Годишник на секция Информатика **Annual of "Informatics" Section**
Съюз на учените в България **Union of Scientists in Bulgaria**
Том 5, 2012, 1-31 **Volume 5, 2012, 1-31**

АНАЛИЗ И ИЗВЛИЧАНЕ НА КОМПЮТЪРНИ ПРОГРАМИ

Калин Георгиев

Факултет по математика и информатика, Софийски университет
E-mail: kalin@fmi.uni-sofia.bg

Резюме: Представен е обзор на областите от компютърните науки Извличане на програми и Анализ на програми, който освен че изброява и дава примери на важни представители на различните методи в тези области, очертава принципните сходства и различния в тези методи и дава тяхна условна класификация. Методите условно се делят на такива, базирани на средствата на математическата логика и такива, предоставящи частични решения с приложение в някакъв специфичен практически аспект на проблема. Изброени са редица конкретни представители на методи за извличане и анализ на програми, както и конкретни примери за приложение на някои от тези представители.

1 Въведение

Извличане на програми

Извличането на програми (или също така – "автоматично програмиране") е широк дял от компютърните науки, обхващащ методите за автоматично генериране на програмен код на базата на определени спецификации. Подходите за специфициране на програми, в зависимост от конкретния метод, могат да бъдат най-разнообразни.

Синтезирането на програми, например, е важен клас методи за извличане на програми. При тези методи автоматично се конструира програмен код, удовлетворяващ определена спецификация, най-често зададена като формула в някаква логическа система [29]. Друг формален метод за генериране на програми е извличането на програми от доказателства на теореми, включително не-конструктивни [44]. За извлечените по тези начини програми е гарантирана тяхната коректност. Някои специални примери за генериране на код включват използването на системи за компютърна алгебра за оптимизация на програмен код, извършващ определени математически изчисления [31].

Друг клас механизми за автоматично генериране на софтуерен код с голямо практическо приложение е т.нар. "генеративно програмиране" [19]. Този клас включва техники като прототипи, шаблони, класове, аспекти и макропроцесори, които въвеждат опростени синтактични конструкции за генериране на често срещани или повторяеми видове код. Тези

техники намират широко приложение поради факта, че съкращават размера на изходния код и оптимизират работата на програмистите.

В модерните среди за програмиране код се генерира и от редица визуални редактори, в които програмистите използват предефинирани компоненти за изграждане на потребителския интерфейс на приложенията. Широко разпространение намират и някои метаезици за специфициране на определени компоненти от сложни софтуерни системи, като например UML (Unified Modelling Language) [46] спецификации, спецификации на Web услуги (напр. Web Service Definition Langiage) [50], и markup езици за дефиниране на потребителски интерфейс (напр. Extensible Application Markup Language) [51].

Автоматизирането на редица практически задачи, като например превеждането на данни от една схема в друга (data mapping), също включва автоматичното генериране на код на програми. Генерираните по този начин програми изпълняват задачата с гарантирана коректност спрямо зададеното съответствие между схемите [14], което може да е с разнообразен формат и също така автоматично извлечено на базата на екземпляри от данни от двете схеми.

Като обобщение ще цитираме заключението на Дейвид Парнас, пионер в софтуерното инженерство, който пръв въвежда концепцията за капсулация на информацията. Той заключва, че “автоматичното програмиране винаги е било евфемизъм за програмиране на език на по-високо ниво спрямо езика, който е бил достъпен за програмиста в дадения момент” [33]. В този смисъл, множеството съвременни средства за генериране на програмен код елиминират “откриването на колелото” в практически задачи. Често повтарящи се програмни конструкции, програмни конструкции, които са сложни или дълги за ръчно описание на съответния език за програмиране и такива, за които коректността е от особено значение, подлежат на автоматично генериране.

Генерирането на програма в голяма част от описаните случаи се състои в транслация на езици от по-високо ниво, които са диалектни за специфична проблемна област. Тъй като езиците за спецификация на определени компоненти от софтуерната система са фокусирани върху проблемната си област, това им позволява да имат по-голяма изразителна сила спрямо езиците за програмиране с общо предназначение, до които в крайна сметка се “превежда” спецификацията.

Анализ на програми

Най-общо казано, анализът на програми е дял от компютърните науки, обхващащ теоретични модели и методи за осигуряване и проверка на разнообразни свойства компютърните програми [2]. Различните видове анализ могат условно да бъдат разделени в няколко големи категории: статичен анализ, динамичен анализ и формална верификация.

Средствата за статичен анализ на програми работят с изходния код на програмата като анализират различни нейни свойства, без да я изпълняват. Разнообразието от свойства е огромно и варира от прости проверки за грешни идентификатори до типов анализ на програмата (static type chechking). Инструментите за статичен анализ намират широко приложение и са част от практически всеки модерен компилатор и среда за разработка. Освен проверка за коректност, статичния анализ има за цел също машинно независими оптимизации на кода на програмата.

Динамичният анализ разчита на серия от тестови изпълнения на програмата в реална или симулирана среда, при които се наблюдава и проверява нейното поведение. Под поведение в случая може да се разбират разнообразни параметри на програмата, като заета памет (и изобщо системни ресурси), стойност на определени променливи, свойства на аргументите на функции, резултати на функции и др. Целта на динамичния анализ е да обхване в максимална широчина пътищата на изпълнение в програмата и да проследни тяхната коректност чрез тестване със серия от подходящо подбрани стойности. Макар да не гарантира коректност от формална гледа точка, подходът намира широко практическо приложение и е важен инструмент за осигуряването на качеството на кода на големи софтуерни системи, особено когато в тях се извършват чести промени. Подходът дори стои в основата на стил за програмиране – разработване, базирано на тестове (test driven development) [43] – при който изграждането на кода на програмата започва с написването на тестовете, които тя трябва да удовлетворява.

Важен клас методи за анализ на програми са методите за формална верификация, при които чрез математически средства се доказва тотална и частична коректност на програмен код при определени условия за входа и изхода на програмата [40]. Подходът включва изграждане на формален модел на системата, на базата на който се доказва съответното свойство. Това е възможно най-"сигурният" подход за проверка на свойства на програми, тъй като математическият апарат позволява доказателство за коректност във всички случаи, за разлика от останалите методи, които гарантират само някои аспекти на коректността на програмите.

2 Извличане на програми

2.1 Програмиране на базата на модели

Програмирането, базирано на модели, е клас от методи за разработка на софтуер, който се базира на специфични за проблемната област (домейна) модели от високо ниво, които са универсални (не зависят от конкретната платформа за реализиране на системата). При програмирането с модели се набляга на абстрактно представяне на знанието и процесите в специфичната проблемна област, вместо на конкретни алгоритми и други изчислителни аспекти.

Ползата от този подход е, че предоставя езици (често визуални), които са по-лесно разбираеми за потребители, запознати добре с дадената проблемна област. Ако тези езици са достатъчно добре специфицирани и с ясна семантика, това позволява части от кода на софтуерните системи да бъдат генерирани автоматично, след което така генерираният програмен код се допълва със специфична бизнес логика.

2.2 Език за моделиране UML

Един от най-популярните езици за моделиране с общо предназначение е UML (Universal Modelling Langiage) [46]. Езикът включва набор от графични нотации за визуално описание на обектно-ориентирани системи. Чрез т.нар. UML диаграми се специфицират, визуали-

зират, модифицират, конструират и документират разнообразни аспекти на софтуерните системи. Тази аспекти включват техники за моделиране на данни (диаграми на същности и връзки), моделиране на бизнес процеси (т.нар. “work flows”) и моделиране на обекти. Макар UML диаграмите да намират широко приложение във всички части от жизнения цикъл на софтуерните системи, от интерес за нас са някои видове диаграми, на базата на които може да бъде генериран програмен код.

Чрез тези диаграми езикът UML е средство за създаване на код сам по себе си [47]. Платформи като Enterprise Archiect на Sparx Systems [41] позволяват визуалното построяване на:

- Диаграми на състоянията (State Machine Diagrams): Фигура 1 представя вид диаграма от вида на т.нар. поведенчески диаграми (behavioral diagrams). Диаграмите от този тип описват софтуерната система като краен набор от състояния и преходи между тях.

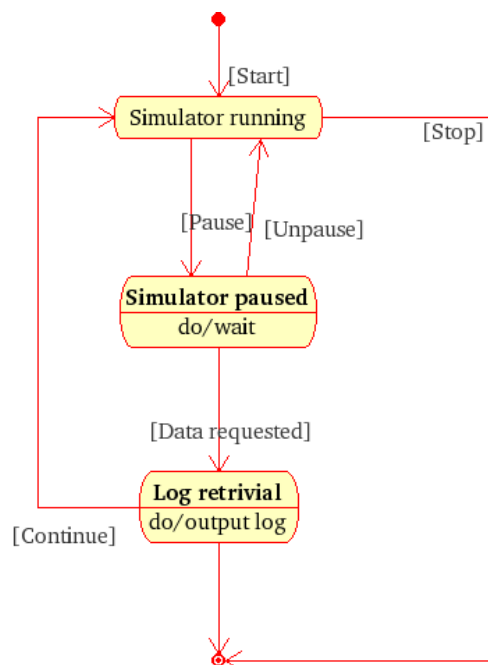

Figure 1: State machine диаграма

- Диаграми на взаимодействието (Interaction diagrams): Фигура 2 представя пример за клас поведенчески диаграми, които наблягат на потока на данни и управлението между елементите в моделирана система.

