Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Кваліфікаційна наукова
праця на правах рукопису

Зилевіч Максим Олегович

УДК 621.3

ДИСЕРТАЦІЯ

Композиційні моделі телекомунікаційних систем в суб'єкто-об'єктному середовищі програмування

17 Електроніка та телекомунікації
172 Телекомунікації та радіотехніка

Подається на здобуття наукового ступеня доктора філософії

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____/Зилевіч М.О.

Науковий керівник: Редько Ігор Володимирович, доктор фізико-математичних наук, професор

Київ–2023

АНОТАЦІЯ

Зилевіч М.О. Композиційні моделі телекомунікаційних систем в суб'єкто-об'єктному середовищі програмування. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 172 “Телекомунікації та радіотехніка”. – Національний технічний університет України “Київський політехнічний інститут імені Ігоря Сікорського”, Київ, 2023.

Дисертаційна робота присвячена вирішенню важливої та актуальної науково-прикладної задачі – технологізації процесів вирішення сучасних задач в людино-машинних, зокрема, телекомунікаційних системах, методом композитологічного уподібнення – логічного ядра суб’єкто-об’єктного середовища програмування (СОСрП).

Дисертаційне дослідження складається зі вступу і чотирьох розділів, які відображають та обґрунтовують основні результати роботи.

У вступі обґрунтовано актуальність дисертаційної роботи, висвітлено зв’язок роботи з науковими програмами, планами та темами НДР КПІ ім. Ігоря Сікорського. Сформульовано мету та вказані задачі, вирішення яких передбачає досягнення мети дослідження. Визначено об’єкт, предмет та методи дослідження, надано інформацію про наукову новизну та практичне значення отриманих результатів. Наведено інформацію про висвітлення результатів роботи в періодичних наукових виданнях та їх апробацію на наукових конференціях.

У першому розділі розглянуто ряд важливих питань, пов'язаних з розробкою програмно-апаратних комплексів у галузі телекомунікацій і його якістю. Головна ідея полягає в тому, що розвиток інформаційних технологій підвищує вимоги до якості та ефективності програм, але сучасний підхід до програмування, не завжди відповідає вимогам сьогодення і більше спирається на інтуїцію та досвід програмістів. Тому, актуальним є перегляд підходів до програмування і питання того, як можна базувати програми на технологічних принципах і теорії інформації. Досліджено реально-

номінальні розбіжності, які виникають між очікуваною роботою програми і тим, як вона фактично функціонує. Акцентовано увагу на важливості реального врахування причино-наслідкового зв’язку між програмуванням та програмою та значущості цього зв'язку для ефективної розробки програмно-апаратних комплексів. Показано особливості та передумови концептомонадної парадигми як інтерсуб’єктивної основи СОСрП. Описано відповідну понятійну систему, що, зокрема, включає такі поняття, як суть, сутність, монада, концепт, композит, оракул тощо. Показано на репрезентативних прикладах оракульне та редукційне концептування, а також конкретний приклад застосування оракульного концептування при розробці та нотації програмного рішення у мові програмування Verilog. Наголошено на питаннях адекватності суб'єкто-об'єктної парадигми та розглядається можливість типізації сутностей у суб'єкто-об'єктних системах для поліпшення їх ефективності та точності.

У другому розділі проведено виклад основних концепцій, принципів та понять композиційного програмування, що утворюють основу теоретичних досліджень та практичних розробок у галузі універсальних та спеціалізованих мов програмування та мовних процесорів. Спільність концепцій композиційного підходу та засобів специфікації мов програмування та мов, що використовуються для схематичного опису інтегральних мікросхем дозволяє поєднати два найважливіших напрями інформатики: мови програмування та схемотехнічного дизайну, адже схемотехнічна розробка близька до програмування, а схемотехнічні рішення – до програм. Досліджено основні змістовні влативості програм. Проаналізовано три основні аспекти програм – прагматика, семантика та синтаксис у їх взаємодоповненні, засадничі принципи, що визначають цю взаємодію та проведене їх прагматико-обумовлене збагачення на область схемотехнічних рішень. Суть описаних аспектів показано на репрезентативних прикладах. Описано основні властивості композицій програм, серед яких особливу увагу приділено адекватності та обчислюваності. Проаналізовано принципи обумовленості, підпорядкованості та віддільності, що визначають

взаємодоповнюваність основних аспектів рішень, визначають три основні етапи конструювання програми: аналіз прагматичних вимог, семантичне конструювання програми, синтаксичне оформлення програми. Опрацьовано програмні дефінітори, що визначають зв’язок між семантикою та синтаксисом мови програмування і як наслідок надають можливість створення інтерпретованої мови програмування. Здійснено принципове збагачення програмного дефінітора як замкненої у конкретній мові програмування системи до програмного дескриптора як відкрито-замкненого середовища існування програмних дефініторів. На репрезентативному прикладі продемонстровано використання концептів програмування у вигляді семантичних шаблонів як ланок програмного ланцюга, які обумовлюють певні класи програм. Використано програмний дескриптор, який виступає у ролі засобу трансляції композитів та базових функцій системи програмування у їх синтаксичні представлення. За допомогою редукційного програмування у заданій системі була отримана програмна специфікація, коректність якої випливає з її побудови. На основі отриманої специфікації за допомогою дескриптора отримано код програми. Визначено композиційні основи СОСрП та обґрунтовано прагматичне положення про концептологічну парадигму програмування. Описано основні парадигми, що характерні СОСрП.

У третьому розділі розглянуто композитосутності основи СОСрП як концептуальноєдиної інтеграційної платформи програмування. Продемонстровано на ряді репрезентативних прикладів та в цілому обгрунтовано, що дане середовище, успадковуючи позитивні сторони традиційних підходів, також суттєво розвиває їх, зокрема, у напрямку врахування взаємодоповнюючої (причино-наслідкової) природи зв’язку вирішення задачі та її рішення, програмування та програми. Такий підхід реально, а не тільки номінально, підтримує взаємодоповнення основних аспектів програмування, забезпечуючи реальну продуктивність отримуваних результатів. Розкрито значення основних загальних властивостей композицій – тотальності, адекватності та замкненості. Ці властивості допомагають обґрунтувати прагматичну

обумовленість та відносність виокремлення композицій як засобів проєктування серед різноманіття алгебраїчних операцій, а також конкретизують важливі взаємозв'язки між ключовими учасниками проєктних рішень – його розробниками та тими, на кого вони орієнтовані. Досліджено логіко-предметні передумови суб'єкто-об'єктної системи програмування (СП) як композитної конкретизації СОСрП. Запропоновано ряд редукційних схем як предметно-орієнтованих шаблонів програмування. Останні предметно розвивають метод редукцій. На репрезентативних прикладах розкрито редукційні аспекти програмної релятивізації, показано приклад логіко-математичної релятивізації рішень задач в побудованій СП. Показано, що композито-композиційна релятивізація рішень задач реально підтримує семантико-синтаксичну підпорядкованість вирішення задач на відміну від синтаксисо-семантичних підходів, що розглядають семантику рішення (програму) виключно через інтерпретацію його коду (тексту у мові програмування). Розглянуті приклади вирішення задач у суб'єкто-об'єктній системі демонструють важливі загальні особливості редукційного концептування оракульних схем.

У четвертому розділі описано основні методи розробки суб'єкто-об'єктних середовиш програмно-апартного проєктування. Визначено, що описані підходи не забезпечують системності. Жоден з них не охоплює процесу від початкового задуму до фактичної реалізації. Вони виступають лише як інструменти організації, але не надають конкретних настанов для організації всього процесу проєктування. Описано основні положення мови Verilog. Розроблено дослідну реалізацію СОСрП, що підтримує розробку програмно-апаратного забезпечення. Користувач може задати дескрипцію вирішення задачі у мові специфікації композицій із подальшою генерацією коду на мові програмування Verilog. Можлива підтримка створення апаратного забезпечення із залученням FPGA як базису апаратної платформи із використанням САПР “Quartus”, що значно спрощує процес розробки програмно-апратного комплексу, отриманого за допомогою суб'єкто-об'єктного середовища програмування.

У дисертаційній роботі отримано наступні нові наукові результати дослідження:

1. Набуло подальшого розвитку предметне збагачення концепто-монадної парадигми програмування видом телекомунікаційних систем програмування та запропоновано відповідну понятійну систему телеконцептування. Зміст їх розкрито у концептомонадному середовищі через оракули «обумовлення», «концепт», «монада», «сутність», «суть». Це забезпечує можливість реальної інтеграції наявних підходів проєктування програмно-апаратних комплексів телекомунікаційних систем у вигляді взаємодоповнення процесів програмування та їх результатів, що складатиме основу реального розуміння програмування і дозволить відійти від сучасного інтуїтивного базису, якісно його розвинувши за допомогою сучасних досліджень та розробок.

2. Вперше розкрито прагматичну обумовленість зведення до телеконцептограм генетичних структур програм. Це дозволяє реально, а не лише номінально підтримувати причинно-наслідкові зв’язки при вирішенні задач, а також способи, методи та засоби їх специфікації. В якості телеконцептограм розглянуто телекомпозити – спеціальні класи суб’єктоорієнтованих базових телекомпозицій. Таким чином, телеконцептування на предметному рівні зводиться до вирішення відповідних рівнянь телекомпозитних редукцій, шо забезпечує коректність отримуваних рішень "за побудовою".

3. Подальшого розвитку набуло застосування підходу оракульного телеконцептування для предметного збагачення СОСрП. На репрезентативних прикладах показані його особливості та перспективи подальшого розвитку. До особливостей відноситься те, що кожна підзадача може бути проконцептована до найпростішої під задачі. Також використання оракульного телеконцептування дає можливість використання традиційного математичного апарату для нотації результату та поєднання його з денотативними методами. Реалізація такого методу на практиці сприяє уніфікації процесу розробки програмно-апаратного продукту, тим самим оптимізує та реально об’єктивізує вплив активної ролі суб’єкта у телеконцептуванні

через механізм оракульних телекомунікацій як технологію телекомунікаційних рішень задач.

4. Вперше запропоновано основні логіко-предметні засади суб'єкто-об'єктної телекомунікаційної системи програмування як предметного замикання СОСрП. Головною особливістю створюваних таким чином систем програмування є те, що вони реально, а не лише номінально підтримують причинно-наслідкове взаємодоповнення двох складових вирішення будь-якої програмістської задачі – програмування як породження та застосування композицій і програми – наслідку програмування.

Отримано наступні практичні та теоретичні результати досліджень:

1. Розроблена дослідна реалізація СОСрП, що підтримує розробку програмного забезпечення як предметного замикання відповідного середовища. Прагматикообумовлені умови такого замикання задаються у дескриптивному середовищі композиційних термів. Синтаксичне оформлення рішення здійснюється Verilog-дескриптором. Можлива підтримка створення апаратного забезпечення із залученням FPGA як базису апаратної платформи із використанням САПР "Quartus".

2. Одержані в дисертації нові результати використані під час виконання науково-дослідної роботи "Композитологічні засади технологічних систем програмування" (№0122U001568), та як матеріали при підготовці та викладанні курсу лекційних і практичних занять з дисципліни "Системне програмування та керування базами даних в телекомунікаціях" другого (магістерського) рівня вищої освіти спеціальності 172 «Електронні комунікації та радіотехніка» освітньо-професійної програми «Інформаційно-обчислювальні засоби радіоелектронних систем», що підтверджено відповідною довідкою та актом.

Ключові слова: програмування, телекомунікації, Verilog, композиція, середовище програмування, редукція, об'єкт, інформаційно - комунікаційні мережі, програмно-апартний комплекс, отології, автоматизація, комп'ютерне моделювання, універсальні та рекурсивні функції, агентне середовище, алгоритм.

ABSTRACT

Maksym Zylevich. Composite models of telecommunication systems in the subject-object programming environment. – Qualifying scientific work on the rights of the manuscript.

Dissertation for obtaining the scientific degree of Doctor of Philosophy in specialty 172 "Telecommunications and radio engineering". - National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, 2023.

The dissertation is devoted to the solution of an important and relevant scientific and applied problem - the technologicalization of the processes of solving modern problems in human-machine, in particular, telecommunication systems, by the method of compositological comparison - the logical core of the subject-object programming environment.

The dissertation study consists of an introduction and four chapters that reflect and justify the main results of the work.

The introduction substantiates the relevance of the dissertation work. The goal and specified tasks, the solution of which involves the achievement of the research goal, are formulated. The object, subject, and methods of research are defined, and information is provided about the scientific novelty and practical significance of the obtained results. Information is provided on the coverage of work results in periodical scientific publications and their approval at scientific conferences.

In the first chapter, some important issues related to the development of software and hardware complexes in the field of telecommunications and its quality are considered. The main idea is that the development of information technologies increases the requirements for the quality and efficiency of programs, but the modern approach to programming does not always meet the requirements of today and relies more on the intuition and experience of programmers. Therefore, it is relevant to review approaches to programming and the question of how programs can be based on technological principles and information theory. The real-nominal discrepancies that arise between the expected work of the program and the way it actually functions have been studied. Attention is focused on the importance of real

consideration of the cause-and-effect relationship between programming and the program and the significance of this relationship for the effective development of hardware and software complexes. The peculiarities and prerequisites of the conceptomonad paradigm as an intersubjective basis of the proposed environment are shown. The corresponding conceptual system is described, which, in particular, includes such concepts as essence, essence, monad, concept, composite, oracle, etc. The oracle and reductive conceptualization, as well as a specific example of the application of the oracle conceptualization in the development and notation of a software solution in the Verilog programming language, are shown in representative examples. Issues of adequacy of the subject-object paradigm are emphasized and the possibility of entity typification in subject-object systems to improve their efficiency and accuracy is considered.

