УДК 004.415.2.043

В.І. САЛАПАТОВ

ФОРМУВАННЯ ОПТИМАЛЬНОГО МАШИННОГО КОДУ ПРОГРАМ ПІД ЧАС ЙОГО СИНТЕЗУ

Анотація. *Запропоновано застосування формального опису синтаксису у вигляді недетермінованих кінцевих автоматів як на етапі синтаксичного аналізу програм, так і на етапі формування машинних кодів. На етапі синтаксичного аналізу для кожного стану автомата, що на відповідній діаграмі відображається своєю верхівкою, певна сукупність вхідних та вихідних дуг можуть виявитися помилковими та виділеними в окремий список. На етапі формування кодів машинних команд помилкові ситуації неможливі, оскільки асемблерні оператори формуються згідно з існуючими правилами опису форматів цих команд. Оптимальність машинного коду досягається за рахунок використання існуючих даних у регістрах та стеку цільового мікропроцесора.*
Ключові слова: *недетермінований кінцевий автомат, направлена діаграма станів, синтез коду.*

Аннотация. *Предложено применение формального описания синтаксиса в виде недетерминированных конечных автоматов как на этапе синтаксического анализа программ, так и на этапе формирования машинных кодов. На этапе синтаксического анализа для каждого состояния автомата, что на соответствующей диаграмме отображается своей вершиной, определенная совокупность входных и выходных дуг могут оказаться ошибочными и выделенными в отдельный список. На этапе формирования кодов машинных команд ошибочные ситуации невозможны, поскольку ассемблерные операторы формируются в соответствии с существующими правилами описания форматов этих команд. Оптимальность машинного кода достигается за счет использования существующих данных в регистрах и стеке целевого микропроцессора.*
Ключевые слова: *недетерминированный конечный автомат, направленная диаграмма состояний, синтез кода.*

Abstract. *The formal description of programming language syntax due to indeterminate finite automates which use on the syntaxes analyses phase as well as on the machine code forming phase was proposed. On the syntaxes analyses phase for every state of automate, which is reflected by its top on the corresponding diagram, a certain combination of the input and output connectors may be considered as syntax mistakes and marked out to a separate list. On the machine code forming phase the mistakes are impossible because assembling operators are formed in accordance with the existing rules of the formats instruction description of these commands. Machine code optimality is provided due to existing data on registers and stack of the target microprocessor.*
Keywords: *indeterminate finite automates, directed diagram of the states, code synthesis.*

1. Вступ

Машинно-незалежна оптимізація програм на даний час достатньо докладно й повно розвинута та широко застосовується в компіляторах. Натомість, машинно-залежна оптимізація коду програм формально не визначена і на даний час все ще залишається актуальною. Особливо це стосується вбудованих обчислювальних засобів, які є частиною спеціалізованих приладів. Надмірність машинного коду, що породжується компіляторами, виникає під час генерації машинного коду і викликана зайвими командами пересилок, збереження та відновлення даних [1]. Алгоритм будь-якої програми, яка обробляє вхідний потік даних, у тому числі компілятора, може бути поданий за допомогою кінцевого автомата [2]. Причому верхівки такого автомата можуть відображати не тільки певний його стан, а й певні дії, які можуть бути описані через основні програмні блоки – лінійні, розгалуження та цикли. Найбільший інтерес являють собою недетерміновані кінцеві автомати (НКА) [2], оскільки на відміну від детермінованих вони не містять додаткових штучних станів з пустими верхівками та додатковими переходами. Більшість мов програмування описується за допомо-

гою контекстно вільної граматики, тому для структурного подання синтаксичного розбору основні конструкції мови можуть бути представлені за допомогою діаграм – спеціальних направлених графів, які відображають перехід алгоритму розбору у черговий стан НКА в залежності від вхідних даних та поточного стану. Верхівки діаграми НКА, окрім фіксації їхнього стану, можуть охоплювати певні послідовності дій, у тому числі й розгалуження. Цикли, у свою чергу, реалізуються також за допомогою розгалуження та на діаграмі відображаються у вигляді зворотних зв'язків, тобто повернення НКА в один з попередніх станів. Направлені дуги діаграми, що зв'язують її верхівки, вказують на певні значення даних, за якими з даного стану НКА виконуються умови переходу.