- Диаграми на дейностите (Activity diagrams): визуализират последователни процеси. На фигура 3 е представена примерна диаграма на дейностите.

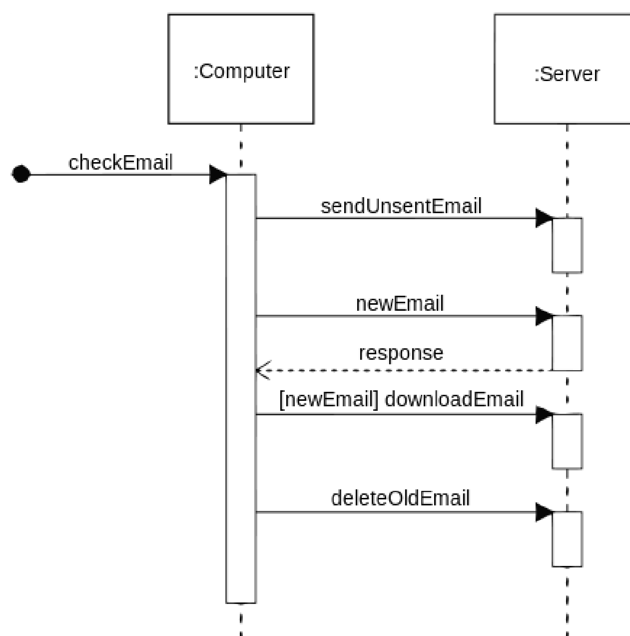

Figure 2: Sequence диаграмата е тип диаграма на взаимодействието

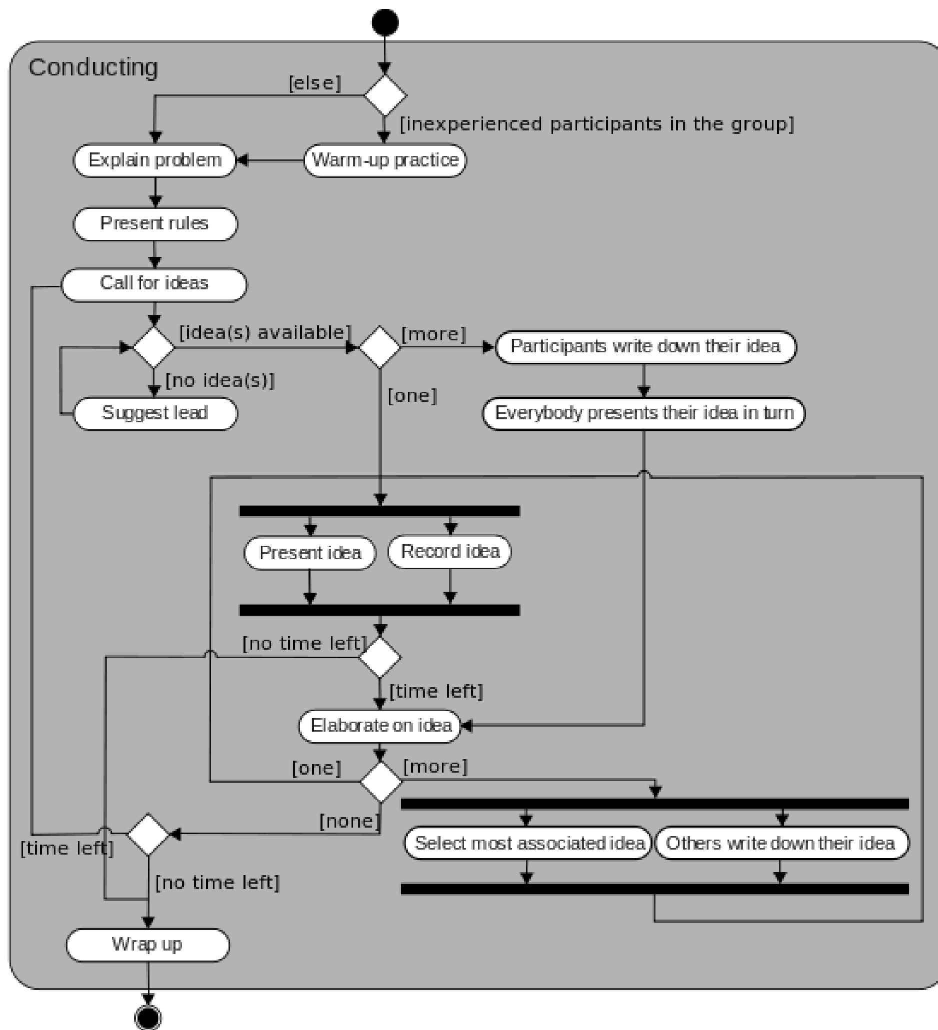

Figure 3: Activity диаграмата описва последователност от действия

Платформата позволява генериране на скелетен програмен код на базата на построени диаграми, който да стане основата на софтуерната система.

Друг разпространен инструмент, поддържащ генериране на код от UML диаграми е IBM Rational Rhapsody [23] – Фигура 4. Rational Rhapsody е популярна платформа за разработка на софтуер, която поддържа генериране на програмен код на базата на UML диаграми. Платформата поддържа генериране на код на разнообразни езици за програмиране – C, C++, Ada, Java, C# от визуални модели.

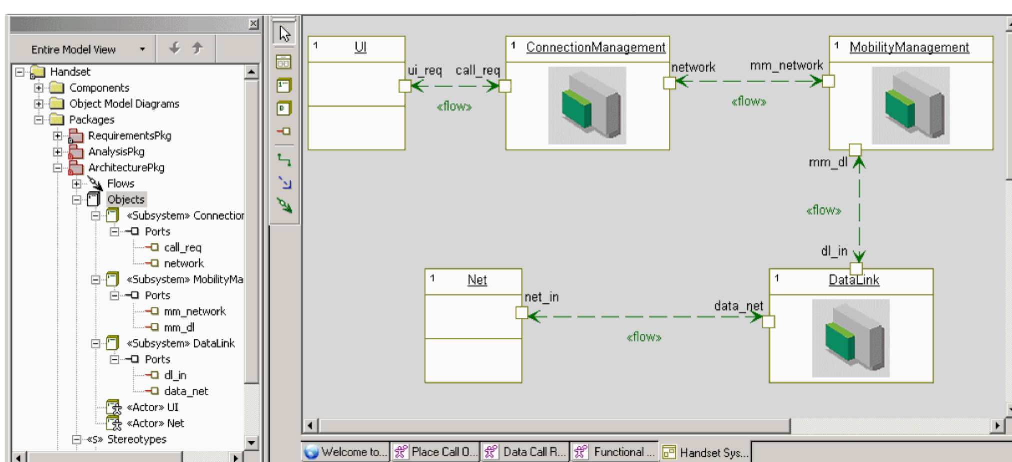

Figure 4: Изглед от IBM Rational Rhapsody

2.3 Генериране на код за потребителски интерфейс

Изграждането на програмен код, реализиращ потребителски интерфейс, често се състои в множество трудоемки, повторяеми дейности. От обектно-ориентирана гледна точка, елементите на потребителския интерфейс в разнообразните платформи са множество инстанции на еднотипни обекти, различаващи се в параметрите при конструирането им (напр. координати, етикети и пр.). В този смисъл кодът, описващ потребителски интерфейс, се състои в серия еднообразни програмни конструкции, чието "ръчно" дефиниране е неефективно.

За да адресират този проблем, модерните платформи за разработка на приложения представят т.нар. езици на "маркиране" (markup languages), които съкращават и улесняват дефинирането на потребителския интерфейс. Това са декларативни езици, описващи елементите на интерфейса, често чрез някакъв тип дърво на документния обектен модел (document object model – DOM).

XAML (Extensible Application Markup Language) [51] е език за описание на потребителския интерфейс в платформата Silverlight на Microsoft Corporation. Езикът, както подсказва името му, е базиран на XML и предоставя множество от примитиви на потребителския интерфейс, които се подреждат в йерархична структура и дефинират изгледа на приложението. На следния пример:

```
<Canvas
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <TextBlock>Hello World!</TextBlock>
</Canvas>
```

се вижда елементарен код на XAML, описващ прост блок с текст. Изходните файлове на XMLS се транслират до езика BAML (Binary Application Markup Language), който се интерпретира от платформата .NET за построяване на визуалното дърво на приложението.

Още по-добър пример за декларативен език за описание на потребителски интерфейс, който се превежда до програмен код, е езикът на Adobe MXML. По подобие на XAML, езикът предоставя примитиви за изграждане на изгледа на приложението. MXML кодът преминава през предварителна фаза на транслация, при която от него се генерира код на езика ActionScript – основният език на платформата. Допълнително, средата за разработка на платформата Adobe Flex – Flash Builder – предоставя визуален редактор, чрез който разработчиците могат да изграждат MXML код автоматично чрез WYSIWYG система.