In the second chapter, the main concepts, principles, and notions of compositional programming, which form the basis of theoretical research and practical developments in the field of universal and specialized programming languages and language processors, are presented. The commonality of the concepts of the compositional approach and the means of specification of programming languages and languages used for the schematic description of integrated circuits allows us to combine the two most important areas of computer science: programming languages and circuit design, because circuit design is close to programming, and circuit solutions are to programs. The main substantive aspects of the programs have been studied. The three main aspects of the programs are analyzed - pragmatics, semantics, and syntax in their complementarity, the basic principles that determine this interaction, and their pragmatically conditioned enrichment in the field of circuit and technical solutions. The essence of the described aspects is shown in representative examples. The main properties of program compositions are described, among which special attention is paid to adequacy and computability. The principles of conditionality, subordination, and separability, which determine the complementarity of the main aspects of decisions, determine the three main stages of program construction: analysis of pragmatic requirements, semantic construction of

the program, and syntactic design of the program. Program definers have been developed that determine the relationship between the semantics and syntax of the programming language and, as a result, provide the opportunity to create an interpreted programming language. The fundamental enrichment of the software definer as a system closed in a specific programming language to the software descriptor as an open-closed environment of the software definers was carried out. A representative example demonstrates the use of programming concepts in the form of semantic templates as links in a program chain that condition certain classes of programs. A program descriptor is used, which acts as a means of translating composites and basic functions of the programming system into their syntactic representations. With the help of reductive programming, a program specification was obtained in the given system, the correctness of which follows from its construction. Based on the received specification, the program code is obtained using the descriptor. The compositional foundations of the proposed environment are determined and the pragmatic position on the conceptual paradigm of programming is substantiated. The main paradigms characteristic of the proposed environment are described.

In the third chapter, the compositional and essential foundations of the proposed environment as a conceptually single integration platform of programming are considered. It is demonstrated in some representative examples and generally substantiated that this environment, inheriting the positive aspects of traditional approaches, also significantly develops them, in particular, in the direction of taking into account the complementary (cause-and-effect) nature of the connection between problem-solving and its solution, programming, and programs. This approach actually, and not only nominally, supports the complementarity of the main aspects of programming, ensuring the real productivity of the obtained results. The meaning of the main general properties of compositions - totality, adequacy, and closure - is revealed. These properties help to substantiate the pragmatic conditionality and relativity of singling out compositions as means of design among the variety of algebraic operations and also specify the important relationships between the key participants of design decisions - their

developers and those on whom they are oriented. The logical and subject prerequisites of the subject-object programming system (PS) as a composite concretization of the proposed environment have been studied. Several reduction schemes are proposed as subject-oriented programming templates. The latter objectively develops the method of reduction. The reductive aspects of software relativization are revealed in representative examples, and an example of logical-mathematical relativization of problem solutions in the constructed PS is shown. It is shown that the composite-compositional relativization of problem solutions really supports the semantic-syntactic subordination of problem-solving, in contrast to syntactic-semantic approaches that consider the semantics of the solution (program) exclusively through the interpretation of its code (text in the programming language). The considered examples of solving problems in the subject-object system demonstrate important general features of the reductive conceptualization of oracle schemes.

The fourth chapter describes the main methods of developing subject-object environments of software-apartment design. It was determined that the described approaches do not ensure systematicity. None of them cover the process from initial conception to actual implementation. They act only as organization tools but do not provide specific instructions for the organization of the entire design process. The main provisions of the Verilog language are described. An experimental implementation of SOSrP has been developed, which supports the development of software and hardware. The user can specify the description of the solution to the problem in the composition specification language, followed by code generation in the Verilog programming language. It is possible to support the creation of hardware with the involvement of FPGA as the basis of the hardware platform using CAD "Quartus", which greatly simplifies the process of developing a software-hardware complex obtained using the subject-object programming environment.

The following new scientific research results were obtained in the dissertation work:

1. The substantive enrichment of the concept-monad paradigm of programming by the type of telecommunication programming systems has gained further development, and a

corresponding conceptual system of teleconceptualization has been proposed. Their content is revealed in the concept-monad environment through the oracles "conditioning", "concept", "monad", "essence", and "entity". This provides the possibility of real integration of existing approaches to designing software and hardware complexes of telecommunication systems in the form of mutual complementation of programming processes and their results, which will form the basis of a real understanding of programming and allow moving away from the modern intuitive basis, qualitatively developing it with the help of modern research and development.

2. For the first time, the pragmatic conditionality of the reduction to teleconceptograms of the genetic structures of programs is revealed. This allows you to actually, and not just nominally, support cause-and-effect relationships when solving problems, as well as ways, methods, and means of their specification. Telecomposites are considered teleconceptograms - special classes of subject-oriented basic telecompositions. Thus, teleconceptualization at the subject level is reduced to solving the corresponding equations of telecomposite reductions, which ensures the correctness of the received solutions "by construction".

3. The use of the oracular teleconceptualization approach for subject-specific enrichment of the proposed environment gained further development. Representative examples show its features and prospects for further development. A special feature is that each subtask can be conceptualized into the simplest subtask. Also, the use of oracular teleconceptualization makes it possible to use the traditional mathematical apparatus for the notation of the result and to combine it with denotative methods. The implementation of such a method in practice contributes to the unification of the software and hardware product development process, thereby optimizing and realistically objectifying the influence of the active role of the subject in teleconceptualization through the mechanism of oracle telecommunications as a technology of telecommunication solutions to problems.

4. For the first time, the main logical and subject principles of the subject-object telecommunication system of programming as a subject closure of the proposed

environmentthe proposed environment were proposed. The main feature of the programming systems created in this way is that they actually, and not only nominally, support the cause-and-effect complementarity of the two components of solving any programming problem - programming as the generation and application of compositions and programs - the consequence of programming.

The following practical research results were obtained:

1. An experimental implementation of SOSrP was developed, which supports the development of software as a subject closure of the corresponding environment. Pragmatically determined conditions of such closure are set in the descriptive environment of compositional terms. The syntactic design of the solution is carried out by a Verilog descriptor. It is possible to support the creation of hardware with the involvement of FPGA as the basis of the hardware platform using "Quartus" CAD.

2. The new results obtained in the dissertation were used during the research work "Compositological principles of technological programming systems" (No. 0122U001568), and as materials for the preparation and teaching of the course of lectures and practical classes in the discipline "System programming and database management in telecommunications ” of the second (master’s) level of higher education, specialty 172 “Electronic communications and radio engineering” of the educational and professional program “Information and computing means of radio electronic systems”, which is confirmed by the relevant certificate and act.

Key words: programming, telecommunications, Verilog, composition, programming environment, reduction, object, information and communication networks, software-hardware complex, otologies, automation, computer modeling, universal and recursive functions, agent environment, algorithm.

Список публікацій здобувача

1. І.В. Редько, П.О. Яганов, М.О. Зилевіч, “Редукційне концептування оракульних схем”, *Системні дослідження та інформаційн ітехнології*, № 1, с. 21-33, 2021. doi: 10.20535/SRIT.2308-8893.2021.1.02 (фахове видання категорії Б, Scopus).

2. І.В. Редько, М.О. Зилевіч, “Редукційне програмування задач у технологічному середовищі програмування”, *Вчені записки Таврійського національного університету імені В.І. Вернадського. Серія: Технічні науки*, т.34, № 2, с.228-233, 2023. doi:https:doi.org/10.32782/2663-5941/2023.2.1/36 (фахове видання категорії Б).

3. І.В. Редько, М.О. Зилевіч, “Теоретичні основи програмної релятивізації у технологічних системах програмування”, *Вісник Вінницького політехнічного інституту*, № 2, с.72-80, 2023. doi: https:doi.org/10.31649/1997-9266-2023-167-2-72-80 (фахове видання категорії Б).

4. І.В.Редько, П.О. Яганов, М.О.Зилевіч, «Концептологічні засади технологічних систем програмування», Вчені записки Таврійського національного університету імені В.І. Вернадського. Серія: Технічні науки, т.34, № 5, с.219-223, 2023. DOI https://doi.org/10.32782/2663-5941/2023.5/34 (фахове видання категорії Б).

5. С.В. Кудлай, М.О. Зилевіч, І.В. Редько, П.О. Яганов, “Концептомонадна модель технологічного середовища програмування”, на *XIII Міжнародній науково-технічній конференції молодих вчених “Електроніка-2020” (ELCONF-2020)*, Київ, 2020, с.45-49. doi: 10.20535/2617-0965.2020.3.3.198584 (матеріали конференції).

6. I.V. Redko, P.O. Yahanov, M.O. Zylevich, “Concept-Monadic Model of Technological Environment of Programming”, on *2020 IEEE 2nd International Conference on System Analysis & Intelligent Computing (SAIC-2020)*, Kyiv,2020, с.125-130. doi: 10.1109/SAIC51296.2020.9239204 (матеріали конференції, Scopus).

7. М.О. Зилевіч, “Застосування оракульного концептування при програмуванні дизайну електронних мікросхем”, на *XIV Міжнародній науково-технічній конференції*

молодих вчених "Електроніка-2021" (ELCONF-2021), Київ, 2021, с.41-45. doi: 10.20535/2617-0965.eae.227740 (матеріали конференції).

8. І.В. Редько, П.О. Яганов, М.О. Зилевіч, "Технологічне середовище програмування з точки зору інтерсуб'єктивної парадигми", на *Міжнародна наукова інтернет-конференція на тему "Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення"*, Тернопіль, 2022, с.30-34. (матеріали конференції).

9. I.V. Redko, P.O. Yahanov, M.O. Zylevich, "Intersubjective paradigm and oracle conceptualization as an open-closed platform for programming technologicalization", on *2022 IEEE 3rd International Conference on System Analysis & Intelligent Computing (SAIC-2022)*, Kyiv, 2022, с.65-70. doi: 10.1109/SAIC57818.2022.9923011. (матеріали конференції, Scopus).

10. I.V. Redko, P.O. Yahanov, M.O. Zylevich, "Reduction programming in a technological programming environment", on *12th International Conference on Electronics, Communications and Computing (IC ECCO-2022)*, Chisinau, Republic of Moldova, 2022, с.194-200. (матеріали конференції).

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

НДР – науково-дослідна робота

ССВ – сутесутнісне відношення

СОСрП – суб’єкто-об’єктнє середовищє програмування

ПОТ – прагматико-обумовлена типізація

КП – композиційне програмування

СПЛ – стандартні програмні логіки

ПСП – проєктносутнісна парадигми

УЗП – універсум засобів проєктування

УСІ – універсальне середовище інтеграції

ЗЕР – засоби еволюційного розвитку

ВСТУП

Актуальність роботи. Вирішення будь-якого завдання в галузі телекомунікації починається зі складання плану. Структура останнього є унікальною в кожному конкретному випадку і залежить від розуміння рішення основної задачі як сукупності рішень її підзадач. Варіанти таких рішень залежать від середовища існування проблеми, складності, необхідних затрат на виконання, рівня обізнаності суб`єкта, доступної програмно-апартної платформи тощо. Сукупність вище описаних умов впливає на кінцевий результат, особливо на достовірність та можливість їх повторного використання як існуючих рішень для майбутніх задач. Таке сприйняття пояснюється парадигмою "розділяй та володарюй" за рахунок якої у галузі телекомунікацій досягнуто значних успіхів. Результативність такого підходу дозволяла нехтувати існуючими труднощами, що характерні для даного підходу. Тим не менш, даний підхід сприяв розвитку теорії телекомунікаційних систем та пов’язаних з ними методів програмування, забезпечуючи змістовну фактографію та фактологію останнього [1].

За останні десятиліття телекомунікації як предметна галузь досягла значних успіхів. Ці досягнення демонструються на прикладі успішного використання реально створених програмно-апаратних продуктів у пристроях і системах різногалузевого призначення. Програма, під керівництвом якої функціонують ці пристрої, розглядається як мета і засіб досягнення успіху. Причому головним якісним показником продуктивності програми, (тобто коректності виконання описаного алгоритму), вважається змістовна діяльність програміста по створенню програми. І хоч синтез програмно-апаратного комплексу передбачає наявність детального плану дій щодо його реалізації, практика свідчить про те, що ця діяльність більше відповідає поняттю “мистецтво програмування”, ніж практичній технології програмотворення. Якщо сповідувати принцип “мета виправдовує засоби”, то діяльність програміста може бути побудована на його суб’єктивних уявленнях про способи досягнення поставленої мети, власному досвіді спроб і помилок, інтуїтивних евристиках, запозичених знахідках тощо.

Проте на сьогоднішній день актуальним стає розуміння того, що програма як продукт повинна базуватись на технологічних засадах теорії інформації та процесів. Зважаючи на те, що значною мірою результативність визначається безпосередньо суб`єктом виконання і оцінюється по кінцевому результату. Обумовлена таким підходом свобода творчості суб'єктів програмотворення розвиває програмування як мистецтво, акцентуючи увагу на фактографічній складовій.

Слід зазначити, що з переваг такого підходу випливають і недоліки – загальне і всебічне спрощення розуміння програмування. Це проявляється у зміщенні уваги на результат творчого процесу – програми. В результаті такого зміщення залежність програми від суб'єкта програмотворення унеможливлює дослідження продуктивних засад програмотворення. Враховуючи те, що основоположні властивості програм формуються на початкових етапах, то така спрощеність значною мірою ускладнює формування технологічних основ програмування [2]. Такий підхід неминуче призводить до накопичення принципових проблем, які за своєю кількістю починають трактуватись як прояви кризи діяльності. Ця властивість притаманна усім системам, що розвиваються, а не тільки галузі телекомунікацій.

Пов'язано це зі всеосяжним проникненням інформаційних технологій у всі області діяльності людини, що спричиняє залучення значних матеріальних і нематеріальних ресурсів на тлі того, що ціна будь-якої прихованої або випадкової помилки неймовірно зростає. Тому вимоги щодо безпеки і надійності роботи програмних продуктів, особливо тих, які використовуються в небезпечних технологічних галузях чи пов'язанні з системами безпеки та життезабезпечення стають визначальним фактором при виборі того чи іншого програмного продукту. Ризик появи техногенної аварії внаслідок розтиражованого мільйонами копій програмного продукту без можливості для користувача проконтролювати його безпечність чи принаймні нешкідливість на етапах його створення стає неприйнятним. Тому у сучасному технологізованому світі будь-яка продуктивна діяльність повинна спиратися на чіткі процедури пошуку рішень, які

окреслені строгими структурами і є знаряддями досягнення гарантованого результату необхідної якості [3].

Якщо програмування розглядати як діяльність, націлену на створення програми, а саму програму як її наслідок, як деякий план логічної послідовності команд, за якими здійснюють процес обчислень, то слід визнати, що звичний метод досягнення результату не обов'язково єдино можливий. Для подолання протиріччя між продуктивним процесом і продуктом, коли надмірна індивідна суб'єктність програмістської діяльності не дозволяє об'єктивізувати та дослідити причинно-наслідкові зв'язки всередині процесу програмування як реальності, невіддільної від його наслідку, доцільно почати зі зміни самої парадигми програмування.