2. Аналіз застосування НКА при синтаксичному розборі

Застосування НКА розглянемо на прикладі опису дійсної константи, що використовується в більшості мов програмування. Так, наприклад, опис і відповідна діаграма НКА алгоритму обробки дійсної константи в розширеній нотації Бекуса-Наура має такий вигляд. Під розширенням розуміється наявність квадратних дужок, які вказують на необов'язковість присутності елемента в конструкції.

<ДК>:= ε | [(+ | -)] [<Ч>] (ε |.) [<Ч>] [<Ех>]; Ц:=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;

<Ч>:=ε | (<Ч> | Ц); <Ех>:= (ε | Е | е) (ε | + | -) [<Ч>].

Позначка ДК означає дійсну константу, Ч – число, Ц – десяткову цифру, Ех – експоненціальну частину, ε – пусто. Подання Ч використовується у першому головному правилі тричі, являє собою самостійне поняття і тому може бути програмно реалізовано окремо у вигляді підпрограми та відповідних звернень до неї. Згідно з наведеними правилами щодо побудови діаграми НКА та допоміжних позначок (Ч, Е), отримуємо діаграму, яка представлена на рис. 1. Треба зазначити, що деякі сукупності переходів в НКА є нечіткими і можуть бути хибними через введення необов'язкових елементів у синтаксичні правила. Тому з метою виявлення хибних сполучень переходів на діаграмі такі сукупності переходів необхідно або перерахувати, або помітити певним чином.

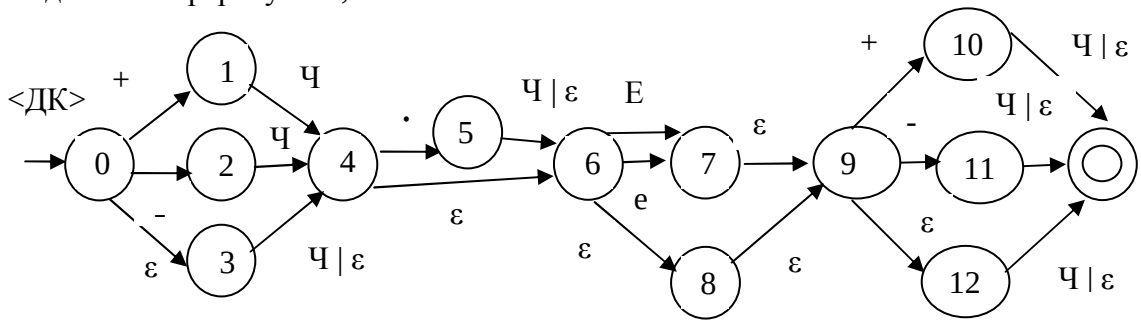

Рис. 1. Діаграма НКА для дійсної константи

Наявність десяткової точки на діаграмі фіксується станом 5 та програмно реалізується встановленням спеціального прапорця для наступного аналізу, оскільки відсутність цілої та дробової частин при наявності десяткової точки неприпустима. А переходи із стану 3 у стан 4 та із стану 5 у стан 6 помічені подвійними значеннями, не дивлячись на те, що ε охоплюється поняттям Ч. Таким чином, ситуація присутності десяткової точки без цілої та дробової частин повинна розпізнаватися як помилкова. Те ж саме стосується поняття Е, коли маємо перехід від стану 6 у стани 7 та 8 і далі у 9, 10, 11, 12 та перехід у кінцевий стан, коли Ч=ε. На діаграмі не розкрите поняття Ч, оскільки правила його опису містять рекурсію, що, у свою чергу, ускладнило б діаграму через появу зворотних зв'язків.

Тому доцільно та візуально зрозуміло для поняття Ч скласти окрему діаграму, яка подана на рис. 2 нижче.

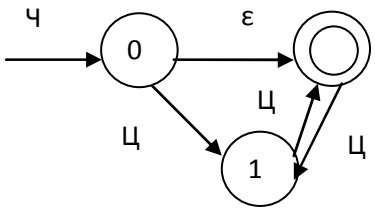

Рис. 2. Діаграма НКА для поняття Ч

Наявність цифр у понятті Ч фіксується станом 1, що, як і в попередньому випадку, також може бути програмно реалізовано встановленням спеціального прапорця та наступним його аналізом, оскільки відсутність цілої та дробової частин вимагає присутності експоненціальної частини. В обох випадках наявність або відсутність як десяткової крапки, так і числа взагалі, у синтаксичних правилах не відображені. Так само не відображені поява хибних вхідних даних та їх обробка. Тому більш детально такі випадки мають бути спеціальним чином помічені та враховані як виключення безпосередньо при програмній реалізації конкретної діаграми.