2.4 Описание на Web услуги

Друга често срещана задача, включваща нуждата от съществени количества повторяем код и съответно подлежаща на автоматизация, е комуникацията с Web услуги (Web services). При архитектурите, ориентирани към услуги, сървърните компоненти на услугата често предоставят описание на интерфейса на услугата, от което директно може да бъде извлечен еднотипен код за прилагане в нейните клиенти.

Такъв език за описание на интерфейса на Web услуги е WSDL (Web Services Description Language) [50]. Езикът дефинира предоставените от услугата методи в достатъчно детайли така, че редица среди за програмиране включват средства за автоматично генериране на клиентски код за комуникация с услугата. Такива среди са, например, Microsoft Visual Studio и Adobe Flash Builder.

2.5 Генеративно програмиране

Преизползването на програмен код (т.е. елиминирането на необходимостта еднотипен програмен код да се програмира "ръчно" за множество от различни програми) се постига чрез "изнасянето" на програмен код с общо предназначение в програмни библиотеки и модули. Това е ключов компонент на съвременната практика при програмирането и представлява едно от основните приложения на обектно-ориентираното програмиране.

Растежът на софтуерните библиотеки се налага поради необходимостта да се поддържат по-големи многообразия от поддържани функционалности и компоненти. С нарастването на обема на възможностите на дадена библиотека, обаче, се проявява класически проблем при преизползването на програмен код – нарастващата сложност на библиотеката. Внасянето на вариации на поддържаните функции и компоненти се облекчава от използването на някои

шаблони за обектно-ориентиран дизайн, но въпреки това увеличаването на универсалността на програмния код, което налага въвеждането на нови класове и обновяването на стари, води до т.нар. "фрагментация на дизайна" [27].

Подходите за увеличаване на възможностите на софтуерните библиотеки условно могат да се разделят на два вида – композиция и трансформация. За композиция се говори, когато стратегията за поддържане на вариации на функционалността е реализиране на множество независими елементи на библиотеката за всяка вариация. Това "хоризонтално" скалиране води до високо бързодействие на резултатния код, използващ библиотеката, но също така и до висока сложност на самата библиотека, поради потенциалния експоненциален взрив на независими компоненти в нея.

Алтернативният подход включва групиране на множество сходни категории от възможности в по-малък брой независими елементи на библиотеката, които чрез по-подробна параметризация успяват да постигнат същото многообразие от вариации на функционалността. По този начин се постига по-високо ниво на абстракция на програмния код. Този подход, обаче, води до влошена производителност. Кодът на отделните функции в библиотеката се разклонява и усложнява поради необходимостта да "взима решения" по време на изпълнение.

Описаният проблем е валиден не само за библиотеки от програмен код, но и за модулите на програмите като цяло, когато тяхната сложност започне да нараства поради необходимостта да поддържат голямо множество функционалности.

Генераторите на код разрешават проблема на двата подхода за скалиране на библиотеки чрез обединяването им. Чрез конструкции от по-високо ниво се дефинират по-силно параметризирани "шаблони" на елементи (функции, методи, класове и др.) на програмата. За разлика от подхода на трансформацията, обаче, за всяка специфична, конкретна употреба на даден програмен елемент автоматично се генерира "специализация", реализираща специфичната логика за дадената употреба. Нещо, което при подхода на композицията се прави ръчно от програмиста. По този начин се постигат едновременно високо ниво на абстракция и запазване на бързодействието.

Прост пример за такъв тип генериране на код са шаблоните в езика C++. Следната функция:

```
template <class T>
void printArray (T array[], int n){
   for (int i = 0; i < n; i++)
    cout << array[i];
}
```

реализира отпечатване на масив с произволен тип на елементите. Ако тази функция се ползва от програмата, например, за цели числа и числа с плаваща запетая, в изпълнимия код на програмата ще се срещат две алтернативи на функцията – едната, компилирана за цели числа, а другата – за числа с плаваща запетая.

Друг прост пример е C/C++-препроцесорът, който предоставя език за дефиниране на т.нар. макроси. Макросите са части от текста на програмата, които се заместват чрез

субституция преди програмата да се компилира и служат за съкращаване и опростяване на програмния текст.

2.6 Аспектно-ориентирано програмиране: AspectJ

"Разделянето на отговорност" (separation of concerns) е базов принцип при дизайна и реализацията на сложни софтуерни системи. Принципът се състои в разделянето на компонентите на програмата по такъв начин, че изолирани нейни части (класове, методи, функции) да имат възможно най-тясна специализация и да реализират по възможност "атомарни" функционалности на програмата. Целта е логиката на отделни части от програмата да не се припокрива с другите части на програмата и по този начин да се минимизират промените в цялата програма, когато някой нейн отделен компонент претърпи модификация или разширение. Модулното програмиране и капсулацията са традиционни техники за постигане на разделянето на отговорности.

Разпространените методи за разделянето на отговорностите в програмата (като модулното програмиране и ООП) традиционно се фокусират върху идентифицирането и композирането на функционални единици, които се представят чрез обекти, модули, процедури и пр. [3]. Съществуват и аспекти на това разделение, обаче, които не са решени добре от класическите подходи. Такива са свойства на системата, засягащи повече от един неин модул, като например синхронизация, взаимодействие между компонентите и записване на данните на постоянен (персистентност). Такива свойства не могат да се изолират добре при функционалната декомпозиция и осигуряването им води до малки фрагменти повторяем код, който е "разпръснат" из множество иначе несвързани модули.

Изобщо, "разпръснати" отговорности (cross-cutting concerns) се наричат такива аспекти на програмата, които оказват влияние на голяма част от останалите аспекти. Или с други думи, код със специфична отговорност, който се среща като част от множество други функции на програмата, чиито задачи не са пряко свързани с въпросната отговорност. Разпръснатите отговорности не могат да бъдат декомпозирани от останалата част на системата по лесен и "чист" начин, което води до дублиране на код и обвързване на код (внасяне на съществени зависимости между отделни части от програмата).

Например, нека имаме приложение, което комуникира с база данни за постигане на определена цел. Често при такива приложения е нужно за всяко обръщение (заявка) към базата данни да се записва информация в дневник (log). Тази информация се записва с цел анализиране на някои параметри на комуникацията, като например обем на обменените данни и време за изпълнение на заявката. Записването на такава информация би наложило във всеки метод, комуникиращ с базата данни, да "оградим" изпълнението на заявката с код, записващ определена информация в дневника. Освен, че тази логиката не е директно свързана с отговорностите на отделните методи, това би довело и до повторение на код. Всяка промяна в интерфейса на функциите за записване в дневника или промяна на данните, които се записват, би довело до промяна и във всички методи, зависещи от тях. Това е типичен пример за "разпръсната" функционалност.

Аспектно-ориентираното програмиране е парадигма, позволяваща постигането на по-висока степен на модуларизация на програмите чрез отделяне на разпръснати функционалности (separation of concerns). Ще дадем пример за решение на описания проблем чрез

средата за аспектно-ориентирано програмиране AspectJ за езика Java.

AspectJ позволява дефиниране на т.нар. "разрези" на програмата (pointcuts). Разрезите на програмата представляват дефинирани по определен начин моменти (точки) от нейното изпълнение. Например, следният код на AspectJ (който дефинира над-множество на езика Java):

```
pointcut query() : execution(*query*(..) )
                   && this(DatabaseConnection);
```

дефинира разрез "query" при всяка точка в програмата, която се обръща към метод на някаква инстанция на класа DataConnection, чието име съдържа думата "query".

За така дефинирания разрез може да се определи програмен код, който да се изпълнява автоматично преди и след разреза:

```
before () : query() {
  Logger.log (/*some information before the query is executed*/);
}
after () : query() {
  Logger.log (/*some information after the query is executed*/);
}
```

По този начин кодът на този "рапръснат" аспект на програмата се генерира автоматично и в изходния код е "изолиран" от останалата част на програмата.

2.7 Адаптиране на код с XVCL

Видяхме как единият от проблемите на софтуерното разработване – преизползването на код – може да се адресира чрез методи за генериране на код. Като последен пример в тази посока ще приведем т.нар. автоматично конфигуриране на варианти (variant configuration) и неговият представител – системата XVCL [52]. Системата е разработена от лабораторията по софтуерно инженерство на Сингапурския национален университет.

При системи като XVCL на кода се гледа по-скоро като на текст, т.е. системата не се базира на неговата семантика и съответно не зависи от нея. Това, което системата предлага, е средства за адаптация на този програмен текст, когато се прилага за различни цели, като така адаптираният код може в последствие също да се адаптира. По този начин се изгражда библиотека от фрагменти (frames), които могат да се комбинират и адаптират за решаването на конкретни задачи, като по този начин се избягва разработването на дублиращ се или аналогичен код за всяка конкретна задача.