Нормативна база програмування є досить вивчена і складає основу індивідно-суб'єктивного його розуміння. Прикладами є об'ємні парадигми програмування: функціональна – орієнтована на об'єм функції [4, 5], об'єктно-орієнтована – на об'єм об'єкта [6], модульна – на об'єм модуля [7, 8] тощо. Це забезпечує можливість розгляду програмування в контексті взаємодоповнення як процесу програмотворення, так і його наслідку в їх причинно-наслідковому зв'язку з переважною роллю саме процесу. Такий підхід дає змогу визначити основи технологізації, тобто зробити загальнодоступними усі надбання кожного суб'єкта програмування для інших, тим самим доповнюючи нормативно-технічну базу галузі і даючи змогу відійти від поняття "мистецтво програмування"[9].

Зважаючи на те, що інтуїтивне сприйняття є визначальним фактором розуміння будь-якої суб'єктної діяльності стає цілком очевидним факт застосування найпростіших підходів на первинних етапах розв'язання будь-яких проблем у будь-якій суб'єктно орієнтовній діяльності. Проте чим розвинутішою стає діяльність, тим глибшим і ґрунтовнішим стає її розуміння, чого не можна сказати про підходи до її вирішення. Звідси слідує поширена проблема – засилля надмірної спрощеності підходів розв'язання, що особливо помітно у галузях пов'язаних із розробницькою діяльністю.

Серед вичерпної кількості основних причин такої проблемності до згаданих вище галузей відносять саме надмірне спрощення і недостатньо змістовне розуміння основоположних причинно-наслідкових взаємозв'язків між процесом та результатом [10]. Змістовоно, сутність таких зв’язків є досить складною, у зв’язку з чим досить поширеною є проблематика повного їх розуміння.

За замовченням, основу роль у таких зв'язках відіграє програма. Дана домінантність визначається результативністю такого підходу при використанні у математичній теорії, та особливо у сферах пов’язаних з комп’ютерним програмуванням та розробкою, що є досить очікуваним, враховуючи тенденцію та напрямок розвитку інформатизації людства. По суті, таке домінування є абсолютизацією ролі програми, що має негативний вплив на кінцевий результат, не дивлячись на можливі переваги. Зведення будь-чого в абсолют, навіть з метою отримання якісного результату, тягне за собою обмеженість, пов’язану із замкненою природою абсолютизації, що унеможливлює адекватне реагування на зовнішні зміни об’єктних процесів діяльності. З цього слідує неможливість усунення основної причини недоліків у програмуванні, в рамках абсолютизації, що вимагає адекватного предметного збагачення цілісного розуміння програмування [11, 12]. Іншими словами, загально прийняті сьогодні парадигми програмування, які по суті своїй актуально замкнені, мають бути адекватно збагачені до відкрито-замкненої парадигми програмування. Під адекватним збагаченням розуміється перехід від продукування індивідуальних підходів розв’язання тих чи інших проблем, які власне можуть бути досить ефективними, але обмеженні домінантністю програми, до розробки єдиного відкрито-замкненого середовища програмування, основоположними в якому є не програма, а програмування як дільність, що націлена на створення програми. Даний підхід дає можливість логічно взаємопов’язувати наявні точки зору на програмування у вигляді взаємодоповнення процесів та їх результатів, що складатиме основу реального розуміння програмування і дозволить відійти від

сучасного інтуїтивного базису, якісно його розвинувши за допомогою сучасних досліджень та розробок [13,14].

У дисертації в якості такого взаємодоповнення запропоновано концепцію композиційної моделі програмування на основі відповідної парадигми та її використання у суб'єкто-об'єктному середовищі програмування. Визначальною складовою запропонованого методу є зміщення акцентів із результатів розв’язання задач та верифікації результату на коректність методу їх досягнення. Таке зміщення дає можливість визначати та ефективно розв’язувати задачі пов’язані з якістю та надійністю отриманих рішень, забезпечити розвиток фактології та фактографії готових рішень, що дозволить заощадити ресурси. Під надійністю та якістю тут слід розуміти гарантовану коректність та функціональну безпеку отриманих рішень при використанні композиційної моделі в суб’єкто-об’єктному середовищі програмування. Під ефективним розв’язуванням тут розуміється збільшення продуктивності діяльності завдяки використанню вже наявних рішень з існуючих варіантів. Такий результат став можливим завдяки адекватній організації процесів розв’язання, через врахування активної ролі суб’єкта у програмуванні. З наведеного вище безпосередньо випливає, що технологізації процесів вирішення сучасних задач в телекомунікаційних системах, методом композитологічного уподібнення – логічного ядра суб’єкто-об’єктного середовища програмування є актуальною та важливою здачею сьогодення.

Зв’язок роботи з науковими програмами, планами, темами.

Дисертаційна робота виконувалась згідно з тематичними планами НДР КПІ ім. Ігоря Сікорського і кафедри КЕОА в межах НДР “Адаптивні середовища проєктування ефективних рішень в галузі автомобльної електроніки”, РК №0119U103292, а також НДР “Композитологічні засади технологічних систем програмування”, РК №0122U001568 згідно основних наукових напрямків діяльності КПІ ім. Ігоря Сікорського та пріоритетного напряму розвитку науки і техніки України “Інформаційні та комунікаційні технології”.

Мета і завдання дослідження.

Метою дисертаційної роботи є вирішення актуальної та важливої науково-прикладної задачі – технологізації процесів вирішення сучасних задач в людино-машинних, зокрема, телекомунікаційних системах, методом композитологічного уподібнення – логічного ядра суб'єкто-об'єктного середовища програмування (СОСрП).

Досягнення мети передбачає розв'язання наступних завдань:

1. Визначення взаємозв'язку концепто-монадної парадигми програмування та телекомунікаційних систем програмування.
2. Формування понятійної системи, що властива телеконцептуванню.
3. Запропонувати зведення до телеконцептограм генетичних структур програм.
4. Удосконалення оракульного телеконцептування для подальшого предметного збагачення суб'єкто-об'єктного середовища програмування.
5. Формування логіко-предметних засад суб'єкто-об'єктної телекомунікаційної системи програмування.
6. Розробка досліднецької реалізація суб'єкто-об'єктного середовища програмування, що підтримує розробку програмно-апаратного забезпечення як предметного замикання СОСрП.

Об'єкт дослідження: процеси активно-пасивного взаємодоповнення діяльностей у телекомунікаційному середовищі.

Предмет дослідження: відношення телекомпозитного уподібнення як основи взаємодоповнення замкненої логіки СОСрП та відкритого різноманіття його предметних суб'єктоорієнтованих продовжень – суб'єкто-об'єктних телекомунікаційних програмно-апаратних комплексів та систем.

Методи дослідження: в основу загальної методології роботи покладено сукупності загальнометодологічних, логіко-епістемологічних, логіко-математичних методів. З поміж загальнометодологічних основними є метод сутесутнісних відношень у поєднанні з синтезом, аналізом, дедукцією та редукційним підходом. Серед логіко-

епістемологічних методів використано абстрагування та конкретизацію. З поміж логіко-математичних методів застосовано композиційний метод проєктування та алгебраїчні методи дослідження у поєднанні з програмно-релятивізаційним підходом.

Наукова новизна отриманих результатів полягає в наступному:

1. Набуло подальшого розвитку предметне збагачення концепто-монадної парадигми програмування видом телекомунікаційних систем програмування та запропоновано відповідну понятійну систему телеконцептування, основу якої складають поняття телеконцепту, телеконцептограми, монади, оракульної телекомунікації, телекомпозиції, телекомпозиту та деякі інші. Телеконцептування конкретизовано покроковістю застосувань оракульних телекомунікацій як суб'єктоорієнтованих телеконцептів – базових логічних інструментів активностей суб'єкта програмування. Зміст їх розкрито у концептомонадному середовищі через оракули «обумовлення», «концепт», «монада», «сутність», «суть». Це забезпечує можливість реальної інтеграції наявних підходів проєктування програмно-апаратних комплексів телекомунікаційних систем у вигляді взаємодоповнення процесів програмування та їх результатів, що складатиме основу реального розуміння програмування і дозволить відійти від сучасного інтуїтивного базису, якісно його розвинувши за допомогою сучасних досліджень та розробок.

2. Вперше розкрито прагматичну обумовленість зведення до телеконцептограм генетичних структур програм. Це дозволяє реально, а не лише номінально підтримувати причинно-насідкові зв'язки при вирішенні задач, а також способи, методи та засоби їх специфікації. В якості телеконцептограм розглянуто телекомпозити – спеціальні класи суб'єктоорієнтованих базових телекомпозицій. Таким чином, телеконцептування на предметному рівні зводиться до вирішення відповідних рівнянь телекомпозитних редукцій. Це забезпечує коректність отримуваних рішень "за побудовою".

3. Подальшого розвитку набув метод редукції, побудовано та досліджено ряд нових редукційних схем, обумовлених відповідними композитами як предметнообумовлених шаблонів програмування, сформульовано та доведено стосовно них корисні необхідні умови редукційності. Парадигмне значення уведених редукцій та їх властивостей у тому, що вони конкретизують вирішення задач як покрокове розкриття структур генезисів їх рішень, з урахуванням яких процедурні, алгоритмічні, програмні реалізації цих рішень, зокрема їх нотація у мовах програмування отримуються вже автоматично, з обумовленою їх побудовою, коректністю. Продемонстровані важливі загальні особливості редукційного концептування оракульних схем. Обгрунтовано, що метод редукцій є дієвим інструментом для технологізації програмування та формування середовищ, а редукційні моделі програм та редукційні методи програмування є прагматико-обумовленою конкретизацією цього середовища.

4. Подальшого розвитку набув спосіб застосування підходу оракульного телеконцептування для подальшого предметного збагачення суб’єкто-об’єктного середовища програмування. На репрезентативних прикладах показані його особливості та перспективи подальшого розвитку. До особливостей відноситься те, що кожна підзадача може бути проконцептована до найпростішої підзадачі. Також використання оракульного телеконцептування дає можливість використання традиційного математичного апарату для нотації результату та поєднання його з денотативними методами. Реалізація такого методу на практиці сприяє уніфікації процесу розробки програмного продукту, тим самим оптимізує та реально об’єктивізує вплив активної ролі суб’єкта у телеконцептуванні через механізм оракульних телекомунікацій як технологію телекомунікаційних рішень задач.

5. Вперше запропоновано основні логіко-предметні засади суб’єкто-об’єктної телекомунікаційної системи програмування як предметного замикання СОСрП – несуперечливої логічної абстракції цілісного різноманіття програмно-апаратних комплексів та систем. Головною особливістю створюваних таким чином

телекомунікаційних систем програмування є те, що вони реально, а не лише номінально підтримують причинно-наслідкове взаємодоповнення двох складових вирішення будь-якої програмістської задачі – програмування як породження та застосування композицій та програми як його наслідку.

Практична значимість отриманих результатів визначається, зокрема:

1. Розроблена дослідна реалізація суб'єкто-об'єктного середовища програмування, що підтримує розробку програмного забезпечення як предметного замикання відповідного середовища. Прагматикообумовлені умови такого замикання задаються у дескриптивному середовищі композиційних термів. Синтаксичне оформлення рішення здійснюється Verilog-дескриптором. Можлива підтримка створення апаратного забезпечення із залученням FPGA як базису апаратної платформи із використанням САПР "Quartus".

2. Створено програму конвертор композиційних термів запропонованої дослідної реалізації суб'єкто-об'єктного середовища програмування у мову програмування Verilog, яка підтримую можливість додавання нових термів. Базові набори функцій і відповідних композицій можуть редагуватися відповідно до поставлених вимог.

3. Одержані в дисертації нові результати використані під час під час виконання науково-дослідної роботи "Композитологічні засади технологічних систем програмування" (№0122U001568). Впровадження та використання результатів роботи підтверджено відповідною довідкою.

4. Результати дисертаційної роботи використані як матеріали при підготовці та викладанні курсу лекційних і практичних занять з дисципліни "Системне програмування та керування базами даних в телекомунікаціях" другого (магістерського) рівня вищої освіти спеціальності 172 «Електронні комунікації та радіотехніка» освітньо-професійної програми «Інформаційно-обчислювальні засоби радіоелектронних систем», що підтверджено відповідним актом.

Отримані в дисертації результати дослідження можуть бути використані:

1. При розробці програмно-апаратних рішень, які висувають підвищені вимоги до надійності.

2. В навчальному процесі вищих навчальних закладів України при підготовці фахівців у галузі конструювання електронно-обчислювальної апаратури та фахівців із розробки ІТ-рішень, зокрема програмних та апаратних обчислювальних засобів.

3. В результатах науково-дослідних робіт за суміжною тематикою.

Особистий внесок здобувача. Основні ідеї та наукові результати дисертаційної роботи отримані автором особисто і висвітлено у десяти наукових працях, представлених у додатку А, серед яких робота [7] виконано одноосібно.

У наукових роботах, опублікованій у співавторстві, здобувачем особисто належить: [1] – формування і реалізація репрезентативних прикладів редукційного концептування; [2, 10] –приклади редукційного програмування, створення дескриптор-дефініторної бази данних; [3] – розробка редукційних підходів програмної релятивізації; [4] – створення композито-композиційних інтерфейсів для взаємодії з системами програмування; [5, 6] – композиційне збагачення та схематизація концептувань; [8, 9] – аналіз засад технологічних систем програмування.

Апробація результатів дисертації. Основні результати дисертації пройшли апробацію на наступних конференціях:

1. XIII Міжнародній науково-технічній конференції молодих вчених “Електроніка-2020” (ELCONF-2020), Київ, 2020.

2. 2020 IEEE 2nd International Conference on System Analysis & Intelligent Computing (SAIC-2020), Київ, 2020.

3. XIV Міжнародній науково-технічній конференції молодих вчених “Електроніка-2021” (ELCONF-2021), Київ, 2021.

4. Міжнародна наукова інтернет-конференція на тему “Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення”, Тернопіль, 2022.

5. 2022 IEEE 3rd International Conference on System Analysis & Intelligent Computing (SAIC-2022), Київ, 2022.

6. 12th International Conference on Electronics, Communications and Computing (IC ECCO-2022), Кишинів, Республіка Молдова, 2022.

Публікації. За темою дисертації опубліковано 10 наукових праць, із них 4 статті у наукових фахових виданнях України, з яких 1 стаття у періодичних наукових виданнях, проіндексованих у Scopus, 6 тез доповідей у збірниках матеріалів науково-технічних конференцій, серед яких 2 матеріали конференцій проіндексованих у Scopus.

Структура та обсяг дисертаційної роботи. Дисертація складається зі вступу, чотирьох розділів основної частини, висновків, списку використаних джерел і додатків. Загальний об’єм дисертації складає 190 сторінок, з них 160 сторінок основного тексту, 8 таблиці, 22 рисунки, 100 літературних джерел та 3 додатки.

