Годишник на секция "Информатика"
Съюз на учените в България
Том 4, 2011, 1-28

Annual of "Informatics" Section
Union of Scientists in Bulgaria
Volume 4, 2011, 1-28

ПОСТРОЯВАНЕ НА КОРЕКТНИ ОБЕКТНО-ОРИЕНТИРАНИ ПРОГРАМИ ЧРЕЗ ИЗГРАЖДАНЕ НА ТЕХНИ ОБОБЩЕНОМРЕЖОВИ МОДЕЛИ

Магдалина Тодорова

СУ "Св. Кл. Охридски", Факултет по математика и информатика
email: magda@fmi.uni-sofia.bg

Резюме: В статията неформално е описан подход за построяване на коректни относно зададени спецификации обектно-ориентирани програми. Подходът интегрира концепцията „design by contract" с подходи за верификация от тип доказателство на теореми и проверка на съгласуваност. Спецификациите, относно които се построяват коректни обектно-ориентирани програми, отразяват връзките между член-функциите на класовете във вид на редици от коректни обръщения. Проверката за съгласуваност се извършва, като се изграждат обобщеномрежови модели на класовете на програмата и на функциите, които се верифицират, след което моделите се изпълняват паралелно. Ако в резултат някой от създадените модели не завърши изпълнението си, в програмата е открита грешка. Характеристиките на ядрата на обобщените мрежи посочват мястото, в което е грешката. Описанието на подхода е опростено, като е представено за обектно-ориентирана програма (ООП), състояща се от един клас и главна функция, която използва класа. Предложен е пример, илюстриращ подробно изпълнението на основните стъпки на подхода.
Ключови думи: Обектно-ориентирано програмиране, Верификация на програми, Обобщени мрежи, Моделиране

1. Въведение

Стандартът IEEE 1012 определя верификацията на програмно осигуряване (ПО) като техника, която проверява дали артефактите (техническо задание, модел на предметната област, описание на архитектурата, програмен код, потребителска документация и др.), създадени в хода на разработката на ПО, съответстват на други артефакти, определени като начални (входни) данни, а също дали тези артефакти съответстват на правилата и стандартите. Известни методи за верификация са: софтуерна експертиза, статичен анализ, формални методи, динамични методи и синтетични методи. Динамичните методи са най-евтините техники за верификация и най-често се използват в практиката. През последните години се провеждат активни изследвания за изгражда-

нето на среди за верификация, в основите на които са динамичните методи. Примери за успешни решения в тази посока са предложени в [1, 2, 3]. Формалните методи за верификация са най-надеждното средство, осигуряващо правилното функциониране на програмното осигуряване. Използват се за верификация на свойства, които могат да се изразят формално в рамките на някои математически модели, а също на тези артефакти, за които могат да се построят съответни формални модели. Примери за програмни средства за формална верификация са дадени в [4, 5, 6, 7, 8, 9]. Недостатък на този вид методи е, че построяването и анализът на формалните модели изискват значителни усилия. Реализират се от висококвалифицирани специалисти по формални модели. Наличието на такива специалисти не е голямо и цената на този вид верификация е висока. Съществуващите формални методи за верификация се прилагат трудно на практика, заради което не се използват масово.

Анализът на ефективността на обучението по програмиране и най-вече на най-масово използваното – обектно-ориентираното [10, 11, 12], показва, че трябва да се разработят образователни средства, повишаващи ефективността му. Създаването на автоматизирани инструменти за подпомагане на учебния процес [13], обучението по прилагане на формални методи за построяване на коректно ПО, започващо от най-ранните курсове по езици за програмиране са крачка в посока на увеличаването на ефективността на обучението по програмиране.

В статията е описан формален подход за верификация на ООП, който лесно може да се прилага за построяване на коректни обектно-ориентирани програми от софтурни инженери, които не са специалисти в областта на формалните методи. Подходът може да се използва и за целите на обучението на студенти за прилагане на формални средства в процеса на създаване на коректно ПО.

Причината за предполаганата по-високата степен на приложимост на метода е, че като средство за моделиране е използван апаратът на обобщените мрежи [14, 15]. Последните са подходящо средство за реализиране на подхода, тъй като:

- Спецификациите, относно които се извършва верификацията на функциите на обектно-ориентираните програми се представят чрез мрежови структури.
- ОМ имат висока степен на приложимост в областта на софтуерното моделиране [16, 17].
- Създадена е методология за построяване на обобщени мрежи [14, 15], с помощта на която лесно може да се построи обобщена мрежа, съответстваща на функция на ООП.
- Изградена е също методология за моделиране на дефинирането на класове чрез обобщени мрежи [18].
- ОМ са средство за моделиране на паралелни процеси, което позволява ефективно паралелно да се изпълняват обобщени мрежи [14].
- Реализирани са и са в процес на усъвършенстване автоматизирани средства за изпълнение на ОМ [19, 20, 21, 22, 23, 24 и 25].

В основите на верификацията на обектно-ориентираните програми е концепцията „design by contract“ [26, 27], предполагаща описание на специални твърдения (договори), които трябва да са в сила в определени места на програмата. Договорите могат да описват: елементарни твърдения, инварианти на класове, предусловия и постусловия на функции и член-функции. Наричат се още спецификация на програмата.

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

Неформално, един клас е коректен когато неговата реализация, зададена чрез член-функциите на класа, е съгласувана с предусловията, постусловията и инвариантата на класа [26].

По-строго коректността на клас по отношение на някаква спецификация е определена в [26] по следния начин. Нека `C` е клас, `Inv` е неговата инварианта, `r` е произволна член-функция на класа. За `r` с $\mathtt{Body}_r$ означаваме тялото на `r`, с $\mathtt{pre}_r(x_r)$ и с $\mathtt{post}_r(x_r)$ – предусловието и постусловието на `r` с допустими аргументи x_r. Ако предусловието или постусловието на член-функция на `C` е пропуснато, счита се, че то е `true`. Чрез $\mathtt{Default}_C$ се означава твърдение, изразяващо връзки по подразбиране между член-данните на класа `C`.

Класът `C` е коректен по отношение на своята спецификация, ако:

i. За всеки конструктор `P` и за всяко допустимо множество от аргументи x_p е в сила тройката на Хоар

$$\{\mathtt{Default}_C \ \mathtt{and} \ \mathtt{pre}_p(x_p)\} \ \mathtt{Body}_P \ \{\mathtt{post}_p(x_p) \ \mathtt{and} \ \mathtt{Inv}\} \quad (1)$$

ii. За всяка член-функция `r` с множество от допустими аргументи x_r е в сила тройката на Хоар

$$\{\mathtt{pre}_r(x_r) \ \mathtt{and} \ \mathtt{Inv}\} \ \mathtt{Body}_r \ \{\mathtt{post}_r(x_r) \ \mathtt{and} \ \mathtt{Inv}\} \quad (2)$$

При верифицирането на член-функциите на клас относно свързаните с тях предусловия и постусловия се формулират и доказват теореми, представени чрез тройки на Хоар [28] от вида (1) и (2) и означаващи тотална коректност. За доказателство на теоремите може да се използва техниката на преобразуващите предикати [29], както и някои системи за автоматично доказателство на теореми.

Верифицирането на функция на ООП относно зададена спецификация реализираме, като изграждаме два обобщеномрежови модела – на функцията, която предстои да бъде верифицирана, и на спецификацията, относно която тя ще се верифицира, след което проверяваме съответствието (съгласуваността) на модела на функцията с модела на спецификацията.

