1. **Състояние на проблема и постановка на задачата**
   1. **Приложимост на обектно ориентираните подходи и езици за реализиране на реално-времеви управляващи системи**

В областта на обектно ориентираното програмиране има голямо количество разработки на различни кодо-генератори. Обектно ориентираните езици са изключително удобни за създаване на сложни и гъвкави софтуерни продукти. Възможностите за наследяване на класове, изграждането и използването на статични и динамични библиотеки дават възможност за преизползване на кода и създаване на мащабен, ефективен и лесен за модификация софтуер. Основен техен недостатък е невъзможността за реализиране на реално-времеви управляващи системи, поради факта, че програмите писани с тези езици не подлежат на валидация по отношение на времето за тяхното изпълнение [LO11]. Според критериите на Кардели [CARD96] процедурните езици превъзхождат обектно ориентираните езици, в областта на реално-времевите системи. От обектните езици, единствено езика C++ е пригодим за използване в реално-времеви системи, поради това, че в него не се използва система за събиране на боклука (garbage collector). В литературата са описани различни подходи за обектно-ориентирано програмиране за реално-времеви системи : RTC++++ [ITM92], Octopus [AKZ96], ROOM [SGW94], HRT-HOOD [BW95]. От изброените източници единствено разработката HRT-HOOD представя метод за гарантиране на детерминистично времево поведени в системи за твърдо реално време, базиран на теорията за изпреварващо диспечиране на приоритетите [ABRT93]. Резултатите са постигнати като се елиминира концепцията за наследяването, въвежда се малък брой класове стереотипи (активни, пасивни защитени, циклични и спорадични) и се ограничават синхронизационните механизми за операциите върху класовете. Агрегацията и връзките между класовете също са ограничени, за да може да се гарантира, че времевите свойства на финалната разработка подлежат на анализ. Някои от тези ограничения биха могли да бъдат смекчени, без да се загуби времевия детерминизъм [MAP97], но като цяло в момента изглежда невъзможно да се запази цялата гъвкавост на общите методи за обектно-ориентиран дизайн в системите за реално време (СРВ).

* 1. **Подход на програмната генерация за създаване на фамилии софтуерни системи.**

Концепцията за автоматична програмна генерация е известна отдавна, но до скоро този метод на разработка на софтуер не беше използван широко. Развитието на хардуерните платформи, промените в софтуерните технологии и бързо нарастващите нужди на пазара на софтуер за управление са на път да обърнат тази тенденция. Интегрирането на формални методи и обектно-ориентирани графични представяния имат значителни предимства пред конвенционалните подходи за програмиране [JAP2000] :

* По-добра спецификация на конкурентност, синхронизация и времеви свойства.
* Възможност за валидация на разработените модели посредством графично анимирано представяне.
* Анализ на диспечируемостта на системата, при условие че се спазва набора от ограничения.
* Възможност да се разработят инструменти, които използват формалните методи като ядро, а скриват детайлите от разработчика.

Програмната генерация е свързана с подходи, техники и инструменти за генериране на код на програма, който в последствие бива компилиран или интерпретиран. Подхода на програмната генерация е особено подходящ при наличие на следните сценарии [MV03]:

* Целевата система трябва да има висока степен на адаптируемост към нуждите на клиента, а стандартните подходи на обектно-ориентираното програмиране не могат да бъдат използвани поради неприложимост в СРВ.
* Минимизиране на размера на изпълнимия код, поради ограничение в програмната памет (този проблем се появява основно в областта на вградените системи).
* Необходимост от статичен анализ на кода по отношение на консумация на ресурси, конкурентност и т.н. Генерираният код обикновено е много по-лесен за анализ в сравнение с код писан върху платформите с общо предназначение.
* Избягване на динамичното заделяне на памет по време на работа на системата, без да се губи гъвкавост по отношение на използваните обекти.
* Разработване на програмна логика с по-високо ниво на абстракция от нивото на стандартните програмни езици. Това може даде възможност на експертите в конкретната област да „програмират“, като след това програмната генерация да намали нивото на абстракция, като генерира имплементация на системата.
* Използване на подходи, които програмния език не позволява (Например използването на класове и обекти в процедурен език.).
* Наличие на големи разпределени системи, като върху всеки модул на системата трябва да се зареди само част от програмата и необходимост да се конфигурират методите за комуникация между отделните модули.
* Възможност за ранно откриване на грешките, не по време на тест на системата, а по време на генериране на програмния код/конфигурация и/или последваща компилация.

Програмната генерация е особено важна в контекста на разработката на продуктови линии, тъй като разработката на генератор за еднократно приложение обикновено не се изплаща. Разработката на продуктови линии е насочена към създаване на фамилии от софтуерни системи насочени в определена област. В [DLP76] определението за фамилия софтуерни системи гласи: „Набор от приложения създават фамилия, когато може да се счита за необходимо да се изучат първо общите свойства на целия набор от приложения, а след това да се разглеждат свойствата на индивидуалните приложения от фамилията.“. Това определение е твърде общо, и може да бъде по-добре обяснено с няколко конкретни примера:

* Набор от проекти в една и съща област, ползващи обща или повтаряща се в много отношения бизнес логика (банкови системи, превключване между телекомуникационни системи, автомобилна диагностика).
* Реализиране на една и съща бизнес логика върху различни хардуерни платформи (например вградена система, мобилно устройство, PC и PLC). В този случай трябва да се генерира платформено зависима реализация на кода от общия модел.
* Набор от продукти реализирани на база една и съща парадигма за моделиране. В този случай цялата реализация е заложена в структурата на модела и неговото представяне като реализация на по-ниско ниво.