РОЗДІЛ 1. Передумови суб’єкто-об’єктних середовищ програмування

Упродовж останніх десятиліть технологічні галузі зробили неймовірний вклад у розвиток людства. Інформаційні технології у всіх їх проявах стали буденною, а іноді навіть незамінною складовою життя сучасної людини будь-якого віку і професії. Цілком логічним є відповідне зростання вимог до характеристик технологічних продуктів, чи то побутова техніка чи програмно-апартний комплекс. В обох випадках актуальними є питання якості, надійності, ефективності виконання поставлених проблем тощо. Навіть якщо взяти до уваги сучасний рівень розвитку та досягнень інформаційних технологій, яким в загальному випадку важко щось протиставити, стає очевидно, що високий рівень цих результатів ґрунтується здебільшого на екстенсіональних методах досягнень результатів, що рано чи пізно призведе до проблеми підтримки стабільності якості результату. Основною причиною такої ситуації, зокрема у галузі телекомунікацій, є те, що сучасний розвиток теоретичної складової галузі відбувається внаслідок інтуїтивного і часто дуже спрощеного шляху пізнання, що визначається світоглядом суб’єкта проєктування і може зовсім не залежати від об’єкта.

Проте на сьогоднішній день актуальним стає розуміння того, що проєкт як продукт повинен базуватись на технологічних засадах теорії інформації та процесів. Зважаючи на те, що значною мірою результативність визначається безпосередньо суб`єктом виконання і оцінюється по кінцевому результату. Обумовлена таким підходом свобода творчості суб’єктів програмотворення розвиває проєктування як мистецтво, акцентуючи увагу на фактографічній складовій. В результаті такого зміщення залежність програми від суб’єкта програмотворення унеможливлює дослідження продуктивних засад програмотворення. Враховуючи те, що основоположні властивості програм формуються на початкових етапах, то така спрощеність значною мірою ускладнює формування технологічних основ. Логічною є необхідність у визначенні та формалізації важливих аспектів і підходів, що сформувались у програмуванні, визначенні та дослідженні основних причини реально-номінальних розбіжностей та шляхи їх розв'язання.

1.1 Передумови концептомонадної парадигми

Відповідно до загально прийнятого уявлення про причинно-наслідкові зв’язки, результативність будь-якої дії визначається розумінням початкової мети суб’єктом виконання. Таке розуміння визначає особливості підходу, що використовуватиметься для отримання результату і як наслідок його переваги та недоліки у порівнянні з іншими можливими варіантами. Відповідно до цього уявлення вирішення проблем, що постають в процесі проєктування програмно-апаратних рішень можливо лише на основі усвідомлення суті розуміння проблематики. Враховуючи всеосяжність такого усвідомлення, шлях до останнього цілком логічно визначається поетапним прагматичним збагаченням уявлень про суть розуміння. Такий шлях природно починається з константи розуміння, а саме притаманної їй здатності до еволюційного розвитку. Дана притаманність визначається у паралельності еволюції розуміння, разом із прагматичним збагаченням уявлень про нього, що безпосередньо стосується теорії програмування. Описана еволюційність обумовлено варіативністю самого розуміння, внаслідок чого можливе залучення до розгляду як процесу, так і його результату. Хоча здебільшого адекватнішим вважається прагматико-обумовлена точка зору. Основою такого погляду є прийняття розуміння в загальному контексті як такого, що визначається взаємодоповненням результату і його передумови. Такий погляд визначається в основному сутністю причинно-наслідкового зв’язку у поєднанній з інтуїтивним розумінням природи речей. Розуміння відповідно до сказаного вище необхідно розглядати як діяльність, результат якої визначається відповідною причиною. Сама ж причина може визначатись будь-яким чином, наприклад, виходячи з особливостей усвідомлення причинно-наслідкового зв’язку суб’єктом проєктування.

На основі наведеної вище інформації, програмування це діяльність, що обумовленна програмою. Таке розуміння реально збагачує інтуїтивне уявлення про програмування, що визначає подальшу інтенсіалізацію розуміння програмування до інтенсіонального збагачення розуміння програми. Під час такого зведення важливим є

збереження значущості усіх складових, що вимагає максимально широкого опису як програми у її реальних та ідеальних проявах, так і її обумовлення [8]. Задля подальшого полегшення розуміння та враховуючи значущість уведеної експлікації вирішено таке визначення програмування називати сутністю. А враховуючи необхідність сутностей не просто покривати реальне та ідеальне розуміння, а бути обумовленими сутностями, можна вивести розуміння суті, як того, що обумовлене сутністю.

Таке визначення суті реально, а не номінально об'єктивізує вплив суб'єкта на розвиток і при цьому є прагматико-обумовленим зразком для наслідування для майбутніх експлікацій, що можуть виникати в подальшому еволюційному розвитку розуміння. Означення суті та сутності саме по собі не є настільки важливим, наскільки важливий є їх причинно-наслідковий зв'язок, що визначає сутність як причину обумовлення із суттю, що є обумовленням передумови. Цей взаємозв'язок формує основу сутесутністної парадигми, яка визначає розуміння програмування як експлікативне зведення до визначення суті сутності. Зважаючи на обумовленість сутностей і сутей означений причинно-наслідковий зв'язок є сутесутнісним відношенням (ССВ), що за своєю структурою є бінарним відношенням яке може розглядатись як рефлексивно-транзитивне замикання ССВ, яке власне визначає програмування відповідним чином.

Відповідно до експлікаційного розуміння описане відношення являє собою алгоритм, який можна використовувати як інструмент еволюційного збагачення понятійної системи програмування, що розвивається шляхом використання ССВ як дорожньої карти еволюційного процесу збагачення. При цьому обумовленість має інтерсуб'єктивний характер визначеності, тобто є загальною на стільки, що може вважатися обумовленою за замовчуванням.

Описане вище ССВ, суть, сутність визначають базис понятійної системи суб'єкто-об'єктних середовищ програмування (СОСрП), що визначають розуміння та подальший розвиток предметного збагачення на основі ССВ, яке виступає взаємодоповненням суті

та сутності. Саме ж збагачення визначається прагматико-обумовленим застосуванням ССВ у поєднанні з замиканням відкритості обумовленості.

1.2 Інтерсуб'єктивна парадигма програмування

Якщо розглядати програмування як процес створення програм на основі логічної послідовності команд, то очевидно, що існує багато можливих методів досягнення результату. Для того, щоб подолати суперечність між творчим процесом програмування і його продуктом – програмою, де індивідуальний підхід програміста заважає об'єктивно вивчити причинно-наслідкові зв'язки всередині процесу, доцільно переосмислити підхід до програмування. Замість традиційної індивідуально-суб'єктивної парадигми, де програмування сприймається через призму програми, можна перейти до інтерсуб'єктивної парадигми. В цій парадигмі програма розглядається як інтерсуб'єктивний об'єкт, на який спрямована дія програміста.

Причиною винекнення будь-яких парадигмальних конструкцій є суперечливість між явищем та його розумінням. Нова парадигма це певний компроміс який забезпечує прийнятне узгодження описаної суперечності. Здатність такого узгодження до еволюційного розвитку розуміння, забезпечує адекватність його існування. Внаслідок такої еволюції розуміння виникає об'єктивна необхідність зміни парадигм.

Глибокий аналіз різних парадигм довів корисність їх інтеграційного розгляду, що дозволяє виділяти спільні риси в різних парадигмах. Визначальним в такому виділенні спільного є те, що будь-яку суб'єкто-об'єктну взаємодію слід розуміти як суб'єктне обумовлення об'єкта. Значущі парадигми розуміння суб'єкто-об'єктної взаємодії розглядаються як деяке обумовлення, а суб'єкто-об'єктне взаємодоповнення є логічно слідуючий активно-пасивний зв'язок предмета впливу з наслідком цього впливу, який визначається суб'єктом. Таке трактування дає змогу визначити програмування як суб'єкто-об'єктну відкрито-замкнуту систему з домінуванням в ній саме активної ролі суб'єкта. Взаємодоповнення активної і пасивної форм є ключовою характеристичною особливістю породжуваного ним причинно-наслідкового, а в разі обумовлення – ССВ.

Ключовим для розуміння програмування є те, що воно, по-перше, спрямоване на взаємодоповнення процесу програмотворення та його наслідку, а по-друге, допускає можливість його цілком природного родовидового збагачення, основу якого складає адекватне цілісне розуміння програми. Для того, щоб залишатись у тренді реальних побудов, найбільш прийнятним є греко-латинська трактовка терміну «програма» як нарису істотної риси (pro (лат.) – префікс, що вказує на уподоблення чомусь, γραμα (грец.) – риса, властивість, особливість).

Таке розуміння програми, з огляду на його загальність є відкритим для реальних збагачень. Точками їх здійснення тут є об'єктивно незводимі один до одного типи абстракцій – риса, її нарис та взаємодоповення першого з другим. Тут об'єктивна незводимість розуміється у сенсі неможливості об'єктивізації такого зведення.

Продовжуючи лінію на реальні збагачення програми та програмування звернемось спочатку до згаданого взаємодоповнення. Очевидно, тут маємо справу з причино-наслідковим зв'язком, що пов'язує між собою модальність риси, у сенсі можливості задіяння її у створення її ж нарису та реальність нарису як цілісну реалізацію цієї модальності. Характеристичні ознаки риси та нарису – їх модальність та реальність, відповідно, як безпосередньо, так і опосередковано збагачують здійснювані побудови. З об'єктивної незводимості типів абстракції «модальність» – (те, що може бути) та «реальність» – (те, що є), а також з того, що з самого початку на рису та нарис не накладалось ніяких штучних обмежень, безпосередньо випливає безальтернативність біабстрактності будь-яких реальних побудов. Зважаючи на те, що терміни риса та нарис переобтяжені вже існуючими інтерпретаціями, домовимось надалі модальний та реальний типи абстракцій називати *сутністю* та *суттю*, відповідно, а відношення, у якому вони знаходяться – *сутесутнісним відношенням*. Розглянемо його більш докладно з метою його подальшого реального збагачення.

Важливим наслідком реальності здійснюваних побудов є інтенсіональність сутесутнісного відношення, обумовлена об'єктивною незводимістю сутності та суті як

типів абстракції. Конкретний сутесутнісний зв'язок у ньому "з'являється" через обумовлення сутності – опосередковане зв'язування її деякою умовою. Ключову роль у цьому зв'язуванні, виходячи з вищезаначеного, відіграє носій таких умов – суб'єкт обумовлення. Тому об'єктивізація участі суб'єкта тут є засадничою передумовою реального осучаснення програмування.

З вищезазначеного безпосередньо випливає, що участь суб'єкта в обумовлені здійснюється в активній та пасивній формах. Перша проявляється в обумовленні безпосередньо – суб'єкт зв'язує сутність умовою, друга – опосередковано, через наслідок такого зв'язування. Сказане підкреслює надважливу роль умов у діяльності суб'єкта обумовлення. Поки що дещо розмите, аморфне їх використання як сутності у побудовах не шкодило останнім. Але в контексті об'єктивізації участі суб'єкта в обумовленні неможливо не здійснити збагачення цієї аморфності до відповідної їй суті. Воно буде здійснено опосередковано, через обумовлення аморфності ролі суб'єкта обумовлення як носія таких умов формою непротирічної логічної абстракції цілісного їх різноманіття. Виходячи з того, що згадана абстракція як суть є неодмінним атрибутом будь-якого суб'єкта обумовлення, використання її не вплине негативно на реальність побудов. Будемо називати таку суть концептом. Таким чином, активна форма участі суб'єкта в обумовленні визначає концепт, як суть, що обумовлена сутністю.

Що ж до пасивної форми участі суб'єкта в обумовленні, то тут вона виражена опосередковано, через наслідок обумовлення сутності концептом. Головним атрибутом усіх здійснюваних побудов, як зазначалося вище, є їх реальність. Утілена в цілісність, єдність отримуваних наслідків. Реальне введення концептів та монад дозволяє об'єктивізувати участь суб'єкта в обумовленні формою активно-пасивної взаємодії концепта та монади:

$$\begin{cases} \textit{концепт} = \text{суть, що обумовлює сутність}; \\ \textit{монада} = \text{сутність, що обумовлюється концептом} \end{cases}$$

Проведені побудови і зокрема остання дефініція, очевидно, зумовлені інтерсубєктивною прагматикою осучаснення програмування і суттєво збагачують

розуміння програмування. Зокрема, вводять важливі види роду сутей – концепти, монади, обумовлення. Причому, останні, так само, як і сутності в цілому презентують собою не стільки кількісні обсяги конкретних сутностей, сутей, концептів, монад і обумовлень, скільки якісні носії як абстракції відповідних суб’єктивних обумовлень, звані оракулами. Іншими словами, остання дефініція є оракульную системою, що презентує собою найбільш загальну реальну схему обумовлень, що спирається на широку фактографію програмування і складає основу всіх інших відомих схем або парадигм програмування. Однак обмежиться тільки нею неможливо з огляду на її недостатню змістовність. Необхідно її збагатити. У першому наближенні збагаченя полягає у розгляді згаданих вище оракулів та залучення до розгляду оракулів вищих типів – оракульних структур або схем. Серед усього різноманіття таких структур особливе місце займають композиційні і, перш за все, генні (базисні) структури.

Для їх конкретизації необхідно дослідити оракульні структури або схеми як відкрито-замкнені взаємодії суб’єктно-об’єктних обумовлень.

1.3 Концептування як схема взаємодії оракулів

Суб’єкт концептування, як творець програми, безпосередньо, у відповідності з концептом, створює алгоритм (план) вирішення задачі. Суб'єктно-об'єктна природа концептування сильно залежить від активної і пасивної форми обумовлень. Активна форма презентує безпосередній вплив суб’єкта обумовлення на відповідний предмет. Пасивна форма визначається наслідком концептування – обумовлення концептом. Тут участь суб’єкта реалізується опосередковано та впливає на наслідок концептування.

Таким чином, концептування є схемою взаємодії оракулів: обумовлення, концепту, монади, сутності, суті. Використовуючи останні суб'єкт здійснює свою активну роль у концептуванні, і тим самим окреслює схему “розділяй і володарюй”. Оракул “монада” є наслідком, що обумовлений реалізацією суб’єктом своєї активної ролі.

