R.

Дефиниция 3. $Wp(x = e, R) = domain(e) \wedge R(x \leftarrow e)$, където *domain(e)* е предикат, описващ множеството от всички състояния, в които е дефиниран изразът *e*.

Дефиниция 4. $Wp(S_1; S_2; ...; S_n, R) = Wp(S_1, Wp(S_2; ...; S_n, R))$, където $S_1, S_2, ..., S_n$ са произволни оператори от подмножеството, описано по-горе.

Дефиниция 5. $Wp(\{S_1; S_2; ...; S_n\}, R) = Wp(S_1; S_2; ...; S_n, R)$, където $S_1, S_2, ..., S_n$ са произволни оператори от подмножеството, описано по-горе.

Дефиниция 6. $Wp(\text{if } (B)\ S_1; \text{ else } S_2, R) = domain(B) \wedge (B \Rightarrow Wp(S_1, R))$

$$\wedge (\neg B \Rightarrow Wp(S_2, R)).$$

В частност

$$Wp(\text{if } (B)\ S, R) = domain(B) \wedge (B \Rightarrow Wp(S, R)) \wedge (\neg B \Rightarrow R).$$

Често не е необходимо да се намери преобразуващият предикат *Wp(if (B)* S_1*; else* S_2*, R)* или *Wp(if (B) S, R)*, а само да се провери дали е в сила импликацията *Q => Wp(if (B)* S_1*; else* S_2*, R)* или *Q => Wp(if (B) S, R)*. В тези случаи е полезна следната теорема.

Теорема 1. Нека за оператора *if (B)* S_1*; else* S_2 и предикатите *Q* и *R* са в сила условията:

а) Q => domain(B)
б) $Q \wedge B \Rightarrow Wp(S_1, R)$
в) $Q \wedge \neg B \Rightarrow Wp(S_2, R)$.

Тогава (и само тогава) е в сила $Q \Rightarrow Wp(\text{if } (B)\ S_1; \text{ else } S_2, R)$.

Тази теорема е адаптиран запис на теорема 10.5 [I.84] за пълната форма на условен оператор на езика C++. Доказателството й може да се получи лесно като се използва доказателството на теорема 10.5 [I.84].

Следствие. Нека за оператора *if (B) S;* и предикатите *Q* и *R* са в сила условията:

а) Q => domain(B)

б) $Q \wedge B$ => Wp(S, R)

в) $Q \wedge \neg B$ => R.

Тогава (и само тогава) е в сила Q => Wp(if (B) S, R).

Тъй като практически не е лесно да се използва дефиницията на преобразуващия предикат за оператора *while*, в практиката се използва формулираната по-долу теорема 2, чрез която се проверява верността на импликацията

Q => Wp(while (B) S, R).

За целта с оператора за цикъл *while (B) S;* се свързат:

а) инварианта *P* - предикат, който е в сила преди изпълнението и след изпълнението на всяка стъпка на оператора *while*;

б) функция *t* за завършване на изпълнението на оператора за цикъл - целочислена функция, явяваща се горна граница на броя на стъпките на цикъла, които остават да бъдат изпълнени. Функцията *t* трябва да е ограничена отдолу от 0 и при всяка стъпка от изпълнението на оператора за цикъл да намалява поне с 1.

Теорема 2. Нека предикатът *P* и целочислената функция *t* удовлетворяват условията:

а) $P \wedge B$ => Wp(S, P)

б) $P \wedge B$ => $(t > 0)$ и

в) $P \wedge B$ => Wp(t_1 = t; S, $t < t_1$),

където t_1 е нов идентификатор. Тогава е в сила условието:

P => Wp(while(B) S, $P \wedge \neg B$).

Тази теорема е адаптиран запис на теорема 11.6 [I.84] за оператора за цикъл *while* на езика *C*++. Доказателството й може да се получи лесно като се използва доказателството на теорема 11.6 [I.84].

Като следствие от теорема 2 може да се формулира списък от условия за верифициране на оператора за цикъл *while*.

Нека е даден *while* цикъл, подходящо анотиран с предусловие, инварианта, ограничаваща функция и постусловие:

```
{Q: предусловие}
{P: инварианта}
{t: ограничаваща функция}
  while (B) S;
{R: постусловие}
```

Списък от условия за верификация на оператора за цикъл while

1) P е в сила преди изпълнението на оператора за цикъл, т.е. в сила е $Q => P$;

2) $P \wedge B => Wp(S, P)$ е в сила, т.е. P е инварианта на цикъла.

3) $P \wedge \neg B => R$ е вярно, т.е. когато завърши изпълнението на оператора за цикъл е в сила постусловието.

4) $P \wedge B => (t > 0)$ е в сила, т.е. t е ограничена отдолу от 0, докато изпълнението на оператора за цикъл не е завършило.

5) $P \wedge B => Wp(t_1 = t; S, t < t_1)$ е вярно, т.е. всяка стъпка от изпълнението на оператора за цикъл води до строго намаляване на ограничаващата функция t.

2.1.3. Моделът „дизайн чрез договор“

За целите на провежданите изследвания се използва и моделът „дизайн чрез договор“ („design by contract“ или „programming by contract“) [I.86, I.52], предлагащ методология за създаване на коректен софтуер. Създаден е от Бертранд Мейер и е реализиран в проектирания от него език за обектно-ориентирано програмиране Eiffel. Моделът „design by contract“ изисква от проектантите на софтуер да определят формални и верифицируеми спецификации за интерфейса на компонентите на програмните системи. За целта се описват специални твърдения, наречени *договори*, които трябва да са в сила в определени места на програмите. Договорите могат да описват: елементарни твърдения, инварианти на класове, предусловия и постусловия на функции и член-функции на класове. Наричат се още *спецификация* на програмата.

Спецификацията на функция (процедура) на процедурна програма се задава чрез указване на предусловието и постусловието ѝ. Ако функцията съдържа оператори за цикъл, за всеки от тях се задават инварианта и функция за завършване на изпълнението ѝ. Предусловието на функцията изразява ограниченията, които са необходими за коректната работа на функцията. Постусловието изразява свойство, описващо състоянието, което трябва да е в сила при завършване на изпълнението на функцията. Предусловието и постусловието на функцията определят договора ѝ с всички нейни клиенти.

Спецификацията на клас на обектно-ориентирана програма (ООП) се задава чрез указване на: инвариантата на класа, предусловията и постусловията на член-функциите (методите) на класа. Ако член-функциите на класовете използват оператори за цикъл, за всеки от тях се задават още инварианта и ограничаваща функция. Предусловието на член-функция изразява ограниченията, които са необходими за коректната работа на член-функцията. Постусловието изразява свойство, описващо състоянието, което трябва да е в сила при завършване на изпълнението на член-функцията. Предусловието и постусловието на член-функция определят договора ѝ с всички нейни клиенти. Всеки обект на клас удовлетворява свойства, които се задават чрез предикат, наречен инварианта на класа. Инвариантата на клас трябва да е в сила след изпълнението на всеки конструктор, както и преди и след изпълнението на всеки метод на класа.

Неформално, един клас е коректен относно зададена за него спецификация, когато неговата реализация, зададена чрез член-функциите на класа, е съгласувана с предусловията, постусловията и инвариантата на класа.

По-строго коректността на клас по отношение на някаква спецификация е определена в [I.86, глава 11] по следния начин.

Дефиниция 7. Класът *C* е коректен относно своята спецификация, тогава и само тогава, когато:

а) За всеки конструктор *P* и за всяко допустимо множество от аргументи x_p е в сила тройката на Хоар

$$\{ Default_C \wedge pre_p(x_p) \} \; Body_P \; \{ post_p(x_p) \wedge Inv \} \qquad (1)$$

б) За всяка член-функция *r* с множество от допустими аргументи x_r е в сила тройката на Хоар

$$\{ pre_r(x_r) \wedge Inv \} \; Body_r \; \{ post_r(x_r) \wedge Inv \}, \qquad (2)$$

където *Inv* е инвариантата на класа *C*, $Body_r$ е тялото на член-функцията *r*, $pre_r(x_r)$ и $post_r(x_r)$ са предусловието и постусловието на *r* с допустими аргументи x_r. Ако предусловието или постусловието на член-функция на *C* е пропуснато, счита се, че то е *true*. Чрез $Default_C$ е означено твърдение, изразяващо връзки по подразбиране между член-данните на класа *C*.

При верифицирането на член-функциите на клас относно свързаните с тях предусловия и постусловия се формулират и доказват теореми, представени чрез тройки на Хоар от вида (1) и (2) и означаващи тотална коректност. Доказателство на тези теореми може да стане ръчно като се използва методът на преобразуващите предикати, както и като се използват някои системи за автоматично доказателство на теореми. Сред най-успешните системи за доказателство на теореми според изследванията, публикувани в [I.88], са:

- HOL Light – 69 теореми от 100,
- Mizar – 45 теореми от 100,
- ProofPower – 42 теореми от 100,
- Isabelle – 40 теореми от 100,
- Coq – 39 теореми от 100.

Това подкрепя направения от автора на настоящия труд избор за използване на системата за доказателство на теореми HOL.

2.1.4. Система за доказване на теореми HOL

Предназначена е да реализира доказателства на теореми в логика от по-висок ред. Множествено-теоретичната семантика на HOL е разработена от A. Pitts [I.64]. Използва полиморфен вариант на теорията на типовете на Чърч, разработен от М. Гордън.

Системата притежава богата колекция от теории, библиотеки и структури, които са основно средство за извършване на доказателствата [I.64]. Наследник е на системата за доказване на теореми Logic for Computable Functions и е преминала през редица версии, някои от които са: HOL88, HOL90, HOL98, HOL4 (Kananaskis 1, 2, 3, 4, 5, 6, 7).

В изследването, описано в [II.3, II.4, II.6, II.7] е използвана версията Kananaskis 4 [I.64]. При създаването на среда за верификация на процедурни и обектно-ориентирани програми [II.3, II.4, II.6, II.7] са използвани средствата на HOL за дефиниране на нови типове и функции.

Въпреки, че логиката на HOL позволява дефинирането на нови типове данни, в HOL е дадена възможност за дефиниране на типове, взаимствана от съвременните езици за функционално програмиране. Тази възможност дава мотивация за интерфейс от високо ниво за определяне на алгебрични типове данни.

Синтаксис на дефиницията на тип:

```
Hol_datatype '[<binding> ;]* <binding>'
<binding> ::= ident = <constructor-spec>
           | ident = <record-spec>
<constructor-spec> ::= [<clause> |]* <clause>
<clause> ::= ident
           |ident of [hol_type =>]* hol_type
<record-spec> ::= <|[ident : hol_type ;]* ident : hol_type |>
```

За дефиниране на функции се използва обръщението

```
Define `<spec>`
```

където

```
<spec> ::= <eqn>
       | (<eqn>) /\ <spec>
<eqn> ::= <alphanumeric> <pat> ... <pat> = <term>
<pat> ::= <variable>
       | <wildcard>
       | <cname>                        (* 0-ary constructor *)
       | (<cname>_n <pat>_1 ... <pat>_n) (* constructor appl. *)
<cname> ::= <alphanumeric> | <symbolic>
<wildcard> ::= _
       | _<wildcard>
```

При доказване на теоремите се използват различни тактики. Примери за вградени в системата тактики са: *REWRITE_TAC*; *ASM_REWRITE_TAC*, *PURE_REWRITE_TAC*, *RW_TAC*, *CONJ_TAC*; *EQ_TAC*; *DISCH_TAC*; *GEN_TAC*; *PROVE_TAC*; *STRIP TAC*; *ACCEPT_TAC*; *ALL_TAC*; *NO_TAC* [I.64]. В приложенията най-често се използват тактиките: *REWRITE* и нейните подобни, *ARW_TAC*, *CONJ_TAC* и *EQ_TAC*.

Тактиката *ARW_TAC* е дефинирана чрез тактиката *RW_TAC*, която е от тип

simpLib.simpset -> thm list -> tactic,

където *simpset* е специална колекция от теореми и други подобни функционалности. *ARW_TAC* се дефинира по следния начин:

val ARW_TAC = RW_TAC arith_ss;

където *arith_ss* е стандартното множество от правила за опростяване и преобразуване на аритметични изрази. *ARW_TAC* се използва с указване на списък, който или е празен, или в него се изброяват допълнителни твърдения, които се добавят като правила за преобразуване към правилата от *arith_ss*.

Тактиката *CONJ_TAC* редуцира целта A ?- t1 /\ t2 на две подцели A ?- t1 и A ?- t2. Тактиката *EQ_TAC* редуцира целта A ?- t1 = t2, където *t1* и *t2* са от тип булев, в следните две подцели A ?- t1 ==> t2 и A ?- t2 ==> t1, където = => означава импликация.

2.2. Основни резултати

Основните резултати на автора на хабилитационния труд в тази област са:

- Описание на подход за проверка на коректността на процедурни и обектно-ориентирани програми относно зададени спецификации чрез използване на описания в раздел 2.1 апарат.
- Проектиране и реализация на среда за верификация на процедурни и обектно-ориентирани програми, базираща се на описания подход.
- Проектиране на грид-среда за електронно обучение по тематиката.
- Прилагане на методи и средства за дедуктивен анализ при обучението по програмиране на студентите от направление „Информатика и компютърни науки“ и на ученици в профилираното обучение по информатика.

2.2.1. Подход за проверка на коректността на процедурни и обектно-ориентирани програми относно зададени спецификации

Подходът интегрира логиката на Хоар, разширена с правилото за доказване на тотална коректност на програми, с модела „дизайн чрез договор“ и метода на преобразуващите предикати. При него проверката на коректността на програма относно зададена спецификация се свежда до доказване на една или повече теореми, което отнася подхода към формалните методи за дедуктивен анализ. За доказване на формулираните теореми се използва системата за доказване на теореми HOL. Описание на подхода е направено в [II.3, II.1].
Подходът е предназначен за образователни и приложни цели.

Кратко описание на подхода за:

а) *процедурни програми*

С програмата (програмния фрагмент) *S* се свързват предикатите *P* и *Q*, изразяващи съответно пред- и постусловията, които задават входно-изходната ѝ спецификация. Верификацията на *S* относно *P* и *Q* се свежда до доказване на теоремата {P} S {Q}. Доказването на верността на тази теорема се осъществява като тя се преобразува във вид на теорема на езика на системата HOL и се оценява от системата. За целта, като се използват аксиомите и правилата за извод на логиката на Хоар, доказателството на

теоремата се свежда до доказване верността на други теореми, изразени чрез тройки на Хоар за операторите, съставящи *S*.
От дефинициите на {P} S {Q} и Wp(S, Q) следва, че доказателството на верността на {P} S {Q} може да се сведе до доказателство на импликацията P => Wp(S, Q). Като се използва този факт, а също дефинициите 2, 3, 4, 5 и 6 (раздел 2.1.2) и правилата за конкатенацията и следствието, след елементарни преобразувания, доказателството на {P} S {Q} се свежда до доказателство на конюнкция от импликации, верифициращи всеки оператор от редицата оператори, съставящи *S*. Например, ако *S* има вида:

```
x = e;
while (B) Stat;
```

доказателството на {P} S {Q} се свежда до доказателство на теоремата:

```
{P} x = e {I} ∧
{I} while (B) Stat {Q}
```

и съответно до доказателство на

```
(P => Wp(x = e, I)) ∧
(I ∧ B => Wp(Stat, I)) ∧
(I ∧ B => (t > 0)) ∧
(I ∧ B => Wp(t1 = t; Stat, t < t1)) ∧
(I ∧ ¬ B => Q)
```

където *I* е инвариантата на оператора *while*, *t* – функцията за завършване изпълнението на оператора *while*, а t_1 е нов идентификатор. След намиране на преобразуващите предикати, се получава теорема, която системата за доказване на теореми HOL може да оцени евентуално след допълване на HOL с още помощни твърдения, дефинирани от потребителя.

б) *обектно-ориентирани програми*

Освен посочените по-горе средства, за верификацията на класовете на ООП се използват още моделът програмиране чрез договор.

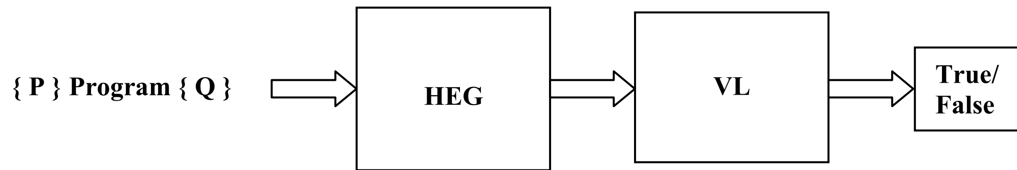
За всеки клас се дефинира *инварианта на класа*. Тя е предикат, който трябва да е в сила след изпълнението на всеки конструктор и преди, и след изпълнението на всеки метод на класа. Моделът „дизайн чрез договор" изисква за всеки метод на клас да се дефинират договори (предусловие и постусловие), специфициращи изискванията към входните за метода данни и към търсения резултат. В рамките на означенията в раздел 2.1.3. конюнкцията на всички тройки на Хоар от (1) и (2) дефинира теорема, доказателството на която верифицира класа. Верификацията на функциите на обектно-ориентирана програма се реализира по начина, описан в а) като се използват договорите на методите на класовете.
Примери за реализацията на подхода са дадени в [II.1, II.3, II.4]. Извършените експерименти и анализи на получените резултати мотивираха създаването на среда за верифициране на програми, основаваща се на него.

2.2.2. Среда за верификация на процедурни и обектно-ориентирани програми

Описанието на средата е направено в [II.6, II.4, II.7, II.1]. Съставена е от две компоненти (фиг. 1):

- генератор на тройки на Хоар (Hoare Expression Generator, съкратено HEG);
- език за проверка на коректност (Verification Language, съкратено VL).

Разработена е за целите на обучението по програмиране и софтуерно инженерство, но освен за обучение на студенти може да се използва и за обучение на софтурни специалисти.



Фиг. 1. Среда за верификация на процедурни и обектно-ориентирани програми

HEG превежда теоремата, верността на която ще се доказва, в Hoare израз на езика VL. VL предоставя средства, чрез които системата за доказване на теореми HOL да докаже теоремите, получени в резултат от работата на модула HEG.

2.2.2.1. *Генератор на Hoare изрази*

Генераторът на Hoare изрази е транслатор, който превежда програма написана на процедурен или на обектно-ориентиран език, подходящо анотирана с предусловия, постусловия и инварианти на цикли и класове, в Hoare израз на езика VL. Реализиран е като компилатор.

Пример. HEG превежда процедурата на езика *C*++

```
## Precondition dividend >= 0 && divisor > 0;
   Postcondition dividend == quot*divisor + rem &&
                             0 <= rem && rem < divisor
##
void div_mod(int dividend, int divisor, int& quot, int& rem)
{ quot = 0;
  rem = dividend;
  while(divisor <= rem)
  { ## Loop_invariant dividend == quot*divisor + rem &&
                      0 <= rem && divisor > 0;
        Limited_function rem
     ##
     quot++;
     rem = rem-divisor;
  }
}
```

анотирана с предусловие, постусловие, инварианта и ограничаваща функция на цикъла и намираща частното *quot* и остатъка *rem* от целочисленото деление на *dividend* на *divisor*, в следния Hoare израз:

```
HOR ((#0 LE %"dividend") AND (#0 LS %"divisor"))
(procedure "div_mod"(val "dividend", val "divisor",
                      ref "quot", ref "rem")
(("quot" :== # 0);;
 ("rem" :== %"dividend");;
 (while ((% "dividend" EQ % "quot" MUL % "divisor" ADD % "rem") AND
         (#0 LE % "rem") AND (#0 LS % "divisor"))
        (% "rem")
        (% "divisor" LE % "rem")
         (("quot" :== % "quot" ADD # 1);;
         ("rem" :== % "rem" SUB % "divisor")))))
        ((% "dividend" EQ % "quot" MUL % "divisor" ADD
          % "rem") AND (# 0 LE % "rem") AND
        (% "rem" LS % "divisor"))`;
```

на езика VL. Някои означения за анотиране на процедурни и обектно-ориентирани програми, които се използват са:

Precondition P	P е предусловие
Postcondition P	P е постусловие
Loop_invariant P	P е инварианта на цикъл
Limited_function f	f е ограничаваща функция на цикъл
##…##	текстът между ## и ## задава предусловия, постусловия, инварианти и/или ограничаващи функции

Полученият израз е предикат, който ако е в сила процедурата *div_mod* ще е тотално коректна относно указаната входно/изходна спецификация. Може да се разглежда като теорема, но за съжаление тази теорема не може да се докаже от никоя от известните системи за доказване на теореми, включително и от HOL. Последното може да стане възможно, ако системата за доказване на теореми HOL се надгради с език, позволяващ й да интерпретира израз от горния вид като теорема. Тази функция се изпълнява от езика VL на средата за верификация.

2.2.2.2. VL - език за верификация на програми

VL е процедурен език, надграден с аксиоматична семантика. Вграден е в системата за доказване на теореми HOL чрез вграждане в дълбочина. Аксиоматичната семантика на езика се реализира чрез Hoare изрази.

Основни типове данни на VL са цял и булев. Програмите му са редици от процедури и функции, съставени от операторите: празен, за присвояване, съставен, условен, за цикъл, за вход и изход, за обръщение към процедура или функция. От съображения за краткост следва описание на подмножество на езика.

Граматика

```
arith_expr ::=
      variable
    | integer
    | arith_expr POW arith_expr
    | arith_expr MUL arith_expr
    | arith_expr QUO arith_expr
    | arith_expr REM arith_expr
    | arith_expr ADD arith_expr
    | arith_expr SUB arith_expr
bool_expr ::=
      arith_expr EQ arith_expr
    | arith_expr LS arith_expr
    | arith_expr LE arith_expr
    | arith_expr GR arith_expr
    | arith_expr GE arith_expr
    | bool_expr IMP bool_expr
    | NOT bool_expr
    | bool_expr AND bool_expr
    | bool_expr OR bool_expr
command ::=
      skip
    | variable :== arith_expr
    | command ;; command
    | iff bool_expr command command
    | while bool_expr arith_expr bool_expr command
    | procedure string form_param command
    | function string form_param command
specification ::=
      { bool_expr } command { bool_expr }
    | [bool_expr]
```

където:

- POW, MUL, QUO, REM, ADD и SUB са аритметичните оператори съответно за степенуване, умножение, целочислено деление, остатък от целочислено деление, събиране и изваждане;
- IMP, NOT, AND и OR означават импликация, отрицание, конюнкция и дизюнкция съответно;
- операторите за сравнения са: EQ (за равенство), LS (за по-малко), LE (за по-малко или равно), GR (за по-голямо), GE (за по-голямо или равно).
- *command* определя синтаксиса на операторите: *skip* е празният оператор, *variable :== arith_expr* е операторът за присвояване, *command ;; command* е съставният оператор, *iff bool_expr command command* е пълна форма на условния оператор, *while bool_expr arith_expr bool_expr command* е операторът за цикъл, като първите два израза са инвариантата и ограничаващата функция, а следващите ги *bool_expr* и *command* са условието и тялото на цикъла;

- *procedure string form_param command* дефинира процедура, където *string* и *form_param* задават името и формалните параметри, а *command* определя тялото на процедурата;
- *function string form_param command* дефинира функция, където *string* и *form_param* задават името и формалните параметри, а *command* определя тялото на функцията.

Формалните параметри на процедурите и функциите се дефинират по следния начин:

```
form_param =
        val string
      | ref string
      | form_param , form_param
```

където формален параметър, предшестван от *val* е параметър-стойност, а формален параметър, предшестван от *ref* е параметър-променлива.
Изразът

{ bool_expr } command { bool_expr }

е тройка на Хоар, а [bool_expr] е стойността на *bool_expr*.