2. Описание на подхода

Нека ООП се състои от клас *C* и функция *M*, която използва класа. Спецификацията, относно която ще се проверява и построява коректна програма, ще задава възможните

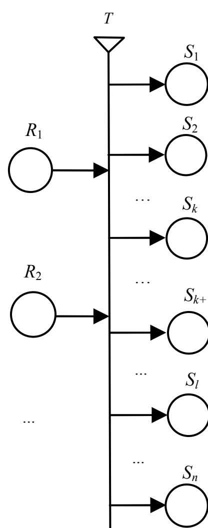
коректни връзки между член-функциите на класа *C*. Създаването на коректна относно зададена спецификация ООП преминава през следните стъпки:

а) *Създаване на формален мрежов модел на класа*

Изгражда се обобщеномрежов модел, специфициращ връзките между член-функциите на класа. Ще го наричаме *формален ОМ модел на класа.* В следващото описание формалния обобщеномрежов проект на *C* ще означаваме с GN_C. Той дефинира възможните коректни начини за използване на член-функциите на класа. Всяка функция, която прилага класа трябва да има поведение, което съответства на формалния му мрежов проект.

Формалният обобщеномрежов модел на клас е обобщена мрежа, която се състои от множество от преходи от вида, представен на фиг. 1., където

$$T = \langle \{R_1, R_2, \ldots\}, \{S_1, S_2, \ldots, S_n\}, t' \rangle$$



Фигура 1. Дефиниция на преход на обобщена мрежа

t' =	S_1	S_2	…	S_k	S_{k+1}	…	S_l	…	S_n
R_1	*Член-функцията е f. AND В сила е условие* P_1.	*Член-функцията е f. AND В сила е условие* P_2.	…	*Член-функцията е f. AND В сила е условие* P_k.	*Член-функцията е g. AND В сила е условие* P_{k+1}.	…	*Член-функцията е g. AND В сила е условие* P_l.	…	*Член-функцията е h. AND В сила е условие* P_n.
R_2	*Член-функцията е f. AND В сила е условие* P_1.	*Член-функцията е f. AND В сила е условие* P_2.	…	*Член-функцията е f. AND В сила е условие* P_k.	*Член-функцията е g. AND В сила е условие* P_{k+1}.	…	*Член-функцията е g. AND В сила е условие* P_l.	…	*Член-функцията е h. AND В сила е условие* P_n.
…	…	…	…	…	…	…	…	…	…

Предикатите P_1, P_2, …, P_k, както и P_{k+1}, P_{k+2}, …, P_l отнасящи се до прилагането на една и съща член-функция, са взаимно-изключващи се, а f, g, …, h са различни член-функции на класа.

Всяка позиция на прехода представя логическо състояние, в което може да попадне обект на класа. Ще го наричаме *логическо състояние на обекта в позицията* или накратко *логическо състояние на позицията*. Логическото състояние се задава чрез булев израз. Ядрата в позициите на $\mathrm{GN_C}$ съответстват на обекти на класа. С всеки обект на клас е свързано *множество от данни*, определено от член-данните на класа. Характеристиката на ядро ще задаваме чрез двойка от вида (*обект, позиция*), където *обект* е идентификатор, задаващ име на обект на класа (ще го наричаме и *име на ядрото*), а *позиция* е името на позицията, в която е ядрото (обектът) в GN_C.

Освен тези две компоненти, в характеристиката на ядро чрез параметъра *обект* неявно участва множеството от данни за обекта (ще ги наричаме още *данни на ядрото*), а чрез параметъра *позиция* – логическото състояние на обекта в позицията (ще го наричаме още и *логическо състояние на ядрото*).

Преходите задават член-функциите, които могат да се изпълнят коректно при предпоставки, следващи от логическите състояния на позициите. Моделът GN_C започва с преход с една входна позиция, реализиращ изпълнението на конструкторите на класа.

б) *Дефиниране на формална спецификация на класа и верификация на класа*

На тази стъпка се построява формалната спецификация на класа. За целта се задават инварианта на класа, предусловията и постусловията на всяка негова член-функция. Реализират се методите на класа. Извършва се формална верификация на класа и доказателство, че формалният мрежов проект на класа съответства на реализацията му.

Реализацията на класа е на някой език за обектно-ориентирано програмиране. В статията използваме C++, но това не ограничава описанието на подхода. Верификацията на класа се осъществява, като за всяка член-функция на класа се построява теорема във вид на тройка на Хоар (1) и (2). Доказателството на теоремите може да се реализира с помощта на техниката на преобразуващите предикати, а също и с помощта на някоя от системите за автоматично доказателство на теореми HOL, Isabelle, Coq и др.

Ако в някоя член-функция се използват оператори за цикъл, за всеки от тях се задават инварианта и ограничаваща функция.

Доказателството, че формалният ОМ модел на класа съответства на реализацията на класа се осъществява, като за логическите състояния на позициите на всеки преход t на GN_C се докаже верността на следните импликации (ще използваме означенията за прехода от фиг. 1):

логическо състояние на позиция R_i =>
предусловието на член-функцията f

логическо състояние на позиция R_i =>
предусловието на член-функцията g

… (3)

логическо състояние на позиция R_i =>
предусловието на член-функцията h ($i = 1, 2, \ldots$)

логическо състояние на позиция S => *Inv*, за всяка позиция S на GN_C (4)

и тройки на Хоар

{*логическо състояние на позиция* R_i && P_j}
$Body_f$
{*логическо състояние на позиция* S_j }, за всяко $i = 1, 2, \ldots$ и $j = 1, 2, ..., k$.

(5)

{*логическо състояние на позиция* R_i && P_j}
$Body_g$
{*логическо състояние на позиция* S_j}, за всяко $i = 1, 2, \ldots$ и $j = k+1, ..., l$.
…
{*логическо състояние на позиция* R_i && P_n }
$Body_h$
{*логическо състояние на позиция* S_n }, за $i = 0, 1, \ldots$

Верността на горните твърдения, съчетана с верификацията на класа осигурява коректността на всички редици от изпълнения на преходи в GN_C.

Нека ядро е в позиция R_i ($i = 0, 1, \ldots$) и са в сила условията:

логическо състояние на позиция R_i => $pre_f(x_f)$

и

Член-функцията е f && P_j ($j = 1, 2, \ldots, k$).

След изпълнението на прехода t, ядрото преминава в позиция S_j. От верността на импликациите

логическо състояние на позиция R_i => $pre_f(x_f)$

и

логическо състояние на позиция R_i => *Inv*

следва

логическо състояние на позиция R_i => $pre_f(x_f)$ && *Inv*

От коректността на член-функцията f относно дефинираната за нея спецификация, следва че в позицията S_j, в която е преминало ядрото, за данните му са в сила постусловието на f и инвариантата на класа, т.е.

{$pre_f(x_f)$ && *Inv*} $Body_f$ {$post_f(x_f)$ && *Inv*}

От последните две твърдения следва верността на тройката на Хоар

{*логическо състояние на позиция* R_i}
$Body_f$
{$post_f(x_f)$ *and Inv*}

От нея и импликацията

логическо състояние на позиция R_i && P_j =>
логическо състояние на позиция R_i

следва

{*логическо състояние на позиция* R_i && P_j}

Body$_f$

{$post_f(x_f)$ *and Inv*}

От тук и от верността на тройката на Хоар

{*логическо състояние на позиция* R_i && P_j}

Body$_f$

{*логическо състояние на позиция* S_j}

следва

{*логическо състояние на позиция* R_i && P_j}

Body$_f$

{$post_f(x_f)$ && *Inv* && *логическо състояние на позиция* S_j}

т.е. движението на ядрото от позиция R_i до позиция S_j съответства на коректно изпълнение на член-функция на класа.