Конкретизація цієї схеми взаємодії оракулів призводить до конкретизації породжуваного концептуванням ССВ. З точки зору прагматики найбільш цікаві конкретизації, пов'язані з об'єктивізацією концепту. Вони зводять вихідне ССВ до відношення особливого виду – функціональної монадосутнісної залежності. Це зведення є істотно інтенсіональним, оскільки спирається на відповідний інтенсіонал, яким є актуалізований концепт. Екстенсіональне ж зведення такої функціональної залежності у вигляді заданого об’єму усіх монадосутнісних зв'язків трактується в контексті її обумовленості інтенсіоналом. Обидва зведення відображають одну і ту саму сутність, але під різними кутами зору.

Схематизація концептування за допомогою оракульних структур дозволить використати можливості традиційного математичного апарату для нотації результату та інтеграції його з денотативними методами. Базовим оракулом концептування є “сутність”. Оракульна природа концептування може бути проілюстрована простим наочним прикладом.

При позначенні операцій і предикатів використовують як операторну, так і термальну форми запису. Так, записи $S^2(\times, S^2(+, I_1^3, I_2^3), I_3^3)_{|N^3 \to N}$ і $(x + y) \times z_{|x,y,z \in N}$ позначають одну і ту ж саму арифметичну операцію. Перша, операційна форма, відображає генезис цієї операції з простіших, обумовлений інтенсіоналом – оператором суперпозиції. Друга термальна описує цю ж арифметичну операцію екстенсіонально як функцію через зазначення об’єму властивих їй арифметичних дій. Форми цих текстів, в певних сенсах, взаємозамінні, але у першій занотована абсолютно природна домінанта генезису по відношенню до його результату, друга ж є описом деякої сутності – актуально заданої множини дій, про генезис якої можна говорити лише опосередковано і частіше за все, недетерміновано, як про “функцію об’єму дій”.

Отже, суб’єкт свідомо чи підсвідомо об'єктивізує концепт схемою взаємодії оракулів, підтверджуючи оракульну природу концептування. Свідомо, (інтенсіонально, змістовно), якщо володіє методами концептування програмування, підсвідомо,

(екстенсіонально, об’ємно), якщо ні. Оракульні структури інтенсіонально збагачують концептування, формують точку зору на монаду, зокрема, програму, як на структурну сутність, а на концепт, як на відповідну структуру.

Серед усього розмаїття структур таких, що складають своєрідний “спільний знаменник”, стосовно якого всі інші структури природно розглядати як похідні від нього, є генетичні структури. Програми, як і монади в цілому, характеризуються перед усім генезисом. Як випливає з самої значущості генезису, він визначає основну парадигму концептування, зокрема, програмування. Серед усього різноманіття генетичних структур особливе місце займають композиційні структури.

1.4 Концептування як покроковість композитних обумовлень

Активна роль суб’єкта у програмотворенні проявляється у тому, що він визначає деталізацію сутності. Інтенсіональність оракула “активна роль суб’єкта” проявляє себе у покроковому процесі деталізації та його результаті, а екстенсіональність у нотації результату. Парадигма “розділяй і володарюй” обумовлюється необхідністю “розділяти”, тобто сприймати сутність як складену структурну, та “володіти”, розглядаючи її як наперед заданий цілісний об’єкт. Цим фактично визнається, що складена цілісна сутність якимось чином склалась. Ми будемо розглядати це як наслідок генезису або концептування, а концептом його є генетична структура – композиція.

Особливістю цих структур є те, що вони не можуть бути задані заздалегідь, актуально та вичерпно. Вони є наслідками суб’єктивних обумовлень і носіями суб’єктивних розумінь розглядуваних сутностей. Такі структури можуть бути як завгодно складно влаштовані. Концептами композицій є базові засоби генезису – генні структури або композити, на тлі яких інші композиції породжуються як похідні покроковості таких базових структур.

Таким чином, концептування конкретизується як покроковість активностей суб’єкта, концептом якої є композиція. При цьому, кожен крок покроковості теж є концептуванням, концепт якого презентується деяким композитом-оракулом, а

відповідна монада зводиться до представлення досліджуваної сутності як складеної сутності виду композиту сутностей-складових.

Сказане дозволяє виділити два основних види оракула “композит”. До першого належать композити, що підтримують розуміння композиції як покрокової сутності композитів. До другого – значущі для суб’єкта базисні композити, що формують носій його активної ролі в концептуванні. При цьому концептування обумовлене розумінням згаданої вище покрокової сутності як концепта композиції.

Щодо композитів першого виду слід зазначити, що цей вид композитів складає логічне ядро середовища програмування. З огляду ж на викладене можна зробити обґрунтований висновок про те, що його складають композити аплікації та суперпозиції. Аплікація для будь-якого композиту K і кортежу функцій $< f_1, \dots, f_n >$ ставить у відповідність нову функцію, яку позначатимемо $K(f_1, \dots, f_n)$, що являє собою результат застосування композиту K до аргументу $< f_1, \dots, f_n >$. Композит $S^p|_{p\in N}$ є суперпозицією k-арних функцій ($k \in N$), а саме: нехай дані m функцій $f_1, \dots, f_m$ однакової арності (наприклад, k) виду $A^k \to B$, визначені на деякій наперед заданій множині A зі значеннями з множини B (у загальному випадку $\boldsymbol{A \cap B \neq \emptyset}$**)** та нехай на множині B визначена m -арна функція f, що приймає значення з деякої множини C. Розглянемо нову k -арну функцію $g: A^k \to C$ таку, що її значення на аргументі $< a_1, \dots, a_k >$ задається так: $g(< a_1, \dots, a_k >) \cong f(f_1(< a_1, \dots, a_k >), \dots, f_m(< a_1, \dots, a_k >))$. Будемо казати, що функція g є результатом застосування $(m+1)$ -арной *суперпозиції* S^{m+1} до кортежу функцій $< f, f_1, \dots, f_m >$, тобто $g = S^{m+1}(< f, f_1, \dots, f_m >)$. Змістовно кажучи, перший підтримує покрокове зведення складно влаштованих композицій до більш простих, друга ж – є засобом специфікації прагматико-обумовлених поділів сутностей на взаємодоповнюючі частини.

Ключовим моментом композитів другого виду є їх релятивність відносно суб’єкта концептування. Іншими словами цей вид композитів виступає в середовищі програмування засобом його предметної адаптації до суб’єкта концептування.

Фундаментальне значення для композитів цього виду мають засадничі загальні властивості композитів: *тотальність*, *адекватність* та *замкнутість*. Ці властивості підтримують прагматичну обумовленість і релятивність виокремлення композицій як засобів концептування серед різноманіття алгебраїчних операцій. Змістовно кажучи, саме вони об'єктивізують визначальні для прагматики проблеми сутесутнісні відношення. Так властивість *адекватності* забезпечує обмеження композиту вимогою відповідності його змісту суб'єктивним уявленням про процес вирішення задачі. Властивість *замкнутості* також є релятивним обмеженням композиту вимогою до наслідку концептування бути монадою. Таке трактування є суттєвим узагальненням традиційного розуміння обчислюваності та природно зводиться до нього у випадку, коли рішення орієнтовано на комп'ютер. Вимога ж *тотальності* фіксує той факт, що сьогодні відсутні прагматичні мотивації розгляду не всюди визначених концептів як засобів концептування.

Зазначені властивості суттєво збагачуючи клас релятивних композитів, разом з тим дозволяють цілком природні їх подальші збагачення.

Одне з головних опосередкованих збагачень класу композитів другого виду здійснюється за рахунок типізації сутностей, на яких можуть бути задані композити. Інформатико-технологічна проблематика звужує збагачення до спеціальних класів функцій над носіями різних типів абстракції та рівнів загальності.

Щодо функціональних властивостей, то тут цікавими є ті з них, що використовуються композитами. Одна з них фактично закладена у визначенні композиту як n-арної операції. Інша – у вимозі обчислюваності композитів і полягає в тому, що передбачається можливість знаходження значення будь-якої вихідної функції на будь-якому даному, у разі його визначенності. Що ж стосується тих або інших властивостей даних, то їх аналіз показує, що обумовлені програмістською проблематикою композити використовують властивості даних бути абстрактними, кортежними та множинними даними.

Таким чином, актуалізація характеристичного для суб’єкта класу релятивних композитів відкриває шлях до концептомонадного моделювання систем у середовищі програмування.

1.5 Редукційне концептування оракульних схем

Розглянемо зміст програмування, яке прийнято сприймати як діяльність з отримання результату – готової програми, з точки зору процесу реалізації плану (програми) отримання цього результату. В такому разі потрібно зазначити, що програмування як діяльність здійснюється в певних умовах, які формуються у невідривному зв’язку з програмою. Можемо стверджувати про наявність обумовленості між програмуванням і програмою. Цей причинно-наслідковий зв’язок є суб’єкто-об’єктним, в якому суб’єкт – програміст – реалізує свій задум, спираючись на власні уявлення про способи, методи, евристики, знахідки і запозичення досягнення результату в межах об’єкту – процесу створення програми. Об’єктивізм процесу проявляється у можливості схематизації програмування генетичними структурами з властивими їм структурними сутностями. Генезис визначає структуру програми, її внутрішню логічну цілісність, якій підпорядковується результат програмування і нею обумовлюється.

В такому причинно-наслідковому зв’язку місце і роль суб’єкто-об’єктної взаємодії для сучасного розуміння програмування є ключовою і повинна розглядатись не поза рамками програмістської діяльності, як це зазвичай відбувається, а безпосередньо як самостійний об’єкт дослідження. З одного боку суб’єкт програмування не обмежений ніякими умовами в процесі програмотворення, крім головної, яку висловлює замовник – програма повинна забезпечувати очікуваний результат. При цьому програміст, обираючи на власний розсуд методи і засоби досягнення результату, використовуючи власний арсенал прийомів, підтверджуючи своїми діями поняття “мистецтво програмування”. Його активна роль проявляється не у залученні до схематизації об’єктивного причинно-наслідкового зв’язку між результатом і процесом його отримання у вигляді конкретних схем, а у формуванні власної авторської “функції

об’єму дій”, за допомогою якої він знаходить розв’язок поставленої задачі без належної конкретизації проміжних результатів. Про технологію програмування у цьому випадку говорити важко, оскільки досягнення кожного суб’єкта програмування реально невіддільні від нього, бо не підтримують взаємодоповнення фактографії і фактології. Загальновідомим у програмістському середовищі і підтверджуючим сказане є твердження про те, що невдалу програму простіше створити заново, ніж виправити.

Активна роль суб’єкта повинна проявити себе у конкретизації діяльності обумовлення, здійснюваної за допомогою плану програмування, шляхом актуалізації значимих для процесу інтенсіональних суб’єктивних обумовлень. Таким чином, на зміну традиційній індивідно-суб’єктивній парадигмі, в межах якої програма розглядається як результат програмування, має прийти інтерсуб’єктивна парадигма, згідно якої програмування – це діяльність суб’єкта зі створення програми (плану) цієї діяльності.

Суб’єкт програмування здійснює обумовлення у активній і пасивній формі. Активна форма виявляє себе у безпосередньому накладанні умови носієм таких умов – суб’єктом обумовлення – на програмування в залежності від того, яким арсеналом засобів він оперує. Програміст впливає на результат і на спосіб досягнення цього результату – програмування як процесу. Активна форма проявляється безпосередньо в прагматиці використання для досягнення мети активних видів обумовлень, які народжені не тільки в рамках вирішення конкретної проблеми, але і залучені, зокрема, з інших методів, що розвиваються на основі традиційного математичного апарату для нотації результату, а також денотативних прийомів.

Пасивна форма пов’язана з наслідком активізації діяльності суб’єкта і опосередковано впливає на наслідок планування програми. Цей вплив проявляє себе у залученні до процесу рішень, які вже відбулись або в межах цього процесу, або поза ним. Ці рішення матеріалізовані у результатах діяльності суб’єкта і його власного розумінні цієї діяльності. Зазвичай з цим пов’язують досвід, набутий при здійсненні активної

діяльності і активного обумовлення цієї діяльності. В інтерсуб'єктивній парадигмі активно-пасивне обумовлення збагачує поняття програми як сутності, оскільки визначає роль суб'єкта не поза межами процесу програмування, а безпосередньо у ньому, знаходячи своє відображення, зокрема, у формі семантичного терму, фіксуючи ключові моменти розуміння діяльності, що призвела до появи результату.

Вищенаведені міркування дозволяють конкретизувати ключові поняття нової моделі середовища програмування через "концепт", "сутність", "суть", "монаду", "оракул", "обумовлення". За їх допомогою можливо перейти до покрокової схематизації активностей суб'єкта програмування "всередині процесу". Її загальнозначиму основу складає поняття "концептування", представлене суб'єкто-об'єктною схемою взаємодії вищенаведених понять як оракулів. Схематизація здійснюється за рахунок актуалізації оракулів, що входять до неї та похідних від неї схем. І, що дуже важливо, вона може бути реально здійснена в рамках інтерсуб'єктивної парадигми. Редукція і редукціювання є дієвим інструментом для технологізації програмування та формування середовищ, а редукційні моделі програм та редукційні методи програмування є прагматико-обумовленою конкретизацією цього середовища.

1.6 Оракульна схематизація концептування

Для інтерсуб'єктивної парадигми програмування справедливим є положення, що поза цілісним розумінням програмування немає його продуктивного розуміння. Концептуванню при цьому відведена роль носія можливих збагачень даного змістовного положення. Неможливість об'єктивного зведення один до одного цих видів розумінь обумовила відкрито-замкненість та суб'єкто-об'єктність схеми концептування. Об'єктивізована замкнутість схеми забезпечується активно-пасивною взаємодією оракулів, що входять до неї. Суб'єктивна ж відкритість підтримується відкритістю самих оракулів "сутність" – Ent, "суть" – Gst, "концепт" – Con, "монада" – Mon, а також активною і пасивною формами обумовлень – "обумовлює" – $\succ$, "обумовлюється" – $\prec$. Це дозволяє реально збагатити, а в подальшому адекватно технологізувати відкрито-

замкнене розуміння активної ролі суб'єкта програмування як поєднання цілісної та продуктивної його складових.

