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 персонализирания и оптимизиран краен продукт да може да бъде автоматично генериран. Крайният 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 е въведен от Дийкстра [DIJ76] и 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).
* 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**
* **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 новогенерираните 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.

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).

[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

[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

[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)*