Верността на условията:

{*логическо състояние на позиция* R_i}

Body$_f$

{*логическо състояние на позиция* S_1 ||

логическо състояние на позиция S_2 ||

...

логическо състояние на позиция S_k

}

{ *логическо състояние на позиция* R_i}

Body$_g$

{ *логическо състояние на позиция* S_{k+1} ||

логическо състояние на позиция S_{k+2} ||

...

логическо състояние на позиция S_l

}

не е задължителна, но ако е налице, показва пълнотата на изследваните пътища от коректни обръщения към член-функции, т.е. не са пропуснати позиции, в които ядрото да може да премине за някои стойности на член-данните си.

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

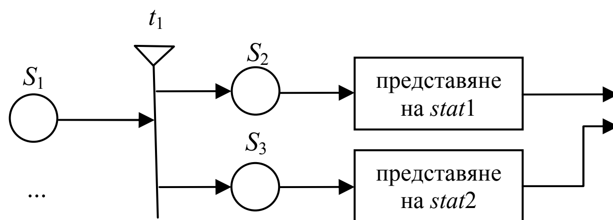
в) *Дефиниране на модел на функцията и проверка за съвместимост на модела на функцията и формалния проект на класа*

На тази стъпка се реализира функцията, която използва класа. Построява се обобщеномрежов модел, който ѝ съответства и се проверява дали обобщеномрежовият модел на реализацията на функцията съответства на формалния мрежов проект на класа.

Тези действия се осъществяват от програмиста, който използва класа. Последният получава реализацията на класа, като със съпътстващата реализацията на класа документация е и формалният мрежов проект, относно който ще се осъществява верификацията на функцията.

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

На операторите за вход/изход, за присвояване и за обръщение към функция или член-функция на клас съответства преход с една или повече входни позиции и една изходна позиция. На условния оператор *if* (*B*) *stat*1; *else stat*2; съответства обобщеномрежов елемент със структура от вида

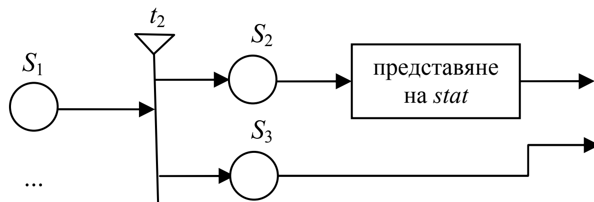


където

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

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

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

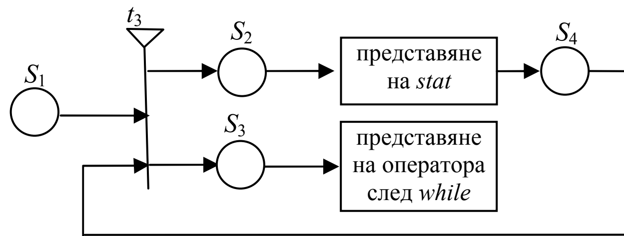


където

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

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

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

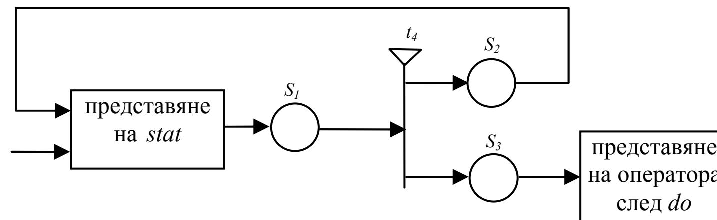


където

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

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

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



където

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

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

След построяването на GN_M се проверява дали обръщенията към член-функциите на класа C в GN_M съответстват на формалния мрежов проект GN_C на класа.

Ще представим накратко идея за реализирането на проверката за съответствието. За целта отначало ще разгледаме случая когато в M е дефиниран само един обект, а след това − когато в M са дефинирани повече от 1 обекти.

1) *Нека в M е дефиниран само един обект.*

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

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

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

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

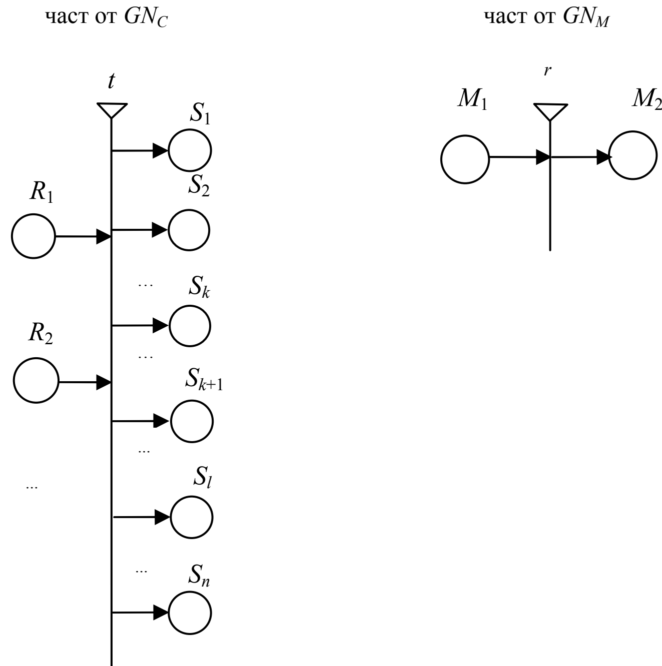
- *Ядрото на мрежата* GN_M *е в позиция, която е входна за преход, свързан с член-функция на класа C*

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

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

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

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



където преходът t се дефинира по същия начин, както на фиг. 1, а $r = \langle \{M_1\}, \{M_2\}, r' \rangle$, където

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

Преходът r ще се изпълни ако са в сила условия Q и P_1, или Q и P_2, или ..., или Q и P_k. В случай, че r може да се изпълни, се изпълнява и преходът t. Ядрото на GN_C ще се премести в позиция S_1 (ако е в сила P_1) или S_2 (ако е в сила P_2) или ... S_k (ако е в сила условие P_k). Ако в условието на прехода t не се съдържа предикатът „Член-функцията е *f*.”, преходът r няма да се изпълни. Последното съответства на „съгласно спецификацията, представена чрез формалния мрежов проект на C в текущото място на функцията M обръщението към член-функцията f не е допустимо (не е коректно)”.

Допълнителното изискване за изпълнение на преход в мрежата GN_M на верифицираната функция M осигурява за използваните член-функции да са в сила предусловията им и коректен резултат след изпълнението им.

Ако съществува изпълнение на GN_M, което не завършва (стига се до преход, който не може да се изпълни), функцията M не е коректна относно зададената спецификация. От позицията в GN_M, в която е спряло изпълнението ѝ, може да се разбере мястото във функцията M, където е грешката и след анализ да се поправи кодът.

2) *Нека в M са дефинирани повече от един обекти.*

Тогава в мрежите GN_M и GN_C ще има повече от едно ядра, съответстващи на обекти на класа. Всяко изпълнение на преход в GN_M, съдържащ обръщение към конструктор на

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

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

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

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

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

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

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

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

Този случай разгледахме по-горе.

3. Пример

Ще илюстрираме описания подход за построяване на коректни обектно-ориентирани програми като дефинираме ООП на езика C++, която съдържа клас *queue*, реализиращ структурата от данни опашка и главна функция main, която използва класа. Опашката ще е представена последователно и нециклично чрез редицата от не повече от *MaxQue* ($MaxQue > 0$) цели числа a_{front}, $a_{front+1}$, …, a_{rear-1}, където *front* и *rear* са указатели към първия елемент и след последния елемент на опашката съответно. В случай, че *front* съвпада с *rear*, опашката е празна, а ако *rear* е равно на *MaxQue*, опашката е пълна. Предикатът $Default_{queue}$ е $MaxQue > 0$. Функцията *main*, която ще използва класа *queue*

ще създава празна опашка, след което ще я модифицира като в нея включва и от нея изключва цели числа. Изборът на операция ще става в зависимост от въведен символ: при въвеждане на '*i*' ще се включва елемент в опашката, при въвеждане на '*r*' ще се изключва елемент и ако въведеният символ е различен от '*i*' и '*r*' изпълнението на програмата ще завършва.