Следният прост пример илюстрира как фрагментът “A”, описан на XML-базирания език на XVCL:

```
<x-frame name="A" >

Contents of A: before B
<adapt x-frame="B"/>
Contents of A: after B

</x-frame>
```

може да адаптира друг фрагмент – “B”:

```
<x-frame name="B" >
Contents of B
</x-frame>
```

При което се генерира следният “код”:

```
Contents of A: before B
Contents of B
Contents of A: after B
```

Показаният прост пример онагледява подхода на системи като XVCL – текстови трансформации над библиотеки от код (или каквито и да е файлове) с цел адаптирането му към определена задача. Системата позволява конструкции като “adapt” да се структурират в цели сложни йерархии, по този начин произвеждайки разнообразни варианти на първоначалния фрагмент.

Системата се прилага за управление на т.нар. “варианти на продукти” (Software Product Lines). При тях целта е производството на серия от еднотипни продукти, които обаче могат да се различават на всички нива – потребителски изисквания, дизайн, целева платформа и др. В такива ситуации различните варианти могат да имат много различния в детайлите на процеса на разработка, но и много големи общи сегменти от код. Ключовата дейност при този метод е идентификацията на тези общи части и дефинирането на просто правила за адаптацията им към специфичен вариант (или конфигурация) на системата. По този начин се избягва “ръчното” извършване на редица банални, повторяеми и подлежащи на грешки дейности коло преизползването на съществуващ код.

2.8 Генериране на код от формални спецификации

В предишните няколко секции разгледахме методи за генериране на (части от) програмен код на базата разнообразни спецификации – XML файлове с описание на потребителски интерфейси или Web услуги, UML модели, аспекти и формални граматики. При всички разгледани случаи изходният език, на базата на който се генерира код, също е формално специфициран. Въпреки това, причината да отделим следващите няколко метода в категорията "формални" е, че са базирани на средствата на математическата логика под някаква форма и позволяват приложението на формален математически апарат за изследване, описание и доказателство на свойствата на генерираните програми.

2.9 Lex & Yacc

Много интересен пример за генератори на програми са генераторът на лексически анализатори Lex [26] и генераторът на компилатори Yacc (Yet another compiler compiler) [53]. Лексическите анализатори и синтактичните анализатори (парсери) са стандартна част от всеки компилатор и интерпретатор. При фиксирана граматика, описваща съответния език, за конструирането на лексическия и синтактичния анализатори обикновено съществуват алгоритми, което позволява автоматизацията на процеса.

Lex се управлява от таблици от регулярни изрази и съответните им програмни фрагменти. От тях Lex генерира програми, които интерпретират входен поток от текст, като го раздробяват на последователност от низове, удовлетворяващи съответните регулярни изрази. Разпознаването на изразите се реализира чрез краен автомат, построен от Lex.

Компютърната програма Yacc е генератор на парсери, разработена от Stephen C. Johnson за операционната система Unix през 1970. Yacc генерира парсери на входни програми, описани чрез аналитична граматика сходна с BNF (Backus–Naur Form).

2.10 Системи за доказателство на теореми HOL и Coq

HOL (Higher Order Logic) [21] е фамилия системи за автоматично доказателство на теореми (proof assistants), базирани на обща логическа система. Системата предоставя абстрактен тип данни, с който се описват доказани теореми и "библиотека" от такива основни теореми. Нови теореми могат да се построяват чрез надграждане на съществуващите теореми чрез определени правила за извод в логика от по-висок ред.

Освен като среда за доказателство на теореми, HOL предоставя и средства за генериране на код от формални спецификации в съответната логическа система. Спецификациите се "превеждат" в изпълним код на езиците SML, OCaml, Haskell и Scala [9, 22].

При процеса на генериране на код, логически единици като константи, типове и класове се съпоставят на езикови конструкции в целевия език за програмиране. "Носителят" на семантиката на генерираната програма са теореми за равенство в логическата спецификация.

На генерираната програмата може да се гледа като на система за пренаписване на термове (term rewriting system). Тъй като всяка отделна операция на програмата си има логически еквивалент в спецификацията, за генерираната програма е гарантирана нейната частична коректност.

Да разгледаме следния пример от [22] за спецификация на "амортизирана" опашка. Амортизираната опашка е метод за реализация на структурата от данни опашка на език за програмиране без странични ефекти (каквито са повечето функционални езици за програмиране). Проблемът с класическия алгоритъм за опашка се състои в това, че операцията за вмъкване на елемент налага модификация на последния елемент на опашката. Тъй като това не е възможно при немутираща (immutable) структура от данни, се налага алтернативна реализация, като например дадената тук, която разчита на два стека.

```
datatype 'a queue = AQueue 'a list  'a list

definition empty :: 'a queue where
  empty = AQueue [] []

primrec enqueue :: 'a -> 'a queue -> 'a queue where
  enqueue x (AQueue xs ys) = AQueue (x # xs) ys

fun dequeue :: 'a queue -> 'a option ? 'a queue where
  dequeue (AQueue [] []) = (None, AQueue [] [])
  | dequeue (AQueue xs (y # ys)) = (Some y, AQueue xs ys)
  | dequeue (AQueue xs []) =
      (case rev xs of y # ys -> (Some y, AQueue [] ys))
```

Примерната спецификация води до генериране на следния код на езика Haskell:

```
module Example where {
     data Queue a = AQueue [a] [a];
     empty :: forall a. Queue a;
     empty = AQueue [] [];
     dequeue :: forall a. Queue a -> (Maybe a, Queue a);
     dequeue (AQueue [] []) = (Nothing, AQueue [] []);
     dequeue (AQueue xs (y : ys)) = (Just y, AQueue xs ys);
     dequeue (AQueue (v : va) []) =
       let {
         (y : ys) = reverse (v : va);
       } in (Just y, AQueue [] ys);
     enqueue :: forall a. a -> Queue a -> Queue a;
     enqueue x (AQueue xs ys) = AQueue (x : xs) ys;
}
```

По подобие на системата HOL, Coq е система за доказателство на теореми, разполагаща със собствена логическа система и "вграден" език за функционално програмиране. Системата позволява работа с логически формули, изразяващи свойства за програми на вградения език, като тези програми могат да бъдат "експортирани" като изпълним код [10].

2.11 Системи за компютърна алгебра

Компютърната алгебра (или още "символни изчисления" е термин, обобщаващ разнообразни алгоритми и методи за манипулиране на математически изрази и уравнения в символна форма, описани на определен формален език. Системите за компютърна алгебра (Computer Algebra Systems – CAS) са софтуерни системи, предоставящи автоматични средства, улесняващи боравенето със символна математика.

Инструментите за преобразуване на изрази, предоставени от системите за компютърна алгебра са действително разнообразни – опростяване, субституция, диференциране, интегриране, намиране на решения (включително на някои ДУ), матрични операции, статистически изчисления и др. Благодарение на тези разнообразни средства, при нуждата от разработка на код, извършващ математически изчисления, дефинирането на прототипи на съответните функции се облекчава значително. Езиците на системите за компютърна алгебра са от много високо ниво и позволяват лесното и бързо дефиниране на функции, извършващи широк клас от изчисления. Веднъж дефинирани и тествани в системата за компютърна алгебра (което обикновено е сложен итеративен процес), ръчното "превеждане" на тези функции на език за общо предназначение е тежка задача с голяма вероятност за допускане на грешки, поради което системите за компютърна алгебра позволяват това превеждане да стане автоматично [31].

Следният пример е за системата за компютърна алгебра Mapple, която позволява генериране на код на езиците Visual Basic, MATLAB, Java, C, C# и Fortran. Изразът на езика на Mapple:

$$cs := [s = 1.0 + x, t = ln(s)e^{-x}, r = e^{-x} + xt]$$

един математик би изписал лесно на Mapple, но не и на Java. Генерираният от горния израз Java код изглежда по следния начин:

```
s = 0.10e1 + x;
t1 = Math.log(s);
t2 = Math.exp(-x);
t = t1 * t2;
r = t2 + x * t;
```

2.12 Трансформатори на предикати

В статията си "Guarded commands, nondeterminacy and formal derivation of programs" [16] W. Dijkstra въвежда т.нар. семантика на трансформаторите на предикати, която задава денотационна семантиката на императивни програми. Семантиката на програмите се описва чрез набор от твърдения за състоянието на програмата, които се запазват в сила при изпълнение на всеки от операторите на съответния език за програмиране. "Състоянието" на програмата при императивните езици обикновено представлява множеството от стойностите на всички променливи, с които програмата оперира.

Ако гледаме на един оператор S като на правило, задаващо еднократна промяна на състоянието на програмата, то можем да разглеждаме твърдения, верни за състоянието преди "изпълнение" на оператора (предусловия) и след изпълнението му (постусловия). Ако P и Q са съответно верни пред- и постусловия за оператора S, пишем $\{P\}S\{Q\}$. Последното се нарича също "тройка на Хоар" по името на логиката на Флойд-Хоар за доказателство на свойства на компютърни програми [20] и може да се разглежда като твърдение, описващо свойство на оператора S.