У прикладі з дійсним числом його було розбито на такі складові: знак, ціла частина, дробова частина та експонента. Подальша деталізація описана правилами, які припускають допоміжні поняття (у нашому випадку це Ч та Ех). Якщо ці поняття містять рекурсію, тобто являють собою нетермінали, то вони описуються як окремий НКА з відповідною діаграмою. Натомість нетермінал Ех описаний за допомогою поняття Ч і не використовує інших допоміжних понять, тому при подальшій деталізації не потребує окремої діаграми. Якщо нове поняття складається тільки з термінальних символів (в нашому прикладі Ц), то воно окремо не виділяється і присутнє в одній з основних діаграм. Основні діаграми опису дійсної константи представлені на рис. 1 та 2. Такий процес проектування програм за допомогою НКА дозволяє виділити та деталізувати окремі самостійні поняття (нетермінали), для яких створюються свої власні діаграми згідно з правилами їхнього опису. Кожне нове поняття, у свою чергу, також може містити самостійний нетермінал, що також містить рекурсію, тобто може мати місце вкладення нетерміналів одне в одне. Такий процес проектування програм зверху донизу за допомогою НКА в деякій мірі нагадує процес нормалізації БД і є структурно впорядкованим і зрозумілим для сприйняття.

3. Методика синтезу оптимального машинного коду програм

Задачею програмної реалізації НКА є неоднозначність багатьох виходів на діаграмі НКА з однієї верхівки та неоднозначність кількох входів в одну верхівку, які, крім стану, охоплюють певні дії. Розгалуження кількох дуг, що виходять з однієї верхівки, в середині цієї верхівки програмно можуть бути реалізовані за допомогою конструкції CASE. В разі входження кількох дуг в одну верхівку необхідно визначити точку входу алгоритму обробки для кожної такої дуги в цій верхівці. Наведений процес деталізації програми через діаграми НКА передбачає проектування програм зверху донизу. При програмній реалізації діаграми для кожної такої верхівки необхідно визначити алгоритм обробки у цьому стані і на рівні інструментальної мови описати цей алгоритм.

У сучасних компіляторах внутрішнє уявлення програм перед формуванням машинних кодів подається, як правило, у трьохадресному коді. Більш гнучким внутрішнім поданням є подання у вигляді самостійної бази даних внутрішнього подання компілятора [5], в якому внутрішнє уявлення програм представлено через інструкції, що за семантикою наближені до системи команд (ІСНК) цільового мікропроцесора (ЦМП), а посилання на операнди здійснюється за іменами змінних або за індексами відповідних масивів. Опис інструкції являє собою код семантики, а кількість операндів разом з їх типами та іншими атрибутами в операторі Асемблера не мають жорсткої прив'язки, як це має місце у трьохадресному коді. Синтаксичний аналіз кожної такої верхівки породжує внутрішнє подання програми, яке остаточно після машинно-незалежної оптимізації повинно перетворитись у ІСНК, який складається з абстрактних команд із семантикою, що максимально наближені

до семантик команд ЦМП. Зрозуміло, що кількість операндів та їх призначення в абстрактних командах співпадають за цими ж показниками у відповідних машинних командах ЦМП. Машинні команди ЦМП, опис яких також являє собою окрему самостійну БД, конкретизують, на відміну від абстрактних команд, формати машинних команд з їхніми видами адресації [3].

Для остаточного перетворення внутрішнього уявлення програми у машинний код через асемблерний еквівалент для кожної такої інструкції необхідно створити окремий недетермінований кінцевий підавтомат з урахуванням особливостей архітектури та системи команд ЦМП і наявних ІР. При цьому для кожної верхівки підавтомата необхідно створити ефективні діаграми НКА на рівні машинного коду ЦМП. Такі діаграми, в першу чергу, мають контролювати наявність ІР згідно з існуючими форматами, та з усіх можливих шляхів реалізації абстрактної команди обирати той, що максимально відповідає обраному критерію оптимальності [4].

Процес формування асемблерного еквіваленту команди ІСНК з урахуванням наявних ІР охоплює формування допоміжних команд, які мають підготувати операнди згідно з існуючими форматами машинних команд, поданий на рис. 3. Цей процес фактично являє собою синтез асемблерних машинних команд та відповідного машинного коду з таких складових, як формати команд з їхніми видами адресації та наявної на даний момент компіляції/виконання програми ІР.