Понастоящем разработката на продуктови линии може да се разглежда в контекста на програмната генерация. В [CE2000] е дадено следното определение: „Генеративното програмиране е парадигма от софтуерното инженерство базирана на моделиране на фамилии софтуерни системи, отговарящи на специфични изисквания, така че персонализиран и оптимизиран на краен продукт да може да бъде автоматично генериран Крайния продукт трябва да бъде съставен от елементарни, преизползваеми компоненти, на базата на познание за софтуерната му конфигурация“.

* 1. **Основни цели на подхода на програмната генерация**

Друго определение за генеративно програмиране е дадено в [EIS97]: “Генеративното програмиране е подход за имплементиране на софтуерни модули, които могат да бъдат комбинирани, за да генерират специализирани и оптимизирани системи, отговарящи на специфични изисквания“. Основните цели на генеративното програмиране са:

* Да се намали концептуалната пропаст между програмния код и концепциите на конкретната област. Това се нарича постигане на висока целенасоченост на програмата (high intentionality).
* Да се постигне голяма преизползваемост и адаптируемост.
* Да се опрости поддържането на множество персонализирани варианти на дадена система/продукт.
* Да се увеличи ефективността (по отношение на време за разработка на системата, време за изпълнение и размер на програма).

За да се постигнат тези цели е необходимо да се следват следните принципи [CEGVV98]:

* Разделяне на задачите (Separation of concerns): Този термин въведен от Дийкстра [DIJ76], отразява важността на решаването на само един проблем в един момент от времето. За да се избегне програмен код, който едновременно решава много проблеми, генеративното програмиране цели да раздели всеки проблем в отделна част от кода. Тези части от кода в последствие се комбинират за да генерират необходимия компонент.
* Параметризация на различията: Параметризацията дава възможност да се представят компактно фамилии от компоненти (компоненти с много общи неща между тях). НИКЪДЕ ДО МОМЕНТА НЕ Е ВЪВЕДЕН ТЕРМИНЪТ «КОМПОНЕНТ»
* Анализ и моделиране на зависимостите и взаимодействията: Не всички комбинации от стойности на задаваните параметри са валидни, а стойностите на някои параметри могат да съдържат или да влияят върху други параметри. Тези зависимости се наричат познание за хоризонталната конфигурация, тъй като се проявяват между параметри от едно и също ниво на абстракция.
* Разделяне на пространството на проблема от пространството на решението: Пространството на проблема се състои от специфични за конкретната област абстракции с които разработчиците на приложението биха желали да взаимодействат, докато пространството на решението съдържа реализационни компоненти. Двете пространства имат различна структура и съответствието между тях се нарича познание за вертикалната конфигурация. Под вертикална се разбира взаимодействие между параметри от различни слоеве на абстракция. И вертикалното и хоризонталното познание се използват при автоматичното конфигуриране.
* Премахване на излишния товар и оптимизация спрямо конкретната проблемна област: При генериране на компонентите статично (по време на компилация), голяма част от излишния изчислителен товар, като проверки за коректност по време на работа, разход на програмна памет поради включване на неизползван код и излишни нива на косвеност, могат да бъдат премахнати. Могат да бъдат извършени и по-сложни специфични оптимизации (например трансформация на циклите при научни изчисления и т.н.).
  1. **Връзки на генеративното програмиране с други парадигми**

С генеративното програмиране са свързани още три важни парадигми, които имат подобни цели:

* Генерично програмиране
* Езици специфични за областта (Domain Specific Languages – DSLs)
* Аспектно ориентирано програмиране

Генеративното програмиране е по-широко като обхват, но използва важни идеи от всяка от изброените области.

Генеричното програмиране може да бъде сумаризирано като „преизползване чрез параметризация“. То позволява създаването на компоненти, които се поддават на персонализиране и промяна, а в същото време запазват ефективността на статично конфигурирания код. Тази техника може да елиминира зависимостите между типове данни и алгоритми, които не са концептуално необходими. Например итераторите позволяват да се използват генерични алгоритми които работят ефективно едновременно за сгъстени и разредени матрици [SL98]. В същото време генеричното програмиране ограничава генерацията на кода до заменяне на конкретни типове с обобщени типове параметри и свързва предварително съществуващи парчета код във фиксиран модел. Генеративното програмиране е много по широко обхватно, тъй като предоставя възможност за автоматична конфигурация на генерични компоненти от абстрактни компоненти и много по-мощна параметризация.