При зададено свойство $\{P\}S\{Q\}$, където P и Q са дадени, а S е неизвестен оператор, методът на трансформаторите на предикати дефинира правила, по които се намира такъв оператор S, удовлетворяващ зададеното условие. Тук "оператор" може да означава произволна сложна комбинация от базови конструкции на езика за програмиране – присвоявания, условни оператори, цикли, редици от оператори.

Методът се състои в намирането на т.нар. най-слабо предусловие $wp(S, Q)$. Най-слабото предусловие за даден оператор S и постусловие Q е функция, за която е вярно, че $\forall x, P \to wp(S, Q)$, където x е векторът от променливи на програмата, а P – някое произволно вярно предусловие за S. Тоест, $wp(S, Q)$ е минималното (най-слабо) предусловие, необходимо за запазване на верността на Q при изпълнението на S. Например, за оператора $S = x := x-5$ и постусловието $Q = x > 10$ имаме $wp(x := x - 5, x > 10) = x > 15$. На базата на така намереното най-слабо предусловие за неизвестния оператор S, методът дефинира списък от правила, по които може да се избере операторът, който ще измени състоянието на програмата от такова, за което е верен P, до такова, за което в верен Q. Ако се гледа на P и Q като на спецификация на програма, то с метода на трансформиращите предикати може да се намери програма, удовлетворяваща съответната спецификация, като за нея е гарантирана тотална коректност относно P и Q.

3 Анализ на програми

Като следствие от Стоп проблема на Тюринг може да се покаже, че не е възможно по произволен код на програма и произволна спецификация на изхода й, да се намери ефективен (в смисъл на изчислим) алгоритъм, който да установява дали програмата би проявила грешки по време на изпълнение. Все пак, както при много неразрешими проблеми, съществуват редица полезни методи за намиране на приблизителен отговор на въпроса за "коректност" на програмите спрямо дадени спецификации на изхода им. Такива методи имат обобщеното наименование "методи за анализ на програми".

3.1 Изчерпване на пространството на състоянията

Един прост метод за проверка на коректността на дадена програма е да се построят в явен вид всички нейни състояния [42]. Състояние на програмата може да се дефинира, например, като състояние на паметта и при всички възможни пътища на изпълнение. Ако тези състояния могат да бъдат генерирани в явен вид, то проверката дали удовлетворяват някакво свойство за коректност е тривиална. Това, обаче, е възможно само за много прости програми, тъй като е лесно да се генерира голямо множество от възможни състояния със сравнително кратки програми.

3.2 Код с доказателство

В някои случаи, когато дадена система се налага да изпълни външен за нея код, е особено важно на системата да се гарантира коректността на този код. Например, т.нар. "филтри на пакети" при някои операционни системи са процедури, които проверяват дали даден мрежов пакет е от значение за някоя програма или не. Всяка програма може да поиска от операционната система да добави такъв филтър, който, за да се изпълнява ефективно, се изпълнява като част от ядрото на системата.

Ясно е, че ако този код е зловреден (или просто грешен), той би могъл да доведе до сериозни последици за системата, ако достъпва нейни вътрешни структури от данни. Проверката в реално време дали програмата върши непозволени операции би довела до значително забавяне и поради това не е приложима.

Едно възможно решение на проблема е изпълнимият код на програмата да бъде придружен от доказателство за коректност (по отношение на зададени от системата свойства), което да бъде проверено от автоматично средство (proof assistant) по време на инсталацията ѝ. По този начин, коректността на програмата ще бъде гарантирана и няма да се налага проследяването ѝ по време на изпълнение [35].

3.3 LTL. Автомати на Бюхи

Автоматите на Бюхи са вид ω-автомати, т.е. крайни автомати, разпознаващи безкрайни думи. За един един ω автомат казваме, че разпознава някаква безкрайна дума, ако изпълнението му над думата преминава безкраен брой пъти през някое от финалните състояния на автомата. Автоматите на Бюхи са интересни с това, че се еквивалентни на формулите от Линейната темпорална логика [32].

Линейната темпорална логика (LTL) е вид модална логика с модалност по времето. За всяка пропозоционална променлива се въвежда вярност относно даден целочислен момент от времето. Например, следната формула на LTL – $\psi U \varphi$ твърди, че формулата ψ е вярна поне докато и φ е стане вярна.

С формули от LTL могат да се описват свойства на състоянията на програми с крайно множество от състоянията. Такива програми могат да работят безкрайно, обработвайки някакъв безкраен поток (например индустриален контролер, следящ поток от данни от сензори). Преходите между състоянията на такива програми могат да се опишат с краен автомат "функциониращ" безкрайно над даден безкраен вход (структура на Крипке).

Нека програмата P е моделирана с автомата (структурата на Крипке) A_P. Думите, разпознавани от структурата на Крипке, са всъщност пътищата на изпълнението на програмата (като последователност от нейни състояния). По този начин проверката дали програмата увовлетворява дадено свойство, изразено чрез LTL формулата f се състои в проверка дали $L(A_P) \bigcap L(A_{\neg f}) = \emptyset$, където $L(A_{\neg f})$ е езикът на автомата на Бюхи, построен от отрицанието на f.

3.4 CTL

CTL (Computational Tree Logic) [12] е логика с разклоняващо се време. Моделът на времето при CTL е дървовидна структура, при която бъдещето може да приеме различни посоки. Например, чрез CTL може да се изрази свойството, че ако някакви начални условия са удовлетворени (например променливите на програмата имат положителни стойности), то всички пътища на изпълнението на програмата избягват някакви нежелани ситуации (например – деление на нула). Следната примерна формула на CTL – $EG.AF.P$ означава, че при възникване на определени условия (E – съществува изпълнение), винаги след това (G – globally), ще дойде момент (AF – по всички изпълнения, в някой техен момент), в който е вярна формулата P. Инструменти, които извършват верификация на програми на базата на CTL формули са, например, BANDERA [5], Cadence SMV [6], CWB-NC [13], RED [38] и други.

3.5 Метод на Флойд

Робърт В. Флойд (1936–2001) е изтъкнат учен в компютърните науки. Един от най-важните му приноси в областта на верификацията на програми е методът за доказателство на коректност на програми, описани като блок-схеми, публикуван през 1967 година в статията “Assigning meaning to programs” [18]. Идеята стои в основата на логиката на Флойд-Хоар, намираща широко приложение за доказателството на коректност на програми (вж. следващата секция).

Методът се състои в дефинирането на “разрези” на програми, описани като блок схеми. Това са точки от изпълнението на програмата $t_1, ..., t_k$, с които се асоциират предикатни формули, изразяващи свойства на стойностите на променливите на програмата. По този начин всяка възможна “траектория” t на изпълнението на програмата дефинира редица от разрези $t_1, ..., t_f$, където t_1 е разреза при стартовия оператор на програмата, t_f е разреза в края на изпълнението й, а всеки два разреза в редицата са непосредствени съседи в така образувания граф от разрези.

Нека t_1 и t_2 са два съседни разреза в програмата, а p_{t_1} и p_{t_2} са съответните им асоциирани предикати. Да допуснем, че за всяка такава двойка съседни разрези е показано, че от верността на p_{t_1} за някакви стойности на променливите на програмата следва верността на p_{t_2} след изпълнението на операторите между двата разреза. Тогава за всяка траектория $t = t_1, ..., t_f$, от верността на предиката за входното условие в началото на изпълнението на програмата ще следва верността на предиката за изходното условие, след като програмата приключи.

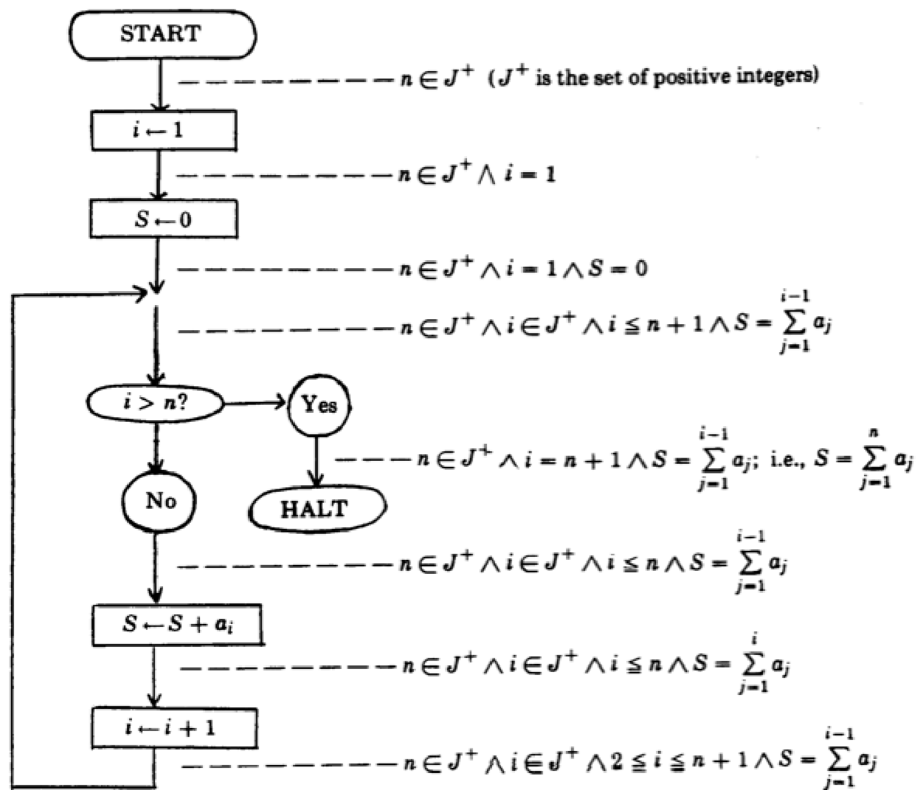

Figure 5: Блок-схема с разрези за програма, изчисляваща $S = \sum_{j=1,..,n} a_j$