Змістовно, саме присутні у схемі концептування оракули є основою реалізації активної ролі суб'єкта в концептуванні. Виходячи з того, що ключовою ланкою реалізації активності суб'єкта є композит як концепт активності суб'єкта і його зв'язок з композицією, можна сказати, що активність суб'єкта є концептуванням, концептом якого є композит – базова для суб'єкта генна структура. Зі сказаного випливають наступні конкретизації як властивих концептуванню оракулів, так і самої оракульної схеми. Перша стосується специфіки використання композитів:

$$\begin{cases} Comt \overset{def}{=} Gst \overset{B}{\succ} Ent \\ CtMon \overset{def}{=} Ent \prec Comt \end{cases},$$

де $Comt$ – оракул "композит", $\overset{B}{\succ}$ – оракул "базисно обумовлює", тобто обумовлює "за один крок", природа якого не потребує додаткової деталізації для суб'єкта, а $CtMon$ – збагачення оракула Mon за рахунок актуалізації концепта композитом. Друга відображає специфіку використання композитів:

$$\begin{cases} Comp \overset{def}{=} Gst \overset{G}{\succ} Ent \\ CMon \overset{def}{=} Ent \prec Comt \end{cases},$$

де $Comp$ – оракул "композиція", $\overset{G}{\succ}$– оракул "генезисно обумовлює", тобто обумовлює покроково у зазначеному вище сенсі, а $CMon$– збагачення оракула Mon за рахунок актуалізації концепта композицією.

Наведені схеми є концептами для різних композитних і композиційних концептувань. Останні отримуються за рахунок актуалізації оракулів, що входять до схем. З самої побудови цих схем випливає, що вони продукують функціональні сутесутнісні залежності виду $Ent \overset{Comt}{\rightarrow} CtMon$ або $Ent \overset{Comp}{\rightarrow} CMon$. Отже, і композитні, і композиційні монади можуть бути експліковані як композиції сутностей:

$$\begin{cases} CtMon \Rightarrow Comt(Ent_{i_1}, \ldots., Ent_{i_k}) \\ CMon \Rightarrow Comp(Ent_{j_1}, \ldots., Ent_{j_p}) \end{cases}'$$

де $Ent_{i_1}, \ldots., Ent_{i_k}, Ent_{j_1}, \ldots., Ent_{j_k}$ – деякі сутності, а $\Rightarrow$ означає "експлікативно зводиться". Розглядаючи ці експлікативні зведення в контексті концептування і проєктуючи їх на відповідні функціональні залежності, отримуємо наступні оракульні схеми:

$$\begin{cases} Ent \overset{Comt}{\rightarrow} Comt(Ent_{i_1}, \ldots., Ent_{i_k}) \\ Ent \overset{Comp}{\rightarrow} Comp(Ent_{j_1}, \ldots., Ent_{j_p}) \end{cases}.$$

Особливість кожної з них у тому, що перша використовує базову композицію, але при цьому на сутності $Ent_{i_1}, \ldots., Ent_{i_k}$ ніяких додаткових обмежень не накладається. У другій же, навпаки, композиція не обов'язково базова, скоріше вона похідна від композитів, зате сутності $Ent_{j_1}, \ldots., Ent_{j_k}$ є елементарними, тобто достатньо деталізованими з точки зору суб'єкта. Перша є концептом реалізації активної ролі суб'єкта в будь-якому кроці концептування, друга являє собою рефлексивно-транзитивне замикання породжуваної концептуванням функціональної залежності. Це створює можливість передбачення наслідку концептування як композиції елементарних, з точки зору суб'єкта, сутностей.

У наведених схемах явно відображений факт взаємодоповнення двох методів дослідження концептування – синтезу та аналізу, композиції і декомпозиції. Синтез представлений в них оракулами $Comt$ і $Comp$, а аналіз (декомпозиція) – $Ent_{i_1}, \ldots., Ent_{i_k}, Ent_{j_1}, \ldots., Ent_{j_k}$.

Звернемо увагу на те, що перша схема оперує базовими для суб'єкта композиціями (композитами). Це дозволяє здійснити ще один крок до продуктивного збагачення розуміння програмування. Будемо говорити, що кортеж $< Ent_{i_1}, \ldots., Ent_{i_k} >$ є $Comt$ - редукцією сутності Ent, якщо існує функціональна залежність

$Ent \overset{Comt}{\rightarrow} Comt(Ent_{i_1}, \ldots., Ent_{i_k})$. З наведеного вище безпосередньо випливає наступне твердження про оракульну схематизацію редукції.

Кортеж $< Ent_{i_1}, \ldots., Ent_{i_k} >$ є $Comt$ -редукцією сутності Ent, якщо справедливо $Ent = Comt(E_{i_1}, \ldots., E_{i_k})$.

Змістовний сенс наведеної оракульної схеми полягає у тому, що вона природним чином імплементує парадигму “розділяй і володарюй” в розумінні активної ролі суб’єкта в концептуванні, підтримуючи реальне взаємодоповнення декомпозиційного і композитного методів концептування. З наведеного вище безпосередньо випливає, що композит $Comt$ як актуально задана базова композиція є концептом Ent. Тобто, Ent є монадою.

Природа композитів і композицій, в цілому, релятивна та істотно залежить як від суб’єкта, так і від розглядуваних сутностей. У свою чергу розуміння сутності залежить від композитів, що залучаються у розгляд. Тому реальне проникнення в інтерсуб’єктивну природу композитів і композицій можливе тільки у взаємодії з прагматико-обумовленим збагаченням цього взаємодоповнення.

Рішення будь-якої задачі, як відомо, є інтеграцією рішень її підзадач. Якщо проблема проста, то інтеграція тривіальна і, як правило, явно не виділяється. У разі ж, коли остання є складною, інтеграційний аспект її рішення домінує, адже власне саме ним і визначається складність. Зважаючи на це, подальші побудови проведемо “від простого до складного”, розглянувши на прикладах нескладних задач чисельного аналізу їх вирішення в середовищах мікро-, макро- і біпольної інтеграції. В якості платформи розгляду використаємо композиційне програмування і іменну модель даних, функцій і операцій, а в якості композитів – операції мультиплікування $\circ$, розгалуження IF, циклування WD і найпростіші похідні від них композиції, що уточнюють найбільш вживані та прості способи генезису одних програм з інших. Надалі під даними, функціями та операціями, якщо не зазначено інше, розуміємо іменні дані, іменні функції та іменні операції відповідно.

Проведене змістовне розгортання і роз'яснення концептування в достатній мірі обгрунтовує точку зору на нього як на покроковість редукувань, яке зводиться до пошуку підходящої $Comt$ -редукції для індукованого відповідною функціональною залежністю рівняння $Ent = Comt(E_{i_1}, \ldots., E_{i_k})$. Під рішенням тут розуміється кортеж $< Ent_{i_1}, \ldots., Ent_{i_k} >$ такий, що справедливою є тотожність $Ent \equiv Comt(Ent_{i_1}, \ldots., Ent_{i_k})$. Запис $Comt(Ent_{i_1}, \ldots., Ent_{i_k})$ означає застосування k -арного композиту $Comt$ до кортежу $< Ent_{i_1}, \ldots., Ent_{i_k} >$, тобто:

$$Comt(Ent_{i_1}, \ldots., Ent_{i_k}) \stackrel{def}{=} Ap(Comt, < Ent_{i_1}, \ldots., Ent_{i_k} >),$$

де Ap розуміється традиційно як аплікація, а $\stackrel{def}{=}$ як рівність за визначенням.

Розглянемо застосування апарату простої WD -редукції, рухаючись від простого до складного. Відповідно до вищевикладеного, така редукція є кортежем $< g, p >$, що є рішенням рівняння $f = WD(g, p)$, де g, p – деякі функція та предикат. Для пошуку рішення корисними є наступні наслідки вищезгаданого твердження про схематизацію редукції, які продуктивно збагачують його, при цьому істотно звужуючи коло пошуку редукції.

- $< g, p >$ є WD -редукцією функції f якщо $f = WD(p, g)$.
- Щоб кортеж $< g, p >$ був WD -редукцією функції f, необхідно, щоб виконувалась рівність $g \circ f = f$, де $\circ$ – операція мультиплікування.

Проілюструємо сказане на конкретних прикладах. Для цього, звернемося до найпростішого класу задач чисельного аналізу, що складається з однієї задачі – обчислення $\sqrt{x}$ із заданою точністю ε, де x і ε – додатні дійсні числа.

Відомо, що послідовність y_0, y_1, y_2, ..., в якій $y_0 = a, y_{i+1} = \frac{1}{2}(y_i + \frac{x}{y_i})|_{i=0,1,2,\ldots}$, де a – деяке додатне дійсне число, незалежно від a збігається до $\sqrt{x}$. Звідси випливає, що процес обчислення $\sqrt{x}$ із заданою точністю може бути зведений до деталізації іменної функції f, яка іменній множині $\{(u, x), (v, \varepsilon), (w, 0)\}$ ставить у відповідність іменну

множину $\{(w, y_n)\}$, де y_n – перший член зазначеної послідовності, для якого виконується умова $|y_n^2 - y_{n-1}^2| < \varepsilon$.

Знайдемо WD -редукцію функції f. Розглянемо наступні іменну функцію $g:\{(u,x),(v,\varepsilon),(w,a)\} \to \{(u,x),(v,\varepsilon),(w_{pr},a),(w,\frac{1}{2}(a+\frac{x}{a}))\}$ та іменний предикат $p:\{(v,\varepsilon),(w,a),(w_{pr},b)\} \to \begin{cases} True, |a^2 - b^2| \geq \varepsilon \\ False, |a^2 - b^2| < \varepsilon \end{cases}$. Легко переконатися в справедливості рівності $f = g \circ f$ і, як наслідок, у тому, що $< g, p >$ дійсно є WD -редукцією функції. Адже дійсно, якщо $|a^2 - (\frac{1}{2}(a+\frac{x}{a}))^2| < \varepsilon$, то $f(\{(u,x),(v,\varepsilon),(w,\frac{1}{2}(a+\frac{x}{a}))\}) = \{(w, y_n)\}$. Отже, можна зробити висновок про те, що $f \equiv WD(g,p)$. При цьому правильність висновку безпосередньо випливає з побудови.

Даний приклад демонструє задачу, що вирішується “в один крок”. Для рішень таких задач не потрібно залучати оракульні схеми концептування, тому і інтеграційна складова тут є тривіальною. Наступний приклад дещо глибше зачіпає інтеграційний аспект концептування. Розглянемо клас спеціальних рівнянь виду $x = \phi(x)$, де функція $\phi(x)|_{x \in R}$ задовольняє наступним двом умовам:

1) вона визначена і неперервно диференційовна на всій числовій прямій;

2) існує таке дійсне число $p < 1$, що для всіх x справедливо $|\phi'(x)| \leq p$.

Відомо, що стосовно такого класу рівнянь метод простих ітерацій збігається. Причому рішенням є границя послідовності $\{x_i\}_{i=0,1,2,..}$, де x_0 – будь-яке дійсне число, а $x_{i+1} = \phi(x_i), i = 0,1,2, ...$

З наведених вище умов випливає, що концептування задачі пошуку рішення рівнянь виду $x = \phi(x)$ можна звести до деталізації функції $f:\{(v,x_0),(u,\varepsilon)\} \to \{(v,x_n)\}$, де x_n – перший член послідовності наближень, для якого виконується умова $|x_{n-1} - x_n| < \varepsilon$. Очевидно, що кортеж $< g, p >$, де $g:\{(v_{pr},a),(v,b)\} \to \{(v_{pr},b),(v,\phi(b))\}$, а $p:\{(v_{pr},a),(v,b),(u,\varepsilon)\} \to \begin{cases} True, |a-b| \geq \varepsilon \\ False, |a-b| < \varepsilon \end{cases}$, є рішенням рівняння $f = WD(E_1, E_2)$. Тобто, $f \equiv WD(g,p)$.

Особливість цього рішення полягає в тому, що воно є вже оракульною схемою вирішення класу задач, іншими словами, є схемою монади. Оракулом тут виступає $\phi:\{(v,b)\}\to\{(v,\phi(b))\}$ – іменна специфікація $\phi(x)|_{x\in R}$. Схема перетворюється в конкретну монаду після заміни ϕ в схемі конкретною функцією, що задовольняє вищенаведеним двом умовам. Такими, наприклад, є функції $\phi:\{(v,b)\}\to\{(v,\frac{cos(b)}{2})\}$, $\phi:\{(v,b)\}\to\{(v,\frac{sin(b)+cos(b)}{3})\}$ тощо. В цій схемі оракульну взаємодію представлено рудиментарно, тому що в ній є всього один оракул – ϕ.

Відзначимо, що концептування надає розробнику можливість отримання схем рішень. Розглянемо для прикладу концептування операцій підсумовування, в якому враховуються взаємодії вже декількох оракулів.

Під функцією, заданою операцією підсумовування, розуміють функцію, яка визначається рівністю $f(x_1,\dots,x_{n-1},m)=\sum_{i=1}^{m}g(x_1,\dots,x_{n-1},i)$, де $g(x_1,\dots,x_{n-1},i)$ – довільна, але фіксована функція, що залежить від дійсних змінних $x_1,\dots,x_{n-1}$ і змінної i, що приймає, як і m, натуральні значення. Безпосередньо зі сказаного випливає, що концептування операції підсумовування зводиться до деталізації іменної оракульної схеми (іменної функції з оракулом) $f:\{(v_1,a_1),\dots,(v_{n-1},a_{n-1}),(v_n,m)\}\to\{(w,f(a_1,\dots,a_{n-1},m)\}|_{a_1,\dots,a_{n-1}\in R,m\in N}$. Що ж стосується схеми рішення, то вона обумовлена наступною характеристичною властивістю оракула $f(x_1,\dots,x_{n-1},m)$:
$f(x_1,\dots,x_{n-1},1)=g(x_1,\dots,x_{n-1},1)$, $f(x_1,\dots,x_{n-1},k)=f(x_1,\dots,x_{n-1},k-1)+$
$+g(x_1,\dots,x_{n-1},k)$ і, вочевидь, може бути задана рівнянням $f=\circ(E_1,E_2)$, рішенням якого є $\circ$-редукція $<f_1,f_2>$, де $f_1:\{(u,a),(w,s)\}\to\{(u,m),(w,0)\}|_{a\in N,s\in R}$ і
$f_2:\{(v_1,a_1),\dots,(v_{n-1},a_{n-1}),(v_n,m),(w,s)\}\to\{(w,s+g(a_1,\dots,a_{n-1},m)+$
$+g(a_1,\dots,a_{n-1},m-1)+\dots+g(a_1,\dots,a_{n-1},1)\}|_{a_1,\dots,a_{n-1}\in R,m\in N}$.