Езиците специфични за областта (DSLs) предоставят специализирани езикови възможности, които увеличават нивото на абстракция. Те позволяват на потребителите да работят с концепциите на проблемната област, но на цената на езиково обобщаване. Този тип езици намират приложение както в системи за числови и символни изчисления (например Mathematica), така и в специализирани езици за телекомуникационни превключватели и финансови калкулации. Тези езици дават възможност да се правят специфични за конкретната област оптимизации и проверки за грешки. От друга страна те рядко имат поддръжка за концепциите от генеричното програмиране.

Аспектно ориентираното програмиране (АОП) цели да декомпозира проблемите във функционални единици и аспекти (като например проверка за грешки или синхронизация) [KLM97]. В една АОП система компонентите и аспектите са „преплетени“ заедно. „Преплитането“ (weaving) между компонентите и аспектите може да стане по време на компилация, посредством използването на компилатор или препроцесор, или по време на работа с използване на динамично отражение (dynamic reflection). При всички случай реализацията на „преплитането“ изисква някаква форма на мета-програмиране. Тук под мета-програмиране се разбира писане на програми, които могат да манипулират други програми, като ги ползват за набор от данни. Такова поведение имат например компилатори, препроцесори, генератори по шаблон. Друг пример за мета-програмиране са програми, които реализират абстракциите на програмния език по рефлективен начин, например мета-класовете в езика SmallTalk.

Концепцията на генеративното програмиране обхваща техниките от изброените по-горе три подхода, както и някои допълнителни техники за да бъдат постигнати основните цели описани в секция 1.3, като :

* DSL техниките се ползват, за да се подобри целенасочеността на програмния код, да се реализират специфични за областта оптимизации и проверки за грешки. Мета-програмирането дава възможност да се реализират необходимите разширения на езика. Тук под разширение на езика се има предвид възможности да се разширява изразната мощност на езика, които не са традиционно включени в конвенционалните библиотеки.
* АОП техниките се използват, за да се постигне разделяне на задачите като се изолират аспектите от функционалните компоненти. Мета-програмирането позволява в последствие компонентите и аспектите да бъдат „преплетени“.
* Техниките на генеричното програмиране се използват, за да има възможност за параметризиране независимо от конкретните типове данни.
  1. **Основни понятия в програмната генерация и генерацията на код**

Основните понятия в програмната генерация и връзките между тях са илюстрирани на фигура 1.5.1.



Фигура 1.5.1 Основни понятия в програмната генерация и връзки между тях

* **Дефиниция на код и модел.** В контекста на програмната генерация е важно да се дефинира за какво се отнасят понятията **код** и **модел**. Моделът представлява принципно описание на обект, което трябва да бъде трансформирано до нещо конкретно, за да бъде изпълнено. Кодът от друга страна е описание, което може да бъде компилирано и изпълнено директно. Това определение е важно защото в контекста на многостъпковото генериране на програми има много трансформации от модел към модел и една финална генерация на изпълним код или конфигурация. Така генерираният код след това може да бъде компилиран или интерпретиран и изпълняван. Трябва да се отбележи, че графичното представяне на модела не е задължително. Той може да се представи и в текстов вид описан с DSL [DS00]. В този случай правилата и синтаксиса на езика за описание на модела представляват негов мета-модел.
* **Моделна трансформация.** Моделната трансформация генерира нови модели от вече съществуващи [HC06]. Обикновено ново-генерираните модели са специализирани (по-конкретни) по отношение на някое свойство.
* **Генерация на изходен код.** Генерацията на изходен код описва процеса на генериране на изпълним код. Това обикновено включва някакъв вид де-абстракция или конкретизация на модела. Генерираният код обикновено изисква компилация преди да може да бъде изпълнен. В някой случай обаче, генератора директно генерира байтов или машинен код готов за изпълнение.
* **Трансформация на изходен код.** Трансформацията на изходен код обозначава създаването на код, на базата на вече съществуващ такъв. Тази техника обикновено се използва когато различни части от кода биват обединявани или променяни по някакъв начин.
* **Пренаписване на байтов код.** Тази техника е се ползва от интерпретаторите на байтов код или от виртуалните машини и добива голяма популярност в контекста на платформи като Java или .Net. Тук кодът, който е бил създаден при компилиране на изходния код, се създава или модифицира.
* **Отражение (reflection).** С тази техника са свързани още техниките „посредничество“ (intercession), деперсонализация (reification) и самонаблюдение (introspection). Отражението описва как една програма може да модифицира сама себе си по време на изпълнение.
  1. **Преглед на основните подходи в генерацията на код**

В литературата са описани седем основни подхода за генерация на код. В последващите секции ще разгледаме в детайли всеки един от тях.