В този пример главната функция дефинира и използва само един обект на класа.

а) *Стъпка* 1. *Дефиниране на формален мрежов проект на класа*

След анализ на дефиницията на структурата от данни опашка, представена последователно и нециклично чрез масив се вижда, че обект на класа *queue* може да попадне в следните четири състояния:

- обектът представя опашка, която е празна, и не е запълнен масивът, съдържащ опашката, т.е. *front* >= 0 && *front* = = *rear* && *rear* < *MaxQue*;
- обектът представя опашка, която не е празна, и не е запълнен масивът, съдържащ опашката, т.е. *front* >= 0 && *front* < *rear* && *rear* < *MaxQue*;
- обектът представя опашка, която е пълна, но от нея могат да се изключват елементи, т.е. *front* >= 0 && *front* < *rear* && *rear* = = *MaxQue*;
- обектът представя опашка, която е празна, и е запълнен масивът, съдържащ опашката, т.е. *front* >= 0 && *front* = = *rear* && *rear* = = *MaxQue*.

В първото състояние над опашката могат да се изпълняват член-функциите *push*, *empty* и *full* на класа *queue*. Във второто състояние над опашката могат да се изпълняват член-функциите *push*, *pop*, *empty* и *full* на *queue*. В третото състояние над опашката могат да се изпълняват член-функциите *pop*, *empty* и *full* и в четвъртото състояние над опашката могат да се изпълняват член-функциите *empty* и *full*. Във всяко от тези четири състояния може да се изпълнят още неявният деструктор на класа *queue* и конструкторът *queue*. С цел да не се усложнява графичното изображение и тъй като условието на примерната програма не изисква дефинирането на повече от един обект, ще пропуснем възможността за изпълнение на конструктора в описаните по-горе състояния.

На фиг. 2 е представена обобщена мрежа, дефинираща непълно формалния ОМ модел на класа. Ще я означаваме с GN_1. Ядрата в тази мрежа имат характеристики от вида: (*обект*, *позиция*), където параметърът *обект* освен с име неявно е свързан и с данни, в случая (*front*, *rear*, *a*), а параметърът *позиция* освен с име на позиция, неявно е свързан и с логическо състояние на позицията.

Логическите състояния на позициите в GN_1 са:

- *front* >= 0 && *front* = = *rear* && *rear* < *MaxQue* на позиции S_1, S_2, S_3 и S_4;
- *front* >= 0 && *front* < *rear* && *rear* < *MaxQue* на позиции U_1, U_2, U_3, U_4 и U_5;
- *front* >= 0 && *front* < *rear* && *rear* = = *MaxQue* на позиции V_1, V_2, *V*3, V_4 и V_5;
- *front* >= 0 && *front* = = *rear* && *rear* = = *MaxQue* на позиции W_1, W_2 и W_3.

Преходите представят допустимите за изпълнение член-функции от съответните позиции на GN_1 и условията, при които тези член-функции могат да се изпълнят. Изпълнението на конструктора на класа съответства на изпълнение на прехода t_1 и

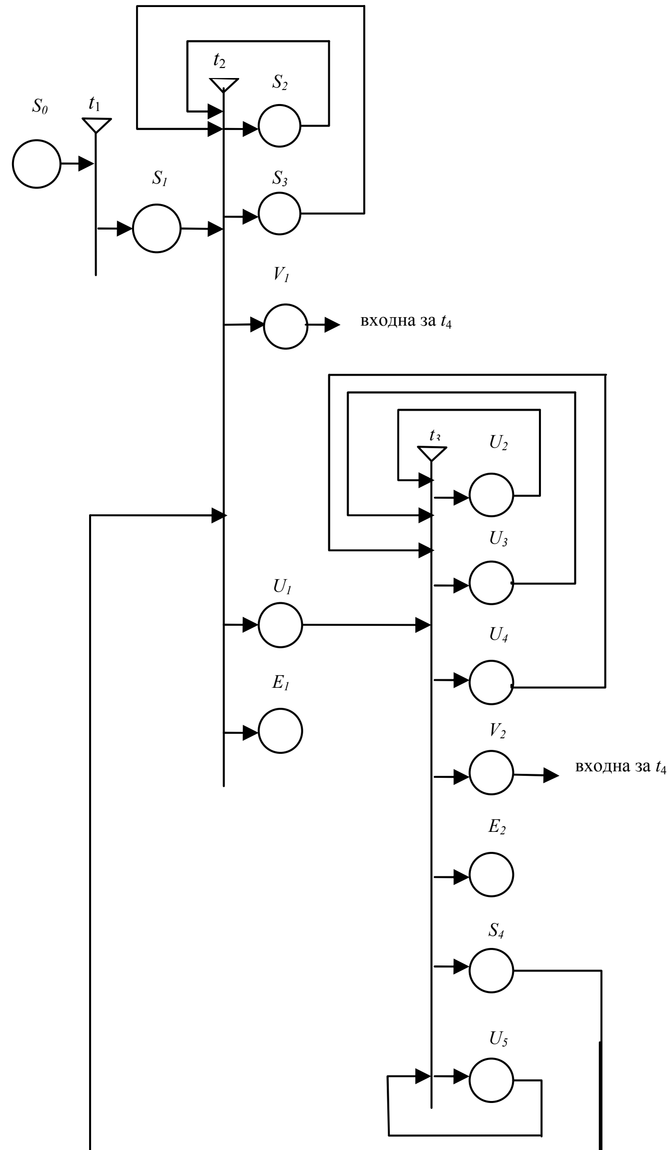
активиране на ядро в позиция S_1 на GN_1, а изпълнението на деструктора съответства на разрушаване на ядрото в позициите E_1, E_2, E_3 и E_4 на GN_1.

Тъй като $Default_{queue}$ е в сила навсякъде в програмата, след изпълнението на прехода t_1 се активира ядро с данни (*front*, rear, *a*) = (0, 0, *a*), удовлетворяващи *front* = = *rear* && $rear < MaxQue$.

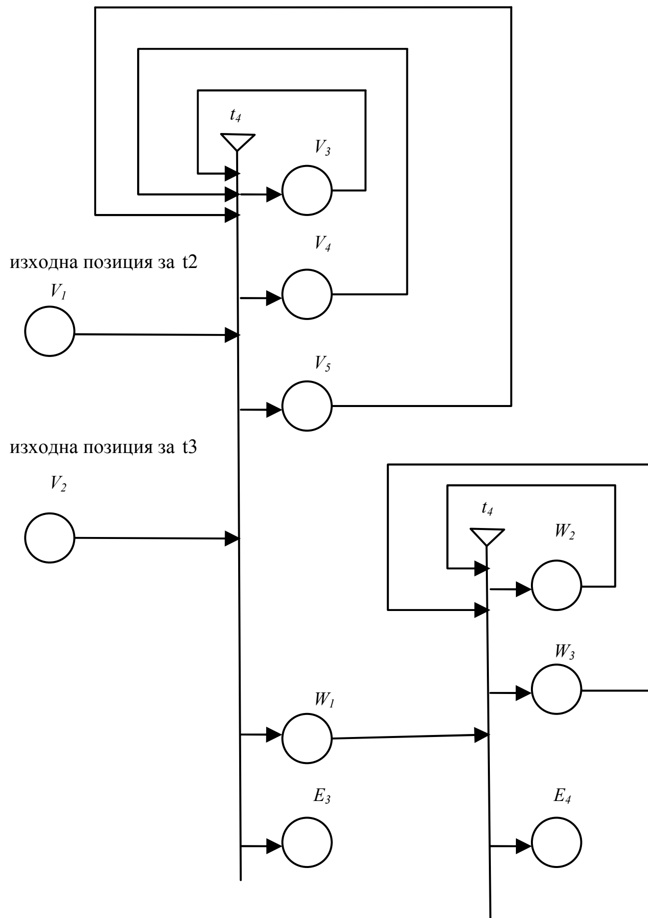
Функцията Φ може да се дефинира по следния начин:

$\Phi((q(0, 0, a), S_1)) = (q(0, 0, a), S_2)$ $\Phi((q(0, 0, a), S_1)) = (q(0, 0, a), S_3)$ $\Phi((q(0, 0, a), S_1)) = (q(0, 1, a), V_1)$ $\Phi((q(0, 0, a), S_1)) = (q(0, 1, a), U_1)$ $\Phi((q(0, 0, a), S_1)) = (–, E_1)$	$\Phi((q(f, r, a), S_i)) = (q(f, r, a), S_2)$ $\Phi((q(f, r, a), S_i)) = (q(f, r, a), S_3)$ $\Phi((q(f, r, a), S_i)) = (q(f, r+1, a), V_1)$ $\Phi((q(f, r, a), S_i)) = (q(f, r+1, a), U_1)$ $\Phi((q(f, r, a), S_i)) = (–, E_1)$ (*i* = 2, 3 и 4)
$\Phi((q(f, r, a), U_j)) = (q(f, r, a), U_2)$ $\Phi((q(f, r, a), U_j)) = (q(f, r, a), U_3)$ $\Phi((q(f, r, a), U_j)) = (q(f, r+1, a), U_4)$ $\Phi((q(f, r, a), U_j)) = (q(f, r+1, a), V_2)$ $\Phi((q(f, r, a), U_j)) = (q(f+1, r, a), S_4)$ $\Phi((q(f, r, a), U_j)) = (q(f+1, r, a), U_5)$ $\Phi((q(f, r, a), U_j)) = (–, E_2)$ (*j* = 1, 2, 3, 4, 5)	$\Phi((q(f, r, a), V_k)) = (q(f, r, a), V_3)$ $\Phi((q(f, r, a), V_k)) = (q(f, r, a), V_4)$ $\Phi((q(f, r, a), V_k)) = (q(f+1, r, a), V_5)$ $\Phi((q(f, r, a), V_k)) = (q(f+1, r, a), W_1)$ $\Phi((q(f, r, a), V_k)) = (–, E_3)$ (*k* = 1, 2, 3, 4, 5)
$\Phi((q(f, r, a), W_n)) = (q(f, r, a), W_2)$ $\Phi((q(f, r, a), W_n)) = (q(f, r, a), W_3)$ $\Phi((q(f, r, a), W_n)) = (–, E_4)$ (*n* = 1, 2, 3)	

където с „–“ е означено отсъствието на характеристика на ядрото (ядрото е разрушено, което съответства на „обектът е разрушен от деструктора на класа”).



Фигура 2. Формален ОМ модел на класа *queue*



Фигура 2. Формален ОМ модел на класа *queue* (продължение)

Следват дефинициите на преходите:

$$t_1 = \langle \{S_0\},\ \{S_1\},\ t_1' \rangle$$

t_1' =

	S_1
S_0	Член-функцията е *queue*.

$$t_2 = \langle \{S_1, S_2, S_3, S_4\},\ \{S_2, S_3, E_1, V_1, U_1\},\ t_2' \rangle$$

t_2' =

	S_2	S_3	E_1	V_1	U_1
S_1	A_1	A_2	A_3	A_4	A_5
S_2	A_1	A_2	A_3	A_4	A_5
S_3	A_1	A_2	A_3	A_4	A_5
S_4	A_1	A_2	A_3	A_4	A_5

където

A_1 = Член-функцията е *empty*.

A_2 = Член-функцията е *full*.

A_3 = Член-функцията е неявният деструктор.

A_4 = Член-функцията е *push*. AND
За данните на ядрото във входните позиции S_i (i = 1, 2, 3, 4) е в сила
$rear$+1 = = $MaxQue$.

A_5 = Член-функцията е *push*. AND
За данните на ядрото във входните позиции S_i (i = 1, 2, 3, 4) е в сила
$rear$+1 < $MaxQue$.

$$t_3 = \langle \{U_1, U_2, U_3, U_4, U_5\}, \{U_2, U_3, U_4, V_2, E_2, S_4, U_5\}, t_3'\rangle$$

t_3' =

	U_2	U_3	U_4	V_2	E_2	S_4	U_5
U_1	A_1	A_2	A_6	A_7	A_3	A_8	A_9
U_2	A_1	A_2	A_6	A_7	A_3	A_8	A_9
U_3	A_1	A_2	A_6	A_7	A_3	A_8	A_9
U_4	A_1	A_2	A_6	A_7	A_3	A_8	A_9
U_5	A_1	A_2	A_6	A_7	A_3	A_8	A_9

където

A_6 = Член-функцията е *push*. AND
За данните на ядрото във входните позиции U_i (i = 1, 2, 3, 4, 5) е в сила
$rear$+1 < $MaxQue$.

A_7 = Член-функцията е *push*. AND
За данните на ядрото във входните позиции U_i (i = 1, 2, 3, 4, 5) е в сила
$rear$+1 = = $MaxQue$.

A_8 = Член-функцията е *pop*. AND
За данните на ядрото във входните позиции U_i (i = 1, 2, 3, 4, 5) е в сила
$front$+1 = = $rear$.

A_9 = Член-функцията е *pop*. AND
За данните на ядрото във входните позиции U_i (i = 1, 2, 3, 4, 5) е в сила
$front$+1 < $rear$.

$$t_4 = \langle \{V_1, V_2, V_3, V_4, V_5\}, \{V_3, V_4, V_5, W_1, E_3\}, t_4'\rangle$$

t_4' =

	V_3	V_4	V_5	W_1	E_3
V_1	A_1	A_2	A_{10}	A_{11}	A_3
V_2	A_1	A_2	A_{10}	A_{11}	A_3
V_3	A_1	A_2	A_{10}	A_{11}	A_3
V_4	A_1	A_2	A_{10}	A_{11}	A_3
V_5	A_1	A_2	A_{10}	A_{11}	A_3

където

A_{10} = Член-функцията е *pop*. AND
За данните на ядрото във входните позиции V_i (i = 1, 2, 3, 4, 5)
е в сила *front*+1 < *rear*.

A_{11} = Член-функцията е *pop*. AND
За данните на ядрото във входните позиции V_i (i = 1, 2, 3, 4, 5)
е в сила *front*+1 = = *rear*.