Семантика на езика

Освен общоприетата процедурна семантика, семантиката на VL включва и логиката на Хоар, допълнена с правила за верифициране на процедури и функции, описана чрез следните правила за извод, записани съгласно синтаксиса на VL:

Правило за оператора skip

$$\frac{\texttt{[P IMP Q]}}{\texttt{\{P\} skip \{Q\}}}$$

Правило за оператора за присвояване

$$\frac{\texttt{[P IMP Q (E/x)]}}{\texttt{\{P\} x:==E \{Q\}}}$$

Правило за съставния оператор

$$\frac{\texttt{\{P\} S1 \{R\} и \{R\} S2 \{Q\}}}{\texttt{\{P\} S1 ;; S2 \{Q\}}}$$

Правило за условния оператор if-else

$$\frac{\texttt{\{P AND B\} S1 \{Q\} и} \quad \texttt{\{P AND (NOT B)\} S2 \{Q\}}}{\texttt{\{P\}(iff B S1 S2)\{Q\}}}$$

Правило за оператора за цикъл

$$\frac{\texttt{[P IMP I] и} \quad \texttt{\{I AND B\} S \{I\} и} \quad \texttt{[(I AND (NOT B)) IMP Q] и} \quad \texttt{[(I AND B) IMP (t GR 0)] и} \quad \texttt{[(I AND B) IMP Wp(t1 :== t ;; S, t LS t1)]}}{\texttt{\{P\}(while I t B S)\{Q\}}}$$

Правило за дефиниция на процедура

$$\frac{\texttt{\{P\} S1 \{Q\}}}{\texttt{\{P\}(procedure N R S1)\{Q\}}}$$

Правило за дефиниция на функция

$$\frac{\texttt{\{P\} S1 \{Q\}}}{\texttt{\{P\}(function N R S1)\{Q\}}}$$

където *P*, *Q*, *R* и *B* са булеви изрази, *E* е аритметичен израз, *S*, *S1* и *S2* са произволни оператори на езика VL, а *x*, *t* и *t1* са променливи. Q(E/x) е булев израз, в който всяко срещане на променливата *x* е заместено с израза *E*. Правилото за оператора *while*

съдържа инвариантата *I* и ограничаващата функция *t* на цикъла. Първите три предпоставки изразяват частичната коректност, а другите две – завършване на изпълнението на оператора за цикъл. Една от предпоставките на правилото за извод на оператора за цикъл съдържа преобразуващия предикат *Wp(command, bool_expr)*, дефиниран в съответствие с правилата от раздел 2.1.2 по следния начин:

```
Wp(skip, Q) = Q
Wp(x :== E, Q) = Q[E/x]
Wp(S1;; S2, Q) = Wp(S1, Wp(S2, Q))
Wp(write E, Q) = Q
Wp(iff B S1 S2, Q) = (B IMP Wp(S1, Q)) ∧
                     (NOT B IMP Wp(S2, Q))
```

$\wedge$ означава конюнкция.

Реализация

Осъществява се чрез разделно вграждане в дълбочина на синтаксиса и семантиката на VL в езика за доказване на теореми HOL [II.6, II.7]. Използвана е версията Kananaskis 4 на HOL (раздел 2.1.4).

Вграждане на синтаксиса

Осъществява се чрез средствата на HOL за дефиниране на типове данни. Следващият фрагмент от програмен код показва вграждане на подмножество на езика, включващо: целите аритметични изрази (чрез типа arith_expr), булевите изрази (чрез типа bool_expr), формалните параметри (чрез типа form_param) и операторите (чрез типа command).
Целите променливи на езика VL са низове, а стойностите им се означават чрез низ, предшестван от знака %. Целите константи се означават чрез цели числа, предшествани от знака #. Аритметичните оператори са инфиксни. Приоритетите им са като в процедурните езици. Операция с по-голям приоритет се задава чрез по-голям приоритетен номер.

```
Hol_datatype `arith_expr =
        % of string
      | # of num
      | ADD of arith_expr => arith_expr
      | SUB of arith_expr => arith_expr
      | POW of arith_expr => arith_expr
      | MUL of arith_expr => arith_expr
      | QUO of arith_expr => arith_expr
      | REM of arith_expr => arith_expr`;

set_fixity "POW" (Infixr 99);
set_fixity "MUL" (Infixl 97);
set_fixity "QUO" (Infixl 97);
set_fixity "REM" (Infixl 97);
set_fixity "ADD" (Infixl 95);
set_fixity "SUB" (Infixl 95);

val bool_expr =
    Hol_datatype `bool_expr =
        EQ of arith_expr => arith_expr
```

```
        | LS of arith_expr => arith_expr
        | LE of arith_expr => arith_expr
        | GR of arith_expr => arith_expr
        | GE of arith_expr => arith_expr
        | IMP of bool_expr => bool_expr
        | NOT of bool_expr
        | AND of bool_expr => bool_expr
        | OR of bool_expr => bool_expr`;
  val form_param =
        Hol_datatype `form_param =
          val of string
        | ref of string
        | , of form_param => form_param`;

  Hol_datatype `command =
          skip
        | :== of string => arith_expr
        | ;;    of command => command
        | write of arith_expr
        | iff of bool_expr => command => command
        | while of bool_expr => arith_expr => bool_expr => command
        | procedure of string => form_param => command
        | function of string => form_param => command`;
  set_fixity "IMP" (Infixl 90);
  set_fixity "EQ" (Infixl 90);
  set_fixity "LS" (Infixl 90);
  set_fixity "LE" (Infixl 90);
  set_fixity "GR" (Infixl 90);
  set_fixity "GE" (Infixl 90);
  set_fixity "AND" (Infixl 83);
  set_fixity "OR" (Infixl 83);
  set_fixity ":==" (Infixr 20);
  set_fixity ";;" (Infixr 10);
  set_fixity "," (Infixr 10);
```

Чрез

```
set_fixity "<name>" (<assoc> <number>);
```

се задава приоритетен номер <number> на инфиксния асоциативен оператор с име <name>. <assoc> е или *Infixl* (лява асоциативност), или *Infixr* (дясна асоциативност). В реализацията по-горе, операторът POW е дясноасоциативен с приоритетен номер 99. Останалите аритметични оператори са лявоасоциативни.

Операторите за сравнения са инфиксни, лявоасоциативни с равни приоритетни номера и сравняват само аритметични изрази. Отрицанието е префиксен оператор, а конюнкцията и дизюнкцията са инфиксни и лявоасоциативни. Операторите “:= =“, “;;“ и “,“ са инфиксни и дясноасоциативни.

Вграждане на семантиката

Реализира се чрез вграждане на семантиката на аритметичните и булевите изрази, на операторите и правилата за извод. За целта се използват средствата на HOL за дефиниране на функции. Вгражданията на процедурната и на аксиоматичната семантики

в реализацията са взаимно независими. Следващият фрагмент съдържа програмен код на езика на системата HOL, реализиращ вграждането на семантиката на подмножество на езика VL.

Вграждането на целите и булевите изрази е реализирано чрез функциите *Se* и *Sb* съответно:

Se: arith_expr → state → num
Sb: bool_expr → state → {true, false}

Вторият параметър на тези функции е *състоянието* (s), в което се изчислява изразът. Състоянието е функция от тип: string ->num.

- *семантика на изрази от тип цял*

```
val Se_def = Define
`(Se (% v) s = s v) /\
  (Se (# c) s = c) /\
  (Se (m POW n) s = ((Se m s) ** (Se n s))) /\
  (Se (m MUL n) s = ((Se m s) * (Se n s))) /\
  (Se (m QUO n) s = ((Se m s) DIV (Se n s))) /\
  (Se (m REM n) s = ((Se m s) MOD (Se n s))) /\
  (Se (m ADD n) s = ((Se m s) + (Se n s))) /\
  (Se (m SUB n) s = ((Se m s) - (Se n s)))`;
```

- *семантика на изрази от тип булев*

```
val Sb_def = Define
`(Sb (m EQ n) s = ((Se m s) = (Se n s))) /\
 (Sb (m LS n) s = ((Se m s) < (Se n s))) /\
 (Sb (m LE n) s = ((Se m s) <= (Se n s))) /\
 (Sb (m GR n) s = ((Se m s) > (Se n s))) /\
 (Sb (m GE n) s = ((Se m s) >= (Se n s))) /\
 (Sb (a IMP b) s = ((Sb a s) ==> (Sb b s))) /\
 (Sb (NOT a) s = ~(Sb a s)) /\
 (Sb (a AND b) s = ((Sb a s) /\ (Sb b s))) /\
 (Sb (a OR b) s = ((Sb a s) \/ (Sb b s)))`;
```

- *семантика на заместване на всяко срещане на променлива в аритметичен израз*

```
val SUBST_arith = Define
`(SUBST_arith (% p) E V = (if p = V then E else (%p))) /\
 (SUBST_arith (#c) E V = (#c)) /\
 (SUBST_arith (m POW n) E V =
        ((SUBST_arith m E V) POW (SUBST_arith n E V))) /\
 (SUBST_arith (m MUL n) E V =
        ((SUBST_arith m E V) MUL (SUBST_arith n E V))) /\
 (SUBST_arith (m QUO n) E V =
        ((SUBST_arith m E V) QUO (SUBST_arith n E V))) /\
 (SUBST_arith (m REM n) E V =
        ((SUBST_arith m E V) REM (SUBST_arith n E V))) /\
 (SUBST_arith (m ADD n) E V =
        ((SUBST_arith m E V) ADD (SUBST_arith n E V))) /\
 (SUBST_arith (m SUB n) E V =
        ((SUBST_arith m E V) SUB (SUBST_arith n E V)))`;
```

- *семантика на заместване на всяко срещане на променлива в булев израз*

```
val SUBST_bool = Define
`(SUBST_bool (m EQ n) X V =
       ((SUBST_arith m X V) EQ (SUBST_arith n X V))) /\
  (SUBST_bool (m LS n) X V =
       ((SUBST_arith m X V) LS (SUBST_arith n X V))) /\
  (SUBST_bool (m LE n) X V =
       ((SUBST_arith m X V) LE (SUBST_arith n X V))) /\
  (SUBST_bool (m GR n) X V =
       ((SUBST_arith m X V) GR (SUBST_arith n X V))) /\
  (SUBST_bool (m GE n) X V =
       ((SUBST_arith m X V) GE (SUBST_arith n X V))) /\
  (SUBST_bool (a IMP b)X V =
       ((SUBST_bool a X V) IMP (SUBST_bool b X V)))  /\
  (SUBST_bool (NOT a) X V = NOT (SUBST_bool a X V))   /\
  (SUBST_bool (a AND b) X V =
       ((SUBST_bool a X V) AND (SUBST_bool b X V)))  /\
   (SUBST_bool (a OR b) X V =
       ((SUBST_bool a X V) OR (SUBST_bool b X V)))`;
```

Вграждането на правилата за извод е реализирано чрез дефиницията на функцията:

HOR: bool_expr → command → bool_expr → {true, false}

Дефиницията на функцията *HOR* използва преобразуващия предикат *Wp*:

Wp: command → bool_expr → bool_expr

Реализацията на *Wp* за оператора за присвояване е свързано с реализация на заместване на всяко срещане на променлива с аритметичен израз в булев израз. Последното се осъществява от обръщението (*SUBST_bool Q E V*), чрез което всяко срещане на *V* в булевия израз *Q* се замества с аритметичния израз *E*. За булевите изрази, които са сравнения на аритметични изрази, заместването се свежда до заместване на всяко срещане на променлива с израз в аритметичен израз. Обръщението (*SUBST_arith A E V*) към функцията *SUBST_arith* реализира заместване на всяко срещане на променивата *V* с израза *E* в аритметичния израз *A*.

- *семантика на преобразуващия предикат Wp*

```
val Wp_def = Define
` (Wp skip Q = Q) /\
   (Wp (V :== E) Q = (SUBST_bool Q E V)) /\
   (Wp (S1 ;; S2) Q = (Wp S1 (Wp S2 Q))) /\
   (Wp (write E) Q = Q) /\
   (Wp (iff B S1 S2) Q =
      ((B IMP (Wp S1 Q)) AND (NOT B IMP (Wp S2 Q))))`;
```

- *семантика на правилата за извод*

```
val HOR_def = Define
 `(HOR P skip Q = (!s. (Sb P s) ==> (Sb Q s))) /\
   (HOR P (V :== E) Q = (!s. (Sb P s) ==>(Sb (Wp (V :== E) Q) s))) /\
   (HOR P (write E) Q = (!s. (Sb P s) ==> (Sb Q s) )) /\
   (HOR P (S1 ;; S2) Q = ((HOR P S1 (Wp S2 Q)) /\
         (HOR (Wp S2 Q) S2 Q))) /\
   (HOR P (iff B S1 S2) Q = (!s. (Sb P s) ==>
                                  (Sb (Wp (iff B S1 S2) Q) s))) /\
```

```
 (HOR P (while B1 E B2 S1) Q =
         (!s. ((Sb P s) ==> (Sb B1 s)) /\
         ((Sb B1 s) /\ (Sb (NOT B2) s) ==> (Sb Q s)) /\
         (HOR (B1 AND B2) S1 B1) /\
         ((Sb B1 s) /\ (Sb B2 s) ==> (((Se E s) > 0) /\
         (Sb (Wp (("t1" :== E ) ;; S1)(E LS % "t1")) s) )))) /\
(HOR P (procedure N R S1) Q = (HOR P S1 Q)) /\
(HOR P (function N R S1) Q = (HOR P S1 Q))`;
```

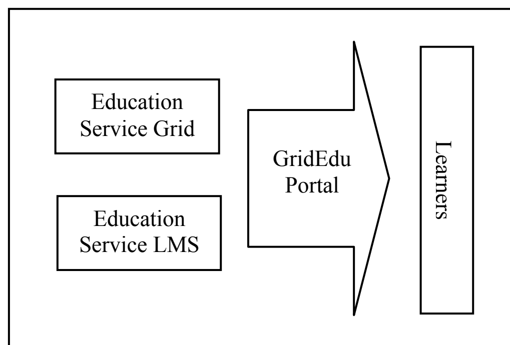
Описаният подход за верификация на процедурни и обектно-ориентирани програми и създадената за него среда за верификация са приложени при обучението по програмиране само на група студенти, притежаващи отлична подготовка по програмиране и математика. Съображенията за това са:

- не е предвиден голям хорариум за обучение по създаването на качествен програмен код в учебните програми,
- недостатъчно добрата подготовка по математика и програмиране на цялата аудитория.

За реализирането на това обучение с цел повишаване на ефективността на обучението по програмиране, от автора на този труд е разработен проект на Грид среда за електронно обучение, наречена GridEdu [II.8].

2.2.3. Грид среда за електронно обучение по тематиката

Проектираната среда интегрира: грид компютинг, електронно обучение, Web услуги и онтологии. Архитектурата й е илюстрирана на фиг. 2, а фигури 3 и 4 представят архитектурите на нейните основни модули.



Фиг. 2. GridEdu архитектура

Компонентата Education Service Grid се основава на Грид услуги, а Education Service Learning Management System (LMS) – на Web услуги и образователни онтологии. GridEdu portal реализира връзката между обучаемите и модулите на системата.

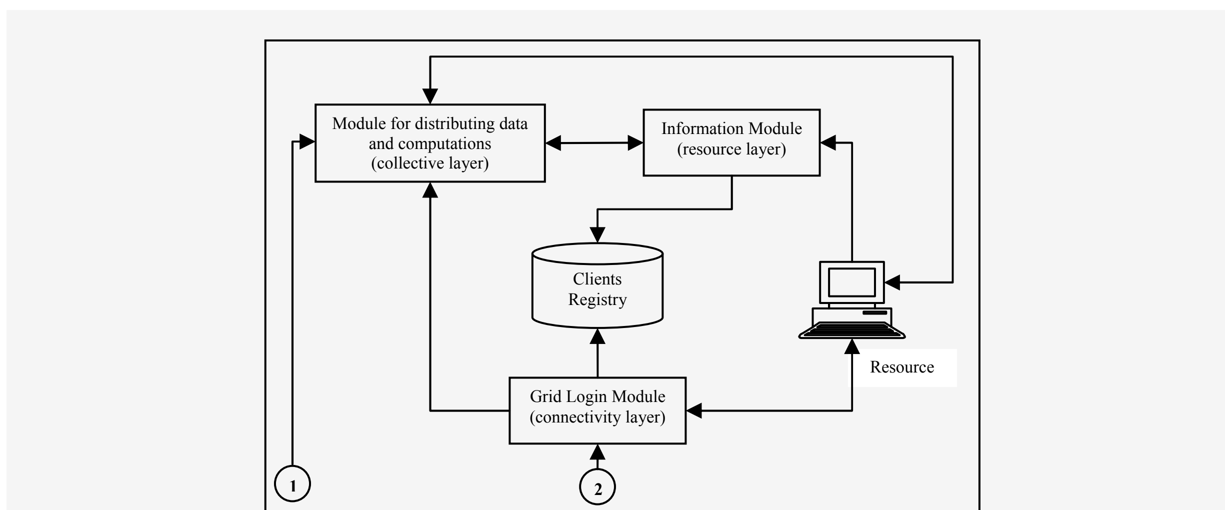
Education Service Grid (фиг. 3) представя слоевете на Грид архитектурата, адаптирани за електронно обучение.

Слоят *fabric* е *Java* аплет, който осигурява единен интерфейс за всички ресурси в Education Service Grid.

Слоят *connectivity* представя услугите, свързани с входа в мрежата. Услугата има следните основни функции: създаване на нови потребители; изтриване на потребител; регистриране на компютри; дерегистриране на компютри, вход и изход в Грид и др. За реализирането й се използва симетричния криптографски алгоритъм.

Слоят *resource* специфицира информационна услуга, която определя статуса и типа на всеки ресурс в мрежата. Основни операции на информационната услуга са: търсене на информация за статуса на всеки компютър; търсене на информация за типа на ресурса на всеки компютър; търсене на ограниченията, използвани за всеки компютър; генериране на списък с информация за статуса на всички компютри и др.

Слоят *collective* отговаря за разпределението на данните и действията в модула Education Service Grid. Услугата извършва следните основни операции: регистрация, копиране, изтриване, търсене на данни и др. За реализирането й е подходящ алгоритъмът за планиране на мрежа (grid scheduling algorithm) [I.89].



Фиг. 3. Education Service Grid

Функциите на Education Service LMS модула са да координира всички дейности, свързани с обучението. Архитектурата му (фиг. 4) е подобна на тази на традиционна LMS, а функционалността й е организирана като уеб услуга. Предлага както съдържание, което използва Education Service Grid, така и съдържание, което не използва Education Service Grid. Изграден е от следните основни компоненти: LMS Login Module, LMS Course Manager Module, Education Ontology Module, Content Module, Authoring Module и Additional Module.

- *LMS Login Module*

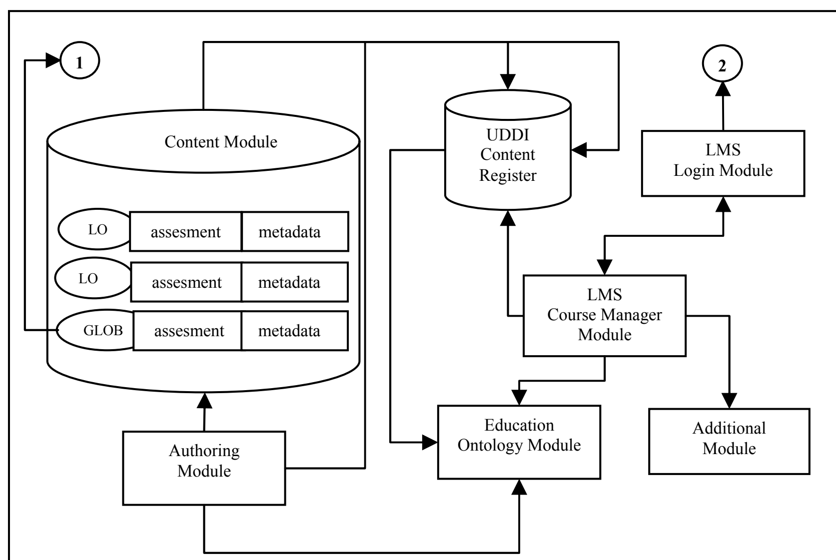
Тази компонента извършва „login“ услугата. Това са функции, свързани с регистрирането на потребители и извършване на някой адаптационни трансформации.

- *LMS Course Manager Module*

Компонентата изпълнява услуги, свързани с управлението на учебните курсове. Обучаемите могат да търсят учебни курсове, реализирани като learning objects (LO), могат да се записват за курсове, да слушат уроци и др.

- *Education Ontology Module*

Компонентата извършва онтологични услуги. Тези услуги поддържат семантично търсене за курсове и реализират операции за вмъкване, изтриване, редактиране и търсене на елементи в онтологията. Позволява се добавяне на нови лекционни единици с цел да се подобри учебното съдържание.



Фиг. 4. Education Service LMS

- *Content Module*

Компонентата изпълнява услуги, които осигуряват съдържанието на електронното обучение. Учебното съдържание се състои от следните части:

- учебен предмет (LO на фиг. 4),
- оценяваща част (assessment на фиг. 4),
- метаданни (metadata на фиг. 4).

LO е урок или част от урок в HML или HTML формат. Частта „оценка“ се състои от онлайн тестове, с които обучаемите и/или техните учители могат да проверят дали учебното съдържание е овладяно. Частта „метаданни“ съдържа информацията за търсене на компютрите, които поддържат съдържанието на електронното обучение.

Модулът „съдържание“ (content module, фиг. 4) съдържа също и Grid Learning Objects (GLOBs). Така Education Service LMS използва функционалността на Education Service Grid компонентата.

- *Authoring Module*

Компонентата изпълнява услуги, които позволяват създаването, редактирането и публикуването на съдържанието на електронното обучение така, че то да може да бъде намерено от Education Service LMS.

- *Additional Module*

Компонентата изпълнява други услуги като чат, дискусии, обявяване на резултати от оценки и др.

Връзката между двата модула на Грид средата за електронно обучение (фиг. 2) се осъществява както е описано в [II.8]. Обучаемият се свързва с GridEdu чрез e-learning PC. Чрез услугата „login" на Education Service LMS компютърът на обучаемия става ресурс на Education Service Grid. След като обучаемият се регистрира, аплетът на слоя „fabric" се прехвърля като код и се изпълнява от компютъра на обучаемия. Това позволява да се осъществи комуникация с Education Service Grid. Услугата „login" на Education Service LMS се обръща към Грид-услугата „login" и изпълнява операцията за зареждане на Грид с параметри: данните и PC-то на обучаемия. Така се реализират действията по проверката за автентичност на компютъра на обучаемия в Education Service Grid. Ако потребителят не е регистриран в grid login service, услугата „login" на Education Service LMS може да предизвика създаването на нов Grid потребител.

Интегриране на компонентите на GridEdu се осъществява и чрез използването на GLOBs. GLOBs разширяват функционалността на обикновените LO, като им добавят Грид функционалност, състояща се от специфичен Грид application слой и потребителски интерфейс. Дизайнът на GLOB е заимстван от [I.90] и съдържа следните компоненти: преглед на урок, метаданни, повторното използване на информационни обекти (re-usable information objects и съкратено RIOs) и обобщение. Метаданните се използват за намиране на GLOBs. RIO се състои от: съдържание, практически единици и оценъчни единици. Частта „съдържание" съдържа учебното съдържание на частта за обучение. Съдържанието може да се запомня в база от данни във формат като XML or HTML. Практическите единици се използват за онлайн генериране на упражнения за обучаемите, а оценъчните единици се използват за генериране на онлайн тестове за финалния изпит. RIO може да съдържа допълнителна грид функционалност, която се реализира като услуга на Грид слоя „application" и се достъпва чрез потребителския интерфейс.

2.2.4. Прилагане на подходи и средства за дедуктивен анализ при обучението по програмиране на студентите от направление „Информатика и компютърни науки" и ученици в профилираното обучение по информатика

В [II.5, II.19] е описан подход за верификация на програми (процедурни и обектно-ориентирани) по време на изпълнение, който се прилага за обучение по програмиране на студентите от бакалавърските курсове на направление „Информатика и компютърни науки". Описанието на подхода в [II.5] е пригодено за учители и студенти, както и за ученици с изявени интереси и възможности по информатика, а това в [II.19] – за студенти, завършили курсовете по увод в програмирането и обектно-ориентирано програмиране.