* “Templates + filtering” (шаблони + филтриране). Това е най-простият подход за генериране на код. Кода се генерира като се прилагат шаблони, върху спецификация на текстови модели (най-често във формат XML/XMI). Обикновено в зависимост от шаблона, който се налага върху модела, част от спецификацията на модела бива филтрирана. Самия код който ще се генерира е вграден в шаблоните.
* „Templates + metamodel” (шаблони + мета-модел). Това е разширение на подхода “templates + filtering”. При този подход, вместо шаблоните да се налагат директно върху модела, първо се създава инстанция на мета-модел от спецификацията на модела. След това шаблоните се налагат върху мета-модела.
* “Frame processing” – (рамкова обработка). Този подход описва начин да се генерира код с т.нар. рамки. Рамките може да се разглеждат като програми или фрагменти от програми, които генерират код в резултат на тяхното оценяване. Рамките могат да бъдат параметризирани с числови или символни данни, както и с инстанции на други рамки.
* “API-based generators” (API – базирани генератори. API – Application Programming Interface – Приложен програмен интерфейс). API базирания генератор предоставя програмен интерфейс, с който се пишат програмите за генериране на код. Този интерфейс обикновено е базиран на мета-модела/синтаксиса на целевия език.
* “Inline code generation”. Този подход описва техника, където кодо-генерацията се прави по време на интерпретиране, компилиране или при подготовка за компилиране на обикновена, негенерирана програма. Този процес обикновено модифицира програмата, която след това се интерпретира или компилира.
* „Code attributes” (Атрибути на кода). Кодовите атрибути описват средствата, с които негенерираният програмен код може да съдържа анотации или атрибути, които не са описани в самия него. На базата на тези атрибути може да се генерира допълнителен код.
* „Code weaving” („Преплитане на кода“). Преплитането на кода осигурява комбинирането на различни части от програмата. Тези различни части обикновено описват различни независими аспекти, които се комбинират в „преплетената“ програма. „Преплитането“ е базирано на спецификации, как различните аспекти пасват заедно.
  + 1. **Подход Templates + Filtering.**

Този подход е приложим, когато е необходимо да бъдат създавани прости кодови фрагменти извлечени от спецификация от по-високо ниво. Тъй като се предполага че спецификацията е обемна и сложна, само определени части от нея трябва да бъдат филтрирани и да се създаде код от тях. Необходимо е, шаблонният език да има достъп до избраната част от модела, за да може да използва информацията от него в целевия код.

По време на генерацията филтрите се прилагат върху цялостното описание на модела, като се изважда само тази част от него, която ще е свързана с кодо-генерацията. Върху така направената извадка от спецификацията се прилагат шаблоните, на база на които се генерира изходен код (фиг. 1.6.1.1).



Фигура 1.6.1.1. Последователност на процеса Templates + filtering

Подхода Templates + filtering е ефикасен начин да се извърши кодо-генерация или моделна трансформация, когато спецификацията на модела е подробна и върху описанието могат да се приложат мощни филтриращи механизми. Създаването на генератори по този подход е относително праволинейно. Недостатъците на подхода са, че той може да стане твърде сложен, когато е необходимо да се правят по-сложни и не тривиални заявки към модела, както и че инфраструктурата на генератора е тясно свързана със синтаксиса на спецификацията на модела. Основното предимство е, че този подход може да бъде базиран на стандартизирани инструменти като XML и XSLT.

Основно тази техника е приложима, когато спецификацията на изходните кодове е много добре структурирана и използва систематичен и добре дефиниран мета-синтаксис. Особено подходяща е за кодо-генерация от тип модел-модел или модел-изпълним код. XML базираните системи в момента са широко използвани за спецификация на модели и по специално XMI. За филтриране на модела може да се използват XSLT или XQuery. За генерация от тип код-код обикновено се използват API-базирани генератори, поради факта, че е трудно да се специфицират правила и филтри за генерирането на изпълним код.

* + 1. **Подход Templates +Metamodel**

Този подход може да се използва, когато са налице добре дефинирани блокове, за изграждане на система и тези блокове имат ясно дефинирана роля в реализационната платформа. Необходимо е шаблоните за кодо-генерация да са специфицирани с концепциите на специфичната проблемна област и да не зависят пряко от детайлите от ниско ниво на данните за моделиране.

За да се състави програмен генератор от такъв тип трябва да се развие настройваем мета-модел на кодо-генератора. Мета-моделирането представлява дефиниране на модели за моделите (фиг. 1.6.1.1.). Тези мета-модели дефинират кои моделиращи елементи могат да се вградят в модела (например инстанции на мета-модел), както и допълнителни ограничения към модела. Например, може да се зададе, че елементите в модел, определен че е от стереотип „услуга“, да са задължени да включват операции инициализиране, стартиране и спиране. Също така мета-модела може да дефинира кои типове входни данни са валидни за операциите на модела от тип „услуга“. Например може да се допускат само примитивни типове входни данни. В този смисъл мета-моделите позволяват на разработчиците да адаптират моделирането според конкретните нужди на проблемната област.



Фигура 1.6.2.1. Процес на моделиране с мета-модели

Най-често срещаното в литературата определение за мета-модел е: „Мета-модела е точна дефиниция на конструкциите и правилата необходими за създаване на семантични модели“ [RP04]. Този мета-модел трябва да бъде базиран на данните от модела. Мета-модела трябва да може да се адаптира, за да включва концепции от проблемната област, които от своя страна да бъдат прилагани върху елементи от модела.

За да се премине към кодо-генерация е необходимо спецификацията на системата да бъде описана по правилата на мета-модела на генератора. След това се създава инстанция на мета-модела, върху която да се прилагат шаблоните за генерация на код (фиг.1.6.2.2.). Шаблоните за кодо-генератора трябва да се напишат така, че да могат да се прилагат спрямо изразите в мета-модела.