и

$$t_5 = \langle \{W_1, W_2, W_3\}, \{W_2, W_3, E_4\}, t_5' \rangle$$

t_5' =

	W_2	W_3	E_4
W_1	A_1	A_2	A_3
W_2	A_1	A_2	A_3
W_3	A_1	A_2	A_3

б) *Стъпка* 2

Следва реализацията на класа *queue*, придружена с коментари, отразяващи формалните спецификации: предусловие и постусловие на всяка член-функция и инвариантата на класа. Тъй като телата на член-функциите на *queue* не съдържат оператори за цикъл, не се налага задаването на инварианти и ограничаващи функции за операторите за цикъл. Инвариантата на класа

```
0 <= front && front <= MaxQue &&
0 <= rear && rear <= MaxQue &&
front <= rear
```

изразява съществуването на редицата, представяща опашка. Тя трябва да бъде в сила след всяко изпълнение на конструктора и пред, и след всяко изпълнение на останалите методи на класа. Логическото състояние на всяка позиция е предикат, от верността на който трябва да следва верността на инвариантата на класа.

```
const int MaxQue = 5;

class queue
{ // class_inv
  // 0 <= front &&
  // front <= MaxQue &&
  // 0 <= rear &&
  // rear <= MaxQue &&
  // front <= rear
  private:
    int a[MaxQue]; // носител
    int front,     // начало
        rear;      // край
  public:
    // PRE
    // true
```

```
    // PRE
    // !full()
    // POST
    // !empty() &&
    // a[rear_old] == x &&
    // rear == rear_old + 1
    void push(int x)
    { a[rear] = x;
      int rear_old = rear;
      rear = rear + 1;
    }

    // PRE
    // !empty()
    // POST
    // x == a[front_old] &&
```

`// POST`
`// front == 0 &&`
`// rear == 0`
`queue()`
`{ front = 0;`
`  rear = 0;`
`}`
`// PRE`
`// true`
`// POST`
`// res == (front == rear)`
`bool empty() const`
`{ bool res =(front == rear);`
`  return res;`
`}`
	`// front == front_old + 1`
	`void pop(int& x)`
	`{ x = a[front];`
	`  int front_old = front;`
	`  front++;`
	`}`
	`// PRE`
	`// true`
	`// POST`
	`// res == (rear == MaxQue)`
	`bool full() const`
	`{ bool res =`
	`      (rear == MaxQue);`
	`  return res;`
	`}`
	`};`

Проверка на коректността на класа относно зададената спецификация

Верификацията на *queue* лесно следва като се провери дали са в сила всички тройки на Хоар, съответстващи на член-функциите на класа. Например верификацията на член-функцията *pop* се осъществява като се докаже, че е в сила тройката на Хоар

```
{ class_inv && !empty() }
   x = a[front];
   int front_old = front;
   front = front+1;
{ class_inv && x == a[front_old] && front == front_old + 1 }
```

т.е.

```
0 <= front && front <= MaxQue && 0 <= rear && rear <= MaxQue &&
front <= rear && !empty() =>
Wp(x = a[front]; front_old = front; front = front+1,
     0 <= front && front <= MaxQue && 0 <= rear && rear <= MaxQue &&
     front <= rear && x == a[front_old] && front == front_old + 1)
```

В резултат от намирането на преобразуващия предикат се получава

```
Wp(x = a[front]; front_old = front; front = front+1,
     0 <= front && front <= MaxQue && 0 <= rear && rear <= MaxQue &&
     front <= rear && x == a[front_old] && front == front_old + 1) =

Wp(x = a[front]; front_old = front,
    0 <= front+1 && front+1 <= MaxQue && 0 <= rear && rear <= MaxQue &&
    front+1 <= rear && x == a[front_old] && front+1 == front_old + 1) =

Wp(x = a[front],
    0 <= front+1 && front+1 <= MaxQue && 0 <= rear && rear <= MaxQue &&
    front+1 <= rear && x == a[front] && front+1 == front+ 1) =
```

```
0 <= front+1 && front+1 <= MaxQue && 0 <= rear && rear <= MaxQue &&
front+1 <= rear && a[front] == a[front] && front+1 == front+1
```

а импликацията, представяща тройката на Хоар за член-функцията pop се свежда до

```
0 <= front && front <= MaxQue && 0 <= rear && rear <= MaxQue &&
front <= rear && !empty() =>
0 <= front+1 && front+1 <= MaxQue && 0 <= rear && rear <= MaxQue &&
front+1 <= rear && a[front] == a[front] && front+1 == front+1
```

Доказателството ѝ лесно следва. Неравенството `0 <= front+1` следва от `0 <= front`. От верността на `front <= rear` и `!empty()`, следва `front < rear`. Тъй като `front` и `rear` са естествени числа, следва `front+1 <= rear`. От тук и от `rear <= MaxQue` следва `front+1 <= MaxQue`. Тъй като опашката не е празна, може да се направят обръщенията `a[front]` и `a[front_old]`.

По подобен начин се доказва верността и на тройките на Хоар за останалите член-функции на класа. Доказателствата на тези твърдения могат да се направят ръчно, както и чрез някоя от системите за автоматично доказателство на теореми.

Проверка дали формалният ОМ модел на queue съответства на реализацията му

Трябва да се провери дали са в сила условията (3), (4) и (5). За позиции S_1, S_2, S_3 и S_4 и преходите t_1 и t_2 тези условия имат вида:

условия (3)

front >= 0 && front = = rear && rear < MaxQue => true
front >= 0 && front = = rear && rear < MaxQue => !full()

условие (4)

front >= 0 && front = = rear && rear < MaxQue =>
front >= 0 && front <= MaxQue && rear >= 0 &&
rear <= MaxQue && front <= rear

условия (5)

```
{ MaxQue >0 && Член-функцията е queue }
      front = 0; rear = 0;
{ front >= 0 && front = = rear && rear < MaxQue }

{ front >= 0 && front = = rear && rear < MaxQue &&
  Член-функцията е empty }
     bool res = (front == rear); return res;
{ front >= 0 && front = = rear && rear < MaxQue }

{ front >= 0 && front = = rear && rear < MaxQue &&
  Член-функцията е full }
      bool res = (rear == MaxQue);
      return res;
{ front >= 0 && front = = rear && rear < MaxQue }
```

{ front >= 0 && front = = rear && rear < MaxQue &&
Член-функцията е push &&
За данните на ядрото във входните позиции S_i (i = 1, 2, 3, 4)
е в сила rear+1 < MaxQue. }
a[rear] = x; rear_old = rear; rear = rear+1;
{ front >= 0 && front < rear && rear < MaxQue }

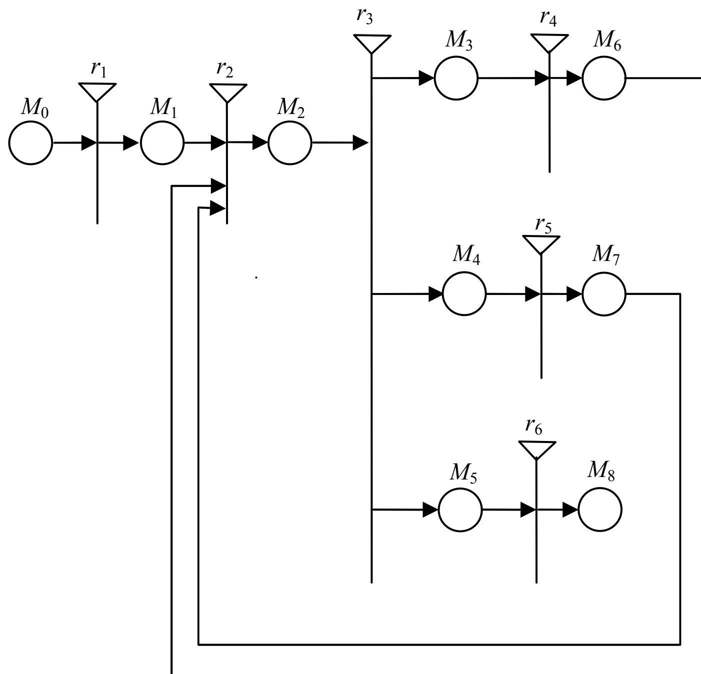
{ front >= 0 && front = = rear && rear < MaxQue &&
Член-функцията е push &&
За данните на ядрото във входните позиции S_i (i = 1, 2, 3, 4)
е в сила rear+1 = = MaxQue. }
a[rear] = x; rear_old = rear; rear = rear+1;
{ front >= 0 && front < rear && rear = = MaxQue }

За другите групи позиции и останалите преходи условията (3), (4) и (5) имат подобен вид. Доказателствата им са елементарни.