В [II.27, глава 12] е описана техника и методология за синтезиране (извличане на тотално коректни програми от спецификация) на програми на езика *C*++. Тя се явява обобщение на метода на преобразуващите предикати и адаптация на идеите на Д. Грис [I.84] за построяване на коректни относно зададени спецификации процедурни програми. В частност тази техника може да се използва и е използвана за формална верификация на процедурни програми в процеса на обучение на студентите по информатика и компютърни науки.

В [II.2] е описан подход за създаване на абстрактни типове данни (АТД), който освен традиционните фази – проектиране (описание на данните и описание на операциите над тях) и реализиране (избор на език за обектно-ориентирано програмиране и реализиране на проектираните операции) включва и фазата верификация и/или валидация на реализацията на АТД. За верификацията се използват техниките, описани в [II.1]. Подходът е предназначен за научни и образователни цели. Използван е при обучението по обектно-ориентирано програмиране и структури от данни и програмиране на студенти по информатика и компютърни науки във Факултета по математика и информатика на СУ.
Опитът от прилагането на подходи и средства за дедуктивен анализ при обучението по програмиране на студентите по информатика и компютърни науки е публикуван в [II.24].

2.2.4.1. Верификация по време на изпълнение

Верификацията по време на изпълнение е синтетичен метод за верификация, който комбинира формална верификация с изпълнение на програмата. Най-често проверяваните свойства се описват чрез формален модел и се вграждат в системата за тестване.

В [II.5] е описан подход за верификация по време на изпълнение за процедурни програми на езика *C*++. Подходът интегрира метода на Флойд за индуктивните твърдения [I.28, I.53, I.91] – метод за формална верификация от тип дедуктивен анализ, адаптиран за програмен код на език от високо ниво, с тестване (от тип *тестване на единична функция*).

Накратко методът на Флойд за верификация на програми се описва от следните теореми [I.28].

Теорема 3. [*Метод на индуктивните твърдения на Флойд*]. За дадена блок-схема *P*, входен предикат $\varphi(\overline{x})$ и изходен предикат $\psi(\overline{x},\overline{z})$ се извършват следните стъпки: 1) срязват се циклите; 2) намира се подходящо множество от индуктивни твърдения; 3) построяват се верифициращи условия. Ако всички верифициращи условия имат стойност *true*, то *P* е частично коректна спрямо φ и ψ.

Теорема 4. [*Метод на добре ограничените множества на Флойд*]. За дадена блок-схема *P* и входен предикат $\varphi(\overline{x})$ се извършват следните стъпки: 1) срязват се циклите и се намират „добри“ индуктивни твърдения и 2) избира се добре ограничено множество и се намират „добри“ частични функции. Ако всички условия за завършване на изпълнението са *true*, то *P* завършва изпълнението си над φ.

Предложеният в [II.5 и II.19] подход за верификация на процедурна програма се състои в следното: Верифицираната процедурна програма (функция) се разделя на части. С всяка част се свързват твърдения, доказващи частичната коректност и завършване на

изпълнението й. Твърденията се вграждат чрез *assert* техника в кода на програмата (функцията) и се проверяват по време на нейното изпълнение. В случай, че някое от твърденията пропадне, системата открива и съобщава за грешка, от където следва, че реализацията на програмата не е коректна.
Ако за някое приложение не е необходимо твърденията за верификация да са вградени в кода, чрез средства на езика е възможно да се деактивира използването на *assert* макроса.

В [II.19] е описан подход за верификация по време на изпълнение на ООП на езика C++. Подходът интегрира логика на Хоар (раздел 2.1.1), метода на преобразуващите предикати (раздел 2.1.2) и модела дизайн чрез договор (раздел 2.1.3) с тестване (от тип: тестване на единична функция, тестване на клас, тестване на група програмни модули и/или на класове).
Верификацията на класове на ООП се осъществява съгласно дефиниция 7 (раздел 2.1.3, глава 2). За верифицирания клас се определя инвариантата му, а за всеки негов метод - предусловие и постусловие. С всеки цикъл, съдържащ се в метод на класа, се свързват инварианта и функция за завършване на изпълнението на цикъла. Твърденията се записват съгласно синтаксиса на използвания език за програмиране и се вграждат в програмата чрез съответни за програмния език средства. В случая на езика C++ вграждането се осъществява чрез макроса *assert*. За верифициране на функция, която използва член-функции на класове се прилага методиката за верифициране на обикновена функция на процедурна програма [II.5, II.19].

2.2.4.2. Синтезиране на програми

От Теорема 1 и следствието от нея (раздел 2.1.2) може да се определи методология за синтезиране на условен оператор [I.84].

***Метод на преобразуващите предикати за*:**

а) синтезиране на оператора if (B) S_1; elseS_2;

1. Намират се условие B, оператори S_1 и S_2, така че да са в сила импликациите:

   $Q \wedge B \Rightarrow Wp(S_1, R)$ и

   $Q \wedge \neg B \Rightarrow Wp(S_2, R)$.
2. Проверява се дали е в сила импликацията $Q \Rightarrow domain(B)$.

б) синтезиране на оператора if – кратка форма

1. Намират се условие B и оператор S, така че да са в сила импликациите:

   $Q \wedge B \Rightarrow Wp(S, R)$ и

   $Q \wedge \neg B \Rightarrow R$.
2. Проверява се дали е в сила импликацията $Q \Rightarrow domain(B)$.

Като следствие от Теорема 2 (раздел 2.1.2) може да се определи методология за синтезиране на оператора за цикъл *while*.

Метод на преобразуващите предикати за синтезиране на оператора за цикъл while (B) S

Нека операторът за цикъл *while (B) S*; е анотиран с подходящи предусловие, инварианта, функция за завършване изпълнението на цикъла и постусловие:

{Q: предусловие}
{P: инварианта}
{t: функция за завършване изпълнението на цикъла}
while (B) S;
{R: постусловие}

Извършват се следните действия:

1. Проверява се дали е в сила Q => P. Ако това не е така, се търси оператор S_0, така че да е в сила импликацията Q => Wp(S_0, P).
2. Намира се булев израз *B*, така че импликацията $P \wedge \neg B \Rightarrow R$ да е в сила.
3. Проверява се дали импликацията $P \wedge B \Rightarrow t > 0$ е в сила.
4. Намира се оператор *S*, така че да са в сила условията:
   $P \wedge B \Rightarrow Wp(t_1 = t; S, t < t_1)$ и
   $P \wedge B \Rightarrow Wp(S, P)$.

където t_1 е помощен идентификатор.

Във формулировката на метода на преобразуващите предикати за синтезиране на условните оператори и оператора за цикъл участва **търсенето на оператори** (**оператор**), съставящи синтезираните оператори. Търсенето на оператор започва от проверка дали търсеният оператор може да е оператор за присвояване. Ако не е възможно, за търсения оператор се проверява дали може да е условен (пълна или кратка форма) и накрая се търси оператор за цикъл.

Важен компонент на метода е построяването на инвариантата на оператора за цикъл. Съществуват различни подходи за построяване на инварианта на оператора за цикъл. Най-често използвани са: отстраняване на конюнктивен член; замяна на константа с променлива; комбиниране на предусловие и постусловие [I.84].

В книгата *Програмиране на C++* [II.27, глава 12] са адаптирани резултати от [I.84] за C++ програми. Дадени са също голямо количество примери за синтезиране на програми, прилагащи почти всички структури от данни, изучавани в уводните курсове по програмиране. Освен това са приложени основните техники на формулираната методология.
Обучението по този метод се осъществява в рамките на учебната дисциплина „увод в програмирането“ . Проверката на знанията се осъществява като се използва подходът „проектно-базирано обучение“.

2.3. Заключителни бележки

В тази глава представихме някои съвременни средства, езици, подходи и методологии за проверка на коректността на процедурни и обектно-ориентирани програми, резултат от научните изследвания на автора на хабилитационния труд. Част от тях са приложени при обучението на студентите от направление „Информатика и компютърни науки" във Факултета по математика и информатика на СУ. Други подлежат да бъдат приложени. Предпоставка за провежданото обучение е паралелното обучение по математика, включващо: дискретни структури; езици, автомати, изчислимост; алгебра; математически анализ и геометрия.

Анализът на проведеното обучение по тематиката показа [II.24], че:

- основите цели на обучението се постигат главно със студентите, които са с много добра подготовка по програмиране и математика;

- в резултат от обучението у обучаемите се изгражда представа за полезността на прилагането на формални методи за верификация на програми. Изгражда се и разбиране за полезността от знания по математика за реализирането на редица приложения;

- се е повишила мотивацията на студентите с по-слаба подготовка по математика към по-задълбочено изучаване на математическите дисциплини.

Глава 3

Подходи за верификация от тип проверка на съгласуваност

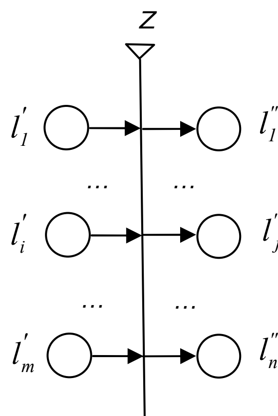
Тази глава е посветена на проверката на коректността относно зададена спецификация на процедурни и обектно-ориентирани програми. Предложени са подходи за верификация, базиращи се на проверка за съгласуваност на модели. Резултатите са публикувани в [II.14-II.18, II.20, II.23, II.25 и II26]. Направеното изследване частично решава отворен проблем, свързан с прилагане на апарата на обобщените мрежи за верифициране на процедурни и обектно-ориентирани програми, формулиран в [I.49].
Изложението на главата започва с кратко описание на използвания математически апарат. Следват описания на подходите за верификация на обектно-ориентирани и на процедурни програми.

3.1. Използван апарат

3.1.1. Обобщени мрежи

Ще дадем кратки сведения за обобщените мрежи (ОМ) – математически формализъм за изграждане на изпълними модели.
ОМ са създадени като средство за моделиране и управление на реални процеси [I.48, I.49]. Явяват се обобщение на мрежите на Петри. Разработеният за тях математически апарат ги прави ефективно и удобно средство за софтуерно моделиране. Чрез ОМ и техните разширения могат да се създават модели на програмни системи, да се дефинират спецификации на свойства, връзки между програмни модули, връзки между класове и член-функции на класове, да се доказват свойства на програмни системи.

Основни компоненти на ОМ са преходите. Графично преход се изобразява както е показано на фиг. 5.



Фиг. 5. Преход в ОМ с m входни и n изходни позиции

Дефиниция 8 (преход на ОМ).

Всеки преход се описва чрез наредена седморка от вида [I.48, I.49]:

$$Z = <L', L'', t_1, t_2, r, M, \quad >,$$

където:

- $L' = \{l_1', l_2', ..., l_m'\}$ и $L'' = \{l_1'', l_2'', ..., l_n''\}$ са крайни, непразни множества от входните и изходните позиции на прехода съответно;
- t_1 и t_2 са съответно момент на активиране на прехода и продължителност на активното му състояние;
- r е условието на прехода, определящо от кои входни към кои изходни позиции могат да преминат ядра. Задава се чрез индексирана матрица (ИМ) от вида

  $$r = \{r_{i,j}\}_{i \in [1,m]; j \in [1,n]}$$

  $r_{i,j}$ е предикат, съответстващ на i-та входна и j-та изходна позиция на прехода;
- M е ИМ, задаваща капацитетите на дъгите на прехода и има вида:

  $$M = \{m_{i,j}\}_{i \in [1,m]; j \in [1,n]}$$

  $m_{i,j} \geq 0$ е естествено число или ∞ и задава капацитета на дъгата от i-та входна позиция към j-та изходна позиция;
- е обект, имащ вид, подобен на булев израз. Нарича се тип на прехода. Когато стойността на типа на прехода, изчислен като булев израз, е „true“, преходът може да се активира, в противен случай – не може.

Дефиниция 9 (обобщена мрежа).

Обобщената мрежа [I.48, I.49] е наредена четворка от вида

$$E = <<A, \pi_A, \pi_L, c, f, \theta_1, \theta_2>, <K, \pi_K, \theta_K>, <T, t^0, t^*>, <X, \Phi, b>>,$$

където:

- A е множество от преходи;
- π_A, π_L и c са функции, дефиниращи приоритета на преходите, приоритета на позициите и капацитета на позициите, съответно;
- f е функция, която намира вярностните стойности на предикатите;
- θ_1 и θ_2 са функции, които задават съответно следващия момент от време, в който може да се активира преходът и продължителността на активното му състояние;
- K е множество от ядра;
- π_K и θ_K са функции, които задават съответно приоритетите на ядрата и моментите на постъпване на ядрата в мрежата;
- T, t^0 и t^* са величини, които задават съответно момента от време, в който ОМ започва да функционира, елементарната времева стъпка и продължителността на функционирането на ОМ;
- X е множество, съдържащо началните характеристики, с които ядрата влизат в мрежата;
- Φ е характеристична функция, която определя следващата характеристика на ядро след преместването му от входна към изходна позиция на преход;

- *b* е функция, която задава максималния брой характеристики, които трябва да се помнят за едно ядро по време на движението му в обобщената мрежа.

Горните дефиниции не са напълно формализирани, което дава възможност на всеки изследовател да дефинира компонентите по начин, съответстващ на предметната област, която е предмет на моделиране. Описанието на ОМ-модели на спецификации и програми може да не съдържа всичките компоненти на мрежата.
Алгоритми за изпълнение на преход и на ОМ са описани в [I.48, I.49].

Обобщените мрежи са избрани като средство за реализиране на описаните в главата подходи, тъй като:

- Някои спецификации, относно които се извършва верификацията на функциите на процедурните и на обектно-ориентираните програми е удобно да се представят чрез мрежови структури. ОМ предлагат формални средства за моделиране и изследване на свойства на мрежови структури.
- Създадена е методология за построяване на обобщени мрежи [I.48, I.49], с помощта на която лесно може да се построи обобщена мрежа, съответстваща на функция на процедурна или обектно-ориентирана програма.
- ОМ са средство за моделиране на паралелни процеси, което позволява ефективно паралелно да се изпълняват обобщени мрежи [I.49]. Последното е необходимо за реализиране на проверка за съгласуваност на два модела.
- Реализирани са и са в процес на усъвършенстване програмни средства за изпълнение на ОМ [I.73, I.74, I.92-I.95].

3.1.2. Други

Използва се също математическият апарат, описан в раздел 2.1 на глава 2.

3.2. Основни резултати

Основните резултати на автора на хабилитционния труд в тази област са свързани с дефинирането, реализирането и изграждането на методология и на образователни средства на подходи за верификация на процедурни и обектно-ориентирани програми [II.14–II.18, II.20, II.23, I.25 и II.26]. Отговорът на отворения проблем, свързан с верификация на програми чрез използване на ОМ и верификация на ОМ, е предложен в статията с обзорен характер [II.26].

3.2.1. Верификация на обектно-ориентирани програми

В изложението ще използваме обектно-ориентирани програми на езика C++, но това не ограничава описанието на подхода.
Основните резултати на автора, свързани с верификацията на ООП, включват:

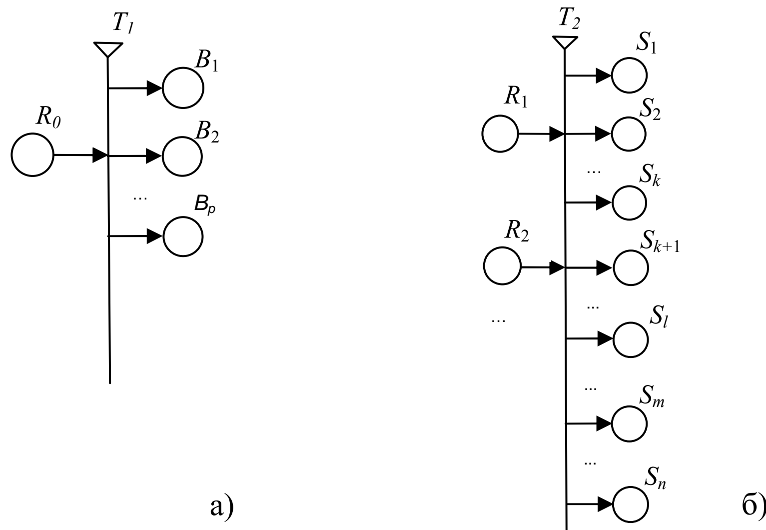
- неформално дефиниране на подход за построяване на коректни обектно-ориентирани програми [II.15];

- реализация на подхода за верификация на ООП [II.14];
- проектиране на образователна среда за верификация на ООП [II.17];
- описание на методология за прилагане на подхода за построяване на коректни ООП [II.18];
- изследване на коректността на модела на спецификацията, относно която се реализира верификацията за ООП за търсене на интерфейсни грешки [II.23]; формулиране на критерии за коректност на спецификацията [II.23].

Спецификацията, относно която се реализира верификацията за ООП за търсене на интерфейсни грешки, наричаме формален обобщеномрежов проект на клас (класове) на ООП или формален проект на класа на ООП.

3.2.1.1. Формален обобщеномрежов проект на клас на ООП

Нека обобщена мрежа се състои от един преход от вида, представен на фиг. 6 а) и множество преходи от вида от фиг. 6 б)



Фиг. 6. Дефиниция на преходи на обобщена мрежа, представяща формален ОМ проект на клас

където

$$T_1 = \langle \{R_0\}, \{B_1, B_2, \ldots, B_p\}, t_1' >$$

$$T_2 = \langle \{R_1, R_2, \ldots\}, \{S_1, S_2, \ldots, S_n\}, t_2' \rangle.$$

Условията на преходите T_1 и T_2 имат вида:

$$t_1' = \begin{array}{c|cccc} & B_1 & B_2 & \ldots & B_p \\ \hline R_0 & V_1 & V_2 & \ldots & V_p \end{array}$$

$t_2' =$

	S_1	S_2	..	S_k	S_{k+1}	..	S_l	…	S_m	…	S_n
R_1	W_1	W_2	..	W_k	W_{k+1}	..	W_l	…	W_m	…	W_n
R_2	W_1	W_2	…	W_k	W_{k+1}	..	W_l	…	W_m	…	W_n
…	…	…	…	…	…	…	…	…	…	…	…

където

V_s = *Член-функцията е конструкторът* $C_s \wedge$ *Условието* Q_s *е в сила* $(1 \le s \le p)$
W_i = *Член-функцията е* $f \wedge$ *Условието* P_i *е в сила* $(1 \le i \le k)$
W_j = *Член-функцията е* $g \wedge$ *Условието* P_j *е в сила* $(k+1 \le j \le l)$
…
W_t = *Член-функцията е* $h \wedge$ *Условието* P_t *е в сила* $(m \le t \le n)$

Предикатите $P_1, P_2, \ldots, P_k$, както и $P_{k+1}, P_{k+2}, \ldots, P_l$ и … $P_m, P_{m+1}, \ldots, P_n$, отнасящи се до прилагането на една и съща член-функция на класа, са взаимно-изключващи се (най-много един предикат може да има стойност *true*), а $f, g, \ldots, h$ са различни член-функции на класа.

Преходът от фиг. 6а) е входен преход, а позицията R_0 е входната позиция на описваната ОМ.
Всяка позиция на преход на тази ОМ представя логическо състояние, в което може да попадне обект на класа. Ще го наричаме *логическо състояние на обекта в позицията* или накратко *логическо състояние на позицията*. Логическото състояние се задава чрез булев израз.
Ядрата в позициите на тази ОМ съответстват на обекти на класа. С всеки обект на клас е свързано *множество от данни*, определено от член-данните на класа. Характеристиката на ядро задаваме чрез двойка от вида:

(*обект*, *позиция*),

където *обект* е идентификатор, задаващ име на обект на класа, а *позиция* е името на позицията, в която е ядрото (обектът) в ОМ.
Освен тези две компоненти, в характеристиката на ядро чрез параметъра *обект* неявно участва множеството от данни за обекта (наричаме ги още *данни на ядрото*), а чрез параметъра *позиция* – логическото състояние на обекта в позицията.

Дефиниция 10 [II.23]. Обобщената мрежа, дефинирана по-горе, наричаме формален ОМ проект на класа C и ще означаваме с GN_C, ако за нея са в сила следните условия:

а) За всяка позиция S на GN_C, различна от входната, е в сила импликацията

логическо състояние на позицията $S \Rightarrow pre_q$ (3)

където q е произволна член-функция на класа C, която участва в условията на прехода, на който S е входна позиция, а pre_q е предусловието на тази член-функция.

За входната позиция на GN_C, е в сила импликацията

$Default_C \Rightarrow pre_{Cs}$ (3′)

за всеки конструктор C_s, $(1 \le s \le p)$. pre_{Cs} е предусловието на конструктора C_s.

б) За всяка позиция S на GN_C, различна от входната е в сила импликацията

логическо състояние на позицията S => Inv (4)

Inv е инвариантата на класа *C*.

в) За всяка двойка от входна и изходна позиции (R, S) на преход, различен от прехода, реализиращ изпълнението на конструктора (конструкторите) на класа *C*, с условие на прехода

Член-функцията е q $\wedge$ *Условието P е в сила*

е в сила тройката на Хоар

{ *логическо състояние на позицията R* $\wedge$ *P*}
$Body_q$ (5)
{ *логическо състояние на позицията S* }

За всяка двойка от входна и изходна позиции (R_0, B_s), $1 \leq s \leq p$, на прехода, реализиращ изпълнението на конструктора (конструкторите) на класа с условие на прехода

Член-функцията е конструкторът C_s $\wedge$ *Условието* Q_s *е в сила*

е в сила тройката на Хоар
{ $Default_C \wedge Q_s$ }
$Body_{Cs}$ (5′)
{ *логическо състояние на позицията* B_s }

Условията (3) и (3′) осигуряват, във входните позиции на преходите на GN_C да са в сила предусловията на всички член-функции, които съответният преход би могъл да изпълни. Условие (4) осигурява верността на инвариантата на класа за текущите стойности на данните на ядрата във всяка позиция на мрежата, различна от входната, а условията (5) и (5′) гарантират запазването на верността на предиката *логическо състояние на позиция* за текущите стойности на данните на ядрата.

За спецификация на формалния мрежов проект GN_C на класа *C* избираме спецификацията на класа *C*.

Дефиниция 11 [II.23]. Формалният ОМ проект GN_C на класа *C* е коректен относно спецификацията си ако:

а) За прехода T_1 (фиг. 6.а), реализиращ изпълнението на конструктора (конструкторите) на класа *C* е в сила:

За всяка двойка (R, B_s) от входна и изходна позиции на прехода с условие

Член-функцията е конструкторът C_s $\wedge$ *Условието* Q_s *е в сила*

е в сила тройката на Хоар

$\{ Default_C \wedge pre_{Cs}(x_{Cs}) \wedge Q_s \} \mathrm{Body}_{Cs} \{ post_{Cs}(x_{Cs}) \wedge Inv \}$ (6)

където x_{Cs} е множеството от допустими аргументи на конструктора C_s, а останалите означения са тези от дефиниция 6 а), $1 \leq s \leq p$.

б) За всеки преход от вида T_2 (фиг 6.б), различен от прехода, реализиращ изпълнението на конструктора (конструкторите) на класа е в сила:

За всяка двойка (R, S) от негови входна и изходна позиции с условие на прехода

Член-функцията е q $\wedge$ *Условието P е в сила*

е в сила тройката на Хоар

$$\{ pre_q(x_q) \wedge Inv \wedge P \} \quad Body_q \quad \{ post_q(x_q) \wedge Inv \} \tag{7}$$

където x_q е множеството от допустими аргументи на член-функцията q.

Условията (6) и (7) са аналогични на условия (1) и (2) от дефиниция 7 съответно.

Теорема 5 [II.23]. Ако класът C е коректен относно спецификацията си, формалният ОМ проект GN_C на класа C също е коректен относно спецификацията си.

Доказателство. Следва от дефиниции 10 и 11 и прилагане на правилото за следствието

$$\frac{P \Rightarrow P', \{P'\}\ S\ \{Q'\}, Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

Дефиниция 12 [II.23]. Формалният ОМ проект GN_C на класа C съответства на реализацията на класа C, ако е коректен относно спецификацията на C.