Да разгледаме примера на фигура 5 от оригиналната статия на Флойд, илюстриращ как се верифицира изходът на описана чрез блок схема програма, изчисляваща $S = \sum_{j=1,..,n} a_j$ по вход n и a_j, $j = 1, .., n$. Да проследим като пример предикатите преди и след оператора `S ← S + ` a_i. Вижда се как от верността на $S = \sum_{j=1}^{i-1} a_j$ и останалите конюнкти в предиката при първия разрез, при добавяне на члена a_i към променливата `S` следва верността $S = \sum_{j=1}^{i} a_j$.

3.6 Логика на Флойд-Хоар

Логиката на Фойд-Хоар (Floyd-Hoare) е формална система за дедуктивни разсъждения относно коректността на компютърни програми, върху която са базирани редица методи за спецификация, синтез и анализ на компютърни програми [20].

Централното понятие в логиката на Флойд-Хоар е тройката на Хоар $\{P\}S\{Q\}$, където P и Q са формули от предикатна логика, а S е оператор на програмата. P се нарича предусловие и изразява свойство, което е вярно преди изпълнението на S, а Q се нарича постусловие и изразява свойство, което е вярно след изпълнението на S.

Логиката на Флойд-Хоар включва аксиоми и правила за извод за всички базови конструкции на прост императивен език за програмиране (като присвояввания, условни оператори, цикли). Съществуват редица разширения на логиката на Флойд-Хоар за описание на по-сложни конструкции като процедури, конкурентност, преходи (jumps) и указатели.

Правилата за извод в базовата логика на Хоар включват:

- $\frac{}{\{P\}skip\{P\}}$, т.е. празният оператор запазва всяко свойство;
- $\frac{}{\{P[E/x]\}x:=E\{P\}}$, където $P[E/x]$ обозначава субституцията в P на всички свободни срещания на x с E. Пример за валидна тройка от този тип е $\{x+1 = 10\}y := x+1\{y = 10\}$.
- $\frac{\{P\}S\{Q\},\{Q\}T\{R\}}{\{P\}S;T\{R\}}$ – правило за редици от оператори (композиция);
- $\frac{\{B\wedge P\}\ S\ \{Q\}\ ,\ \{\neg B\wedge P\}\ T\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{endif}\ \{Q\}}$ - правило за условен оператор;
- $\frac{\{P\wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{\neg B\wedge P\}}$ – правило за оператора за цикъл while;
- $\frac{P'\rightarrow P\ ,\ \{P\}\ S\ \{Q\}\ ,\ Q\rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$ – правило за следствието, където $\rightarrow$ означава импликация.

С прилагане на правилата за извод в логиката на Хоар (и нейните разширения) е възможно "ръчното" доказателство на свойства на програмен код спрямо спецификация, описана в предикатна логика. Редица системи за доказателство на теореми поддържат автоматични средства за такъв тип верификация, например системата KEY.

3.7 Индукционно правило на Скот

При изследването на денотационната семантика на функционални програми [40], програмите се моделират като оператори над частични функции, като най-малката неподвижна точна на оператора задава семантиката на програмата. Например, ако искаме да докажем, че следната функция (на псевдо код):

```
multiply (x,y) = if x == 0 then 0 else y + multiply (x-1,y)
```

е частично коректна относно спецификацията $P(f) = \forall x, y : f(x, y) \simeq x * y$, постояваме следния оператор $\Gamma : (\mathrm{N} \times \mathrm{N} \to \mathrm{N}) \to (\mathrm{N} \times \mathrm{N} \to \mathrm{N})$

$$\Gamma(f)(x,y) \simeq \begin{cases} \text{0, ако x=0} \\ \text{y+f(x-1,y), иначе.} \end{cases}$$

Тук условното равенство $\simeq$ означава равенство в случая, в който функцията е дефинирана (т.е. програмата приключва).

За доказване на верността на свойството P за най-малката неподвижна точка на оператора Γ е достатъчно да покажем верността му за празната функция $\emptyset$ ($\forall x : \neg!\emptyset(x)$) и че ако $P(f)$ за някоя функция f, то от това следва $P(\Gamma(f))$.

За нашият пример $P(\emptyset)$ е вярно, тъй като $\emptyset$ не е дефинирана за никой аргумент. Нека допуснем, че $P(f)$. Тъй като $\Gamma(f)(0, y) = 0$ по дефиниция, то остава да видим, че $\forall x > 0, y : \Gamma(f)(x, y) \simeq x * y$. По дефиниция $\Gamma(f)(x, y) = y + f(x-1, y)$ при $x > 0$. Но от $P(f)$ можем да заключим, че $f(x-1, y) = (x-1)*y$. Следователно $\Gamma(f)(x, y) = y+f(x-1, y) = y + (x-1) * y = x * y$.

3.8 Статично типизиране

Типовете в езиците за програмиране дефинират множеството от допустимите стойности за променливи, резултати от функции и оценки на изрази. В езиците за програмиране, поддържащи типове, сигнатурите на функциите и дефинициите на променливите, например, могат да се разглеждат като "изисквания" или спецификации относно това с какви стойности е коректна тяхната употреба и с какви не е. По този начин се добавя един "слой" от превантивни мерки за коректност в програмата.

Автоматичната проверката дали ограниченията, наложени от типовете в програмата, са спазени, може да се извърши по време на компилация или по време на изпълнение на програмата. В първия случай говорим за статична проверка, а във втория – за динамична. Компилаторите на някои езици за програмиране комбинират и двата подхода (например Perl).

По този начин, статичното типизиране е ограничен вид проверка за коректност (т.нар. type safety), тъй като позволява широк спектър от грешки да бъдат идентифицирани рано в процеса на разработка. Чрез разглеждане на "потока" на данни в една програма, една типова система може да идентифицира редица логически грешки, свързани с неправилни присвоявания, обръщения към функции и др.

Тъй като статичното типизиране разполага само с информацията за програмата, налична по време на компилация, но не и по време на изпълнение, системите за статично типизиране проявяват известна консервативност. Например, следният израз "if <test> then <do something> else <type error>" няма да се компилира при статична проверка, ако <type error> е израз, съдържащ типова грешка. Ако обаче <test> е винаги истина, то една динамична типова система би допуснала изпълнението на програмата и то би било коректно. Ако <test> е израз, който с малка вероятност е лъжа, при динамично типизиран език такава грешка може да остане незабелязана дълго време, дори и при много добри динамични тестове на програмата [45].

3.9 Lint

Софтуерите от фамилията "Lint" получават началото си през 70-те години на 20-ти век с първия в серията статичен анализатор на код на езика C [28]. Чрез серия от правила, базирани на текста на програмата, Lint подчертава някои съмнителни или непреносими (между различни платформи) сегменти от програмите, насочвайки програмистите да им обърнат специално внимание.

Програмите от този тип разчитат на прости правила за откриване на основни грешки, като неинициализирани променливи, константни условия на условни оператори и цикли, изрази, които чиито стойности биха могли да излязат от допустимия интервал. Практически всички модерни компилатори поддържат функции от този вид, а съвременни Lint софтуери правят допълнителни проверки, като например преносимост на кода между различни компилатори (и интерпретатори).

Подобни системи за анализ на код са Astree [4], Polyspace [34], Coverity [11], Klockwork [7] и много други.

3.10 Системата KeY. JML

Системата KeY [25] е платформа за формална верификация на Java програми, която работи със спецификации на езиците JML и OCL. Спецификациите се превеждат до теореми от динамичната логика [17], като и самите програми също се дефинират в термините на динамичната логика [24].

Следният пример показва цикъл на езика Java, анотиран с конструкции на езика JML. JML е език за описание на спецификации за програми на езика Java. Изразите на JML следват стила на логиката на Флой-Хоар.

```
/*@ loop_invariant 0<=i && i <= logArray.length
@ && max!=null &&
@ (\forall int j; 0 <= j && j<i;
@ max.balance >= logArray[j].balance);
@ assignable max, i;
@*/
```

```
while(i<logArray.length){
   LogRecord lr = logArray[i++];
   if (lr.getBalance() > max.getBalance()){
   max = lr;
}
```

Системата Key е способна по такива анотации да потвърждава верността на инварианти и други видове анотации на реален Java код. На JML се базират и други инструменти, например – системата Why [49], която превежда JML спецификацията и Java програмата на езиците на серия от автоматични системи за доказателство на теореми и прави сравнителен анализ на резултатите от всяка от тях.