Фигура 1.6.2.2. Последователност на процеса Templates + metamodel

Основно предимство на този подход на програмна генерация е, че разбора на модела и на инстанцията на мета-модела е напълно разделена от шаблоните, които контролират кодо-генерацията. Съответно промяната на формата на модела става възможна без да е необходимо да се прави промяна в шаблоните. Също така това дава възможност да се създават шаблони от гледна точка на мета-модела (адаптирани към проблемната област), а не от гледна точка на детайлите от ниско ниво свързани с представянето на модела. По този начин мета-модела може да се фокусира върху проблемната област и съответно става по-лесен за разчитане и поддръжка. Възможно е и в мета-модела да се интегрират и специфични за областта атрибути и поведение. Също така мета-модела е идеалното място за залагане на ограничения по време на моделирането. Спазването на тези ограничения ще бъде проверено от генератора когато мета-модела се инстанцира по време на генерация. В следствие на всички изброени съображения езика и начина по който са описани шаблоните могат да бъдат прости и лесни за разбиране, а сложността на описанието на системата ще бъде изнесена в мета-модела.

* + 1. **API-базирани генератори**

API-базирани генератори (API – Application Programming Interface) се използват, когато е необходимо да се генерира относително малко количество код, което обработва добре дефинирана задача. Обикновено този код се генерира в контекста на специализирани специфични за решаването на конкретната задача програмни инструменти. При API базираното генериране не се използват шаблони или модели, а ръчно написана програма използва наличните програмни интерфейси за да създаде или модифицира целевия код (фиг. 1.6.3.1.).



Фигура 1.6.3.1. API-базирани генератори

API базираните генератори са строго специфични по отношение на програмния език, който те трябва да генерират. Поради факта, че не се използват шаблони или мета-модели създаването на по-големи системи става сложно и тромаво. За такива цели подхода Templates + metamodel е много по-подходящ. Въпреки този си недостатък API базираните генератори често се използват като добавка или като основа при работа с други по-сложни типове генератори.

* 1. **Интеграция на генериран с негенериран код**

В много случаи програмистите предпочитат да не използват програмни генератори, поради основанието, че генерирания код е труден за четене. Обикновено това е истина. Проблема на генерирания код не е в подредба, форматиране и стил – модерните генератори създават добре изглеждащ код. Генерираният код обаче е труден за разбиране, поради факта, че съдържа много детайли от ниско ниво, с които програмистите предпочитат да не се занимават.

В повечето случаи не би трябвало да е необходимо да се чете, камо ли да се модифицира генерирания код. Има други начини за успешно интегриране на генериран и негенериран код. За начало трябва да се класифицират подходите за генерация на код. Те могат да бъдат разделени на два отделни класа: Подходи, генерирания код и негенерирания код са отделни и подходи при които генерирания и негенерирания код се сливат. Резултатите от класификацията са показани в таблица 1.7.1.

|  |  |
| --- | --- |
| Клас | Подход |
| Генерирания и негенериран код са отделни | Templates + filtering  Templates + metamodel  Frame processing  API-based generator  Code attributes |
| Генерирания и негенериран код са слети | Inline code generation  Code weaving |

Таблица 1.7.1. Класификация на методите за кодо-генерация на базата на разделяне на генерирания и негенерирания код.

Интеграция между генериран и негенериран код се налага само при подходите, при които генерирания и негенерирания код са отделни. Основните подходи за интеграция са:

* Използване на библиотеки – генерирания код може да извиква негенериран код съдържащ се в библиотеки. Това дава възможност да се генерира възможно най-малко код и да се разчита на предварително имплементирани компоненти, които да се използват от генерирания код.
* Извиквания на генериран код – възможността негенериран код да извиква генерирани функции. На практика това е възможно, когато негенерирания код се обръща към абстрактни класове или интерфейси, които генерирания код имплементира.
* Използване на „фабрики“ (factory – в обектно ориентираното програмиране това е обект, който служи за създаване на други обекти [FRBS04]). Прилагането на този подход дава възможност генериран код да се „инжектира“ в негенерирания.
* Наследяване на негенерираните класове. Негенерираните базови класове могат да съдържат полезни общи методи, които да бъдат извиквани от генерираните наследници.
* Абстрактни методи в базовите негенерирани класове. Базовият клас получава възможност да извиква абстрактни методи, които са имплементирани от генерираният наследник.

Възможно е да се комбинират няколко от изброените подходи. Например генерираният код може да използва абстрактен интерфейс от негенерирания код, който е имплементиран от друг фрагмент генериран код и е достъпен през негенерирана „фабрика“. В контекста на проекта различни части могат да бъдат реализирани с различни средства.

Като обобщена препоръка може да се каже, че интеграцията на генериран и негенериран код може да бъде опростена, като се дефинира ясна архитектура с добре дефинирани отговорности и интерфейси. От своя страна интерфейсите трябва да дефинират кои части от кода са генерирани и кои не.