3.2.1.2. Дефиниране на подход за проверка на коректността на ООП

Подходът се състои в разделяне на верификацията на класовете на ООП от верификацията на функцията, която ги прилага. Верификацията на класовете се осъществява съгласно дефиниция 7 (раздел 2.1.3, глава 2).
За верификацията на функция, която прилага класове на ООП се реализират следните действия:

- изгражда се *ОМ* модел за всеки клас, който модел дефинира връзките между методите на класа във вид на коректни редици от извиквания, а също *ОМ* модел на връзките между класовете на програмата. Тези модели дефинират спецификацията, относно която ще се извършва верификацията на функция на ООП.
- Функцията на ООП, която ще бъде верифицирана относно описаната по-горе спецификация, се представя чрез обобщена мрежа.
- Обобщените мрежи на функцията и на спецификацията, произтичаща от връзките между класовете и връзките между член-функциите на всеки клас, се изпълняват паралелно с цел установяване дали моделът на функцията съответства на (удовлетворява) зададената спецификация.

Чрез този подход могат да се търсят както грешки в класовете на ООП, така и интерфейсни грешки в програмата.

За да се избегне усложняване на описанието следва дефиниране на подхода в най-простия случай, когато ООП се състои от един клас, ще го означаваме със C, и функция M, която използва класа.

Създаването на коректна относно зададена спецификация ООП преминава през следните стъпки:

1) *Създаване на формален обощеномрежов модел на класа*

На тази стъпка се изгражда спецификацията, относно която ще се верифицира ООП. За целта се дефинират:

- *част от формалната спецификация на класа:* инвариантата на класа и предусловието на всяка негова член-функция;
- *обобщеномрежов модел, специфициращ връзките между член-функциите на класа* и се проверява дали този ОМ модел е *формален ОМ модел на класа.*

В следващото изложение формалният *ОМ* модел на класа *C* ще означаваме с GN_C. Той дефинира възможните коректни начини за използване на член-функциите на класа. Всяка функция, която прилага класа трябва да има поведение, което съответства на формалния му ОМ модел.

2) *Дефиниране на формална спецификация на класа и верификация на класа относно дефинираната спецификация*

За реализирането на тази стъпка се извършват следните действия:

- *доизграждане на формалната спецификация на класа*

Задава се постусловието на всяка негова член-функция. За операторите за цикъл в член-функциите на класа се дефинират инварианти и функции за завършване на изпълнението на циклите.

- *верификация на класа относно дефинираната спецификация*

За целта за всяка член-функция на класа се построява теорема във вид на тройка на Хоар (1) и (2) (дефиниция 7, раздел 2.1.3 на глава 2). Доказателството на теоремите може да се реализира ръчно, а също като се използва средата за верификация на програми, описана в раздел 2.2.2 на глава 2.

Тази и предходната стъпки са сред най-трудните, но веднъж реализирани могат да се използват за верифициране на всички функции, които използват класа. Извършват се от програмиста на класа.

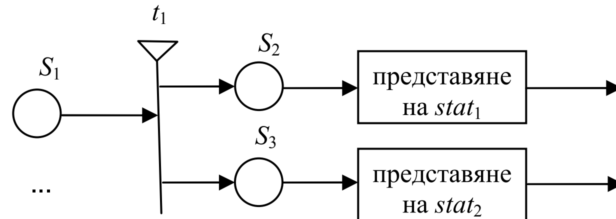
Съгласно дефиниция 12 и теорема 5, след изпълнението на тези две стъпки, е в сила, че формалният ОМ проект GN_C на класа *C* съответства на реализацията на класа *C*.

3) *Дефиниране на модел на функцията*

На тази стъпка се построява ОМ модел, който съответства на функцията на ООП, която предстои да бъде верифицирана относно построената на стъпки 1) и 2) спецификация. Осъществява се от програмиста, който използва класа.
Нека *M* е името на верифицираната функция.

За простота на описанието ще приемем, че *M* съдържа оператори за: вход/изход, присвояване (не съдържащи обръщения към член-функции на класа или към външни за класа функции), разклоняване на изчислителния процес (*if*, *if-else*, *switch*), цикъл (*while* и *do-while*) и обръщения към член-функции на класа *C*. Обобщеномрежовият модел на функцията *M* ще означаваме чрез GN_M. Всеки преход на GN_M съответства на изпълнение на един или няколко оператора на *M*. Ако преход изразява изпълнение на член-функция на класа, с него се свързва и името на обекта, чрез който е активирана член-функцията. На всяка използвана в член-функцията променлива, различна от обект на класа, съответства също ядро, което се премества от позиция в позиция в съответната ѝ обобщена мрежа, но тези ядра и движението им не са предмет на разглеждане, защото не са пряко свързани със спецификацията, относно която се извършва верификацията.

На операторите за вход/изход, за присвояване и за обръщение към функция или член-функция на клас съответства преход с една или повече входни позиции и една изходна позиция. На условния оператор *if* (*B*) $stat_1$; *else* $stat_2$; съответства обобщеномрежов елемент със структура от вида

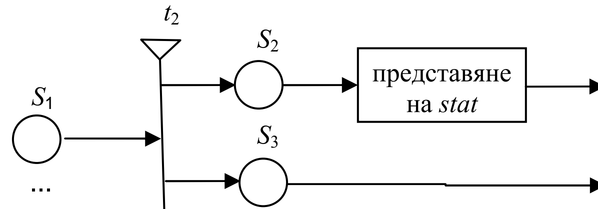


където

$$t_1 = \langle \{S_1, \ldots\}, \{S_2, S_3\}, r_1 \rangle$$

$$r_1 = \begin{array}{c|cc} & S_2 & S_3 \\ \hline S_1 & B & \neg B \\ \ldots & \ldots & \ldots \end{array}$$

На условния оператор *if* (*B*) *stat;* съответства обобщеномрежов елемент със структура от вида

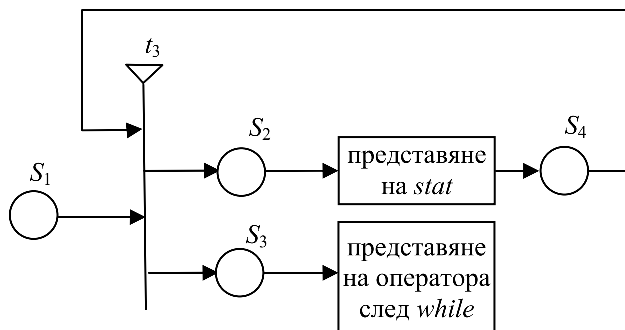


където

$$t_2 = \langle \{S_1, \ldots\}, \{S_2, S_3\}, r_2 \rangle$$

$$r_2 = \begin{array}{c|cc} & S_2 & S_3 \\ \hline S_1 & B & \neg B \\ \ldots & \ldots & \ldots \end{array}$$

На оператора за цикъл *while* (*B*) *stat* съответства обобщеномрежов елемент със структура от вида

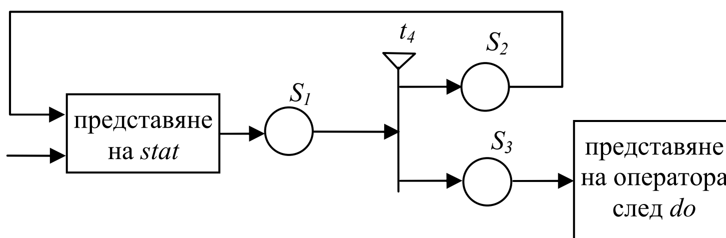


където

$$t_3 = \langle \{S_1, S_4\}, \{S_2, S_3\}, r_3 \rangle$$

$$r_3 = \begin{array}{c|cc} & S_2 & S_3 \\ \hline S_1 & B & \neg B \\ S_4 & B & \neg B \end{array}$$

Операторът за цикъл *do stat while* (*B*); може да се представи чрез обобщеномрежов елемент със структура от вида



където

$$t_4 = \langle \{S_1, \ldots\}, \{S_2, S_3\}, r_4 \rangle$$

$$r_4 = \begin{array}{c|cc} & S_2 & S_3 \\ \hline S_1 & B & \neg B \\ \ldots & \ldots & \ldots \end{array}$$

4) Проверка за съвместимост на модела на функцията и формалния проект на класа

След построяването на GN_M се проверява дали обръщенията към член-функциите на класа *C* в GN_M съответстват на формалния проект GN_C на класа. Установяването на несъответствие между моделите е сигнал за грешка във функцията *M*.

Ще представим накратко идея за реализирането на проверката за съответствието. За целта отначало ще разгледаме случая когато във верифицируемата функция *M* е дефиниран само един обект, а след това − когато в *M* са дефинирани повече от един обекти.

а) *Нека в M е дефиниран само един обект*

Изпълняват се двата ОМ модела – GN_M и GN_C. Изпълнението започва с изпълнение на GN_M. Активира се ядро без начална характеристика във входната позиция на GN_M.

Ядрото се премества от позиция в позиция на GN_M до достигане до входна позиция на преход, съответстващ на изпълнение на член-функция на класа C. Ако член-функцията не е конструктор, е възникнало несъответствие между двата модела. Ако член-функцията е конструктор на класа C се активира ново ядро също без начална характеристика във входната позиция на обобщената мрежа GN_C. Едновременно се изпълняват преходите в GN_M и GN_C, съответстващи на изпълнение на съответния конструктор. В резултат, ядрата се преместват в съответни изходни позиции на тези преходи и получават характеристики от вида (*име_на_ядро*, *позиция_в_*GN_M) и (*име_на_ядро*, *позиция_в_*GN_C). Имената на двете ядра са еднакви и съвпадат с името на дефинирания обект.

Проверката дали моделът GN_M съответства на модела GN_C продължава с изпълнение на модела на М. Възможни са два сценария:

- *Ядрото на GN_M е в позиция, която е входна на преход с условия, които не са свързани с член-функции на класа C*

В този случай след изпълнението на прехода характеристиката на ядрото на GN_M променя само позицията си, а ядрото в мрежата GN_C не се премества от текущата си позиция.

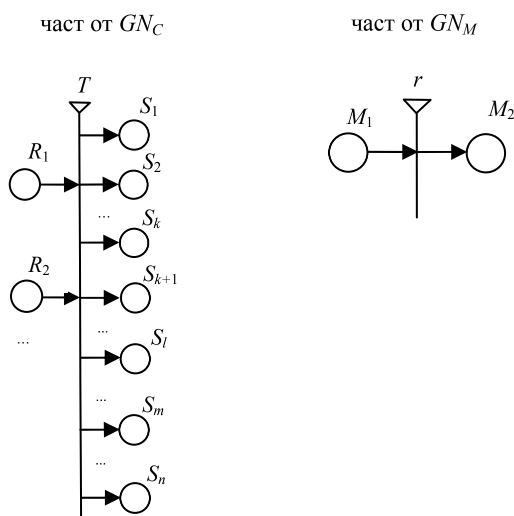
- *Ядрото на мрежата GN_M е в позиция, която е входна за преход, свързан с член-функция на класа C*

Тогава преходът ще се изпълни, ако едновременно са в сила:

- условието на прехода в мрежата GN_M, свързано с член-функцията и
- точно едно от условията на прехода с входна позиция, съвпадаща с текущата позиция на ядрото в мрежата GN_C.

Второто условие може да се разглежда като разрешение за изпълнение на обръщението към член-функцията за да бъде то коректно. Ако преходът в мрежата GN_M се изпълни, се изпълнява и преходът в мрежата GN_C.

Например, ако в частите от обобщени мрежи по-долу са активирани ядра в позициите R_1 и M_1



където преходът T се дефинира по същия начин, както е дефиниран преходът T_2 от фиг. 6 б), а преходът $r = \langle\{M_1\}, \{M_2\}, r'\rangle$, където

$$r' = \begin{array}{c|c} & M_2 \\ \hline M_1 & \textit{Член-функцията е } f \wedge Q \end{array}$$

Преходът r ще се изпълни ако са в сила условия Q и P_1, или Q и P_2, или ..., или Q и P_k. В случай, че r може да се изпълни, се изпълнява и преходът T. Ядрото на GN_C ще се премести в позиция S_1 (ако е в сила P_1) или в позиция S_2 (ако е в сила P_2) или ... в позиция S_k (ако е в сила условие P_k). Ако в условието на прехода T не се съдържа предикатът „Член-функцията е f", преходът r няма да се изпълни. Последното съответства на „съгласно спецификацията, представена чрез формалния мрежов проект на класа C в съответното място на функцията M обръщението към член-функцията f не е допустимо (не е коректно)".
Допълнителното изискване за изпълнение на преход в мрежата GN_M на верифицираната функция M осигурява за използваните член-функции на класа да са в сила предусловията им и коректен резултат след изпълнението им.

Освен посочените по-горе несъответствия, сигнализиращи за съществуващи грешки, ако съществува изпълнение на GN_M, което не завършва (стига се до преход, който не може да се изпълни), функцията M не е коректна относно зададената спецификация. От позицията в GN_M, в която е спряло изпълнението й, може да се разбере мястото във функцията M, където е грешката и след анализ да се поправи кодът.

б) *Нека в М са дефинирани повече от един обекти*

Тогава в мрежите GN_M и GN_C ще има повече от едно ядра, съответстващи на обекти на класа. Всяко изпълнение на преход в GN_M, съдържащ обръщение към конструктор на класа предизвиква активиране на две нови ядра – едно в текущата входна позиция на прехода на GN_M, съответстващ на изпълнението на конструктор, и друго във входната позиция на GN_C. Тези ядра имат еднакви имена, съвпадащи с името на обекта, който е създаден. След изпълнението на преходите, които съдържат обръщението към конструктора на класа тези ядра получават характеристики от определения по-горе вид. Всяко ядро има период на активност, започващ от изпълнението на съответния преход, изпълняващ конструктор и завършващ след изпълнението на преход, изпълняващ деструктора на класа.

Ядро се създава в позиция, която може или да е празна, или в нея може да са натрупани ядра. Всички ядра на мрежата GN_M са събрани в една позиция и се преместват едновременно при изпълнение на преходите. Броят им расте при изпълнение на конструктор и намалява – при изпълнението на деструктора на класа. Съответните ядра в GN_C могат да са в една позиция, но може да са разположени в произволни позиции на мрежата.

В случай, че в позиция на мрежата GN_M има няколко ядра са възможни:

- *Позицията е входна за преход, извършващ действие, което не е свързано с класа C*

Тогава, ако преходът на GN_M може да се изпълни, се изпълнява и всички ядра се преместват от входната в съответната изходна позиция на прехода. Съответните им едноименни ядра в мрежата GN_C не променят позициите си.

- *Позицията е входна за прехода r, извършващ действие, свързано с изпълнение на член-функция на класа C, различна от конструктор*

Нека член-функцията, която е свързана с прехода *r*, да е активирана с обекта *p* на класа *C*. В този случай, ако във входната позиция на *r* няма ядро с име *p*, преходът няма да се изпълни – възниква грешка. В противен случай преходът ще се изпълни, ако са в сила двете условия, разгледани в случая, когато във функцията *M* е дефиниран само един обект. Тогава всички ядра на GN_M се преместват в изходната позиция на прехода (съгласно условията на прехода). Ядрото на GN_C с име *p* променя позицията си в GN_C в съответствие с условията на изпълнилия се за него преход, а останалите ядра на GN_C не променят позициите си.

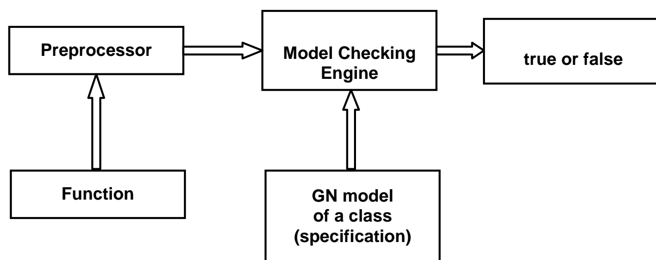
- *Позицията е входна за прехода r, извършващ действие, свързано с изпълнение на конструктор на C*

Този случай разгледахме по-горе.

Детайлно описание на подхода е представено в следващия раздел на този труд. Пример, показващ изпълнението на описания подход за верификация е даден в [II.15].

3.2.1.3. Реализация на подхода

За реализирането на подхода е проектирана програмна система, архитектурата на която е дадена на фиг. 7 [II.14].
Компонентата *Function* задава функцията на ООП, която подлежи на верифициране. *Preprocessor* генерира обобщената мрежа, която съответства на функцията, която ще бъде верифицирана. Модулът Model Checking Engine получава като вход ОМ, получена като резултат от работата на preprocessor-а и формалния ОМ проект на класа (спецификацията) и определя дали функцията удовлетворява формалната спецификация (т.е. дали двете мрежи си съответстват). В случай, че резултатът от проверката е отрицателен model checker-ът съобщава причината за грешката и посочва мястото, където тя е възникнала.



Фиг. 7. Архитектура на Model Checker

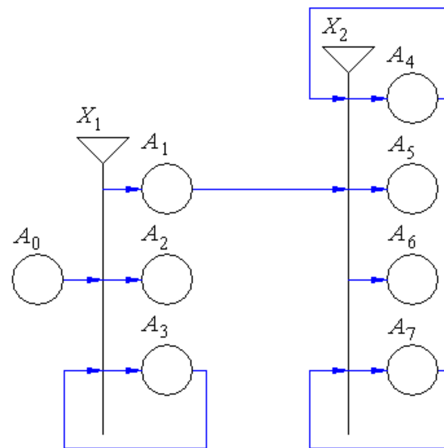
Създаването на формалния ОМ проект на класа (спецификацията) изисква задълбочено познаване на класа, на възможните връзки между член-функциите му, както и владеене на техниките за конструиране на ОМ-и.

Ще представим два варианта на алгоритъма, който *Model Checking Engine* реализира в най-простия случай, когато ООП се състои от един клас *C* и функция *M*, която го прилага.

Вариант 1

(функцията *M*, дефинира и използва само един обект на класа *C*)

Нека GN_C е формалният ОМ проект на класа *C*, а GN_M – ОМ-моделът на функцията *M*. Тъй като в *M* е дефиниран само един обект, всяка от мрежите GN_M и GN_C ще съдържа точно по едно ядро, което ще съответства на обекта. Мрежата GN_M може да съдържа още ядра (съответстващи на локалните променливи, дефинирани в *M*), но тъй като те не са свързани с обекта на класа *C*, не са предмет на разглеждане. Ако *q* е името на обекта в *M*, характеристиките на ядрата на GN_M и GN_C ще са двойки от вида (*q*, име_на_позиция_в_GN_M) и (*q*, име_на_позиция_в_GN_C) съответно.
Обобщената мрежа GN_1, описана на фиг. 8, дефинира алгоритъма за проверка дали GN_M съответства на GN_C.



Фиг. 8. Алгоритъм за проверка за консистентност в случая когато функцията дефинира и използва само един обект на класа

Дизайнът й е резултат от обработка с графичния редактор на ОМ *Gennete*. Обобщената мрежа GN_1 се състои от два прехода. Преходът X_1 извършва инициализиращи действия, а X_2 – проверката за съответствието (консистентността). Мрежата GN_1 също ще има само едно ядро. Характеристиката му е определена като четворка от вида (име_на_ядрото, име_на_позиция_в_GN_1, име_на_позиция_в_GN_M, име_на_позиция_в_GN_C), където името_на_ядрото в случая е *q*, име_на_позиция_в_GN_1 е името на позицията на ядрото *q* в GN_1, а име_на_позиция_в_GN_M и име_на_позиция_в_GN_C са имената на позициите на едноименните на *q* ядра в GN_M и GN_C съответно.
Следват дефинициите на преходите на GN_1.

$$X_1 = <\{A_0, A_3\}, \{A_1, A_2, A_3\}, r_1>$$

$$r_1 = \begin{array}{c|ccc} & A_1 & A_2 & A_3 \\ \hline A_0 & P_1 & P_2 & P_3 \\ A_3 & P_1 & P_2 & P_3 \end{array},$$

P_1 = Текущите за изпълнение преходи на GN_M и GN_C съдържат условие от вида „Член-функцията е конструктор на C“ и могат да се изпълнят едновременно.

$P_2 = \neg P_1 \wedge \neg P_3$

P_3 = Текущият за изпълнение преход на GN_M не съдържа условие, свързано с член-функция на класа C.

$$X_2 = <\{A_1, A_4, A_7\}, \{A_4, A_5, A_6, A_7\}, r_2>$$

$$r2 = \begin{array}{c|cccc} & A_4 & A_5 & A_6 & A_7 \\ \hline A_1 & P_4 & P_5 & P_6 & P_7 \\ A_4 & P_4 & P_5 & P_6 & P_7 \\ A_7 & P_4 & P_5 & P_6 & P_7 \end{array},$$

P_4 = Ядрата на GN_M и GN_C са в позиции, входни за преходи, които могат да се изпълнят едновременно, заради наличие на условия на преходите, които са в сила едновременно.

P_5 = Изпълненията на мрежите GN_M и GN_C са завършили и нямат активни ядра.

$P_6 = \neg P_4 \wedge \neg P_5 \wedge \neg P_7$

$P_7 = P_3$

Изпълнението на GN_1 започва с активиране на ядро в позиция A_0, входна за прехода X_1, което няма начална характеристика. Паралелно с това започва изпълнението и на мрежите GN_M и GN_C. Във входните позиции на GN_M и GN_C също се активират ядра без начални характеристики.

Изпълнение на прехода X_1

В случай, че е в сила предикатът P_1, се изпълняват преходът X_1 на GN_1 и текущите за изпълнение преходи на GN_M и GN_C. В резултат ядрата на GN_M и GN_C се преместват в съответните изходни за изпълнените преходи позиции. Нека имената на новите им позиции са M_i и S_j. Ядрата на GN_M и GN_C получават характеристиките (q, M_i) и (q, S_j) съответно. Ядрото на GN_1 се премества в позиция A_1 и получава за характеристика четворката (q, A_1, M_i, S_j), съдържаща името на обекта и имената на позициите A_1, M_i и S_j, в които са ядрата в мрежите GN_1, GN_M и GN_C съответно. Изпълнението на прехода X_1 завършва, след което предстои изпълнение на прехода X_2 на обобщената мрежа GN_1.

В случай, че е в сила предикатът P_3, се изпълняват преходът X_1 на GN_1 и текущият преход на GN_M. Не се изпълнява преходът на GN_C, в чиято входна позиция е ядрото. В резултат ядрото на GN_M се премества в съответната изходна позиция (нека името й да е M_p) и получава характеристиката (q, M_p). Ядрото на GN_C не се премества и нека е в

позиция с име S. Ядрото на GN_1 се премества в позиция A_3 и характеристиката му става (q, A_3, M_p, S). Тъй като позиция A_3 е входна за прехода X_1, следващата стъпка в изпълнението на GN_1 е повторно изпълнение на X_1.

В случая, когато е в сила предикатът P_2, т.е. не са в сила условията P_1 и P_3, GN_1 моделът GN_M не съответства на GN_C и изпълнението на GN_1 завършва. Условията на прехода X_1 указват причината за несъответствието, което позволява да се намери грешката във функцията M.

Изпълнение на прехода X_2

В случай, че е в сила предикатът P_4, се изпълняват преходът X_2 и преходите с входни позиции, в които са ядрата на GN_M и GN_C. В резултат ядрата на GN_M и GN_C се преместват и нека новите им позиции са M_n и S_m съответно. Характеристиките им имат вида (q, M_n) и (q, S_m). Ядрото на GN_1 се премества в позиция A_4 и получава за характеристика четворката (q, A_4, M_n, S_m), съдържаща името на обекта и имената на позициите A_4, M_n и S_m, в които са ядрата с име q в мрежите GN_1, GN_M и GN_C, съответно. Тъй като позиция A_4 е входна на прехода X_2, изпълнението на мрежата GN_1 продължава с поредно изпълнение на прехода X_2.

В случай, че е в сила предикатът P_7, се изпълняват преходът X_2 на GN_1 и преходът на GN_M, който има за вход позицията, в която е ядрото. Не се изпълнява преход на GN_C. В резултат ядрото на GN_M се премества в изходна за изпълнения преход позиция, която означаваме с M_k. Ядрото на GN_C остава в текущата си позиция и нека тя е S_l. Ядрото на GN_1 се премества в позиция A_7 и получава характеристиката (q, A_7, M_k, S_l). Изпълнението на GN_1 продължава с изпълнение на прехода X_2.