в) *Стъпка 3. Реализиране на функцията, която използва класа, построяване на обобщеномрежов модел за нея и проверка за съгласуваност на обобщеномрежовия модел на функцията с формалния ОМ модел на класа)*

```
int main()
{ queue q;
  char ch; int x;
  do
  { cout << "i -> insert\n"
            "r -> remove\n"
            "another symbol -> quit\n";
    cin >> ch;
    switch (ch)
    { case 'i': cout << "x= "; cin >> x;
                q.push(x);
                break;
      case 'r': q.pop(x);
                cout << x << endl;
    }
  } while(ch == 'i' || ch == 'r');
  return 0;
}
```

Обобщеномрежовият модел на *main* има вида от фиг. 3. Ще го означаваме с GN_2. За простота в него са пропуснати операторите за извеждане.



Фигура 3. Обобщен мрежов модел на функцията *main*

Следват дефинициите на преходите на тази ОМ.

$$r_1 = \langle \{M_0\}, \{M_1\}, r_1' \rangle$$

r_1' =

	M_1
M_0	*Член-функцията е queue.*

В резултат се активира ядро q с данна (*front*, *rear*, *a*) = (0, 0, *a*), която удовлетворява *front* >= 0 && *front* = = *rear* && *rear* < *MaxQue*.

$$r_2 = \langle \{M_1, M_6, M_7\}, \{M_2\}, r_2' \rangle$$

r_2' =

	M_2
M_1	*Операторът е cin >> ch.*
M_6	*Операторът е cin >> ch.*
M_7	*Операторът е cin >> ch.*

Характеристиките на ядрото не се променят.

$$r_3 = \langle \{M_2\}, \{M_3, M_4, M_5\}, r_3' \rangle$$

r_3' =

	M_3	M_4	M_5
M_2	*ch = = 'i'*	*ch = = 'r'*	*ch != 'i' && ch != 'r'*

$$r_4 = \langle \{M_3\}, \{M_6\}, r_4' \rangle$$

r_4' =

	M_6
M_3	*Операторът е cin >> ch. && Следващата го член-функция е push.*

$$r_5 = \langle \{M_4\}, \{M_7\}, r_5' \rangle$$

r_5' =

	M_7
M_4	*Член-функцията е pop.*

$$r_6 = \langle \{M_5\}, \{M_8\}, r_6' \rangle$$

r_6' =

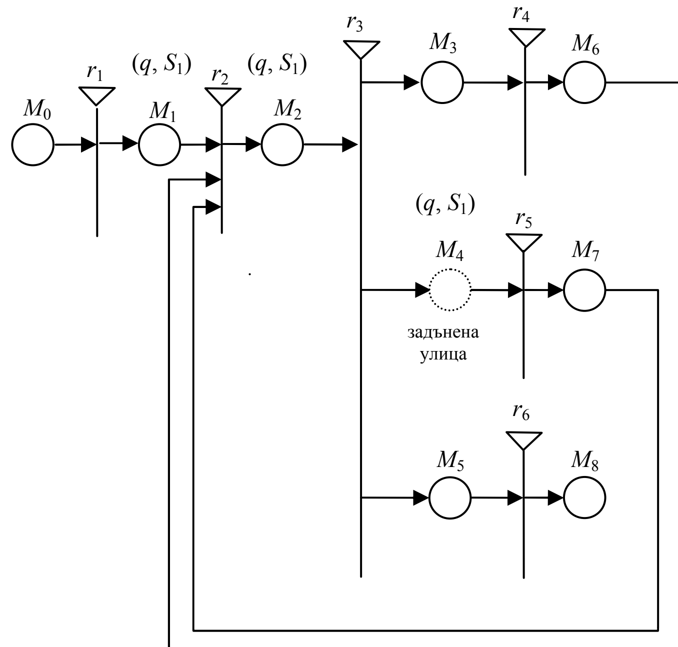
	M_8
M_5	*Член-функцията е неявният деструктор.*

Проверка за съгласуваност

Ще покажем, че направената реализация на функцията *main* не е коректна относно зададената чрез GN_1 спецификация. За целта е достатъчно да намерим един път на движение на ядро в модела GN_2, който не завършва нормално, а достига до неизпълним преход (задънена улица).

Ще изпълним двата обобщеномрежови модела – GN_2 и GN_1 и ще проверим дали изпълнението на GN_2 е съгласувано със спецификацията на класа, представена чрез GN_1. След изпълнението на преходите t_1 на GN_1 и r_1 на GN_2 в позициите S_1 и M_1 се активират едноименни ядра q с данни (*front*, *rear*, *a*) = (0, 0, *a*), удовлетворяващи $front == rear \; \&\& \; front < MaxQue$. Изпълнението на GN_2 продължава с изпълнението на прехода r_2, в резултат на което се въвежда стойност на символната променлива *ch*, ядрото q на GN_2 се премества в позиция M_2 и не променя данните си. Ядрото q на GN_1 остава в позиция S_1. Ще коментираме три изпълнения на прехода r_3 на GN_2, първите две от които показват несъгласуваност на GN_2 с GN_1.

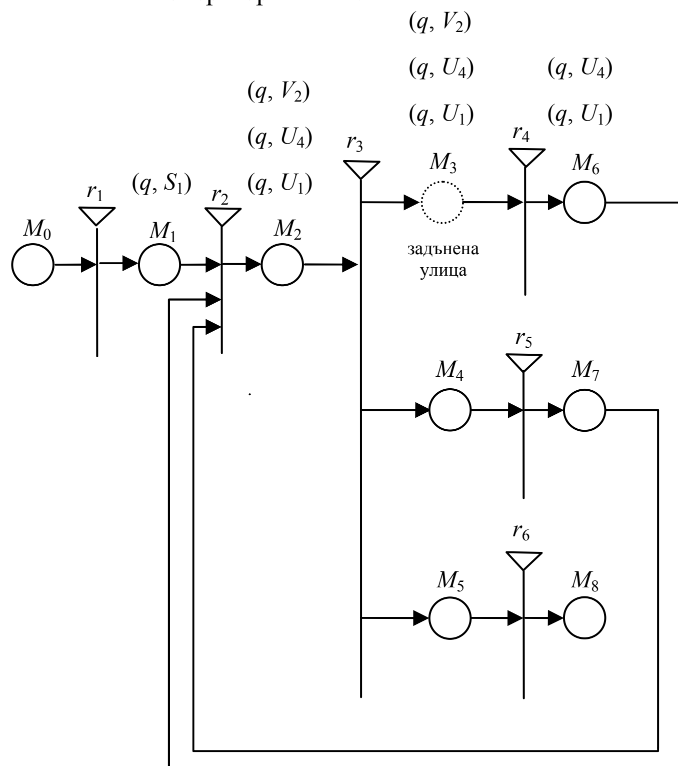
- Нека е въведен символът '*r*'. Тогава ядрото q в мрежата GN_2 попада в позиция M_4 и не променя данните си. Единствената възможност за продължение на движението на ядрото в GN_2 е да премине в позиция M_7, ако се изпълни преходът r_5, съответстващо на „Член-функцията е *pop*". Последното не е възможно, тъй като в GN_1 ядрото q е в позиция S_1, която е вход на прехода t_2, сред условията за изпълнение на който няма условие, което съдържа „Член-функцията е *pop*". Така се получава грешка (достига се до задънена улица) в позиция M_4, което в термините на програмата съответства на опит да се изключи елемент от празна опашка. На илюстрацията по-долу над позициите, през които е преминало ядрото q на GN_2 е записана характеристиката на едноименното му ядро в мрежата GN_1. Позицията M_4, в която възниква „задънената улица" показва, че грешката в main е в случая $ch == 'r'$ на оператора *switch*.