* 1. **Взаимоотношения и връзки между подходите за кодо-генерация.**

По отношение на основния подход на генериране на кода подходите за програмна генерация могат да бъдат разделени на два класа шаблонно генериране и API базирано генериране. Фигура 1.8.1. илюстрира разделението на класовете подходи и връзките между тях.



Фигура 1.8.1. Подходи за програмна генерация и връзки между тях.

При “Templates + filtering” кода се генерира директно от спецификацията на модела и шаблоните, комбинирани с филтри и правила. „Templates + metamodel” може да се разглежда като разширение на “Templates + filtering”, тъй като мета-модела и инстанциите са „поставени в средата“, за да разделят шаблоните от синтаксиса на модела.

В света на API базираните подходи, очевидно от фигурата най-общ е директният “API-based” подход. “Code attributes” основно се занимава с начина по който информацията за генериране се специфицира в кодовите атрибути. Сам по себе си този подход не съдържа информация за начина по който се генерира крайния код. В повечето инструменти (.NET, XDoclet) този подход се използва като разширение на “API-based”. „Code weaving” се занимава повече с начина, по който различните аспекти са специфицирани и как те могат да бъдат обединени. За произвеждането на крайния код обикновено отново се използва „API-based“ подход. “Inline code generation” също може да използва „API-based”, въпреки че в повечето случаи се прилага само просто обработване на текста от типа, включване или изключване на определени фрагменти в зависимост от зададените условия. „Frame processing” също може да ползва базовия подход, като вътрешно инстанциите на рамките са базирани на синтаксиса/мета-модела на целевия език.

В таблица 1.8.1 е направена категоризация на подходите за кодо-генерация по отношение на основните им характеристики разгледани по-горе.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Генериран/ негенериран код | Шаблон/ API подход | Сложност за първоначално обучение | Пригодимост за сложни задачи | Гъвкавост |
| Template + filtering | Отделни | шаблонен | ниска | Не добра | Не добра |
| Template + metamodel | Отделни | Шаблон + API на модела | висока | Много добра | Много добра |
| Frame processors | Отделни | Шаблон | висока | Много добра | Много добра |
| API based | Отделни или интегрирани | API | ниска/висока | Зависи от нивото на абстракция | Зависи от нивото на абстракция |
| Inline code generation | Интегрирани | Шаблон/API | ниска | Не добра | добра |
| Code attributes | Отделни | Шаблон/API | ниска | Добра | Не добра |
| Code weaving | интегрирани | API | висока | Добра | добра |

В таблица 1.8.2. е показано в кои от най-широко разпространените технологии се използват гореописаните подходи и техники за кодо-генерация.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Моделна трансформация | Генерация на изходен код | Трансформация на код | Пренаписване на байтов код |
| Template + filtering | XML + XSLT | XML+XSLT |  |  |
| Template + metamodel |  | b+m generator framework | Open c++  Open Java |  |
| Frame processors |  | ANGIE  XVCL |  |  |
| API based | XMI APIs | JEnerator | Compost | BCEL  .NET CodeDom  Reflection.Emit |
| Inline code generation |  |  | Preprocessors,  C++ templates |  |
| Code attributes |  | XDoclet |  | .NET |
| Code weaving |  |  | AspectJ  AspectC++ | HyperJ |

* 1. **Взаимоотношения и връзки между подходите за кодо-генерация.**

Глава 3. Архитектура на програмен генератор.

Основна структура на обектен модел на система за програмна генерация на реално времеви управляващи устройства.

Предложеният модел представлява йерархична система от графи, подчинени един на друг и описание на хоризонталните и вертикални връзки между тях. Всяка генерирана система се представя от един функционален блок. Функционалният блок представя интерфейса на системата, който я свърза с външния свят. Такива връзки могат да бъдат специализирани съобщения за връзка с други системи от същия тип, списък с комуникационни интерфейси за връзка с други системи за управление, събиране и съхранение на данни, драйвери за управление на входно-изходни устройства. Всеки функционален блок съдържа един или повече автоматни графи. Извършването на действие във всеки един от автоматните графи се активира от настъпило вътрешно или външно събитие за функционалния блок. Такова събитие може да бъде изминал период от време (графа представлява периодична задача) или друг вид събитие (подаден сигнал на вход на системата, заявка за обработка на данни от друг функционален блок и т.н.). След като един граф бъде активиран, той може да продължи да работи докато не достигне крайна точка или да се активира еднократно, да извърши зададените за текущото състояние действия=, да извърши преход към друго състояние (ако има изпълнени необходимите условия и след това да чака следващо активиращо събитие. Всеки от автоматните графи има предварително зададена начална точка, в която той се намира след стартирането на системата. Допустимо е да съществуват графи, които нямат крайна точка. Веднъж стартирани те ще работят докато работи цялата система. Изпълнението на такъв граф може да спре само ако системата получи сигнал за спиране на работа (например при преминаване в авариен режим, поради преминаване в режим на готовност с цел пестене на енергия и т.н.). В такъв случай целият функционален блок преустановява работата си.