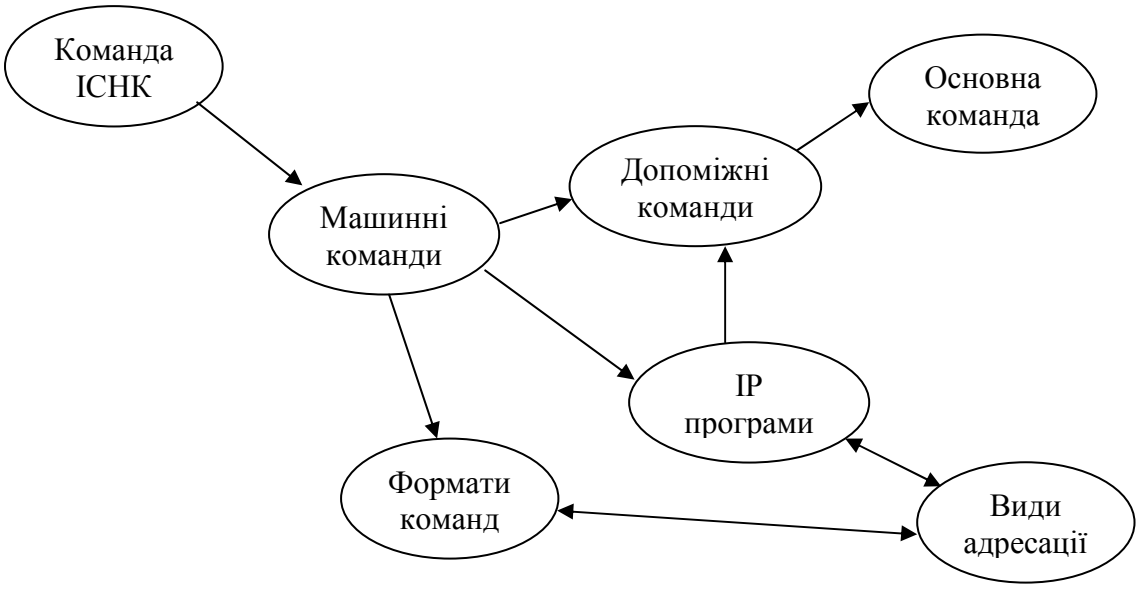

Рис. 3. Узагальнений процес синтезу машинного коду

Двосторонні стрілки на діаграмі означають перегляд для ІР видів адресації операнда, а для форматів команд – наявність адресації ІР в одному з форматів конкретної машинної команди. Допоміжні машинні команди формують операнди для основної команди із семантикою ІСНК і разом з основною командою програмно реалізують команду ІСНК і на останній фазі – синтезі оптимального машинного коду, мають перетворитися в остаточний код програми із формуванням об’єктного файла. Множина верхівок (ІР програми) цього підграфа разом з форматами команд та видами адресацій визначають поточний стан програми для ЦМП, який залежить від наявності ІР. Стани цього автомата повинні визначатися вмістом програмно доступних регістрів (ПДР) ЦМП, а також станом та вмістом стеку.

Стосовно нашої задачі щодо побудови якісного компілятора, то для кожної мовної конструкції необхідно мати подібні діаграми. Далі, з усіх таких діаграм необхідно виділити спільні нетермінали і в кожному конкретному випадку перелічити типові помилкові си-

туації, які не висвітлені у діаграмах НКА. Наприклад, в умовній конструкції IF <вираз> THEN ... значення виразу має відображати логічний тип (ТАК або НІ), а в операторах присвоєння значення виразу повинно відповідати типу змінної, для якої це значення надається. Поняття <вираз> є одним з основних нетерміналів, який використовується в усіх мовах програмування та досить докладно описаний у різних джерелах, зокрема, в [2]. Для програмної реалізації виразу доцільно застосувати модифікований стековий алгоритм [3]. Нетермінали, які є спільними для кількох конструкцій або зустрічаються досить часто, на практиці реалізуються у вигляді підпрограм. Це, в першу чергу, відноситься до лексичного аналізатора, який виконує пошук лексем та обробку типових її елементів (ідентифікатори, ключові слова, числові, символьні та логічні константи).