Фигура 4.

- Нека е въведен символът '*i*'. Тогава ядрото на GN_2 попада в позиция M_3 и не променя данните си. Единствената възможност за продължение на движението на това ядро е да премине в позиция M_6, ако се изпълни преходът r_4, съответстващ на условието “Член-функцията е *push*”. Последното е възможно, тъй като в GN_1 ядрото е в позиция S_1, която е входна за прехода t_2. Условието A_5 на t_2 е в сила едновременно с “Член-функцията е *push*”. След изпълнението на прехода r_4 се изпълнява и преходът t_2 на мрежата GN_1. Ядрото на GN_1 преминава в позиция U_1. Единствената възможност за продължение на движението на ядрото на GN_2 е да премине от позиция M_6 в позиция M_2, ако се изпълни преходът r_2. Последното не зависи от GN_1 и приемаме, че се е осъществило. Ако още *MaxQue*-1 пъти се въведе символът '*i*', след *MaxQue*-1 пъти изпълнение на преходите r_3, r_4 и r_2 на GN_2, ядрото на GN_2 преминава *MaxQue*-1 пъти през позициите M_3, M_6 и M_2 на GN_2. При това движение ядрото на GN_1 променя позицията си, като през първите *MaxQue*-2 движения преминава от позиция U_4 в позиция U_4, а при последното изпълнение на движението преминава от позиция U_4 в позиция V_2 (масивът, съдържащ опашката е запълнен с елементи). Опитът за изпълнение на следващо включване на елемент в опашката ще доведе до грешка, тъй като позиция V_2 на GN_1 е вход на прехода t_4, който не съдържа условие „Член-функцията е *push*”. Така се достига до грешка (до задънена улица) в позиция M_3, което в термините на програмата съответства на опит да се включи елемент в пълна опашка. На илюстрацията по-долу над позициите, през които е преминало ядрото q на GN_2 са записани характеристиките на ядрото q на мрежата GN_1 в редицата от изпълнения на

преходи. Позицията M_3, в която възниква „задънената улица” показва, че грешката в main е в случая $ch == 'i'$ на оператора *switch*.



Фигура 5.

- Нека е въведен символ, различен от '*i*' и '*r*'. Тогава ядрото на GN_2 попада в позиция M_5. Единствената следваща възможност на ядрото е през прехода r_6 да попадне в позиция M_8, ако е възможно да се изпълни преходът r_6. Това съответства на „изпълнение на неявния деструктор на класа *queue*”. Последното е възможно, тъй като ядрото на GN_1 се намира в позиция S_1, от която може да премине в позиция E_1 след като се изпълни преходът t_2 на GN_1.

След извършения анализ поправката на главната функция *main* може да се извърши лесно.

4. Заключение

В статията описахме метод за верификация на обектно-ориентирани програми, който интегрира концепцията „design by contract” с подходи за верификация от тип доказателство на теореми и проверка за съгласуваност на модели. Методът може да се

използва за построяване на коректни, относно указани спецификации, обектно-ориентирани програми. Експерименти за прилагане на метода чрез използване на средата GN Lite, с която могат да се изпълняват ОМ, не са извършвани заради това, че все още не са реализирани модулите, които извличат обобщени мрежови модели за функциите, подлежащи на верификация. Не са правени също и опити за обучение на студенти с цел овладяване на техники за формална верификация. Но ограничените средства за изразяване чрез обобщени мрежи в сравнение с езиците на формалната логика, лесният начин за работа и изграждане на модели чрез тях карат автора да смята, че предложеният метод е практически приложим и го мотивират за разработване на образователна среда за обучение на студенти и софтуерни инженери за използване на формални методи за построяване на коретно ПО.

Благодарности

Направените изследвания са подпомогнати от договор 182/2011 на ФНИ на СУ „Св. Кл. Охридски".

Литература

[1] Dong W. L, H. Yu, Y. B. Zhang. Testing BPEL-based web service composition using high-level Petri Nets. Proceedings – IEEE International Enterprise Distributed Object Computing Workshop, (2006).

[2] Pavlov, V., B. Borisov, S. Ilieva, D. Petrova-Antonova, Framework for Testing Service Compositions. 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing Timisoara, Romania, September, 2010.

[3] Ilieva, S., V. Pavlov, I. Manova. A Composable Framework for Test Automation of Service-Based Applications. 7th International Conference on the Quality of Information and Communications Technology (QUATIC), Porto, Portugal, 2010.

[4] http://mtc.epfl.ch/software-tools/blast/index-epfl.php (последно посетена на 30.08.2011).

[5] http://en.wikipedia.org/wiki/Construction_and_Analysis_of_Distributed_Processes (последно посетена на 28.08.2011).

[6] http://www.uppaal.com/ (последно посетена на 26.08.2011).

[7] http://nusmv.fbk.eu/ (последно посетена на 02.09.2011).

[8] http://embedded.eecs.berkeley.edu/research/hytech/ (последно посетена на 30.09.2011).

[9] Lindstrom, B., L. Wells, Design/CPN – Performance Tool Manual, 1999. http://www.daimi.au.dk/designCPN/man/Misc/Performance.pdf (последно посетена на 01.09.2011).

[10] Дончев, И., Э. Тодорова, Операции с обьектами и коммуникация между обьектами в обучении обьектно ориентированному программирования, The Sixth International Conference "INTERNET–EDUCATION–SCIENCE" Vinnytsia, Ukraine, October 7–11, 2008.

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

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

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

[14] Atanassov, K., On Generalized Nets Theory, Prof. Marin Drinov Academic Publishing House, Sofia, 2007.

[15] Atanassov, K., Generalized Nets, World Scientific, Singapore, 1991.

[16] Orozova D., V. Jecheva, Generalized Net Model of E-Learning System with Privacy Protection Module, Technological Developments in Education and Automation, M. Iskander et al. (eds.), Springer, 2010, 309–313.

[17] Жечев Т., Д. Орозова, Java базирана система за консултации, Годишник на Бургаски свободен университет, 2006, Бургас, том XIV, 185–189.

[18] Sotirova, E., P. Tcheshmedjiev, D. Orozova, Modelling the process of defining Java class using generalized net, Proceedings of the Jangjeon Mathematical Society, Vol. 9, No.2, December, 2006, pp. 201-215.

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

[20] Trifonov T., K. Georgiev, K. Atanassov, Software for modelling with Generalised Nets. Issues in Intuitionistic Fuzzy Sets and Generalized Nets, Vol. 6, 2008, 36–42.

[21] Gochev V., An implementation of generalized nets using object-oriented programming in .NET framework, Management and education, vol. VI (4) 2010, 227–231.

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

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

[24] Dimitrov, D.G., Software Products Implementing Generalized Nets, Annual of “Informatics”, Section Union of Scientists in Bulgaria, Vol. 3, 2010, 37–50.

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

[26] Meyer, B., Object-Oriented Software Construction, SECOND EDITION, ISE Inc. Santa Barbara, California, 1997.

[27] Meyer, B., Applying “design by contract”, Computer 25(10), 40-51, 1992.

[28] Hoare, C.A.R., Proof of Correctness of Data Representations, Acta Informatica, Vol. 1, N 4, 1972

[29] Gries, D., The Science of Programming, Springer-Verlag, Berlin and New York, 1981.