Всеки автоматен граф има определен брой състояния. Във всяко състояние може да са дефинирани едно или повече условия за извършване на преход към друго състояние (или за оставане в същото състояние). Ако в едно състояние има дефиниран само един преход и той сочи към него самото, то графа остава в това състояние до края на работата на функционалния блок.

Преходите между състоянията на графа могат да бъдат синхронни или асинхронни. Синхронен преход в случая означава че след извършването на прехода графа ще изчака следващо активиращо събитие, преди да продължи работата си в новото състояние. Асинхронен преход означава че след извършването на прехода графа ще премине веднага към изпълняването на действията заложени в новото състояние и след това ще оцени отново условията за преход в следващо състояние.

Всеки от автоматните графи има заложен приоритет на изпълнение. Този приоритет се подава на диспечера на задачите на операционната система и пряко влияе върху скоростта на изпълнението на графа. Трябва да се отбележи че при изпълнението на функционалния блок върху еднозадачна операционна система, графът с най-висок приоритет може да блокира работата на останалите графи. В този случаи при проектирането на графите трябва да се вземат специални мерки тяхното изпълнение да бъде неблокиращо. При многозадачна операционна система, ако високо приоритетен граф бъде блокиран, за да изчака настъпването на някакво събитие, графите с по-нисък приоритет ще получат възможност да извършват действия. Друг фактор, който трябва да се взема предвид при проектирането на автоматните графи е изчислителния товар, който системата трябва да поеме при изпълнението на всяко тяхно състояние. Ако в граф с висок приоритет на изпълнение е заложено извършването на сериозен обем изчисления, това може да доведе до значително изоставане на работата на останалите графи, което би могло да доведе до влошаване на качеството на работата им и дори до нарушаване на функциите на системата.

\*\*\*(Да се развие текст със стратегии за избягване на подобни проблеми – например динамично повишаване на приоритета на изоставащи задачи, механизми за самовъзстановяване на системата в случай на блокиране на задача с цел гарантиране на отказоустойчивост)

За всеки от дефинираните преходи между състоянията на графа може да има по едно или повече условия за извършване на преход. Ако в едно състояние има повече от един преход то те се приоритизират, като условията за извършване на преход с най-висок приоритет се оценяват първи. Ако те са изпълнени прехода се извършва, а ако не се преминава към оценка на следващо условие за преход и т.н. При проектирането на системата е заложен преход по подразбиране, който е с най-нисък приоритет и се активира в случай че никое от предхождащите го условия не е изпълнено. Такъв преход обикновено е преход към същото състояние или е преход, който поставя графа в състояние на изчакване на събитие, което ще промени текущите условия за изпълнение на преход от това състояние.

Обектния модел на системата теоретически позволява динамично променяне на условията за изпълнение на преход, както и динамично добавяне или премахване на условия. Също така модела дава възможност за динамично създаване, премахване или пренасочване на преходи. За да може това да бъде осъществено е необходимо за всеки граф да се реализира супервайзорна задача, която при нужда ще изпълнява гореописаните промени. Пример за система от подобен тип е разпределена ad-hoc система, при която регистрирането на нов възел (ново устройство, нов потребител, нова комуникационна връзка) поражда събитие, което сигнализира супервайзорната задача да създаде нов преход, условия и при нужда ново състояние в графа с предварително дефинирано поведение. За реално-времева система с цел бързодействие и предсказуемост на системата всички преходи и състояния, които биха могли да се добавят по време на работа трябва да са предварително дефинирани. При възникване на нужда от тяхното използване те съответно ще се активират и изпълняват докато е необходимо.

Наличието на гъвкав модел дава възможност да се изграждат модулни системи за работа в реално време, които позволяват добавяне или премахване на интелигентни модули по време на работа на системата без нужда от спиране на работа и преконфигуриране.

Във всяко едно от състоянията на системата може да се дефинират множество сигнални графи, които извършват обработка на постъпващите в системата данни и реално представляват изчислителният товар. Всеки сигнален граф съдържа един или повече елементарни или сложни (съставни) функционални модули свързани помежду си с връзки за активация (определящи реда на изпълнение на функционалните модули) и връзки за предаване на данните. При изпълнение на сигналния граф първо се изпълняват функционалните модули, които нямат входни активиращи връзки. След тяхното изпълнение те ре регистрират като изпълнени и изпълнението продължава с тези модули, които имат пълен набор от активирани входове.

Системата за генерация не позволява създаване на сигнален граф, б който има функционални модули с активиращи входове, които не са свързани към изходи на други функционални модули. Това гарантира че изпълнението на сигналния граф винаги ще завършва и всички функционални модули ще бъдат изпълнени. Всеки функционален модул може да се изпълни само по веднъж в рамките на едно изпълнение на сигналния граф. След завършване на изпълнението на текущия граф се стартира изпълнението на следващ сигнален граф регистриран за изпълнение в текущото състояние на автоматния граф или се преминава към проверка на условията за извършване на преход.

Предикатите на условията за извършване на преход в конкретната реализация на обектния модел представляват изходи на функционални модули.

\*\*\* ДА се разпише по-подробно как се реализират видовете условия (аналогови, булеви, и т.н.) и как се генерира BDD.