3.11 Системата ACL2

ACL2 (A Computational Logic for Applicative Common Lisp) [1] е автоматична система за доказателство на теореми в логика от първи ред с поддръжка на диалект на езика Common LISP. Аксиоматиката на ACL2 описва семантиката на езика за програмиране и вградените в него функции. Потребителски дефинираните функции разширяват теорията на системата.

Следният пример демонстрира как на ACL2 се верифицира функция за сортиране на списък чрез последователното вмъкване на елементите му в нов, сортиран списък:

```
(defun insert (e x)
        (cond ((endp x) (cons e x))
              ((< e (car x)) (cons e x))
              (t (cons (car x) (insert e (cdr x))))))
(defun isort (x)
        (if (endp x)
            nil
          (insert (car x)
                  (isort (cdr x)))))
(defun ordered (x)
        (cond ((endp x) t)
              ((endp (cdr x)) t)
              (t (and (<= (car x) (car (cdr x)))
                      (ordered (cdr x))))))
```

При така направените дефиниции може да се провери следната теорема:

```
(defthm ordered-isort
         (ordered (isort x)))
```

3.12 Системата PVS

PVS (Prototype Verification System) [36] е език за дефиниране на спецификации, който е интегриран в система за автоматично доказателство на теореми. Ядрото на PVS е базирано на разширение на теорията за зависимите типове [15].

На следния пример е разработена примерна програма [37], реализираща телефонен указател.

```
BEGIN
   N: TYPE % names
   P: TYPE % phone numbers
   B: TYPE = [N -> P] % phone books

   n0: P
   emptybook: B
   emptyax: AXIOM FORALL (nm: N): emptybook(nm) = n0

   FindPhone: [B, N -> P]
   Findax: AXIOM FORALL (bk: B), (nm: N): FindPhone(bk, nm) = bk(nm)

   AddPhone: [B, N, P -> B]
   Addax: AXIOM FORALL (bk: B), (nm: N), (pn: P):

   AddPhone(bk, nm, pn) = bk WITH [(nm) := pn]

END phone_1
```

А чрез следната конструкция се описва и проверява свойство за коректност на FindPhone и AddPhone:

```
   FindAdd: CONJECTURE FORALL (bk: B), (nm: N), (pn: P):
      FindPhone(AddPhone(bk, nm, pn), nm) = pn
```

3.13 Coq и HOL

Както беше споменато в частта за извличане на програми, популярните системи за доказателство на теореми Coq и HOL включват езици за програмиране, върху програми на които може да се извършва дедуктивен анализ с цел доказване на техни свойства.

3.14 Динамичен анализ

Динамичният анализ на компютърни програми обединява методи, разчитащи на реално изпълнение на програмите за проверка на тяхната коректност. За ефективността на динамичното тестване е важно параметрите на изпълнението на програмата да бъдат подбрани по такъв начин, че чрез серия от изпълнения да се "премине" през голяма част от пътищата на изпълнение на програмата. Мярката за степента на покритие на пътищата на изпълнение на програмата се нарича Code coverage [8].

Следващите секции съдържат описание на основните методи за динамичен анализ на програми.

3.15 Автоматизирано тестване

Въпреки множеството формални и автоматични методи, тестването на софтуерните системи чрез контролирано "ръчно" експериментално изпълнение на програмата за специално подбрани потребителски сценарии продължава да бъде основен инструмент за откриване на грешки. Тези неавтоматични подходи за тестване водят до откриването на над 50% от грешките при индустриалното разработване на софтуер [30].

Особено често ръчно тестване се налага за програми, предоставящи богат графичен потребителски интерфейс. За частите от програмата, тясно свързани с потребителския интерфейс, е трудно да се напишат автоматични тестове, тъй като интерфейсът обикновено е предвиден да реагира на събития от входните контролери, което трудно се формализира.

За автоматизирането на такъв тип тестове съществуват програми, които позволяват симулиране на потребителски вход и последващо изпълнение на верифициращ код, който да потвърди коректността на някакво свойство на състоянието на програмата. Този верифициращ код е външен за програмата, но получава достъп до нейната вътрешна памет и състояние в определени моменти от изпълнението ѝ. Такъв софтуер е например QuickTest Pro на Hewlett-Packard.

3.16 Верификация на изпълнението

Верификацията по време на изпълнението (Runtime Verification) [39] е метод, който проверява верността на свойства на програмата на базата на нейно реално изпълнение. Подходите за верификация на изпълнението обикновено разчитат на свойства, изразени като предикати, крайни автомати, LTL формули и др. Подходът се състои в генерирането на "следящ" код, който се "инжектира" в проверяваната програма и следи за верността на зададените свойства.

Проверката на изпълнението заобикаля сложността на формалните методи, тъй като има значително по-простата задача да провери някакво конкретно свойство на състоянието на програмата в конкретен момент от нейното изпълнение. Цената за това улеснение е непълното покритие на всички възможни пътища за изпълнение на програмата.

Т.нар. assertions са прост пример, при който програмистите още при написването на кода на програмата влагат условия, чието нарушаване би довело до нейното прекратяване. Често влагани са условия за параметрите на функции, които се подразбират от дефиницията на функцията, но не се гарантират от избрания тип (например null указатели). Тази техника е полезна не само за локализиране на грешки по време на изпълнение на програмите, но и за тяхното документиране, тъй като изразените в явен вид изисквания за параметрите на функциите често говорят много за тяхната семантика.

3.17 Изолирани тестове

Изолираните тестове (или Unit Tests) са метод, при който отделни единици от кода на програмата се тестват независимо чрез серия от специално подбрани тестови параметри [48]. Тези "единици" се дефинират като най-малките "тестваеми" части от програмата – например функции и методи, макар че в някои случаи се тестват цели класове и модули.

В идеалния случай, всеки такъв тест е изолиран от останалите. За изолация на тествания код от останала част на програмата се налага дублиране на свързаните към него други модули с т.нар. "имитиращи" (mock) методи и класове. Имитиращите методи и класове имат същия прототип като техните оригинали, но връщат константи. Изолираните тестове са важна част от модерния процес на софтуерна разработка, тъй като служат за ранна идентификация на нарушени свойства на кода в части от програмата, в които често се внасят промени.

Следва пример за тестване на функцията abs за намиране на абсолютната стойност на число с плаваща запетая.

```
function testAbs(){
    assert (abs(-5) == 5);
    assert (abs(5) == 5);

    assert (abs(-0.00000005) == 0.00000005);
    assert (abs(0.00000005) == 0.00000005);

    for (var i = 0; i < 1000; i++){
        var x = random(-10000,10000);
        assert (abs(-x)/x == -1);
    }
}
```

Както се вижда от примера, стандартно при тези тестове се използват параметри, осигуряващи покритие на възможно най-много пътища на изпълнение на функциите, гранични стойности и множество от произволни стойности.

4 Заключение

Методите за извличане (или "синтез") на програми са разнообразни, като някои от тях имат фундаментални различния по между си. Въпреки това, всички те споделят две ключови характеристики: вход – спецификация под някаква форма и изход – синтезиран код.

При формалните методи, залагащи на средствата на математическата логика, спецификацията най-често е някаква логическа формула, описваща "поведението" на желаната програма. Тези методи гарантират, че синтезираната програма удовлетворява спецификацията си и тяхната мотивация е именно в това – получаването на програми, чието поведение може да бъде предвидено с абсолютна сигурност.

От друга страна, мотивацията на редица методи с широка практическа употреба е да предоставят език от по-високо ниво, който е близък до предметната област на решавания проблем. Това позволява с по-малко усилия, по-голяма точност и по-добра "четимост" да се формализират обекти и процеси, свързани с някакъв специфичен практически проблем. При болшинството методи от този вид, от спецификацията на проблема често се генерира непълен, скелетен код, който трябва да се допълни в последствие със специфични реализационни детайли. Следователно ползата от тези методи е не толкова гарантираното поведение на програмите, а в това, че решението на практически проблеми се описва с по-естествен за предметната област начин. Това от своя страна намалява вероятността за грешки в модела, подобрява неговата изразителност и яснота и не на последно място спестява значително време от реализацията на повторяеми сегменти програмен код.

Макар формалните методи за синтез на програми да гарантират тяхното поведение, обхвата (концептуалния размер) на редица практически задачи представлява пречка за приложението им. Формалните методи обикновено се прилагат за по-малки, критични сегменти код, например, в животоспасяваща апаратура, за сертифициране на индустриални контролери и др., но практически не се прилагат в широката индустрия на потребителския софтуер. Поради това се налага употребата на методи за последващ анализ на програмите относно тяхното поведение.

И в тази посока на задачата математическата логика ни дава разнообразни мощни инструменти за гарантиране на коректността на програмен код спрямо формули, специфициращи различни аспекти на поведението на програмата. И в този случай те са приложими за по-кратки "откъси" от код, в случая, в който се търси пълна гаранция за коректността на кода. И в този случай, ограничавайки задачата до коректността относно отделни аспекти на поведението, получаваме редица методи с широко практическо приложение.