В случая, когато е в сила предикатът P_5, ядрото на GN_1 се премества в позиция A_5. В този случай обобщеномрежовият модел на функцията M съответства на формалния мрежов проект на класа. Това означава, че не е намерена интерфейсна грешка във функцията M.

Когато е в сила условие P_6, т.е. не е в сила нито едно от условията P_4, P_5 и P_7, GN_M не съответства на спецификацията GN_C. Ядрото на GN_1 преминава в позиция A_6 като променя само втория елемент на характеристиката си. Условието на прехода задава причините за несъответствието, а характеристиката на ядрото в позиция A_6 показва местата в GN_C и GN_M, в които е възникнало несъответствието. Тази информация позволява да се намери мястото и причината за грешката във функцията M.

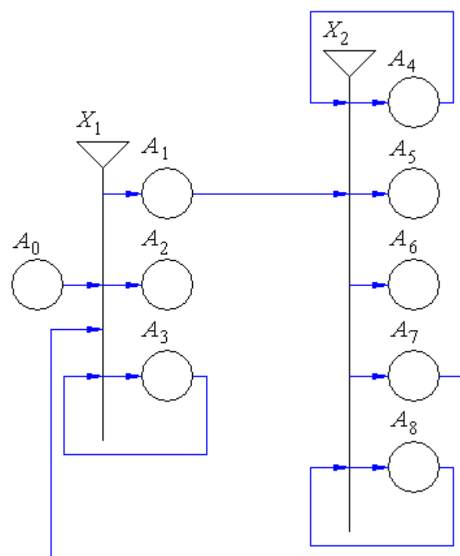
Вариант 2

(функцията M, дефинира и използва повече от един обект на класа C)

Обобщената мрежа GN_2, описана на фиг. 9, дефинира алгоритъма за проверка дали GN_M съответства на GN_C в този случай. Обобщените мрежи GN_2, GN_M и GN_C ще имат по толкова ядра, свързани с обекти, колкото е броят на дефинираните обекти във функцията M. Мрежата GN_M може да има и други ядра, но тъй като те не са свързани с обекти на класа (а с обикновени променливи), не са предмет на разглеждане. Множествата от ядрата на обобщените мрежи GN_2, GN_M и GN_C означаваме с K, K_M и K_C съответно. Характеристиките на ядрата от K_M и K_C са двойки от вида (име_на_ядро,

име_на_позиция_в_GN_M) и (име_на_ядро, име_на_позиция_в_GN_C) съответно, а характеристиката на ядро от K е четворка от вида (име_на_ядро, име_на_позиция_в_GN_2, име_на_позиция_в_GN_M, име_на_позиция_в_GN_C) и за всяко ядро задава името му, името на позицията му в GN_2 и имената на позициите на едноименните му ядра в GN_M и GN_C. Приоритетът на ядрата се определя от реда, в който се дефинират и използват обектите във функцията M. Всеки преход на мрежата GN_M, който е свързан с обръщение към член-функция на класа C е свързан и с името на обекта, за който се извършва обръщението към член-функцията. Предстоящото изпълнение на такъв преход на GN_M определя името на активните ядра в трите мрежи.

Подобно на GN_1, мрежата GN_2 се състои от два прехода. Преходът X_1 извършва инициализиращи действия, а X_2 – проверката на съответствието.



Фиг. 9. Алгоритъм за проверка за консистентност в случая когато функцията дефинира и използва повече от един обект на клас

Следват дефинициите на преходите на обобщената мрежа GN_2.

$$X_1 = \langle \{A_0, A_3, A_7\}, \{A_1, A_2, A_3\}, r_1 \rangle$$

$$r_1 = \begin{array}{c|ccc} & A_1 & A_2 & A_3 \\ \hline A_0 & P_1 & P_2 & P_3 \\ A_3 & P_1 & P_2 & P_3 \\ A_7 & P_1 & P_2 & false \end{array},$$

P_1 = Преходите с входни позиции, в които са активираните ядра на GN_C и GN_M съдържат условие от вида „Член-функцията е конструктор на класа C“ и могат да се изпълнят едновременно.

$P_2 = \neg P_1 \wedge \neg P_3$

P_3 = Активното ядро на GN_M е в позиция, която е входна за преход, който не съдържа условие, свързано с член-функция на класа C.

$$X_2 = <\{A_1, A_4, A_8\}, \{A_4, A_5, A_6, A_7, A_8\}, r_2>$$

$$r2 = \begin{array}{c|ccccc} & A_4 & A_5 & A_6 & A_7 & A_8 \\ \hline A_1 & P_4 & P_5 & P_6 & P_7 & P_8 \\ A_4 & P_4 & P_5 & P_6 & P_7 & P_8 \\ A_8 & P_4 & P_5 & P_6 & P_7 & P_8 \end{array},$$

P_4 = Активираните едноименни ядра на GN_M и GN_C са в позиции, входни за преходи, които не изпълняват конструктори на класа C и които могат да се изпълнят едновременно, заради наличие на условия на преходите, които са в сила едновременно.

P_5 = Изпълненията на мрежите GN_M и GN_C са завършили и в тях няма активни ядра.

$P_6 = \neg P_4 \wedge \neg P_5 \wedge \neg P_7 \wedge \neg P_8$

P_7 = Активното ядро на GN_M е в позиция, която е входна за преход, изпълняващ конструктор на класа C.

$\mathrm{P}_8 = \mathrm{P}_3$

Изпълнението на GN_2 започва с активиране на ядро в позиция A_0, входна за прехода X_1, което няма начална характеристика. Паралелно с това започва изпълнението и на мрежите GN_M и GN_C. Във входните им позиции също се активират ядра без начални характеристики.

Изпълнение на прехода X_1

В случай, че е в сила предикатът P_1, се изпълняват преходът X_1 и преходите с входни позиции, в които са активираните ядра на GN_M и GN_C. В резултат активираните ядра на GN_M и GN_C се преместват в съответните изходни за изпълнените преходи позиции. Нека ги означим с $\mathrm{M_i}$ и S_j. След предвижването ядрата получават характеристиките (q, M_i) и (q, S_j) съответно, където q е името на обекта, който е дефиниран след изпълнението на конструктора. Ядрото на GN_2 се премества в позиция A_1 и получава за характеристика четворката (q, A_1, M_i, S_j), съдържаща името на обекта и имената на позициите A_1, M_i и S_j, в които са активираните ядра в мрежите GN_2, GN_M и GN_C, съответно. Изпълнението на прехода X_1 завършва, след което предстои изпълнение на прехода X_2 на GN_2.

Случаят, когато предикатът P_3 е в сила, е аналогичен на описания за мрежата GN_1.

В случай, че е в сила предикатът P_2, мрежата GN_M не съответства на GN_C и изпълнението на GN_2 завършва. Условията на прехода X_1 указват причината за несъответствието, а характеристиката на ядрото (ако има такава) задава позициите в GN_M и GN_C, в които то е възникнало.

Изпълнение на прехода X_2

В случай, че е в сила предикатът P_4, се изпълняват преходът X_2 и преходите с входни позиции, в които са активираните ядра на GN_M и GN_C. Нека името на активното ядро е q и това ядро в мрежите GN_M и GN_C се е предвижило в позициите M_n и S_m съответно. Тогава техните характеристики са (q, M_n) и (q, S_m). Ядрото q на GN_2 се предвижва в позиция A_4 и получава за характеристика четворката (q, A_4, M_n, S_m). Тъй като позиция A_4 е входна на прехода X_2, изпълнението на мрежата GN_2 продължава с поредно изпълнение на прехода X_2.

В случай, че е в сила предикатът P_8, се изпълнява преходът X_2 на GN_2 и преходът на GN_M, който има за вход позицията, в която е активното ядро и не се изпълнява преход на GN_C. В резултат едноименното активирано ядро на M се предвижва в съответната изходна позиция (нека тя е M_k). Едноименното ядро на GN_C остава в текущата си позиция (нека тя е S_l). Ядрото на GN_2 се премества в позиция A_8 и получава характеристиката (q, A_8, M_k, S_l), където q е името на активното ядро. Изпълнението на GN_2 продължава с изпълнение на прехода X_2.

В случая, когато е в сила предикатът P_5, ядрото на GN_2 се премества в позиция A_5 и загубва характеристиката си. В този случай обобщената мрежа на функцията съответства на формалния мрежов проект на класа. Последното показва, че функцията M е коректна относно указаната чрез GN_C спецификация.

В случая, когато е в сила предикатът P_7 текущите активирани ядра в трите мрежи се деактивират. Активират се три нови ядра без начални характеристики, които съответстват на обекта, който ще се създаде след обръщението към конструктора. За целта отначало се изпълнява преходът X_2. В резултат всички текущи деактивирани ядра на GN_2 се преместват в A_7 и към тях в A_7 се активира новото ядро без начална характеристика. Активира се ядро в текущата входна позиция в GN_M, в която е било последното активирано ядро и още едно ядро - във входната позиция на мрежата GN_C. Следващото действие е да се изпълнят преходът X_1 на GN_2 и преходите в GN_M и GN_C, съответстващи на изпълнение на конструктор, ако последното е възможно.

Когато е в сила условие P_6, т.е. не са в сила условията P_4, P_5, P_7 и P_8 на прехода X_2, GN_M не съответства на GN_C. Ядрото на GN_2 преминава в позиция A_6 като променя само втория елемент на характеристиката си. Условията на прехода задават причините за несъответствието, а характеристиката на ядрото в позиция A_6 показва местата в GN_C и GN_M, в които е възникнало несъответствието. Интерпретацията на този резултат е, че във функцията M е намерена грешка.

Анализът на ефективността на обучението по програмиране и най-вече на най-масово използваното – обектно-ориентираното показва, че трябва да се разработят образователни средства, повишаващи ефективността му [I.96-I.98]. Създаването на автоматизирани инструменти за подпомагане на учебния процес [I.99], обучението по прилагане на формални методи за построяване на коректно ПО, започващо от най-ранните курсове по езици за програмиране са крачка в посока на увеличаването на ефективността на обучението по обектно-ориентирано програмиране.

В следващата част от изложението се описват елементи от създаването на такива инструменти за целите на обучението по програмиране.

3.2.1.4. Образователна среда за проверка на коректността на ООП

Верификацията на програми, включително алгоритмите за формална верификация и валидация на софтуерни компоненти и системи е важна тема в учебните програми за специалностите компютърни науки и информатика [I.100]. Студенти, завършили тези програми и стремящи се към специализации по софтуерно инженерство, управление на проекти, както и към професията системен анализатор, трябва да бъдат насърчавани да се справят с увеличаващата се сложност на софтуерните проекти и произтичащите от тях системи.

Обектно-ориентираното програмиране е признато като един от най-успешните методи за разработка на софтуер и е широкоприето като стандарт за сериозен дизайн и реализация на софтуерни системи. Студентите от специалностите компютърни науки и информатика се обучават по ООП още в най-ранните години на университетското си образование. По време на това обучение, обаче, малко внимание се обръща на специфични подходи и техники за верификация на обектно-ориентирани програми. Поне частично оправдание за това е липсата на подходящи образователни среди и инструменти за улесняване на тези образователни усилия.

С цел преодоляване на този недостатък при обучението по ООП е разработена образователна среда [II.17], базираща се на описания подход. С прилагането ѝ ще се подпомогне придобиването на знания и умения в областта на формалните методи за верификация на обектно-ориентирани програми. Предназначена е за студенти от бакалавърските класове (започвайки от първокурсници и второкурсници), до студенти от магистърските програми в областта на софтуерното инженерство и докторанти. Средата може да се използва и от софтуерни инженери и програмисти с цел обучение и прилагане при реализирането на качествен софтуер.
Тя поддържа три нива на достъп на: начинаещи, междинни и напреднали потребители.
Основните знания и предварително необходимите умения, свързани с всяко ниво на достъп, са добре определени и отразени в интерфейса на взаимодействието, който средата предоставя на потребителите. За пълнота, в средата са включени описания на прилаганите формални методи за верификация, заедно с основните математически средства, свързани с метода за верификация.

а) *ниво „начинаещи„*

Интерфейсът на това ниво е прост и лесен за използване без да е необходимо задълбочено познаване на съвременни формални методи за проверка на коректността на ООП. Предлага на студентите, софтуерни инженери и програмисти средства, чрез които те могат да верифицират функции на ООП, използвайки предварително създадени формални ОМ проекти на класовете (спецификации) и ОМ-модели на функциите, които ще бъдат верифицирани. На това ниво се изисква обучаемите да са запознати с

основните елементи на теорията на обобщените мрежи и да могат да ги прилагат за моделиране. Те също трябва да знаят основните елементи на теорията на формалната верификация. В резултат на обучението, потребителите на образователната среда на ниво „начинаещи“ ще се научат как да използват средата за верификация и да анализират получените резултати. В случаите, когато е възникнала грешка, ще могат да намират място на грешка в програмата, ще могат да анализират причините и да направят необходимите подобрения на програмния код.

б) *ниво „междинно*“

Въпреки че интерфейсът на това ниво се проектира да е удобен за работа, за ефективното му използване се изискват знания за формални методи за верификация на програми – поне на тези, които са описани в раздел 2.1 на глава 2. Това ниво предлага на студентите и другите потребители на средата средства за верифициране на функции на ООП относно спецификации, определени с помощта на формални ОМ проекти на класовете на ООП. На това ниво студентите би трябвало вече да притежават необходимите знания и умения, които им дават възможност да създават ОМ-модели на функциите на една ООП програма в подходящ за верификация формат и да ги интегрират в образователната среда. Те също трябва да могат ръчно да прилагат алгоритъма за проверка за съгласуваност на обобщеномрежовите модели на функцията и на спецификацията (стъпка 4 на подхода, описан в раздел 3.2.1.2). Чрез ръчно прилагане на алгоритъма за проверка на съгласуваността студентите могат да получат достъп до проследяване функционалността на алгоритъма на модула *Model Checking Engine* стъпка по стъпка (раздел 3.2.1.3).

в) *ниво „напреднали*“

На това ниво се очаква потребителите да са усвоили всички стъпки на подхода (раздел 3.2.1.2). Трябва да могат да създават ОМ-модели на функции на ООП програма в подходящ за верификация формат, да дефинират ОМ-модели на класовете на ООП, да проверяват дали последните са формални ОМ проекти на класовете и да ги интегрират в образователната среда. Те също трябва да владеят прилагането на алгоритмите за проверка за съгласуваност и да могат да проследяват работата на model checker-а стъпка по стъпка. Обучаващият се трябва да е запознат с някоя от системите за автоматично доказателство на теореми, с техниката на преобразуваите предикати, необходими за реализиране на верификацията на член-функциите на класовете на програмата. Очертаните изисквания са от съществено значение за работата в ниво „напреднал“, където на потребителите се дава пълен контрол над процеса на верификацията на програмата и доказателството на съгласуваността.

Описаната накратко образователна среда все още е в процес на разработка. Съществено внимание се обръща на подготвянето на подходящи примери за грешки в софтуера, които трудно могат да се намерят със съществуващите динамични и други методи за верификация. Целта на тези примери е да се направи убедителна илюстрация за превъзходство на представения формален подход пред другите подходи. Надяваме се, че по този начин обучението по формална верификация на програмно осигуряване може успешно да бъде въведено още в ранните години на обучение по програмиране на

студентите от специалностите компютърни науки, софтуерно инженерство и информационни системи.

3.2.1.5. Методологически аспекти на подхода за построяване на коректни ООП

Ще напомним, че проверката на коректността на ООП в случая, когато програмата се състои от един клас и една функция, която го използва, се осъществява като се изпълнят следните стъпки:

- Дефиниране на спецификацията, относно която ще се верифицира функцията;
- Верифициране на класа и на спецификацията;
- Построяване на обобщеномрежов модел на функцията, която ще бъде верифицирана относно построената спецификация;
- Проверка дали обобщеномрежовият модел на функцията удовлетворява дефинираната спецификация.

В две от стъпките (първата и третата) се изисква създаването на обобщени мрежи.
При третата стъпка изграждането на ОМ е свързано с прилагането на обобщените мрежи на основните оператори на управление на изчислителния процес.
При реализиране на първата стъпка се оказва полезна методологията за изграждане на ОМ, дефинирана в [I.48, I.49].
В тези литературни източници методологията за изграждане на обобщени мрежи е определена в най-общия случай. За всяко приложение се налага това описание да бъде конкретизирано.
В [II.18] е направена такава конкретизация за първата стъпка за реализиране на подхода за верификация на ООП. Изложението на методологията е съпроводено с пример, илюстриращ съответните действия.
В следващото изложение е описана тази методология в случая, когато ООП съдържа само един клас (означаваме го с *C*).

Процесът на изграждане на спецификацията преминава през следните две стъпки:

1) ***Дефиниране на ОМ-модела на спецификацията***

а) *Построява се статичната структура на обобщената мрежа, задаваща спецификацията*

За целта се изпълняват следните действия:

Определят се събитията, които могат да се случат на обект на класа

На всяко събитие се съпоставя (без да се дефинира формално) преход в ОМ-модела на спецификацията. За дефинирането на преходите се определят началните и крайните състояния на всяко събитие. На всяко начално състояние се съпоставя входна позиция на прехода, на която се уточняват приоритетът й спрямо другите позиции на прехода, за който се отнася, и капацитетът й. На всяко крайно състояние се съпоставя изходна позиция на прехода, на която се уточняват приоритетът й спрямо другите позиции на

прехода, капацитетът й, а също и каква характеристика трябва да придобие всяко ядро, което влезе в някоя от тези позиции.

Определят се приоритетите на преходите, съответстващи на събитията

За целта се проследяват основните действия, които могат да се изпълнят в съответните на преходите събития и връзките между тях. Приоритетът на основните действия над обект на класа задава приоритета на преходите.

Определят се капацитетите на позициите

Във всяка позиция на ОМ-модела на класа може да се съдържат от 0 до толкова на брой ядра, колкото е броят на обектите в ООП, която ще се верифицира относно спецификацията, зададена чрез ОМ-модела на спецификацията.

С всяка позиция на преход се свързва булев израз, характеризиращ състоянието, в което се намира обект на класа

Този булев израз е *логическото състояние на позицията*. За целта се анализират предпоставките, при които могат да се изпълнят отделните събития, протичащи в класа и се записват чрез булеви изрази. Всички входни позиции на преход, съответстващ на конкретно събитие, имат едно и също логическо състояние. Логическото състояние на всяка позиция е предикат, от верността на който трябва да следва верността на инвариантата на класа.

б) *Добавя се динамиката на обобщената мрежа, задаваща спецификацията*

Динамичният характер на мрежата се определя от ядрата и от условията на преходите.

Дефиниране на ядрата в ОМ

Обектите на класа се моделират чрез ядрата на ОМ, моделираща класа. Ядро влиза в мрежата когато във функцията, която се верифицира, се създава обект на класа (чрез конструктор) и излиза, когато съответният му обект се разрушва (след изпълнение на деструктора на класа). Ядрата в ОМ имат характеристики от вида: (*обект*, *позиция*), където параметърът *обект* е име на обекта, моделиран от ядрото, а параметърът *позиция* съдържа името на позицията, в която е ядрото в даден момент на движение. В характеристиката на ядро неявно участват още два компонента: член-данните на класа, които са свързани с параметъра *обект* и логическо състояние на позицията, свързано с параметъра *позиция*. Параметърът *позиция* се използва за да натрупа информация за пътя, който ядрото е изминало и в случай на грешка, да укаже мястото й.

Дефиниране на приоритета на ядрата

Приоритетът на ядрата в ОМ-модела на класа се определя от изпълняваните преходи на ОМ-модела на функцията, която се верифицира относно спецификацията, определена от класа.

Дефиниране на преходите на мрежата

За целта се определят условията на преходите и характеристичната функция *Φ*, задаваща новата характеристика, която ще получи ядро, след като се премести от входната позиция, в която се намира, в съответната изходна позиция (съгласно условието на прехода).

в) *Задава се времевата характеристика на обобщената мрежа, дефинираща спецификацията*

Времевите компоненти T, t^0 и t^* (дефиниция 9) на ОМ-модела на класа се определят както следва: T съвпада с момента от време, в който за пръв път в ОМ-модела на функцията е изпълнен преход, съответстващ на конструктор на класа; t^0 и t^* съвпадат със съответните параметри на ОМ-модела на функцията.

2) *Проверяване дали изграденият ОМ-модел на класа е формален ОМ проект на класа*

Тази стъпка се реализира като се провери дали изграденият ОМ-модел на класа, означаваме го с GN_C, е формален ОМ проект на класа. За целта се проверява дали за GN_C са в сила условията от дефиниция 10:

а) За всяка позиция S на GN_C, различна от входната, е в сила импликацията

логическо състояние на позицията $S \Rightarrow pre_q$

където q е произволна член-функция на класа C, която участва в условията на прехода, на който S е входна позиция, а pre_q е предусловието на тази член-функция.

За входната позиция на GN_C, е в сила импликацията

$Default_C \Rightarrow pre_{C_s}$

за всеки конструктор C_s, $(1 \le s \le p)$. pre_{C_s} е предусловието на конструктора C_s.

б) За всяка позиция S на GN_C, различна от входната е в сила импликацията

логическо състояние на позицията $S \Rightarrow Inv$

Inv е инвариантата на класа C.

в) За всяка двойка от входна и изходна позиции (R, S) на преход, различен от прехода, реализиращ изпълнението на конструктора (конструкторите) на класа C, с условие на прехода

Член-функцията е $q \wedge$ *Условието* P *е в сила*

е в сила тройката на Хоар

{ *логическо състояние на позицията* $R \wedge P$}
$Body_q$
{ *логическо състояние на позицията* S }

За всяка двойка от входна и изходна позиции (R_0, B_s), $1 \le s \le p$, на прехода, реализиращ изпълнението на конструктора (конструкторите) на класа с условие на прехода

Член-функцията е конструкторът $C_s \wedge$ *Условието* Q_s *е в сила*

е в сила тройката на Хоар

$$\{ Default_C \wedge Q_s \}$$
$$Body_{Cs}$$
$$\{ \textit{логическо състояние на позицията } B_s \}$$

3.2.2. Верификация на процедурни програми

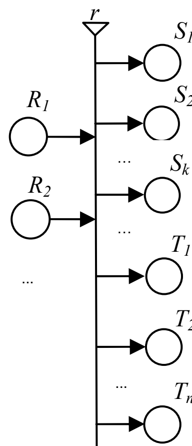
В тази част са описани резултати на автора на хабилитационния труд, свързани с верификацията на процедурни програми от тип проверка на съгласуваност.
Резултатите включват:

- дефиниране на подход за верификация на процедурни програми [II.16];
- изследване на коректността на модела на спецификацията, относно която се осъществява верификацията на процедурна програма [II.25];
- реализация на подхода [II.20].

За илюстрация на подхода за верификация на процедурни програми е използван езикът *C++*.

3.2.2.1. Формален обобщеномрежов проект на връзките между функциите на процедурна програма

Нека обобщена мрежа се състои от множество от преходи *r*, представени на фиг. 10.



Фиг. 10. Преход на формален ОМ проект на връзките между функциите на процедурна програма

където

$$r = <\{ R_1, R_2, ...\}, \{ S_1, S_2, ..., S_k, ..., T_1, T_2, ..., T_n\}, r' >$$

r' =	S_1	S_2	...	S_k	...	T_1	T_2	...	T_n
R_1	V_1	V_2	...	V_k	...	W_1	W_2	...	W_n
R_2	V_1	V_2	...	V_k	...	W_1	W_2	...	W_n
...	...	...	...	...	...	...	...	...	...

V_i = Функцията е *f*. $\wedge$ В сила е условие P_i $(1 \leq i \leq k)$;

...

W_j = Функцията е *g*. $\wedge$ В сила е условие Q_j $(1 \leq j \leq n)$.

Тук f и g са произволни, различни функции на програмата. Предикатите $P_1, P_2, \ldots, P_k$, а също и $Q_1, Q_2, \ldots, Q_n$, отнасящи се до прилагането на една и съща функция, са взаимноизключващи се (не могат да са едновременно истина). Последното изискване не е задължително. В случай, че не е изпълнено, разрешаването на движението на ядрата ще се осъществява и в зависимост от приоритетите на позициите.