Отже, $f\equiv f_1\circ f_2$. Треба зазначити, що дане рішення є коректним з побудови. Крім того, воно, як і будь-яке з наведених вище рішень, може бути при необхідності деталізовано. Припустимо, що функція f_1 цього не потребує, а от f_2 необхідно

деталізувати. В цьому випадку, деталізація, очевидно, може зводитись до знаходження рішення рівняння виду $f_2 = WD(E_1, E_2)$. Неважко помітити, що $< h, p >$, де $p :\{(u,k)\} \to \begin{cases} True, k \neq 1 \\ False, k = 1 \end{cases}$ і $h :\{(v_1, a_1), \dots, (v_{n-1}, a_{n-1}), (v_n, m), (u,k), (w,s)\} \to \{(u, k-1), (w, s + g(a_1, \dots, a_{n-1}, k))\}$ є WD -редукцією функції f_2, тобто $f_2 \equiv WD(h,p)$. Отже, $f \equiv f_1 \circ f_2 = f_1 \circ WD(h,p)$.

Відзначимо, що отримане рішення є оракульною схемою, оракул якої – функція $g(x_1, \dots, x_{n-1}, i)$, точніше її представлення у вигляді іменної функції $g :\{(v_1, a_1), \dots, (v_{n-1}, a_{n-1}), (u,i)\} \to g(a_1, \dots, a_{n-1}, i)|_{a_1,\dots,a_{n-1} \in R, i \in N}$.

Схема перетворюється на конкретну монаду після заміни $g(x_1, \dots, x_{n-1}, i)$ в схемі конкретною функцією, яка, в свою чергу може потребувати редукування. Наприклад, обравши у якості неї функцію $g(i) = \sqrt{i}$, отримаємо задачу, для вирішення якої спочатку залучається поточна схема монади, а далі, у випадку необхідності деталізації функції $g(i)$, може бути використане отримане вище рішення для функції $\sqrt{x}$. У цьому випадку вже матимемо рішення задачі, що використовує нетривіальну покроковість композитних концептувань виду $Ent \overset{Comt}{\to} CtMon$.

Не складає особливих труднощів, відштовхуючись від отриманої оракульної схеми, розглянути концептування для більш специфічних задач, наприклад:

$f(x_1, \dots, x_{n-1}, k, m)|_{k \in N} = \sum_{i=k}^{m} g(x_1, \dots, x_{n-1}, i)$,

$f(x_1, \dots, x_{n-1}) = \sum_{i=k(x_1,\dots,x_{n-1})}^{m(x_1,\dots,x_{n-1})} g(x_1, \dots, x_{n-1}, i)$, де $m(x_1, \dots, x_{n-1})$ і $k(x_1, \dots, x_{n-1})$ – натуральнозначні функції такі, що $m(a_1, \dots, a_{n-1}) > k(a_1, \dots, a_{n-1})|_{\forall a_1,\dots,a_{n-1} \in R}$. Відповідні монади для цих задач можуть бути отримані аналогічно до вищенаведеного. Відзначимо, що на відміну від розглянутих раніше прикладів, останні модифікації характеризуються нетривіальним взаємозв'язком трьох оракулів – $g(x_1, \dots, x_{n-1}, i)$, $m(x_1, \dots, x_{n-1})$ і $k(x_1, \dots, x_{n-1})$. Кожний з них може бути, в свою чергу, розглянутий у контексті його подальшої деталізації. Наведені репрезентативні приклади редукційного

концептування, обґрунтовують оракульну схематизацію як технологію рішень програмістських задач.

1.7 Приклад застосування оракульного концептування

Оракульним концептуванням вважається опис концептування через оракульні структури. Відповідно концептуванням вважається процес створення концепту, схемою взаємодії оракулів: обумовлення, концепту, монади, сутності, суті. За допомогою цих оракулів суб’єкт здійснює свою активну роль у концептуванні, актуалізуючи всі, або деякі з них, і тим самим конкретизуючи вихідну схему “розділяй і володарюй”.

Концептом називають план певної дільності, спрямований на розв'язання конкретної проблеми. Така задача може бути будь-якого рівня складності, але згідно з підходом оракульного концептування її можна розбити на скінченну послідовність елементарних підзадач, які називаються оракулами. Під оракульними структурами слід розуміти сукупність певних елементарних підзадач, що виникли внаслідок концептування конкретної задачі. Схематичне визначення оракульного концептування показано на (Рис. 1.1).

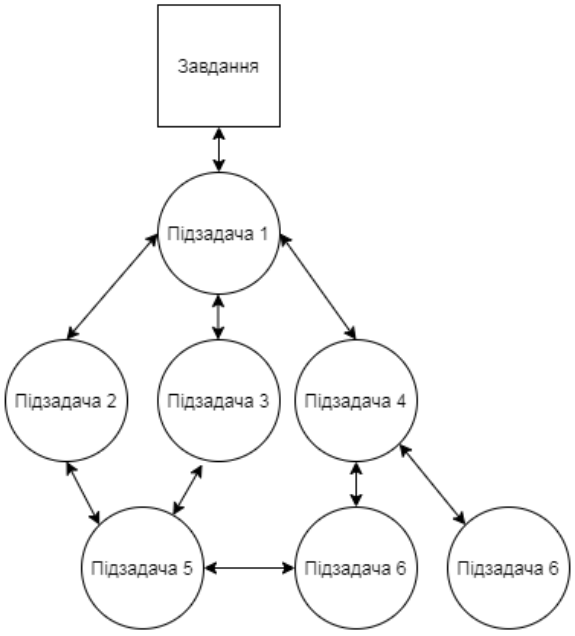


Рис. 1.1. Схематичний вигляд оракульного концептування

На зображеній вище схемі у квадраті позначено початкове завдання. У колах позначені елементарні підзадачі, які вважаються оракулами. Двонаправлені стрілочки між оракулами відповідають елементарним концептам, а відповідно розбиття певної задачі на елементарні є оракульним концептуванням.

Таким чином, в програмуванні концептування визначається як послідовність дій суб'єкта програмотворення, спрямована на вирішення певного завдання. При цьому кожен етап такого концепту також є певним концептом. Теоретично кожна підзадача аналогічним чином може бути проконцептована нескінченно, проте на практиці все закінчується сукупністю певних елементарних дій, які є характерними для сфери застосування цього підходу.

У процесі програмотворення особа на підсвідомому рівні об'єктивізує концепт схемою взаємодії оракулів. Оракульні структури збагачують концептування, формують точку зору на програму як на структурну діяльність, а на концепт – як на відповідну структуру.

Для кращого розуміння принципів і особливостей оракульного концептування розглянемо його на прикладі роботи спрощеного скінченного автомату "кавоварка", схема якого показана на (Рис. 1.2).

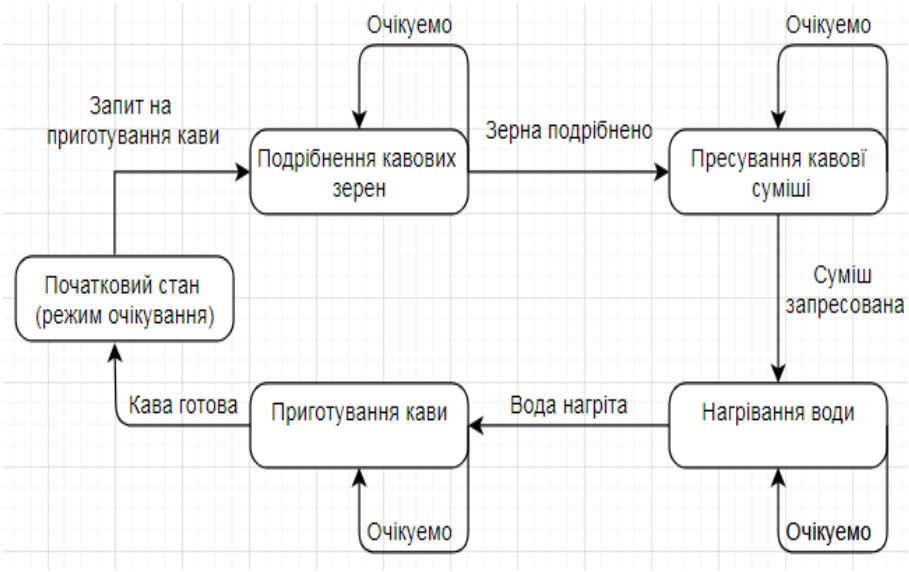


Рис. 1.2. Спрощена схема скінченного автомату "кавоварка"

Зображена вище схема дає спрощене уявлення про алгоритм роботи кавової машини. Згідно з цим, автомат має стани, кожен з яких відповідає за вирішення певної підзадачі, перехід між станами здійснюється при виконанні певних умов. Вже на цьому етапі можна сказати, що кожен стан автомату – це оракул, а умова переходу – концепт цього оракулу. Таке твердження є вірним, але якщо дивитись з точки зору апаратного коду, то на даному рівні концептування ще не має особливого змісту.

Щоб розкрити зміст оракульного концептування, застосуємо його до етапу “Подрібнення кавових зерен”. Схема такого застосування показана на (Рис. 1.3).

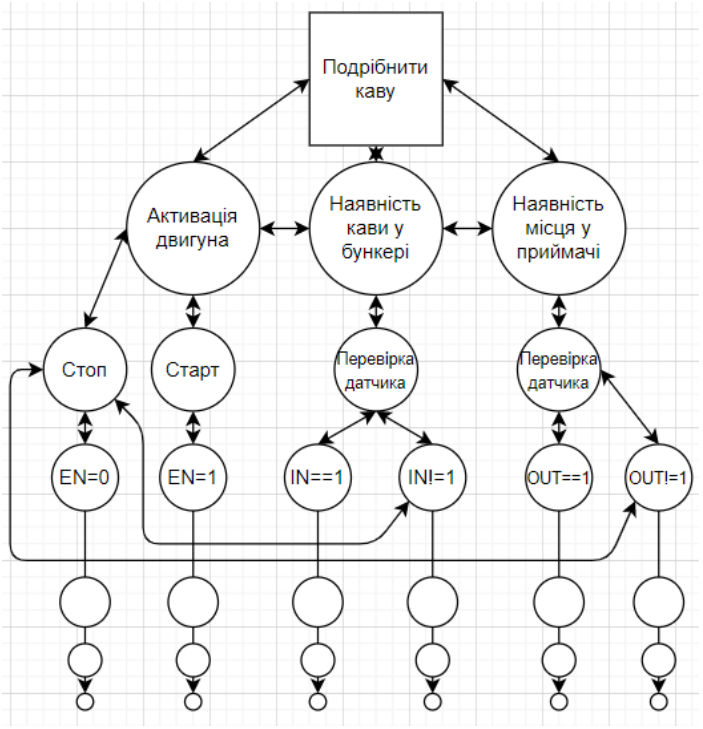


Рис. 1.3. Схема оракульного концептування

Зображена вище схема репрезентує найпростіше застосування оракульного концептування. Відповідно до схеми є початкове завдання “подрібнити каву” і за допомогою концептування початкову задачу розбито на простіші складові – оракули. Чим менше коло, тим простішим є оракул. Таким чином найпростіший оракул являє собою низькорівневий апартний код для процесора. Для програміста рівень композиції відповідає синтаксису мови програмування.

Використання такого підходу дає змогу уніфікувати алгоритми та схеми написання програмного коду для будь-якого продукту чи системи. Починаючи від повсякденних гаджетів і закінчуючи системами життєзабезпечення космічних апаратів. Це досягається шляхом усунення надмірного індивідного впливу конкретного програміста на кінцевий продукт.

Роль суб'єкта під час створення програми визначається тим, що саме він визначає сутність програми. Оракульність тут підкреслюється у поступовості визначення сутності програми і її результату.

Практичне використання оракульного концептування показано на прикладі розробки людино-машинного інтерфейсу за кодом програми зображеної на (Рис. 1.4).

```
1  module ctr (input                up_down,
2                                   clk,
3                                   rstn,
4              output reg [2:0]     out);
5
6      always @ (posedge clk)
7          if (!rstn)
8              out <= 0;
9          else begin
10             if (up_down)
11                 out <= out + 1;
12             else
13                 out <= out - 1;
14         end
15 endmodule
```

Рис. 1.4. Реалізація лічильника на мові Verilog

Показана вище програма є простим лічильником, який рахує від 0 до 7 з можливістю скидання по інверсному значенню сигналу rstn та можливістю вибору "напрямку" рахування по значенню сигналу up_down. Для мінімізації впливу індивідуальних особливостей конкретного програміста на написання такого коду, останній має бутии максимально уніфікований таким чином, що програмісту необхідно обрати лише модуль, який він хоче реалізувати (в даному випадку – лічильник), та

обрати його розрядність. Усе інше здебільшого не потребує втручання. Схематична реалізація такого інтерфейсу показана на (Рис. 1.5).

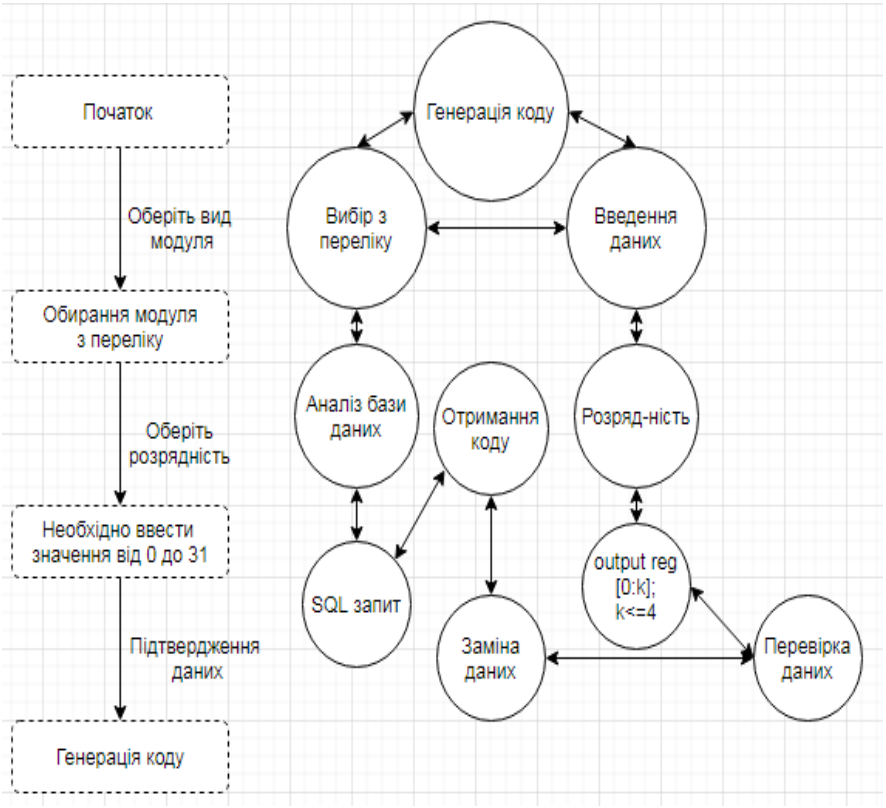


Рис. 1.5. Схематична реалізація інтерфейсу

Відповідно до зображеної схеми програмісту необхідно з існуючого переліку обрати модуль, який йому потрібно реалізувати. Зрозуміло, що перелік має бути стандартизованим відповідно до сучасних норм та не мати доступу до редагування для сторонніх осіб. Допустимі зміни заздалегідь визначаються та можливі до внесення шляхом використання людино-машинного інтерфейсу. Тобто в даному прикладі користувач може змінити лише розрядність лічильника. Усе інше покладається на інтерфейс користувача.