Всеки функционален модул от сигналния граф представлява представяне на изчислителна или комуникационна процедура. Тази изчислителна процедура може да има входове за данни за обработка, входове за параметри и входове за вътрешни състояния. Параметрите и вътрешните състояния на процедурата са дефинирани като входове с цел да се даде възможност при проектирането на сигналния граф тези входове да се свържат като изходи на други функционални модули, като по този начин се реализират обратни връзки, динамична промяна на параметрите в зависимост от резултата от изчисленията (адаптивни регулатори) и т.н.

Всеки вход на функционален модул може да се дефинира като вход тип връзка или като вход тип статична глобална променлива. Когато входа е от тип връзка той трябва задължително да бъде свързан към изход на друг функционален модул, който от своя страна е дефиниран като глобален.

Приноси

В дисертацията е предложен подход на генеративно програмиране като широкообхватна парадигма за постигане на качествено и лесно за възприемане описание на програмираните системи, преизползваемост и адаптируемост на кода, като в същото време не се прави сериозен компромис с производителността и изискванията към системните ресурси на произвеждания софтуер. Предложени са архитектура и реализация на програмен генератор за системи за управление и симулация, архитектура и реализация на реално-времеви интерпретатор и методологии за валидиране на генерираните конфигурации (като програма и като модел на система).

Списък със съкращенията

[ABRT93] N. Audsley; A. Burns, M. Richardson, K. Tindell, Applying new scheduling theory to static priority pre-emptive scheduling, Software Engineering Journal (Volume:8 , Issue: 5 ), 1993

[AKZ96] Awad, Kuusela and Ziegler. *Object Oriented Technology for Real-Time Systems.*

Prentice-Hall. 1996

[BW95] A. Burns and A. Wellings, *HRT-HOOD: A Structured Design Method for Hard*

*Real-Time Systems.* Elsevier Science. 1995.

[CARD96] L. Cardelli, “Bad Engineering Properties of Object-Oriented Languages.” *ACM Computing Surveys*, Volume 28A, Number 4, 1996

[CE2000] Czarnecki & Eisenecker, *Generative Programming, Addison-Wesley, 2000*

[CEGVV98] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. GenerativeProgramming and Active libraries. 1998

[DIJ76] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976

[DLP76] D.L. Pamas, *On the Design of Program Families*, IEEE Transactions on Software Engineering, March, 1976

[DS00] Diomidis Spinellis. Notable design patterns for domain specific languages. Journal of Systems and Software, 56(1):91–99, 2001.

[EIS97] U. Eisenecker. Generative Programming (GP) with C++. In *Proceedings of Modular Programming Languages, JMLC’97, Linz, Austria, 1997*, H. Mössenböck, (Ed.), Springer-Verlag, Heidelberg 1997

[FRBS04] Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra, Head First Design Patterns, O'Reilly Media, 2004

[HC06] K. [Czarnecki](http://ieeexplore.ieee.org/search/searchresult.jsp?searchWithin=p_Authors:.QT.Czarnecki,%20K..QT.&searchWithin=p_Author_Ids:37303741900&newsearch=true), S. [Helsen](http://ieeexplore.ieee.org/search/searchresult.jsp?searchWithin=p_Authors:.QT.Helsen,%20S..QT.&searchWithin=p_Author_Ids:38110844000&newsearch=true), Feature-based survey of model transformation approaches. [IBM Systems Journal](http://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=5288519) (Volume:45 ,  [Issue: 3](http://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=5386619)),2006

[JAP2000] Juan Antonio de la Puente, Real-Time Object-Oriented Design And Formal Methods, Dept. of Telematics Engineering, School of Telecommunication, Technical University of Madrid. 2000

[ITM92] Yutaka Ishikawa, Hideyuki Tokuda, Clifford W. Mercer, An Object-Oriented Real-Time Programming Language, 0018-9162/92/1000-0066 IEEE, 1992

[KLMM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings ECOOP’97 — Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, 1997*

[LO11] Phillip A. Laplante, Seppo J. OvaskaReal-Time Systems Design and Analysis, 4th Edition 2011

[MAP97] M.A. De Miguel, A. Alonso and J.A. de la Puente, Object-Oriented Design of

Real-Time Systems with Stereotypes. *Proc. 9th Euromicro Workshop on Real-Time*

*Systems*. Toledo, Spain. IEEE CS Press. 1997.

[MV03] Markus Voelter, A Catalog of Patterns for Program Generation, voelter - ingenieurbüro für softwaretechnologie Ziegelaecker 11, 89520 Heidenheim, Germany. 2003

[RP04] Elvinia Riccobene, Patrizia Scandurra, Towards an Interchange Language for Abstract State Machines, Abstract State Machines 2004, Advances in Theory and Practice vol. 11.

[SGW94] B. Selic, G. Gullekson and Ward, P.T. *Real-Time Object-Oriented Modeling*.

John Wiley and Sons. 1994.

[SL98] J. G. Siek and A. Lumsdaine. The Matrix Template Library: A Unifying Framework for Numerical Linear Algebra. In *Proceedings of the ECOOP’98 Workshop on Parallel Object-Oriented Computing (POOSC’98)*