Всяка позиция на прехода *r* е свързана с предикат, който задава състоянието, в което може да попадне една променлива (или множество от променливи величини) на функция на програмата. Наричаме го *логическо състояние на променливата* (или *на множеството от променливи*) *в позицията* или накратко *логическо състояние на позицията*.

Логическото състояние се задава чрез булев израз, който комбинира предусловия на функциите на програмата. С променлива (променливите) на програмата може да е свързана стойност (редица от стойности).

Ядрата в позициите на тази ОМ съответстват на променливите величини, дефинирани в програмата. Характеристиките им задаваме като двойки от вида

(*име_на_променлива*, *позиция*),

или

(*множество_от_имена_на_променливи*, *позиция*),

където *позиция* е името на позицията, в която е ядрото в обобщената мрежа. Освен тези две компоненти, в характеристиката на ядро чрез параметъра *име_на_променлива* (*множество_от_имена_на_променливи*) неявно участва стойността на променливата (стойностите на променливите). Наричаме я още *стойност на ядрото*. Чрез параметъра *позиция* в характеристиката на ядро се задава и логическото състояние на променливата (множеството от променливи) в позицията. Наричаме го и *логическо състояние на ядрото*.

Дефиниция 13 [II.25]. Обобщената мрежа, дефинирана по-горе, наричаме формален ОМ проект на връзките между функциите на процедурна програма и означаваме с GN_F, ако за нея са в сила:

а) За всяка позиция *R* на GN_F, различна от входната, е в сила импликацията

логическо състояние на позицията R => pre_{func} (8)

където *func* е произволна функция на процедурната програма, различна от *main*, която участва в условията на прехода, на който *R* е входна позиция, а pre_{func} е предусловието на тази функция.

За входната позиция на GN_F, е в сила импликацията

$Default_F \Rightarrow pre_{INIT}$ (8′)

за всека инициализираща функция *INIT*. Pre_{INIT} е предусловието на *INIT*.

Чрез $Default_F$ е означено твърдение, изразяващо връзки по подразбиране между данните на функциите на програмата.

б) За всяка двойка от входна и изходна позиции (*R*, *V*) на преход, различен от прехода, реализиращ изпълнението на инициализиращите функции, с условие на прехода

Функцията е func. $\wedge$ *Условието P е в сила.*

е в сила тройката на Хоар

$$\{\,\textit{логическо състояние на позицията } R \wedge P\}\ Body_{func}\ \{\,\textit{логическо състояние на позицията } V\,\} \quad (9)$$

За всяка двойка (*R*, *V*) от входна и изходна позиции на прехода, реализиращ изпълнението на инициализиращи функции с условие на прехода

Функцията е инициализираща (INIT). $\wedge$ *Условието Q е в сила.*

е в сила тройката на Хоар

$$\{\,Default_C \wedge Q\,\}\ Body_{INIT}\ \{\,\textit{логическо състояние на позицията } V\,\} \quad (9')$$

Условията (8) и (8′) осигуряват, във входните позиции на преходите на GN_F да са в сила предусловията на всички функции, които съответният преход може да изпълни. Условията (9) и (9′) гарантират запазването на верността на предиката *логическо състояние на позиция* за текущите стойности на данните на ядрата.

За спецификация на формалния обобщеномрежов проект GN_F на функциите на процедурната програма избираме предусловията и постусловията на функциите на процедурната програма.

Дефиниция 14 [II.25]. Формалният ОМ проект GN_F на връзките между функциите е коректен относно спецификацията си, ако за всяка двойка (*R*, *V*) от входна и изходна позиции на преход с условие

Функцията е q. $\wedge$ *Условието P е в сила.*

е в сила тройката на Хоар

$$\{\,pre_q(x_q) \wedge P\,\}\ Body_q\ \{\,post_q(x_q)\,\} \quad (10)$$

където x_q е множеството от допустими аргументи на функцията *q*.

Теорема 6. Ако функциите на процедурната програма са коректни относно спецификациите си, формалният ОМ проект GN_F на връзките между функциите също е коректен относно спецификацията си.

Доказателство. Следва от дефиниции 13 и 14 и прилагане на правилото за следствието.

Дефиниция 15. Формалният ОМ проект на връзките между функциите на процедурна програма съответства на реализацията на функциите на програмата, ако е коректен относно спецификацията си.

3.2.2.2. Подход за верификация на процедурни програми

Подходът за верификация на ООП, описан в раздел 3.2.1.2 може да се адаптира за процедурни програми.
За целта разделно се верифицират функциите, които изграждат програмата относно спецификации според предназначението им. Изгражда се формален ОМ проект на връзките между функциите на програмата. Главната функция не се включва във формалния ОМ проект на връзките между функциите на програмата.
За главната функция на програмата се построява обобщеномрежов модел и се проверява дали той съответства на обобщеномрежовия проект на връзките между функциите на програмата. Всяка от функциите на програмата, която използва други функции, дефинирани в програмата, също се верифицира и относно спецификацията, зададена чрез ОМ модела на връзките между функциите на програмата.

Чрез този подход могат да се търсят както грешки във функциите на програмата, така и интерфейсни грешки.
Изпълнението на подхода преминава през следните стъпки:

1) *Построяване на формален ОМ проект на връзките между функциите на процедурната програма*

На тази стъпка се изгражда част от спецификацията, относно която ще се верифицира процедурната програма. За целта се дефинират:

- *предусловието на всяка функция на програмата*;
- *обобщеномрежов модел, специфициращ връзките между функциите на програмата.*

Извършва се проверка дали този ОМ модел е *формален ОМ проект на връзките между функциите.* За целта за ОМ модел, специфициращ връзките между функциите на програмата се проверяват условията от дефиниция 13. Ако условията са изпълнени, се изпълняват следващите стъпки на подхода. В противен случай се променя ОМ моделът на връзките така, че получената ОМ да е формален ОМ проект на връзките между функциите.

В следващото изложение формалният *ОМ проект на връзките между функциите на процедурната програма* ще означаваме с GN_F. Той дефинира възможните коректни начини за използване на функциите на програмата. Всяка функция, която използва други функции, дефинирани в програмата, трябва да има поведение, което съответства на GN_F.

2) *Дефиниране на формална спецификация на процедурната програма и верификация на всички функции на програмата, различни от main*

За реализирането на тази стъпка се извършват следните действия:

- доизграждане на формалната спецификация на процедурната програма

За всяка дефинирана в програмата функция се определя постусловие, а за функциите, които съдържат оператори за цикъл – съответни инварианти и ограничаващи функции.

- верификация на всички функции на програмата, различни от main

Всяка от функциите, дефинирана в програмата се верифицира съгласно зададената за нея спецификация. За целта се построява теорема във вид на тройка на Хоар. Доказателството на теоремата може да се реализира с помощта на техниката на преобразуващите предикати, чрез някоя от системите за автоматично доказателство на теореми HOL, Isabelle, Coq и др., а също като се използва средата, описана в раздел 2.2.2.

Тази и предходната стъпки са най-трудните, но веднъж реализирани могат да се използват за верифициране на всички функции, които трябва да бъдат верифицирани относно формалния ОМ проект на връзките между функциите на програмата.

3) *Построяване на ОМ-модел на главната функция и на функциите, които ще бъдат верифицирани относно формалния ОМ проект на връзките между функциите*

Ако функция се обръща към други функции, дефинирани в програмата, тя трябва да се верифицира освен относно спецификацията, произтичаща от спецификата й и относно спецификацията, зададена чрез формалния ОМ проект на връзките между функциите на програмата.
Верификацията на функция относно спецификацията, зададена от формалния ОМ проект на връзките между функциите се осъществява като се построи обобщеномрежов модел на функцията и се провери дали двата модела – на функцията, която подлежи на верификация и на формалния ОМ проект на връзките между функциите си съответстват (дали са консистентни).

За целта като се използват правилата за съответствие на процедурните оператори и обобщеномрежовите елементи, представени в 3) на раздел 3.2.1.2, процедурният запис на всяка подлежаща на този вид верификация функция се преобразува в ОМ.

4) *Проверка за съгласуваност*

Нека функцията, за която ще се извършва проверката за съгласуваност е M, а нейният ОМ-модел е GN_{M}.
Проверката за съгласуваност на моделите – GN_{M} и GN_{F} се осъществява като моделите се изпълнят паралелно. Алгоритъм за паралелното изпълнение е даден на фиг. 11.
Следва кратко описание на алгоритъма за проверката за съгласуваност на двата модела. По-подробно описание е дадено в следващия раздел. За простота на описанието ще се ограничим до случая, когато фактическите параметри на обръщенията към функциите са променливи величини.

Всяко изпълнение на преход в GN_M, съдържащ обръщение към дефиниция на променлива величина предизвиква активиране на две нови ядра – едно в текущата позиция на прехода на GN_M, съответстващ на изпълнението на декларацията и друго в текущата или във входната позиция на GN_F. Новите ядра в текущите позиции на обобщените мрежи GN_M и GN_F се получават в резултат от разцепване на вече съществуващи ядра. Тези ядра имат еднакви имена, съвпадащи с името на дефинираната променлива. След изпълнението на преходите, които съдържат обръщението към декларацията, новосъздадените ядра получават характеристики от определения по-горе вид. Ако дефиницията е с инициализация, ядрата имат за стойност стойността на инициализиращия израз. Ако дефиницията е без инициализация, ядрата са с неопределени стойности. Всяко ядро има период на активност, започващ от изпълнението на прехода, реализиращ дефиниция на променлива и завършващ след изпълнението на прехода, реализиращ завършване изпълнението на блока (оператора), в който дефинираната локална променлива е „видима".
Ядро се създава в позиция, която може или да е празна, или в нея са натрупани ядра. Всички ядра на мрежата GN_M, са събрани в една позиция и се преместват едновременно при изпълнение на преходите. Броят им расте при изпълнение на дефиниция и намалява – при завършване на блока, в който променливата е дефинирана. Ако преход съответства на изпълнение на обръщение към функция на програмата, с него се свързва и редица от фактическите параметри на обръщението към функцията.
В случай, че в позиция на мрежата GN_M има ядра са възможни:

- *Позицията е входна за преход, съответстващ на действие, което не е обръщение към функция на програмата*

Тогава ако преходът може да се изпълни се изпълнява и всички ядра в позицията се преместват от входната в съответната изходна позиция на прехода. Съответните им ядра в GN_F не променят позициите си.

- *Позицията е входна за преход r, съответстващ на изпълнение на обръщение към функция на програмата*

Нека функцията, съответстваща на прехода r да има име f и фактически параметри – променливите x, y, …, z. В този случай, ако във входната позиция няма ядра с имена x, y, …, z, преходът няма да се изпълни – възникнала е грешка. В противен случай преходът ще се изпълни, ако паралелно с него може да се изпълни преход на GN_F с входна позиция, съдържаща ядра с имена x, y, …, z и с условие, което съдържа предиката „функцията е f". Изискването за паралелно изпълнение на двата прехода осигурява коректност на обръщението към функцията f в M. Ако преходът r не може да се изпълни, моделите GN_M и GN_F не са съгласувани. Последното съответства на „налице е некоректно обръщение към функция на програмата", т.е. намерена е интерфейсна грешка.

3.2.2.3. Реализация на подхода за верификация на процедурни програми

Реализацията на подхода ще опишем за програма, която се състои от функциите f_1, f_2, …, f_n, някои от които са инициализиращи, и функция *main*, която прави обръщения към тези функции. За простота ще се ограничим до случая когато:

- фактическите параметри на обръщенията към функциите са променливи;
- функциите $f_1, f_2, \ldots, f_n$ не съдържат обръщения към други функции.

Реализацията е представена графично на фиг. 11 в случая когато в *main* са дефинирани повече от една променливи. Алгоритъмът е описан чрез ОМ. Означаваме я с GN_1. Тази мрежа изпълнява паралелно обобщените мрежи GN_{main} и GN_F. Състои се от два прехода. Преходът X_1 изпълнява действия, свързани с дефинирането на променливи величини, а преходът X_2 – с проверката за съгласуваност. Характеристиките на ядрата на GN_{main} и GN_F имат вида

(*множество_от_имена_на_променливи*, *име_на_позиция_в_*GN_{main})

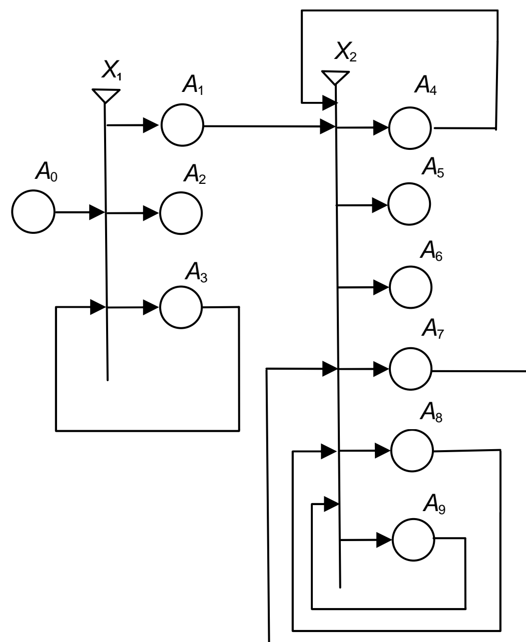
и

(*множество_от_имена_на_променливи*, *име_на_позиция_в_*GN_F)

съответно. Характеристиката за ядрата на GN_1 е редица от вида

(*множество_от_имена_на_променливи*, *име_на_позиция_в_*GN_1,
*име_на_позиция_в_*GN_{main}, *име_на_позиция_в_*GN_F).

и определя множеството от имена на променливите, името на позиция в GN_1 и имената на позициите в GN_{main} и GN_F на множеството от ядрата. Всеки преход в обобщената мрежа GN_{main}, който е свързан с обръщение към функция на програмата е свързан и с имената на формалните параметри на обръщението към функцията.



Фиг. 11. Алгоритъм за проверка за консистентност в случая на процедурни програми

Дефинициите на преходите на GN_1 имат вида:

$$X_1 = < \{A_0, A_3\}, \{A_1, A_2, A_3\}, r_1 >$$

където

$$r_1 = \begin{array}{c|ccc} & A_1 & A_2 & A_3 \\ \hline A_0 & P_1 & P_2 & P_3 \\ A_3 & P_1 & P_2 & P_3 \end{array}$$

P_1 = Преходите с входни позиции, в които са ядрата на GN_F и GN_{main}, съдържат условие от вида „*Изпълнява се дефиниране на променлива*“.

$P_2 = \neg P_1 \ \& \ \neg P_3$

P_3 = Ядрото на GN_{main} е във входна позиция за преход, който изпълява действие, което не използва променливи величини и може да се изпълни.

$$X_2 = < \{A_1, A_4, A_7, A_8, A_9\}, \{A_4, A_5, A_6, A_7, A_8, A_9\}, r_2 >$$

$$r_2 = \begin{array}{c|cccccc} & A_4 & A_5 & A_6 & A_7 & A_8 & A_9 \\ \hline A_1 & P_4 & P_5 & P_6 & P_7 & P_8 & P_9 \\ A_4 & P_4 & P_5 & P_6 & P_7 & P_8 & P_9 \\ A_7 & P_4 & P_5 & P_6 & P_7 & P_8 & P_9 \\ A_8 & P_4 & P_5 & P_6 & P_7 & P_8 & P_9 \\ A_9 & P_4 & P_5 & P_6 & P_7 & P_8 & P_9 \end{array}$$

и

P_4 = Ядрата на GN_{main} и GN_F са във входни позиции на преходи, които изпъляват обръщения към някоя от функциите f_1, f_2, …, f_n. Поради наличието на условия на преходите, които са в сила едновременно, преходите могат да се изпълнят паралелно.

P_5 = Изпълнението на мрежите GN_{main} и GN_F е завършило.

$P_6 = \neg P_4 \ \& \ \neg P_5 \ \& \ \neg P_7 \ \& \ \neg P_8 \ \& \ \neg P_9$

$P_7 = P_1$

P_8 = Ядрото на GN_{main} е във входна позиция на преход, свързан с изпълнение на оператор вида: входен, изходен, условен, цикъл или начало на блок.

P_9 = Преходите с входни позиции, в които са ядрата на GN_F и GN_{main}, съдържат условие от вида „*Изпълнява се разрушаване на променлива*“.

Изпълнението на GN_1 започва с активиране на ядро в позиция A_0, която е входна за прехода X_1. Това ядро има характеристика с празно множество от променливи. Паралелно започват да се изпълняват мрежите GN_{main} и GN_F. Във входната позиция на всяка от тях се активира ядро с празно множество от променливи.

Изпълнение на прехода X_1

В случай, че е в сила предикатът P_1, се изпълняват преходът X_1 на GN_1 и преходите с входни позиции, в които са активираните ядра на GN_{main} и GN_F. В резултат, ядрата на GN_{main} и GN_F се предвижват в съответните изходни позиции на изпълнените преходи. Нека ги означим с M_1 и S_1. И получават характеристиките $(\{x\}, M_1)$ и $(\{x\}, S_1)$ съответно,

където x е името на дефинираната променлива. Ядрото на GN_1 се предвижва в позиция A_1 и получава за характеристика редицата $(\{x\}, A_1, M_1, S_1)$. Последната се състои от четири елемента: множество, състоящо се от името на дефинираната променлива и имената на позициите A_1, M_1 и S_1, в които са ядрата в мрежите GN_1, GN_{main} и GN_F, съответно. Изпълнението на прехода X_1 завършва, след което започва изпълнението на прехода X_2 на GN_1.

В случай, че е в сила предикатът P_3, преходът X_1 на GN_1 и текущият преход на GN_{main} се изпълняват, ако това е възможно. Преходът на GN_F, който има ядро във входната си позиция, не се изпълнява. В резултат на това ядрото на GN_{main} се премества в съответната изходна позиция и се променя само вторият елемент на неговата характеристика. Ядрото на GN_F не се предвижва. Ядрото на GN_1 се премества в позиция A_3, като се променя само вторият елемент на неговата характеристика. Тъй като позиция A_3 е входна за прехода X_1, следващата стъпка в изпълнението на GN_1 е да повтори изпълнението на X_1.

В случай, че е в сила предикатът P_2, ОМ-моделът GN_{main} не съответства на GN_F и изпълнението на GN_1 завършва. Условията на прехода X_1 задават причините за несъответствието, а характеристиката на ядрото указва позициите в GN_{main} и GN_F, където то е възникнало.

Изпълнение на прехода X_2

В случай, че е в сила предикатът P_4, се изпълняват преходът X_2 на GN_1 и преходите с входни позиции, в които са ядрата на GN_{main} и GN_F. Нека ядрата на GN_{main} и GN_F се предвижват в позициите M_2 и S_2, съответно и множеството от променливите е V. Тогава техните характеристики са (V, M_2) и (V, S_2). Ядрото на GN_1 се премества в позиция A_4 и получава характеристиката (V, A_4, M_2, S_2). Тъй като позиция A_4 е входна за прехода X_2, изпълнението на GN_1 продължава със следващо изпълнение на прехода X_2.

В случай, че е в сила предикатът P_5, ядрото на GN_1 се премества в позиция A_5 и загубва характеристиката си. В този случай ОМ-моделът на функцията *main* съответства на формалния обобщеномрежов проект на връзките между функциите, т.е. не е намерена грешка.

В случай, че е в сила предикатът P_7, се изпълняват преходът X_2 на GN_1 и преходите с входни позиции, в които са ядрата на GN_{main} и GN_F. В резултат ядрата на GN_{main} и GN_F се преместват в съответните изходни позиции (нека те са M_3 и S_3) и получават характеристиките $(V\cup\{x\}, M_3)$ и $(V\cup\{x\}, S_3)$ съответно, където x е името на новата променлива, дефинирана във функцията *main*. Ядрото на GN_1 се премества в позиция A_7 и получава като характеристика редицата $(V\cup\{x\}, A_7, M_3, S_3)$. Характеристиката съдържа имената на променливите, които са дефинирани до момента в програмата и имената на позициите A_7, M_3 и S_3, в които са ядрата на обобщените мрежи GN_1, GN_{main} и GN_F, съответно. Изпълнението на GN_1 продължава с изпълнението на прехода X_2.

В случай, че е в сила предикатът P_8, преходът X_2 на GN_1 и преходът на GN_{main}, с входна позиция, в която е ядрото на GN_{main}, се изпълняват, ако е възможно. Преходът на GN_F не се изпълнява. Като резултат ядрото на GN_{main} се премества в съответната изходна позиция. Нека тя е с име M_4. Ядрото на GN_F остава в текущата си позиция (нека тя е S_4). Ядрото на GN_1 се премества в позиция A_8 и получава характеристиката (V, A_8, M_4, S_4), където V е

множеството от имената на променливите, свързани с ядрото. Изпълнението на GN_1 продължава с изпълнението на прехода X_2.
Ако не е възможно да се изпълни текущият преход на GN_{main}, GN_{main} не съответства на GN_F.

В случай, че е в сила предикатът P_9, преходът X_2 на GN_1 и преходите с входни позиции, в които са ядрата на GN_{main} и GN_F, се изпълняват. Като резултат ядрото на GN_{main} се премества в съответната изходна позиция. Нека я означим с M_5. Ядрото на GN_F се предвижва в позиция, която нека означим с S_5. Ядрото на GN_1 се премества в позиция A_9 и получава характеристиката (V', A_8, M_5, S_5), където $V' = V \setminus \{x\}$ е множеството от имена на променливи, които са свързани с ядрото, а x е разрушената променлива. Множествата от променливите на характеристиките на ядрата на GN_{main} и GN_F стават V'. Изпълнението на GN_1 продължава с изпълнението на прехода X_2.

В случай, че е в сила предикатът P_6, GN_{main} не съответства на GN_F. Ядрото на GN_1 се премества в позиция A_6 и променя само втория елемент на характеристиката си. Условията на прехода задават причините за несъответствието, а характеристиката на ядрото в позицията A_6 показва позициите в GN_F и GN_{main}, където несъответствието е възникнало.

3.3. Заключителни бележки

В тази част описахме нови подходи за верификация на процедурни и обектно-ориентирани програми. Подходите прилагат модела „дизайн чрез договор“ и обобщени мрежи – средство за моделиране и управление на реални процеси. Могат да се прилагат както за цялостна верификация на процедурни и обектно-ориентирани програми, така и за търсене само на интерфейсни грешки на програмите.

Направеното изследване е още в начален етап. Предстоят изследвания за верификация на ООП, съдържащи йерархии от класове, както и верифицирането на абстрактни типове данни, представящи хетерогенни структури. Предстои също по-масово внедряване на елементи от този подход при обучението по програмиране във Факултета по математика и информатика на СУ.

Заключение

В хабилитационния труд са описани основните научни и научно-приложни резултати в областта на *подходи, програмни среди и езици за верификация на програми и прилагането им при подготовката на софтуерни специалисти*, получени от доц. Магдалина Тодорова основно през последните 5 години. Само книгата [II.27] е написана през 2002 година.
Всички публикации, включени в хабилитационния труд, без тези с номера II.17 и II.19 са самостоятелни.

Тематично резултатите, описани в хабилитационния труд, могат да се систематизират по следния начин:

Резултати по темата *Методи за верификация на програми*

- Направен е сравнителен обзор на известни в литературата методи за верификация на програми [II.12].
- Направен е обзор на формалните методи и средства за верификация [II.12].

Основни резултати, свързани с темата *Формални методи и средства за дедуктивен анализ. Модели на приложни програмни системи, подпомагащи обучението по формални методи за дедуктивен анализ*

- Описан е подход за верификация на процедурни и обектно-ориентирани програми чрез използване на: логиката на Хоар, метода на преобразуващите предикати, модела „дизайн чрез договор“ и логически системи за автоматично доказване на теореми [II.3, II.1].
- Проектирана, реализирана и експериментирана е среда за верификация на процедурни и обектно-ориентирани програми, базираща се на описания подход [II.6, II.4, II.7].
- Опитът от прилагането на формални методи и средства за верификация при обучението по програмиране на студентите от направление „Информатика и компютърни науки“ е отразен в [II.2, II.5, II.19, II.24, II.27].
- Проектирана е среда за електронно обучение с цел прилагане на формалните методи и средства за целите на обучението [II.8].