Оскільки вміст ПДР ЦМП у процесі компіляції не може бути заздалегідь визначеним і змінюється у процесі як компіляції, так і виконання програми, то кількість станів таких підавтоматів у загальному випадку є достатньо великою. Але в кожному конкретному випадку для формування кодів машинних команд програм потрібні певні дані в ПДР та стеку. Тобто, йдеться про направлений пошук необхідних даних, які повинні використовуватись як операнди команд при синтезі машинних кодів програм. Більшість сучасних компіляторів використовують символічну мову машинних команд ЦМП – Асемблер як проміжну, тому при синтезі кодів програм також використовується формування асемблерного еквівалента програми та наступна її трансляція. Синтез машинного коду програм складається з формування окремих операторів Асемблера, головним чином операторів машинних команд та їх складових. Класичні компілятори готують операнди командами пересилок, що не завжди є доцільним, оскільки ці операнди можуть знаходитись у ПДР від попередніх формувань і визначають стан підавтомату конкретної верхівки та найкоротший шлях до кінцевої верхівки підавтомата. Якщо для схожих фрагментів програм вводити нові проміжні поняття або нетермінали, то в подальшому вони можуть бути оформлені у вигляді підпрограм. У нашому випадку такою конструкцією є число – Ч. Доцільність застосування підпрограм полягає у тому, щоб витрати на програмну реалізацію підпрограм були меншими, ніж звичайний повтор змістовних частин цих підпрограм. Стосовно розміру пам’яті необхідно, щоб $Pi + Si + n*Qi < n*S$, де і Pi – програмні витрати на оформлення підпрограми, n – кількість повторів фрагмента у програмі або викликів, Qi – програмні витрати на звернення до підпрограм, і S – програмні витрати змістовної частини підпрограм.

Опис форматів для кожної машинної команди являє собою обмежену множину припустимих сполучень видів адресацій до операндів, які використовуються у цій команді. Опис таких форматів складає спеціальну базу даних (БД), яка формується на етапі опису системи команд та особливостей архітектури цільового процесора. Набір форматів для кожної команди розпізнається за ідентифікатором цієї команди у загальній БД опису форматів команд [4]. На етапі синтезу асемблерних команд внутрішнє уявлення програми фактично являє собою низку машинних операторів із семантиками команд ЦМП та вказівками на операнди. Спочатку виконується пошук потрібного інформаційного ресурсу (ІР) з існуючими способами адресації до нього, потім обираються ті види адресацій, що відповідають поточній команді із заданою семантикою і з цієї множини обираються ті, що найкраще відповідають оптимальному формату згідно з обраним критерієм [4]. Операція вибірки потрібних форматів з ІР повинна здійснюватись за допомогою реляційної операції пошуку за зразком. Для кожної команди таким зразком або ознакою є її внутрішній ідентифікатор, який фактично визначає код семантики команди і виконує роль первинного ключа, а також код виду адресації згідно з набором форматів команди з обраною семантикою. Формати вказують на спосіб доступу до операндів в машинній команді, тобто, на вид адресації, який пов’язаний з певним набором ПДР. Треба зазначити, що певні групи команд мають, як правило, однакові набори форматів, наприклад, арифметико-логічні команди, команди зсуву, пересилок тощо. Тому тип групи команд з однаковими форматами може використо-

вуватись як слабкий зв'язок [3, 5] між відношеннями у БД опису системи команд та особливостей архітектури цільового мікропроцесора.

Під час синтезу кодів команд у ПДР залишаються дані від попередньо синтезованих команд, які у кожному конкретному місці виконання програми можуть бути використані для доступу до потрібних операндів при формуванні нових кодів машинних команд. Таким чином, замість безумовної підготовки операндів у стандартних ПДР, як це зараз здійснюється в сучасних компіляторах під час генерації машинного коду, виконується пошук існуючих даних, і в разі їх наявності знайдений операнд використовується при синтезі поточних команд програми. Традиційна підготовка операндів у ПДР виконується лише в разі відсутності в них потрібних операндів. Якщо більша частина підготовчих команд, що складає від 60 % машинного коду, виявиться непотрібною через наявність операндів від попередньо синтезованих машинних кодів, то синтез кодів програм з використанням згаданих інформаційних ресурсів ІР дасть можливість, як мінімум вдвічі, скоротити коди програм і тим самим значно підвищити продуктивність роботи процесора.