Анализът на програми може да бъде базиран на техния код (статичен анализ), например, за проверка на типовото съответствие между променливи, функции и изрази. Редица практически методи, от друга страна, не разглеждат кода на програмата, а с разнообразни средства "следят" нейното поведение по време на изпълнението ѝ.

References

[1] Kaufmann, T., Moore, J. S.,*"ACL2 Demo"*, `http://www.cs.utexas.edu/users/moore/publications/flying-demo/script.html`

[2] Nielson, F., Nielson, H. R., Hankin, C.,*"Principles of Program Analysis"*, Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1999.

[3] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J.,*"Aspect-Oriented Programming"*, Proceedings ECOOP'97 — Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 1997, Mehmet Aksit and Satoshi Matsuoka (Eds.), LNCS 1241, Springer-Verlag, 1997.

[4] absint.com,*"Astrée Run-Time Error Analyzer"* `http://www.absint.com/astree/index.htm`

[5] cis.ksu.edu,*"About Bandera"*, `http://bandera.projects.cis.ksu.edu/`

[6] kenmcmil.com, *"The Cadence SMV Model Checker"*, `http://www.kenmcmil.com/smv.html`

[7] *"Clockwork Website"*, `http://www.klocwork.com/`

[8] Miller, J. C., Maloney, C. J.,*"Systematic mistake analysis of digital computer programs"*, Communications of the ACM (New York, NY, USA: ACM) 6 (2) February 1963: 58–63.

[9] Berghofer, S., Strecker, M.,*"Extracting a formally verified, fully executable compiler from a proof assistant"*, Electronic Notes in Theoretical Computer Science – ENTCS, Vol. 82, 2004, No. 2, 377–394.

[10] Letouzey, P.,*"Coq Extraction, an Overview"*, Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008, LNCS, vol. 5028.Springer (2008)

[11] coverity.com,*"Coverity SAVE"*, `http://www.coverity.com/products/coverity-save.html`

[12] Huth, M., Ryan, M.,*"Logic in Computer Science (Second Edition)"*, Logic in Computer Science (Second Edition). Cambridge University Press. p. 207.

[13] cs.sunysb.edu,*"Concurrency Workbench of the New Century"*, `http://www.cs.sunysb.edu/~cwb/`

[14] Lenzerini, M.,*"Data Integration: A Theoretical Perspective"*, PODS 2002: 233–246.

[15] Nordstrom, B., Petersson, K., Smith, J. M.,*"Programming in Martin-Lof's Type Theory: An Introduction"*, Programming in Martin-Lof's Type Theory: An Introduction. Oxford University Press. 1990.

[16] Dijkstra, E. W.,*"Guarded commands, nondeterminacy and formal derivation of programs"*, Communications of the ACM, CACM, Volume 18, Issue 8, Aug. 1975, 453–457.

[17] Harel, D., Kozen, D., Tiuryn, J.,*"Dynamic Logic"*, MIT Press, 2000.

[18] Floyd, R. W.,*"Assigning Meaning to Programs"*, Proceedings of Symposium on Applied Mathematics, Vol. 19, J.T. Schwartz (Ed.), A.M.S., 1967, 19–32.

[19] Wilcox, J.,*"Paying Too Much for Custom Application Development"*, `http://edgewatertech.wordpress.com/`

[20] Hoare, C. A. R.,*"An axiomatic basis for computer programming"*, Communications of the ACM, 12(10): 576–580,583 October 1969.

[21] Aagaard, M., Leeser, M. E., Windley, P. J.,*"Toward a super duper hardware tactic"*, Higher Order Logic Theorem Proving and its Applications: 6th International Workshop, HUG'93, Vancouver, B.C., August 11-13 1993: Proceedings, volume 780 of Lecture Notes in Computer Science, 399–412. Springer-Verlag, 1994.

[22] Haftmann, F., Bulwahn, l.,*"Code generation from Isabelle/HOL theories (tutorial)"*, `http://isabelle.in.tum.de/doc/codegen.pdf`

[23] IBM Corporation,*"C++ Tutorial for Rational Rhapsody"*, `http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.rhapsody.doc/pdf75/tutorialcpp.pdf`

[24] Beckert, B., Hähnle, R., Schmitt, P. H.,*"Verification of Object-Oriented Software: The KeY Approach"*, Springer, 2007.

[25] Karlsruhe Institute of Technology, Technische Universität Darmstadt, Chalmers University of Technology,*"The KeY Project. Integrated Deductive Software Design"*, `http://www.key-project.org/`

[26] Lesk, M. E., Schmidt E.,*"Lex – A Lexical Analyzer Generator"*, Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey, 1975.

[27] Biggerstaff, T. J.,*"The Library Scaling Problem, and the Limits of Concrete Component Reuse"*, Technical Report, MSR-TR-94-19, Microsoft Research, Advanced Technology Division, Microsoft Corporation, November 1994.

[28] Johnson, S.,*"Lint, a C program checker"*, Computer Science Technical Report 65, Bell Laboratories, December 1977.

[29] Тодорова, М.,*"Синтезиране на програми"*, Софтех, София, 1998.

[30] Wong, Y. K.,*"Modern Software Review: Techniques and Technologies"*, IRM Press, 2006.

[31] Gomez, C., Scott, T. C. ,*"Maple Programs for Generating Efficient FORTRAN Code for Serial and Vectorized Machines"*, Comput. Phys. Commun. 115, 1998, 548–562.

[32] Clarke, E.M., Grumberg, O., Peled, D. A.,*“Model Checking”*, The MIT Press, 1999.

[33] Parnas, D. L.,*“Software Aspects of Strategic Defense Systems”*, American Scientist, November 1985.

[34] mathworks.com,*“Static Analysis with Polyspace Products”*, `http://www.mathworks.com/products/polyspace/`

[35] Necula, G. C., Lee, P. ,*“Safe kernel extensions without run-time checking.”*, SIGOPS Operating Systems Review 30, SI (Oct. 1996), 229-–243.

[36] Owre, Shankar, and Rushby,,*“PVS: A Prototype Verification System”*, CADE 11 conference proceedings, 1992.

[37] Crow, J., Owre, S., Rushby, J, Shankar, N. J., Srivas, M.,*“A Tutorial Introduction to PVS”*, WIFT ’95: Workshop on Industrial-Strength Formal ,Specification Techniques, Boca Raton, Florida, April 1995.

[38] REDLIB Project,*“Red & Redlib: Automated Verification Support for Timed Automatas”*, `https://sites.google.com/site/redlibtw/`

[39] Havelund, K.,*“Using Runtime Analysis to Guide Model Checking of Java Programs”*, 7th International SPIN Workshop, August 2000.

[40] Сосков, И., Дичев, А.,*“Теория на програмите”*, Университетско издателство “Св. Кл. Охридски”, София, 1996.

[41] Sparx Systems 2007,*“MDA Overview”*, `http://www.omg.org/mda/mda-files/MDA-Tool-Sparx-Systmes.pdf`

[42] Mäkelä, M.,*“Proceedings of the conference on Application and theory of petri nets: formal methods in software engineering and defence systems, Volume 12”*, ACM International Conference Proceeding Series, Vol. 145, Laboratory for Theoretical Computer Science, Helsinki University of Technology, Espoo, Finland.

[43] Beck, K. ,*“Test-Driven Development by Example”*, Addison Wesley–Vaseem, 2003.

[44] Trifonov, T.,*“Quasi-linear Dialectica Extraction”*, CiE, 2010, 417–426

[45] Benjamin, C. P.,*“Types and Programming Languages”*, MIT Press. 2002.

[46] Booch, G., Jacobson, I., Rumbaugh, J.,*“OMG Unified Modeling Language Specification, Version 1.3 First Edition”*, `http://formal.iti.kit.edu/~beckert/teaching/Praktikum-Formale-Entwicklung-WS0405/00-03-01.pdf`

[47] Hunt, J.,*“The Unified Process for Practitioners: Object-oriented Design, UML and Java”*, Springer, 2000.

[48] Kolawa, A., Huizinga, D.,*“Automated Defect Prevention: Best Practices in Software Management”*, Wiley-IEEE Computer Society Press. 2007. p. 426.

[49] Filliâtre, J-C, Marché, C.,*"The Why/Krakatoa/Caduceus Platform for Deductive Program Verification"*, CAV, 2007, 173-177

[50] W3C,*"Web Services Description Language (WSDL) 1.1"*, `http://www.w3.org/TR/wsdl`

[51] Microsoft Corp.,*"Xaml Object Mapping Specification"*, Microsoft Corporation. Release: June 2008.

[52] Jarzabek, S.,*"Effective Software Maintenance and Evolution: Reuse-based Approach"*, CRC Press Taylor Francis, May 2007.

[53] Johnson, S. C.,*"Yacc: Yet Another Compiler Compiler"*, Computing Science Technical Report No. 32, Bell Laboratories, Murray hill, New Jersey, 1975.