Основни резултати, свързани с темата *Подходи за верификация от тип проверка на съгласуваност*

- Създаден е подход за верификация на ООП [II.14, II.15, II.17, II.18, II.23, II.26]
  - неформално е дефиниран подход за построяване на коректни обектно-ориентирани програми;
  - проведено е изследване на коректността на модела на спецификацията, относно

която се извършва интерфейсната верификация на ООП;
- създадена е експериментална реализация на подхода за верификация на ООП;
- проектирана е образователна среда за верификация на ООП;
- изследвани са методически аспекти на подхода за построяване на коректни ООП.

- Адаптиран е подходът за верификация на ООП за верификация на процедурни програми [II.16, II.20 и II.25]

- неформално е дефиниран подход за верификация на процедурни програми;
- проведено е изследване на коректността на модела на спецификацията, относно която се извършва интерфейсната верификация на процедурната програма;
- създадена е експериментална реализация на подхода за верификация на процедурни програми.

Част от постигнатите резултати по тематиката са прилагани или се прилагат в учебната дейност на доц. М. Тодорова.

ЛИТЕРАТУРА

I.1. G. Tassey, The Economic Impacts of Inadequate Infrastructure for Software Testing. NIST Report, May 2002, http://www.nist.gov/director/planning/upload/report02-3.pdf (последно посетена на 30.01.2013).

I.2. P. H. Roosen-Runge, Software verification tools, part I, Department of Computer Science York University, Toronto, Canada, http://www.cse.yorku.ca/course_archive/2006-07/W/3341/SVT0-3.pdf, 1999, 2003, 2007 (последно посетена на 30.01.2013).

I.3. IEEE 1012-2004 Standard for Software Verification and Validation. IEEE, 2005, http://pesona.mmu.edu.my/~wruslan/SE2/Readings/detail/Reading-7.pdf (последно посетена на 30.01.2013).

I.4. IEEE Std. 1028-1997, „IEEE Standard for Software Reviews", IEEE Computer Society, 1997.

I.5. R. Radice, Software Inspections, http://www.methodsandtools.com/archive/archive.php?id=29, 2002, (последно посетена на 10.01.2013).

I.6. http://en.wikipedia.org/wiki/Software_inspection (последно посетена на 30.01.2013).

I.7. Y. K. Wong, Modern Software Review: Techniques and Technologies. IRM Press, 2006, http://www.amazon.co.uk/Modern-Software-Review-Techniques-Technologies/dp/1599040131#reader_1599040131 (последно посетена на 10.01.2013).

I.8. http://en.wikipedia.org/wiki/Static_program_analysis (последно посетена на 05.01.2013).

I.9. B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsbarrow, N. J. Ward, and D. W. R. Marsh, Industrial Perspective on Static Analysis, Software Engineering Journal, Mar. 1995, 69-75, http://www.ida.liu.se/~TDDC90/papers/industrial95.pdf (последно посетена на 29.01.2013).

I.10. http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis#C.2FC.2B.2B (последно посетена на 29.01.2013).

I.11. http://en.wikipedia.org/wiki/Software_testing#Testing_methods (последно посетена на 21.01.2013).

I.12. SOA Testing Techniques, http://www.crosschecknet.com/soa_testing_black_white_gray_box.php (последно посетена на 21.01.2013).

I.13. Code Coverage Analysis, http://www.bullseye.com/coverage.html#intro (последно посетена на 21.01.2013).

I.14. Standard glossary of terms used in Software Testing Version 2.2, 2012, http://www.astqb.org/documents/ISTQB_glossary_of_testing_terms_2.2.pdf (последно посетена на 21.01.2013).

I.15. Software Testing-Testing Life Cycles, http://www.etestinghub.com/testing_lifecycles.php#2 (последно посетена на 21.01.2013).

I.16. С. Илиева, Хабилитационен труд, ФМИ на СУ Св. Климент Охридски, 2012.

I.17. J. Lönnberg, Visual testing of software, 2003, http://www.cs.hut.fi/~jlonnber/VisualTesting.pdf (последно посетена на 07.01.2013).

I.18. R. Black, Advanced Software Testing- Vol. 2, Guide to the ISTQB Advanced Certification as an Advanced Test Manager. Santa Barbara, Rocky Nook Publisher. ISBN 1-933952-36-9, 2008.

I.19. J.C. Kaner, Exploratory Testing, http://www.kaner.com/pdfs/ETatQAI.pdf (последно посетена на 04.01.2013).

I.20. A. Bauer, M. Leucker, Ch. Schallhart, Runtime Verification for LTL and TLTL, ACM Transactions on Software Engineering and Methodology (TOCEM), 2009. http://users.cecs.anu.edu.au/~baueran/publications/tosem-rv.pdf (последно посетена на 06.01.2013).

I.21. http://www.time-rover.com (последно посетена на 07.01.2013).

I.22. Y. Cheon, G. Leavens, A runtime assertion checker for the Java Modeling Language (JML). Proc. of International Conference on Software Engineering Research and Practice (SERP'02), CSREA Press, June 2002, http://archives.cs.iastate.edu/documents/disk0/00/00/03/05/00000305-00/jmlrac.pdf, pp. 322-328, (последно посетена на 07.01.2013).

I.23. N. Delgado, A. Gates, S. Roach, A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools, IEEE Transactions on Software Engineering, 30 (12), December 2004, pp. 859-872, (последно посетена на 07.01.2013).

I.24. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, A. Pretschner (eds.). Model Based Testing of Reactive Systems, Springer, 2005 http://is.ifmo.ru/books/_model-based_testing_of_reactive_systems.pdf (646 pages), (последно посетена на 08.01.2013).

I.25. Model-Based Testing, https://goldpractice.thedacs.com/practices/mbt/ (последно посетена на 08.01.2013).

I.26. V. Kuliamin, Integration of Verification Methods for Program Systems, Programming and Computer Software, 2009, Vol. 35, No. 4, pp. 212–222, 2009, http://link.springer.com/article/10.1134%2FS0361768809040057?LI=true#page-1 (последно посетена на 04.01.2013).

I.27. В. Кулямин, Методы верификации программного обеспечения, 2008, http://www.ict.edu.ru/ft/005645/62322e1-st09.pdf (последно посетена на 04.02.2013).

I.28. З. Манна, Математическа теория на информатиката, София, Наука и изкуство, 1983.

I.29. Э. Мендельсон, Введение в математическую логику. Москва, Наука, 1971.

I.30. Х. Барендрегт, Лямбда-исчисление. Его синтаксис и семантика. М., Мир, 1985.

I.31. Р. Фейс, Модальная логика. Москва, Наука, 1974.

I.32. Y. Venema, Temporal Logic, http://staff.science.uva.nl/~yde/papers/TempLog.pdf (последно посетена на 06.01.2013).

I.33. LTL: Linear-time logic, http://www.eecs.qmul.ac.uk/~pm/SaR/2004ltl.pdf (последно

посетена на 06.01.13).

I.34. J. Bradfield, C. Stirling, Modal Mu-Calculi, 2005, http://homepages.inf.ed.ac.uk/jcb/Research/MLH-bradstir.pdf (последно посетена на 03.02.2013).

I.35. St. Roman, Access Database Design & Programming, Third Edition, O'Reilly Media, Inc., 1999, 2nd ed., http://files.zipsites.ru/books/programming/OReilly_misc/O'Reilly%20-%20Access%20Database%20Design%20&%20Programming,%20Second%20Edition.pdf (последно посетена на 03.02.2013).

I.36. J. Guttag, Algebraic Specification of Abstract Data Types, http://www-sst.informatik.tu-cottbus.de/~db/doc/People/Broy/Software-Pioneers/Guttag_new.pdf.

I.37. W. Fokkink, Introduction to Process Algebra, 2nd edition, 2007, http://www.few.vu.nl/~wanf/BOOKS/procalg.pdf (последно посетена на 03.02.2013).

I.38. Finite State Machines, http://spinroot.com/spin/Doc/Book91_PDF/F8.pdf (последно посетена на 03.01.13).

I.39. P. Panangaden, Labelled Transition Systems, 2009, http://www.cs.mcgill.ca/~prakash/Talks/lecture1.pdf (последно посетена на 07.01.2013).

I.40. A. Rajee, M. Yannakakis, Model Checking of Hierarchical State Machines, ACM-TRANSACTION, 2002, http://www.cis.upenn.edu/~alur/Fse98.pdf (последно посетена на 07.01.2013).

I.41. A. Rajee, Timed Automata, http://repository.upenn.edu/cgi/viewcontent.cgi?article=1105&context=cis_reports (последно посетена на 07.01.2013).

I.42. T. Henzinger, The Theory of Hybrid Automta, Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 96), pp. 278-292, http://mtc.epfl.ch/~tah/Publications/the_theory_of_hybrid_automata.html (последно посетена на 17.01.2013).

I.43. J. Peterson, Petri Nets, Computing Surveys, vol. 9, No. 3, 1977, https://www.rose-hulman.edu/class/se/OldFiles/csse373/Spring2009/Resources/peterson77.pdf, pp. 223-252 (последно посетена на 24.01.2013).

I.44. K. Jensen, An Introduction to the Theoretical Aspects of Coloured Petri Nets, Lecture Notes in Computer Science vol. 803, Springer-Verlag 1994, pp. 230-272.

I.45. H. Genrich, Predicate/transition nets. Lecture Notes in Computer Science, vol. 254, 1986, pp. 207-247.

I.46. W. Thomas, Automata on infinite objects, In J. van Leeuwen, editor, Handbook of Theoretical Computer Science, volume B: Formal Models and Semantics, Elsevier Science Publishers, Amsterdam, 1990, pp. 133-192.

I.47. Abstract State Machines, http://www.eecs.umich.edu/gasm/intro.html (последно посетена на 30.01.2013).

I.48. K. Atanassov, On generalized nets theory. Prof. Marin Drinov Academic Publishing House, ISBN 978-954-322-237-7, Sofia, 2007.

I.49. K. Atanassov, Generalized nets. World Scientific, Singapore, New Jersey, London, 1991.

I.50. C. A. R. Hoare, An axiomatic basis for computer programming. Communications of the ACM, 12(10), 1969, pp. 576–585.

I.51. D. Harel, D. Kozen, J. Tiuryn, Review of Dynamic Logic, MIT Press, 2000, 459 pp, ISBN 0262082896, http://www.ccs.neu.edu/home/riccardo/papers/harel-dl.pdf (последно посетена на 30.01.2013).

I.52. B. Meyer, Applying Design by Contract, IEEE Computer 25(10), Oct. 1992, pp. 40-51.

I.53. R. Floyd, Assigning meaning to programs. Proc. of Symposium in Applied Mathematics, http://www.cs.virginia.edu/~weimer/2007-615/reading/FloydMeaning.pdf, in Proceedings of Symposium on Applied Mathematics, Vol. 19, J.T. Schwartz (Ed.), A.M.S., 1967, pp. 19–32.

I.54. E. Dijkstra, Discipline of programming, Prentice Hall, 1976.

I.55. R. Keller, Formal Verification of Parallel Programs, Communications of the ACM, vol. 19, N. 7, July, 1976, pp. 371-384.

I.56. Vampire's Home Page, http://www.vprover.org/ (последно посетена на 25.01.2013).

I.57. M. Giese, Taclets and the KeY Prover, 2003, http://folk.uio.no/martingi/pub/uitp03.pdf (последно посетена на 30.01.2013).

I.58. M. Kaufmann, J. S. Moore, ACL2 Version 4.2, december, 2012, http://www.cs.utexas.edu/users/moore/acl2/ (последно посетена на 24.01.2013).

I.59. E theorem prover, http://en.wikipedia.org/wiki/E_theorem_prover (последно посетена на 30.01.2013).

I.60. H. Schwichtenberg, H. Benl, U. Berger, M. Seisenberger, W. Zuber, Proof theory at work: Program development in the Minlog system, Automated Deduction, W. Bibel and P.H. Schmitt, eds., Vol. II, Kluwer, 1998.

I.61. The MINLOG System, http://www.minlog-system.de/ (последно посетена на 19.01.2013).

I.62. Waldmeister, http://www.mpi-inf.mpg.de/~hillen/waldmeister/index.htm (последно посетена на 12.01.2013).

I.63. Darwin - A Theorem Prover for the Model Evolution Calculus, http://combination.cs.uiowa.edu/Darwin/ (последно посетена на 17.01.2013).

I.64. The HOL System TUTORIAL, for HOL Kananaskis-7, http://www.cs.uu.nl/docs/vakken/pv/resources/kananaskis-7-tutorial.pdf, August 8, 2011 (последно посетена на 17.01.2013).

I.65. Isabelle, http://www.cl.cam.ac.uk/research/hvg/Isabelle/ (последно посетена на 17.01.2013).

I.66. G. Huet, G. Kahn, Ch. Paulin-Mohring, The Coq Proof Assistant - A Tutorial, 8.2 2009, http://coq.inria.fr/V8.2pl1/files/Tutorial.pdf (последно посетена на 17.01.2013).

I.67. J. Rushby, Prototype Verification System (PVS), http://www.csl.sri.com/projects/pvs/ (последно посетена на 17.01.2013).

I.68. Computation tree logic, http://en.wikipedia.org/wiki/Computation_tree_logic (последно посетена на 11.01.2013).

I.69. K. McMillan, Symbolic Model Checking, Kluwer Academic Publishers Norwell, MA, USA, ISBN 0792393805, 1993, http://www.kenmcmil.com/pubs/thesis.pdf (последно посетена на 11.01.2013).

I.70. A. Browne, E.M. Clarke, S. Jha, D.E. Long, W. Marrero, An improved algorithm for the evaluation of fixpoint expressions, Theoretical Computer Science, vol. 178, Issue 1-2, May 30, 1997.

I.71. BLAST: Berkeley Lazy Abstraction Software Verification Tool, http://mtc.epfl.ch/software-tools/blast/index-epfl.php (последно посетена на 27.01.2013).

I.72. Construction and Analysis of Distributed Processes, http://en.wikipedia.org/wiki/Construction_and_Analysis_of_Distributed_Processes (последно посетена на 28.01.2013).

I.73. T. Trifonov, K. Georgiev, GNTicker – A software tool for efficient interpretation of generalized net models. Issues in Intuitionistic Fuzzy Sets and Generalized Nets, vol. 3. Warsaw, 2005.

I.74. T. Trifonov K. Georgiev, K. Atanassov, Software for modelling with Generalised Nets. Issues in intuitionistic fuzzy sets and generalized nets, Vol. 6, 2008, 36-42.

I.75. UPPAAL, http://www.uppaal.org/ (последно посетена на 13.01.2013).

I.76. NuSMV: a new symbolic model checker, http://nusmv.fbk.eu/ (последно посетена на 12.01.2013).

I.77. HyTech: The HYbrid TECHnology Tool, http://embedded.eecs.berkeley.edu/research/hytech/ (последно посетена на 28.01.2013).

I.78. B. Lindstrøm, L. Wells, Design/CPN - Performance Tool Manual, 1999, http://www.daimi.au.dk/designCPN/man/Misc/Performance.pdf (последно посетена на 28.01.2013).

I.79. List of model checking tools, http://en.wikipedia.org/wiki/List_of_model_checking_tools (последно посетена на 18.01.2013).

I.80. Verity-Check, http://www.veritable.com/Computer/Formal_Validation/Verity-Check/verity-check.html (последно посетена на 14.01.2013).

II.81. Bisimulator manual page, http://www.inrialpes.fr/vasy/cadp/man/bisimulator.html (последно посетена на 29.01.2013).

II.82. Reductor manual page, http://www.inrialpes.fr/vasy/cadp/man/reductor.html (последно посетена на 29.01.2013).

II.83. C.A.R. Hoare, Proof of Correctness of Data Representations, Acta Informatica, Vol. 1, N 4, 1972, http://www.cs.cmu.edu/afs/cs/academic/class/17654-f01/www/refs/HJ2.pdf (последно посетена на 29.01.2013).

I.84. D. Gries, The Science of Programming. Springer-Verlag, Berlin and New York, 1981.

I.85. G. Plotkin, Dijkstra's Predicate Transformers and Smyth's Powerdomains,

http://www.cs.mcgill.ca/~prakash/Courses/comp598/plotto.pdf (последно посетена на 29.01.2013).

I.86. B. Meyer, Object-Oriented Software Construction, SECOND EDITION, ISE Inc. Santa Barbara, California, 1997.

I.87. Z. Manna, A. Pnueli, Axiomatic Approach to Total Correctness, Acta Informatica, 3 (1974) 253-262.

I.88. Л. Д. Беклемишев, Компютърные доказательства, 2009, http://www.reflexion.ru/club/16-04-09Beklemishev.pdf, (последно посетена на 29.01.2013).

I.89. V. Hamscher, Schwiegelshohn, A. Streit, R. Yahyapour, Evaluation of Job-Scheduling Strategies for Grid Computing, LNCS 1971, Springer-Verlag, Berlin, pp.191-202 (2000).

I.90. C. Barrit, CISCO systems Reusable Learning Object Strategy – Designing Information and Learning Objects Through Concept, Fast Procedure, Process, and Principle Templates, Version 4.0, White Paper (2001).

I.91. R. Floyd, Toward Interactive Design of Correct Programs, Proc. IFIP, pp.7-10.

I.92. K. Atanassov, D. Dimitrov, V. Atanassova, Algorithms for Tokens Transfer in the Different Types of Intuitionistic Fuzzy Generalized Nets. Cybernetics and Information Technologies, Vol. 10, 2010, No. 4, 22–35.

I.93. D.G. Dimitrov, A Graphical Environment for Modeling and Simulation with Generalized Nets, Annual of „Informatics“, Section Union of Scientists in Bulgaria, Vol. 3, 2010, 51–66.

I.94. D.G. Dimitrov, Software Products Implementing Generalized Nets, Annual of „Informatics“, Section Union of Scientists in Bulgaria, Vol. 3, 2010, 37–50.

I.95 D. G. Dimitrov, Optimized Algorithm for Token Transfer in Generalized Nets, In Recent Advances in Fuzzy Sets, Intuitionistic Fuzzy Sets, Generalized Nets and Related Topics, Vol. 1, 2010, pp. 63–68, ISBN 9788389475350.

I.96. И. Дончев, Э. Тодорова, Операции с обьектами и коммуникация между обьектами в обучении обьектно ориентированному программирования, The Sixth International Conference „Internet–Education–Science“, Vinnytsia, Ukraine, October 7–11, 2008.

I.97. И. Дончев, Э. Тодорова, Полиморфизм в курсе обьектно-ориенторованного программирования – дидактические аспекты, The Sixth International Conference „Internet–Education–Science“, Vinnytsia, Ukraine, October 7–11, 2008.

I.98. И. Дончев, Э. Тодорова, Object-Oriented Programming in Bulgarian Universities' Informatics and Computer Science Curricula, Informatics in Education, 2008, Vol. 7, No. 2, 159–172, 2008 Institute of Mathematics and Informatics, Vilnius.

I.99. А. Семерджиев, Т. Трифонов, М. Нишева, Автоматизирани инструменти за подпомагане на учебния процес по информатика. Международна научна конференция „Приложение на информационните и комуникационни технологии“, 2011, 2-3 декември, УНСС, София, България. 429–434, ISBN 978-954-92247-3-3.

I.100. L. Cassel, A. Clemens, G. Davis, M. Guzdial, R. McCauley, A. McGettrick, B. Sloan, L. Snyder, P. Tymann, B. Weide, 2008. Computer Science Curriculum 2008, An Interim Revision of CS 2001. Report from the Interim Review Task Force (December 2008). ACM and IEEE Computer Society.

Списък
на публикациите на Магдалина Тодорова,
представени за участие в конкурс за професор
по 4.6. Информатика и компютърни науки,
(програмиране)

II.1. Тодорова, М. Дизайн на език за верификация на програми на обектно-ориентирани езици, Конференция на съюза на учените в България, секция Информатика, 2007, Годишник на секция „Информатика“, Съюз на учените в България, ISSN 1313-6852, том 1, 2008, стр. 40-48.

II.2. Тодорова, М. Верификация и валидация на реализациите на абстрактни типове данни, Математика и математическо образование, Доклади на 37. пролетна конференция на СМБ, април, 2008, стр. 414-419.

II.3. Тодорова, М. Формална верификация на процедурни и обектноориентирани програми с използване на системата за доказване на теореми HOL, сп. Автоматика и информатика, издание на Съюза по автоматика и информатика „Джон Атанасов“, ISSN 0861–7562, год. XLII, кн. 2, 2008, стр. 25-27.

II.4. Тодорова, М. Верификация на процедури и функции чрез езика VL, Международна конференция „Автоматика и информатика’08“, 1-4 октомври, София, ISSN 1313–1850, 2008, стр. IX-13–IX-16.

II.5. Тодорова, М. Верификация на програми по време на изпълнение, сп. Математика и информатика, кн. 1, 2009, стр. 31-38.

II.6. Todorova, M. Verification Environment for Imperative and Object-Oriented Programs, Annual of „Informatics“ Section, Union of Scientists in Bulgaria, ISSN 1313-6852, Volume 2, 2009, pp. 14-24.

II.7. Todorova, M. VL: Language for Verification of Procedural programs, Proceedings of the Third International Conference Information Systems & Grid Technologies, 28-29 May 2009, Sofia, Bulgaria, pp. 96-104.

II.8. Todorova, M. Grid Framework for e-learning Services, Proceedings of the 4th International Conference Information Systems & Grid Technologies, May 28 – 29, 2010, Sofia, Bulgaria, pp. 153-163.

II.9. Todorova, M., H. Hristov, E. Stefanova, N. Nikolova. How to Build up Contemporary Software Professionals (Project-Based Learning in Data Structure and Programming), Proceedings of the Second International Conference on Software, Services and Semantic Technologies, September 11-12, 2010, Varna, Bulgaria, pp. 47-54.

II.10. Todorova, M., H. Hristov, E. Stefanova, N. Nikolova, E. Kovatcheva. Innovative Experience in Undergraduate Education of Software Professionals. Project-Based Learning in Data Structure and Programming, Proceedings of the ICERI2010: International Conference of Education, Research and Innovation, November 15–17,

2010, Madrid, Spain, pp. 5141-5150.

II.11. Тодорова, М., Х. Христов, Е. Стефанова, Н. Николова, Е. Ковачева. Проектно-базирано обучение по структури от данни и програмиране, Математика и математическо образование, Доклади на 40. пролетна конференция на СМБ, 2011, стр. 454-465.

II.12. Тодорова, М. Формални методи и средства за верификация на програмно осигуряване, Конференция на съюза на учените в България, секция „Информатика“, 2010, Годишник на секция „Информатика“, Съюз на учените в България, ISSN 1313-6852, том 3, 2010, стр. 26-36.

II.13. Todorova, M. Formal Specification of WEB Services by Means of Generalized Nets with Stop-Conditions, Proceedings of the 5th International Conference Information Systems & Grid Technologies, 28-29 May 2011, Sofia, Bulgaria, pp. 215-227.

II.14. Todorova, M. Model Checker of Object-Oriented Programs based on Generalized Nets, 10th International Workshop on Intuitionistic Fuzzy Sets and Generalized Nets (IWIFSGN'2011), Warsaw, Poland, September 30, 2011, published in *Recent Developments in Fuzzy Sets, Intuitionistic Fuzzy Sets, Generalized Nets and Related Topics. Foundations and Applications*, *vol. II: Applications*, Systems Research Institute of Polish Academy of Sciences, ISBN 83-894-7541-3, pp. 309-320.

II.15. Тодорова, М. Построяване на коректни обектно-ориентирани програми чрез изграждане на техни обобщеномрежови модели, Конференция на съюза на учените в България, секция „Информатика“, 2011, Годишник на секция „Информатика“, Съюз на учените в България, ISSN 1313-6852 том 4, 2011, стр. 1-28.

II.16. Todorova, M. Verification of Procedural Programs via Building there Generalized Nets Models, Proceedings of the 41. Spring Conference of the Union of Bulgarian Mathematicians, Mathematics and Education in Mathematics, 2012, pp. 259-265.

II.17. Todorova, M., K. Kanev. Educational Framework for Verification of Object-Oriented Programs, Proceedings of The Joint International Conference on Human-Centered Computer Environments HCCE’2012, 8-13 march, Hamamatsu, Japan, 2012, pp. 23-27.