Загальна БД компілятора, який синтезує машинні коди за обраною методикою, складається з таких трьох незалежних взаємопов'язаних БД: БД опису системи команд та особливостей архітектури цільового мікропроцесора; БД ІР, яка постійно змінюється під час виконання програми, та БД внутрішнього уявлення програми. При компіляції процес зміни ІР відтворюється на фазі синтезу кодів команд з урахуванням лінійних ділянок програм, циклів та розгалужень [6]. Під час синтезу асемблерного еквівалента програми ПДР фактично використовуються як кеш. Тому для його обслуговування слід застосовувати відповідні дисципліни обслуговування кешу: додавання, витіснення та заміщення. Перші дві дисципліни не викликають труднощів, оскільки пов'язані із використанням вільних на конкретний момент часу ПДР при додаванні та простому використанні тих ПДР, які у подальшому не знадобляться при синтезі в разі витіснення. Заміщення передбачає збереження певних ПДР у стеку або пам'яті та наступне відновлення в разі потреби. Графічно цей процес показаний на рис. 4.

Рис. 4. Структура процесу витіснення ПДР

Позначки Ri , Rj, Rk … вказують на ті ПДР, які підлягають заміщенню. Оптимізація у процесі синтезу машинного коду полягає у зведенні до мінімуму блоків збереження та наступному відновленні вмісту ПДР. Треба зазначити, що надмірність машинного коду, яка породжується компіляторами, пов'язана саме через генерацію непотрібних команд, які реалізують блоки збереження та відновлення ПДР, а також зайвих команд завантаження даних у стандартні ПДР як операнди для породження машинних команд, що реалізують потрібну мовну конструкцію. Збереження та відновлення даних стандартно виконується через стек. Тому у верхівці стеку мають бути розміщені (збережені) ті дані, які повинні бути використані одразу ж після реалізації поточного програмного фрагмента. Стандартне використання конкретних ПДР у класичних компіляторах також є однією з причин надмірності машинного коду, тому що потрібна інформація для синтезу поточних кодів команд може вже знаходитись від синтезу попередніх команд в інших ПДР з урахуванням того, що формат потрібних команд дозволяє використання цих ПДР.

4. Висновки

Створений за описаною методикою компілятор мови С для сигнального мікропроцесора ADSP-2188 підтвердив правильність основних її положень щодо синтезу оптимального машинного коду програм. Отже, оптимальність машинного коду залежить від обраного критерію оптимальності та відповідної цільової функції. Для ADSP-2188 оптимальність вироджується до мінімуму пам'яті під програмний код через стандартну довжину машинних команд та однаковий час їх виконання. У порівнянні із стандартним фірмовим компілятором мови С розмір програм зменшується як мінімум удвічі завдяки багаторазовому використанню IP та усуненню непотрібних команд пересилок, збережень і відновлень даних. Компілятор фактично моделює дії кваліфікованого програміста, який створює програми за допомогою мови Асемблер для ЦМП. Таким чином, цілком доцільно останню фазу роботи компілятора – генерацію машинного коду – замінити на фазу синтезу машинного коду, яка усуває надмірність машинного коду програм. Але програмна реалізація компілятора за такою методикою підвищує його складність.

СПИСОК ЛІТЕРАТУРИ

1. Салапатов В.І. Правила синтеза кодов программ / В.І. Салапатов // Вісник Черкаського інженерно-технологичного університету. – 2001. – № 1. – С. 108 – 111.
2. Ахо А. Компиляторы: принципы, технологии и инструменты / Ахо А., Сети Р., Ульман Дж.; пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 768 с.
3. Салапатов В.І. Подання даних для синтезу коду у нормалізованому вигляді / В.І. Салапатов // Вісник Черкаського інженерно-технологічного університету. – 2001. – № 3. – С. 96 – 99.
4. Салапатов В.І. Структура системи синтезу оптимальних машинних кодів програм / В.І. Салапатов // Вісник Черкаського інженерно-технологічного університету. – 2004. – № 4. – С. 128 – 132.
5. Дейт К.Дж. Введение в системы баз данных / Дейт К.Дж.; пер. с англ. – М.: Издательский дом «Вильямс», 2001. – 1072 с.
6. Салапатов В.І. Особливості формування інформаційних ресурсів при розгалуженні програм / В.І. Салапатов // Вісник Херсонського національного технічного університету. – 2006. – № 3 (26). – С. 141 – 144.

Стаття надійшла до редакції 14.04.2011