II.18. Todorova, M. Methodological Aspects of an Approach for Verification of Object-Oriented Programs, Proceedings of the 6th International Conference IEEE Intelligence Systems’2012, 6-8 sept., Sofia, Bulgaria, 2012, pp. 153-158.

II.19. Todorova, M., P. Armyanov. Runtime Verification of Computer Programs and its Application in Programming Education, Global Science and Technology Forum: International Journal of Mathematics, Statistics and Operations Research, Vol. 1, No. 1, July 2012, Singapore, DOI: 10.5176/2251-3388_1.1.20, pp. 105-110.

II.20. Todorova, M. Implementation of an Approach for Verification of Procedural Programs, Global Science and Technology Forum: International Journal on Computing, Vol. 2, No. 2, June 2012, Singapore, DOI: 10.5176/2010-2283_2.2.170, pp. 70-75.

II.21. Todorova, M. Simulation of the Behavior of a System with Social Structure by Means of Generalized Nets, Proceedings of the 13th International Workshop on Generalized Nets, 29 Oct. 2012, London, UK, ISSN 1313-6860, pp. 62-68.

II.22. Todorova, M. Simulation of the Behavior of a Pure Imperative Synchronous Programming Language by Means of Generalized Nets with Stop-Conditions, Proceedings of the 6th International Conference Information Systems & Grid Technologies, 1-2 June, 2012, Sofia, Bulgaria, pp. 312-325.

II.23. Todorova, M. Correctness of a Formal Generalized Net Project of a Class of an Object-Oriented Program, Proceedings of the 12th International Workshop on Generalized Nets, 17 June 2012, Burgas, Bulgaria, ISSN: 1313-6860, pp. 73-78.

II.24. Todorova, M. Applying Program Verification Methods in Software Specialists Education, Proceedings of the 7th International Technology, Education and Development Conference, 4-6 March, 2013, Valencia, Spain, ISBN: 978-84-616-2661-8, pp. 6260-6270.

II.25 Todorova, M. Correctness of the Formal Generalized Net Project of the Connections Between Procedural Program Functions, International Journal Advanced Studies in Contemporary Mathematics, South Korea, Vol. 23, 2013, No. 3, July 2013 (in print) – with SJR Factor.

II.26 Todorova, M. Using generalized nets for program verification, Comptes rendus de l'Academie Bulgare des Sciences, No 8, Tome 66, 2013, ISSN: 1310-1331 (in print) – with Impact Factor.

II.27. Тодорова, М. Програмиране на C++, Част I, СИЕЛА СОФТ ЕНД ПУБЛИШИНГ, София, първо издание, ISBN 954-649-454-2(1), 2002, 375 стр.; второ преработено и допълнено издание, ISBN 978-954-28-0704-9, 379 стр.

II.28. Тодорова, М. Програмиране на C++, Част II, СИЕЛА СОФТ ЕНД ПУБЛИШИНГ, София, 2002, 2004, ISBN 954-649-480-1 (ч. 2), (483 стр.); второ преработено и допълнено издание, София, ISBN 954-649-480-1 (ч. 2), 2008, 495 стр.

II.29. Тодорова, М. Езици за функционално и логическо програмиране. Първа част. Функционално програмиране, СИЕЛА СОФТ ЕНД ПУБЛИШИНГ, София, ISBN 978-954-28-0828-2, преработено и допълнено издание, 2001, 2003, 2006, 2010, 227 стр.

II.30. Тодорова, М. Езици за функционално и логическо програмиране. Втора част. Логическо програмиране, преработено издание, СИЕЛА СОФТ ЕНД ПУБЛИШИНГ, София, ISBN 954-649-559-X, 2003, 255 стр.

II.31. Азълов, П., Ф. Златарова, М. Тодорова. Информатика за 10. клас – профилирана подготовка, Просвета, София, ISBN 954-011-438-1, 2003, 352 стр.

II.32. Тодорова, М., П. Армянов, Д. Зотева, К. Николов. Сборник от задачи по програмиране на C++. Част първа. Увод в програмирането, ТехноЛогика ЕООД, София, ISBN 978-954-9334-06-7, 2008, 357 стр.

II.33. Тодорова, М., П. Армянов, К. Николов. Сборник от задачи по програмиране на C++. Част втора. Обектно-ориентирано програмиране, ТехноЛогика ЕООД, София, ISBN 978-954-9334-09-8, 2008, 528 стр.

II.34. Тодорова, М. Обектно-ориентирано програмиране на базата на езика C++, СИЕЛА СОФТ ЕНД ПУБЛИШИНГ, София, 2011, ISBN 978-954-28-0909-8, 394 стр.

II.35. Тодорова, М. Структури от данни и програмиране на езика C++, СИЕЛА СОФТ ЕНД ПУБЛИШИНГ, София, 2011, ISBN 978-954-28-0990-6, 334 стр.

Приложение

Хабилитационният труд е съставен въз основа на 21 публикации на доц. Магдалина Тодорова, от които 20 статии, издадени през последните 5 години и 1 книга, издадена през 2002 год. Всички работи с изключение на статиите с номера II.17 и II.19 са самостоятелни.

Към края на март 2013 г. са забелязани следните цитирания на тези публикации:

II.1. Тодорова, М. Дизайн на език за верификация на програми на обектно-ориентирани езици, Конференция на съюза на учените в България, секция Информатика, 2007, Годишник на секция „Информатика“, Съюз на учените в България, ISSN 1313-6852, том 1, 2008, стр. 40-48.

е цитирана в:

1. М. Божилова, Д. Николова, Техники за осигуряване на отказоустойчивост и възстановяване в софтуерни системи, Сборник доклади от научна конференция с международно участие „Военни технологии и системи за осигуряване на отбраната“ 2011 (MT&S 2011) 8-9 декември 2012, ISBN 978-619-900 24 – I – 4, София, 2012, стр. 262 – 269.

II.2. Тодорова, М. Верификация и валидация на реализациите на абстрактни типове данни, Математика и математическо образование, Доклади на 37. пролетна конференция на СМБ, април, 2008, стр. 414-419.

е цитирана в:

2. М. Божилова, Д. Николова, Техники за осигуряване на отказоустойчивост и възстановяване в софтуерни системи, Сборник доклади от научна конференция с международно участие „Военни технологии и системи за осигуряване на отбраната“ 2011 (MT&S 2011) 8-9 декември 2012, ISBN 978-619-900 24 – I – 4, София, 2012, стр. 262 – 269.

II.3. Тодорова, М. Формална верификация на процедурни и обектноориентирани програми с използване на системата за доказване на теореми HOL, сп. Автоматика и информатика, издание на Съюза по автоматика и информатика „Джон Атанасов“, ISSN 0861–7562, год. XLII, кн. 2, 2008, стр. 25-27.

е цитирана в:

3. М. Божилова, Дисертационен труд на тема „Повишаване на надеждността на разпределено софтуерно приложение“ за получаване на образователна и научна степен „Доктор“, 2009.

II.5. Тодорова, М. Верификация на програми по време на изпълнение, сп. Математика и информатика, кн. 1, 2009, стр. 31-38.

е цитирана в:

4. М. Божилова, Д. Николова, Техники за осигуряване на отказоустойчивост и възстановяване в софтуерни системи, Сборник доклади от научна конференция с международно участие „Военни технологии и системи за осигуряване на отбраната“ 2011 (MT&S 2011) 8-9 декември 2012, ISBN 978-619-900 24 – I – 4, София, 2012, стр. 262 – 269.

II.8. Todorova, M. Grid Framework for e-learning Services, Proceedings of the 4th International Conference Information Systems & Grid Technologies, May 28 – 29, 2010, Sofia, Bulgaria, pp. 153-163.

е цитирана в:

5. D. Orozova, V. Jecheva, An Intelligent Approach to Electronic Test Results Evaluation, Proceedings of the 41. Spring Conference of the Union of Bulgarian Mathematicians, Mathematics and Education in Mathematics, 2012, pp. 293-298.

6. M. Nisheva-Pavlova, Digital Libraries and Cloud Computing, Sixth International Conference ISGT'2012, Sofia, Bulgaria, June 1–3., 2012, pp.231-239.

7. D. Orozova, Modeling of Electronic Learning Environment with Generalized Nets, 11-th International Workshop on Intuitionistic Fuzzy Sets and Generalized Nets, october 12, 2012, Warsaw, published in New Trends in Fuzzy Sets, Intuitionistic Fuzzy Sets, Generalized Nets and Related Topics. Volume II: Applications, IBS PAN - SRI PAS, Warsaw Poland, pp. 123-130.

II.14. Todorova, M. Model Checker of Object-Oriented Programs based on Generalized Nets, 10th International Workshop on Intuitionistic Fuzzy Sets and Generalized Nets (IWIFSGN'2011), Warsaw, Poland, September 30, 2011, published in *Recent Developments in Fuzzy Sets, Intuitionistic Fuzzy Sets, Generalized Nets and Related Topics. Foundations and Applications*, *vol. II: Applications*, Systems Research Institute of Polish Academy of Sciences, ISBN 83-894-7541-3, pp. 309-320.

е цитирана в:

8. P. Gocheva, V. Gochev, Verification of a Token Transfer Algorithm in a .NET Implementation of Generalized Nets, Proceedings of the 13th International Workshop on Generalized Nets, 29 October 2012, London, 69–76.

9. В. Атанасова, Изследване на алгоритми за конструиране на обобщеномрежови модели, дисертационен труд за присъждане на ОНС доктор по информатика, Институт по информационни и комуникационни технологии, БАН, 2013, 122 стр.

II.15. Тодорова, М. Построяване на коректни обектно-ориентирани програми чрез изграждане на техни обобщеномрежови модели, Конференция на съюза на учените в България, секция „Информатика", 2011, Годишник на секция „Информатика", Съюз на учените в България, ISSN 1313-6852 том 4, 2011, стр. 1-28.

е цитирана в:

10. K. Kaloyanova, Design from data: how to use requirements for better information system analysis and design, Proc. of the Int. Conference Informatics in Scientific Knowledge, Varna, June, 26-29, 2012, pp. 189-197

11. V. Atanassova, The Minimal Solution of a Problem in Generalized Nets, 6-th International Conference IEEE Intelligence Systems'2012, 6-8 sept., Sofia, Bulgaria, 2012, vol. 2, pp. 159-163.

12. D. Dimitrov, On the Global Operator G2 over Generalized Nets, Proc. of 12th International Workshop on Generalized Nets, 17 June 2012, Burgas, Bulgaria, 1-5, ISSN: 1313-6860.

13. D. Dimitrov, On the Global Operator G6 over Generalized Nets, Annual of „Informatics" Section, Union of Scientists in Bulgaria, 2012, ISSN 1313-6852, Volume 5, 2012, pp. 43-52.

14. P. Gocheva, V. Gochev, Verification of a Token Transfer Algorithm in a .NET Implementation of Generalized Nets, Proceedings of the 13th International Workshop on Generalized Nets, 29 October 2012, London, 69–76.

15. N. Novachev, I. Marinov, D. Stratiev, T. Pencheva, Kr. Atanassov, Generalized net model of the process of evaluation of the environmental impact of refinery activity, Proceedings of the 13th International Workshop on Generalized Nets, 29 October 2012, London, 56–61.

16. D. Stratiev, I. Marinov, T. Pencheva, Kr. Atanassov, Generalized net model of an oil refinery, Proc. of 12th International Workshop on Generalized Nets, 17 June 2012, Burgas, Bulgaria, 10-16, ISSN: 1313-6860.

17. В. Атанасова, Изследване на алгоритми за конструиране на обобщеномрежови модели, дисертационен труд за присъждане на ОНС доктор по информатика, Институт по информационни и комуникационни технологии, БАН, 2013, 122 стр.

II.16. Todorova, M. Verification of Procedural Programs via Building there Generalized Nets Models, Proceedings of the 41. Spring Conference of the Union of Bulgarian Mathematicians, Mathematics and Education in Mathematics, 2012, pp. 259-265.

е цитирана в:

18. D. Dimitrov, On the Global Operator G2 over Generalized Nets, Proc. of 12th International Workshop on Generalized Nets, 17 June 2012, Burgas, Bulgaria, 1-5, ISSN: 1313-6860.

19. D. Dimitrov, On the Global Operator G6 over Generalized Nets, Annual of „Informatics“ Section, Union of Scientists in Bulgaria, 2012, ISSN 1313-6852, Volume 5, 2012, pp. 43-52.

20. P. Gocheva, V. Gochev, Verification of a Token Transfer Algorithm in a .NET Implementation of Generalized Nets, Proceedings of the 13th International Workshop on Generalized Nets, 29 October 2012, London, 69–76.

21. A. Shannon, B. Riečan, E. Sotirova, G. Inovska, Kr. Atanassov, M. Krawczak, P. Melo-Pinto, Taekyun Kim, A generalized net model of university subjects rating with intuitionistic fuzzy estimations, 16th International Conference on IFSs, Sofia, 9–10 Sept. 2012, Notes on Intuitionistic Fuzzy Sets Vol. 18, 2012, No. 3, pp. 61–67.

22. В. Атанасова, Изследване на алгоритми за конструиране на обобщеномрежови модели, дисертационен труд за присъждане на ОНС доктор по информатика, Институт по информационни и комуникационни технологии, БАН, 2013, 122 стр.

II.17. Todorova, M., K. Kanev. Educational Framework for Verification of Object-Oriented Programs, Proceedings of The Joint International Conference on Human-Centered Computer Environments HCCE'2012, 8-13 march, Hamamatsu, Japan, 2012, pp. 23-27.

е цитирана в:

23. B. Bontchev, D. Vassileva, Adaptive Edutainment in UML, Proceedings of the 8th WSEAS International Conference on Educational Technologies (EDUTE'12), Porto, Portugal, July 1-3, 2012, 129-134.

24. K. Kaloyanova, Using the Project Approach in IS Education, Proceedings of the 8th Annual International Conference on Computer Science and Education in Computer Science 2012, 5–10 July 2012, Boston, USA, pp. 36-40.

25. И. Дончев, Э. Тодорова, Опыт обучения объектно-ориентированному программированию и C++11, Восьмая международная научно-практическая конференция „ИОН“ 2012 (IES-2012), 1-5 окт. 2012, Vinnytsia, Ukraine, стр. 74-75, 2012.

26. I. Donchev, Experience in Teaching C++11 within the Undergraduate Informatics Curriculum, International Science Journal Informatics in Education, Vol. 12, No. 1, 2013, Vilnius University, pp. 1-21.

27. A. Shannon, B. Riečan, E. Sotirova, G. Inovska, Kr. Atanassov, M. Krawczak, P. Melo-Pinto, Taekyun Kim, A generalized net model of university subjects rating with intuitionistic fuzzy estimations, 16th International Conference on IFSs, Sofia, 9–10 Sept. 2012, Notes on Intuitionistic

Fuzzy Sets Vol. 18, 2012, No. 3, pp. 61–67.

28. I. Donchev, E. Todorova, Training in Object-Oriented Programming and C++11, Computer and Information Science; Vol. 6, No. 2; 2013, Published by Canadian Center of Science and Education, ISSN 1913-8989 E-ISSN 1913-8997, pp. 84-92 (2 пъти я цитира).

II.18. Todorova, M. Methodological Aspects of an Approach for Verification of Object-Oriented Programs, Proceedings of the 6th International Conference IEEE Intelligence Systems'2012, 6-8 sept., Sofia, Bulgaria, 2012, pp. 153-158.

е цитирана в:

29. K. Kaloyanova, Successful Practices for Learning Information Systems Development, Proceedings of 7th International Technology, Education and Development Conference, 4th-6th March, 2013, Valencia, Spain, ISBN: 978-84-616-2661-8, pp. 4849-4855.

30. В. Атанасова, Изследване на алгоритми за конструиране на обобщеномрежови модели, дисертационен труд за присъждане на ОНС доктор по информатика, Институт по информационни и комуникационни технологии, БАН, 2013, 122 стр.

II.20. Todorova, M. Implementation of an Approach for Verification of Procedural Programs, Global Science and Technology Forum: International Journal on Computing, Vol. 2, No. 2, June 2012, Singapore, DOI: 10.5176/2010-2283_2.2.170, pp. 70-75.

е цитирана в:

31. A. Shannon, B. Riečan, E. Sotirova, G. Inovska, Kr. Atanassov, M. Krawczak, P. Melo-Pinto, Taekyun Kim, A generalized net model of university subjects rating with intuitionistic fuzzy estimations, 16th International Conference on IFSs, Sofia, 9–10 Sept. 2012, Notes on Intuitionistic Fuzzy Sets Vol. 18, 2012, No. 3, pp. 61–67.

32. P. Tcheshmedjiev, Verifying BPMN Processes using Generalized Nets, Annual of „Informatics" Section, Union of Scientists in Bulgaria, 2012, ISSN 1313-6852, Volume 5, 2012, pp. 111-119.

II.27. Тодорова, М. Програмиране на С++, Част I, СИЕЛА СОФТ ЕНД ПУБЛИШИНГ, София, първо издание, ISBN 954-649-454-2(1), 2002, 375 стр.; второ преработено и допълнено издание, ISBN 978-954-28-0704-9, 379 стр.

е цитирана в:

33. Hr. Kostadinova, Kr. Yordzhev, An Entertaining Example for the Usage of Bitwise Operations in Programming, Mathematics and Natural Science, Proceedings of the Fourth International Scientific Conference – FMNS2011, 8 – 11 June 2011 Faculty of Mathematics and Natural Science,VOLUME 1, pp. 159-168.

34. Kr. Yordzhev, An example for the use of bitwise operations in programming, Proceedings of the 38 Spring Conference of the Union of Bulgarian Mathematicians, Borovetz, 2009, pp. 196-202.

35. Hr. Kostadinova, Kr. Yordzhev, A Representation of Binary Matrices, Proceedings of the Thirty Ninth Spring Conference of the Union of Bulgarian Mathematicians, Albena, Bulgaria, April 6–10, 2010, pp. 198-206.

36. И. Дончев, Э. Тодорова, Полиморфизм в курсе обьектно ориентированного программирования – дидактические аспекы, Proceedings of the Sixth International Conference „Internet–Education–Science", Vinnytsia, Ukraine, October 7 –11, 2008, pp. 110-113, (по 2 пъти го цитира).

37. И. Дончев, Э. Тодорова, Операции с обьектами и коммуникация между обьектами в обучении обьектно ориентированного программирования, Proceedings of the Sixth International Conference „Internet–Education–Science“, Vinnytsia, Ukraine, October 7–11, 2008, pp. 106-109 (2 пъти го цитира).

38. И. Дончев, Э. Тодорова, Object-Oriented Programming in Bulgarian Universities' Informatics and Computer Science Curricula, Informatics in Education, 2008, Vol. 7, No. 2, pp. 159–172, 2008, Institute of Mathematics and Informatics, Vilnius, Lithuania.

39. I. Donchev, E. Todorova, Object-Oriented Exception Handling: Implications for Education, Proceedings of the 4-th Annual International Workshop on Computer Science and Education in Computer Science, Borovets June 2008, pp. 82-88, 2008.

40. І. Дончев, Е. Тодорова, Операції з об'єктами й комунікація між об'єктами у навчанні об'єктно-орієнтованому програмуванню, Вісник Вінницького політехнічного інституту, 2009, № 1, стр. 98-101, Украйна.

41. И. Дончев, Класове и обекти в обучението по програмиране във висшето училище, дисертационен труд за присъждане на ОНС „доктор“ по научната специалност 05.07.03 „Методика на обучението по информатика“, Велико Търново, 2009, 263 стр.

42. Hr. Hristov, Review and Outlooks of the Means for Visualization of Syntax Semantics and Source Code. Procedural and Object Oriented Paradigm – Differences, in Anniversary International Conference REMIA2010 (Research and Education in Mathematics, Informatics and their Applications), 10-12 dec. 2010, Plovdiv, pp. 443-450.

43. Н. Киров, Сборник от учебни материали по Въведение в програмирането, (Принципи на програмирането със С++, част I), Издателство „Демократични традиции - Деметра“, ISBN 954-9526-15-1, 2003.

44. Н. Киров, Сборник от учебни материали по Въведение в програмирането, (Принципи на програмирането със С++, част I), електронен учебник, http://www.math.bas.bg/ ~nkirov/book1.html, 2003.

45. Kr. Yordzhev, Bitwise operations related to a combinatorial problem on binary matrices, arXiv:1301.5100v1 [math.CO] 22 Jan 2013, Cornell University Library, http://arxiv.org/abs/ 1301.5100.

46. Н. Киров, Сборник от учебни материали по Програмиране и структури от данни, (Принципи на програмирането със С++, част II), Издателство „Демократични традиции - Деметра“, София, ISBN 954-9526-15-2, 2004.

47. Н. Киров, Сборник от учебни материали по Програмиране и структури от данни, (Принципи на програмирането със С++, част II), електронен учебник, http://www.math.bas.bg/~nkirov/book2/ book2.html, 2004.

48. Н. Георгиева, Програмиране на С++, електронен учебник по програмиране на С++, http://www.c-programing.hit.bg/.

49. К. Иларионов, М. Младенова, д.т.н. Г. Дінински, М. Добрикова, М. Ангелова, Национална изпитна програма за провеждане на държавни изпити за придобиване на втора степен на професионална квалификация, http://www.mon.bg/opencms/*export/ sites/mon/top_menu/ vocational/exam_programs/2007-5/5230901-IIst.pdf 2008.*

второ преработено и допълнено издание, ISBN 978-954-28-0704-9 (379 стр.).

50. Р. Жекова, Обработка на редици от числа, Национална конференция „Образованието в информационното общество“ 2006, http://sci-gems.math.bas.bg/jspui/ bitstream/10525/1503/1/adis-october-2006-099p-0103p.pdf.

51. Р. Жекова, Задачи за програмиране на итеративни процеси, Национална конференция „Образованието в информационното общество“ 2006, http://sci-gems.math.bas.bg:8080/jspui/handle/10525/1504.

52. О. Кър, И. Желязкова, Програмна реализация на средство за обработка и визуализация на данни от учебни сесии, Научни трудове на Русенския университет - 2009, том 48, серия 3.2, 187-191.

53. П. Азълов, Псведослучайни числа: експерименти по математика в час по информатика, сп. Математика и информатика, бр. 4, 2009.

54. П. Азълов, Въвеждане на рекурсията чрез абстракция и редици от задачи, Доклади на 39. пролетна конференция на СМБ, април, 2010, стр. 243-249.

55. Е. Келеведжиев, З. Дженкова, Алгоритми, програми и задачи. Ръководство за начална подготовка по информатика за олимпиади и състезания, Регалия 6, 2005.

56. П. Азълов, Обектно-ориентирано програмиране. Структури от данни и STL, Сиела, 2008.

57. Б. Йовчева, И. Иванова. Първи стъпки в програмирането на С++, изд.„KLMN“, ISBN-10: 954-8212-01-3; ISBN-13: 978-954-8212-01-4, 2006.

58. Б. Йовчева, И. Иванова. Ръководство за лабораторни упражнения по програмиране I част (на базата ба езика С++), изд. „Шуменси университет Епископ К. Преславски“, ISBN-10: 954-577-357-Х; ISBN-13: 978-954-577-357-0, 2006.

59. Р. Жекова, Лекции по информатика с програмиране на С++, http://l00list.hit.bg/ Дата на последната актуализация: 12.08.2009 г. http://www.l00list.hit.bg/ (последна актуализация 19.06.2012 г.)

60. И. Дамянов, Увод в програмирането, http://idamianov.web.officelive.com/introtoprogramming.aspx, електронен учебник, 2010.

61. Св. Енков, Shark’s С++, електронен учебник, http://enkov.com/cpp.htm, 2011.

62. Н. Бъчваров, Обектно-ориентирано програмиране на С++, електронен учебник, 2007-2010, http://www.nikbsoft.com/pgmett/moodle/

Цитиранията на трудовете, включени в хабилитационния труд, са както следва:

Цитирания на:	*Брой цитирани материали:*	*Брой цитирания в национални и чуждестранни издания:*	*Брой цитирания в международни издания:*	*ОБЩО цитирания на:*
статии	11	12	20	32
книга	1	24	6	30
ОБЩО	12	36	26	62