Проте залишається питання реалізації такого способу програмування, адже інтерфейс при оракульному концептуванні згенерує код, який можливо буде редагувати як при звичному способі написання програм. І це буде цілком нормальним явищем, оскільки завдання програмування часто дуже специфічні і потребують всебічного

підходу до вирішення, що реалізується саме завдяки описаній на початку роботи надмірній індивідності процесу.

Слід розуміти, що використання оракульного концептування має полегшити рутинні процеси програмування. Усі загально визначені прийоми в написанні, як і власне конкретні модулі, якщо говорити про мову програмування з прикладу будуть уніфікованими і загальнодоступними для усіх, коли надбання кожного окремого суб'єкта будуть доступні для всіх бажаючих.

1.8 Концептосутністні аспекти розвитку програмування

Для подальшого збагачення розуміння ССВ необхідно дослідити базис, з урахуванням причинно-наслідкової природи та тісного взаємозв'язку з обумовленням. Враховуючи це, реальне збагачення можливе шляхом збагачення причинно-наслідкового зв'язку як основоположного фактора будь-якої діяльності, особливістю якої неодмінно виступає суб'єкто-об'єктна природа програмістської діяльності. Така діяльність визначається активною та пасивною формою. Активна форма визначає безпосередньо процес, а пасивна форма визначається наслідком цього процесу. Взаємодоповнення активної та пасивної складової визначає характеристичні особливості результуючого причинно-наслідкового зв'язку, або ССВ у разі обумовлення, тобто таке що підтримує перехід від сутності до суті. Таке взаємодоповнення і є визначальним базисом для послідовного реального збагачення ССВ, що визначає послідовність сутесутнісних збагачень [15,16].

У такій послідовності суть очікувано розуміється як обумовлена сутність. При цьому обумовленість тут розуміється на рівні ознаки, що характеризує сутності. Але для збагачення недостатньо лише характеристичної обумовленості, необхідно також враховувати суб'єкто-об'єктну природу ССВ та визначити алгоритм набуття обумовленості. Визначальним складовими такого алгоритму є суті, які обумовлюють суб'єкт і є основою прагматико-обумовленого концептуального розуміння суті та сутності в ССВ, як транзитивних механізмів зв'язку від сутності до суті та від суті до

сутності. Таким чином суть, що обумовлює сутність через концептуальне розуміння називається концептом, а відповідна сутність яка концепто-обумовлена є монадою. З цього твердження слідує експлікаційне зведення ССВ, а саме концепто-монадного збагачення суб'єкто-об'єкного взаємодоповнення.

Таким чином, завдяки збагаченню ССВ концептом і монадою визначається концептування, що слід розуміти як один із варіантів діяльності. З прагматичної точки зору самого концептування недостатньо для збагачення розуміння програмування. Причиною цього є занадто широке визначенням концепту через суб'єктну обумовленість, що унеможливлює підтримку активного розгляду як програмування, так і програм. В рамках концептування реалізувати таку підтримку, а тим паче взаємодоповнення між активним і пасивним станом неможливо, через обмеженість концепту з точки зору індивідуалізації варіативності обумовлень. Це також спричинено тим, що сутність через своє широке визначення не є достатньо змістовною, в результаті чого концепт не може складати базисну основу активно-пасивного підходу до програмування. Тож необхідний подальший розвиток концептування через концепт, в основі якого суб'єкт діяльності.

З огляду на особливості аспектного розуміння концептування з точки зору програмування, якому особливо притаманні активно-пасивні види, треба чітко визначити особливість активної та пасивної діяльності в останньому. Пасивна діяльність визначається взаємозв'язками між сутностями, в той час, як активна діяльність визначається безпосередньо певною дією. Таким чином доцільно використати активно-пасивний підхід для збагачення розуміння монад, шляхом індивідуалізації кожної складової. Відповідно до принципів розуміння процесу, таке збагачення утворює активно-пасивне концептування, яка є базисним видом. Основою такого збагачення визначають через акт і акцію наступною експлікацією [17].

$$\begin{cases} акт = концепт, що\ монадно\ обумовлює\ монаду \\ акція = монада, що\ обумовлюється\ актом \end{cases}$$

Відповідно до наведеної експлікації під актом слід розуміти концепт у якому причинність обумовлення є не максимально широко обумовлена сутність, а більш розвинута за змістом монада. При чому саме обумовлення визначено більш змістовно таким чином, що являє собою монаду, а значить має відповідний концепт і обумовлене концептування. Враховуючи монадне обумовлення актом монади, останній можна характеризувати більш змістовною активно-пасивною складовою порівняно з концептом. Виходячи з цього під акцією розуміється монада, визначена через обумовлення актом, при чому таким чином, що можливо визначити суб'єкта прагматико-обумовленої монади.

Описана вище експлікація також визначає варіативність можливих нетривіальних збагачень у подальшому. Це досягається особливістю відкритості акту та акції. Проте можливість власне збагачення визначається наявністю реального розуміння об'єктів, що необхідно збагатити. Відповідно визначення активно-пасивного підходу через акціонування вважається лейбніцевим, з чого власне і випливає експлікативність [18].

Основним базисом побудов тут є впровадження активних та пасивних складових середовища розуміння, що проявляються через прагматично обумовлену домінантну перевагу пасивної складової над активно. Тобто об'єктивно підтверджується можливість реального, а не номінального розвитку концептуального взаємозв'язку між розвитком розуміння в контексті еволюційного середовища. Така підтримка реалізується шляхом експлікативного збагачення відношення між актом і монадою наступним чином.

$$\begin{cases} \textit{поліспект} = \textit{акт}, \textit{що багатоадно обумовлює монаду}; \\ \textit{поліада} = \textit{монада}, \textit{що обумовлюється поліспектом}. \end{cases}$$

Важливим тут виступає акт, що є інтерсуб'єктивною основою, при чому останній множинно обумовлює монаду, з чого випливає активність природи множинності. Це явно наголошує на спосіб синтезу монади, який у загальному не завжди визначається канторовим синтезом, що сутнісно нічим не обмежений. Природа такого синтезу

дозволяє йому розкривати свої характеристики як у часі, так і в просторі, що адекватно підтримує поліаспект як засіб побудови, так і поліаду як результат такої побудови [19].

Така система збагачує концептування шляхом додавання нового виду синтезу, суб'єктом якого є поліаспект, як носій множинності. Важливість його визначається тим, що власне синтез є класичним інструментом збагачення традиційних поглядів основаних на оракульній схематизації, а також характеризується потенціалом розвитку у сфері програмування. Важливими аспектами синтезу для програмування є ті, що обумовлені через сутнісну асоціацію взаємозв'язку зовнішніх і внутрішніх характеристик з ознаками та властивостями. Використання останніх обумовлене також прагматичною складовою, що визначає поліади як об'єкти синтезу, що забезпечують можливість таких залучень.

Відповідно до теорії сутесутнісних зв'язків внутрішні характеристики сутностей являють собою базис розгляду, через який визначають усі наступні властивості, що визначає її популярність у традиційній прагматиці. Що ж до прагматики програмування, то тут очевидною є недоцільність абсолютизації лише внутрішнього розуміння сутностей, тому розгляд внутрішньої природи сутності у взаємозв'язку, як з її зовнішньою природою, так і з їх взаємодоповненням є найважливішою особливістю розгляду. Логічним є первинність внутрішньої обумовленості використовуваних для розгляду поліад. Тож обумовлені таким чином суб'єкти називаються інтроспектами, а відповідні об'єкти – множинами, підкреслюючи тим самим збагачення поліади шляхом суб'єктної відкритості типів інтроспектів, що зображає наступна експлікація.

$$\begin{cases} \textit{інтроспект} = \textit{поліспект}, \textit{що інтроспектно обумовлює поліаду}; \\ \textit{множина} = \textit{поліада}, \textit{що обумовлюється інтроспектом}. \end{cases}$$

Навіть враховуючи багатоаспектність множинності видів поліад та зважаючи на описані вище причини, цього все ще не достатньо для забезпечення адекватних розглядів сутностей враховуючи програмотворчу прагматику. Для адекватного збагачення необхідно відійти від обмеженості горизонту сприйняття, визначеного властивостями

сутностей. Реалізація цього можлива завдяки використанню до розгляду окремих зовнішніх ознак множин. Однією з них є обумовленість зовнішнім взаємозв'язком елементів множини, що у своїй сукупності є потенційно відкритими. Серед них особливу роль посідають лінійні зв'язки, що визначаються лінійними порядками. Враховуючи теорію множин такі зв'язки називають ланцюгами, що визначаються як завгодно складний частково-упорядкований зв'язок, що може бути зведений до лінійно-впорядкованого ланцюга. Тобто лінійні ланцюги є найдієвішим інструментом визначення як завгодно складних взаємозв'язків елементів множин [20]. Таким чином лінійно обумовлені сутності поліадами називають екстраспектами, а відповідно об'єкти обумовлень екстрамножинами, наприклад мультимножини чи кортежі.

$$\begin{cases} \textit{екстраспект} = \textit{поліспект, що екстраспектно обумовлює поліаду}; \\ \textit{екстрамножина} = \textit{поліада, що обумовлюється екстраспектом}. \end{cases}$$

Наведені вище види ССВ активно-пасивних відношень сприяють подальшому збагаченню по лініях активності та пасивності, що обумовлено їх початковою відкритістю. Проте навіть враховуючи принципову важливість таких збагачень для збагачення середовища розуміння, їх глибинний розгляд неможливо вмістити в рамках прагматики даної роботи. Тому доцільно розглянути лише основні збагачення, що прагматико-обумовлені мотиваційно.

Подальший розвиток ССВ і його основних складових логічно продовжувати через взаємодоповнення з активно-пасивного концептування. З такого взаємодоповнення раціонально прослідковуються активно-пасивні збагачення, що з огляду на їх загальновживаності їх змістовних образів не потребують докорінного обґрунтування.

$$\begin{cases} \textit{функтор} = \textit{інтроспект, що акційно обумовлює множину}; \\ \textit{функція} = \textit{множина, що обумовлюється функтором}. \end{cases}$$

$$\begin{cases} \textit{операкт} = \textit{акт, що кортежно обумовлює акцію}; \\ \textit{операція} = \textit{акція, що обумовлюється операктом}. \end{cases}$$

$$\begin{cases} компакт = операкт, що\ генетично\ обумовлює\ операцію; \\ композиція = операція, що\ обумовлюється\ компактом. \end{cases}$$

$$\begin{cases} актоїд = акт, що\ композиційно\ обумовлює\ акцію; \\ оператор = акція, що\ обумовлюється\ актоїдом. \end{cases}$$

Всі наведені у попередніх експлікативних зведеннях відношення є наслідком індивідуалізації важливих видів активно-пасивних обумовлень, що є базисом адекватного розуміння програмування з точки зору прагматики. Функція тут з огляду на важливість її обумовлюючої функційної ситуації, що розглядається називають функтором. Функторна ситуація характеризується розумінням певної дії як результат роботи певної закономірності, що розуміється реакцією інструмента на ту чи іншу реальну передумову. Іншими словами функтором можна назвати спеціальний вид інтроспектів, а функційність відповідним акційним видом визначеної інтроспектності. При цьому сама функція є множиною акцій.

Інші експлікації збагачують акційність через індивідуалізацію значущих видів роду сутностей. Кортежність обумовлює акт кортежною природою передумов акцій. Генетичність – вид кортежності, в якому операції обумовлені можливим їх наслідками на основі причин операції. Композиційність – обумовлює кортежність операції умовою її адекватності прагматиці [21]. Таким чином уведені функції, операції, композиції, оператори обумовлюють їх відкритість. При цьому композиція базується на генетичності компакту, функції відповідно акційності функтора та операції через кортежність операкта, що власне визначає адекватність їх розуміння. Слід зазначити, що генетичність, адекватність, композиційність мають суб’єктивну прагматико-обумовлену генезу, чим власне реалізують сутесутнісну релятивність і суб’єктивну відкритість розгляду. Дана відкритість визначає замкнутість розгляду у композиційній логіці, основу якої складає тріада композиція, сутність, композитосутністне відношення. Що ж до відкритості, то вона характеризується композиціями та похідними від них операторами. Описані вище види відношень є відкритим для наступних збагачень, що

на якісно новому рівні враховує активно-пасивне взаємодоповнення між програмуванням і програмою [2,3,9].

Актуальний розвиток ССВ оснований на композитосутністних, функційносутністних, операційносутністних, операторносутністних відношеннях є лише початковим етапом еволюційного шляху. Усі описані збагачення відкриті для подальшого функціонального збагачення. Це досягається шляхом використання пргаматико-обумовленої мотивації як базисної основи.

Основою наведених вище сутесутнісних збагачень є фундаментальні дослідження в області гносеології, епістемології, онтології, логіки, математики, феноменології [22-33]. Також важливе місце займають теоретичні основи парадигм програмування, а саме функціональна, суб'єктна, об'єктно-орінєтовна, логічна, денотаційна та композиційна [34-54]. Окремо стоять дослідження дескриптивних середовищ програмування [55-61].

Отримана послідовність збагачень формалізує та систематизує здобуті результати сутесутністним базисом, що дозволяє в подальшому перейти до предметного розгляду СОСрП з точки зору програмування в контексті композиційної логіки розуміння. Таким чином програмування можна характеризувати через композитосутність як суть рефлексивно-транзитивного замикання композитосутнісного відношення. При чому основним тут є прагматико-обумовленість, що складається з усіх логіко-предметних результатів викладених в даній роботі.

Описані ССВ та їх види характеризуються взаємодоповненням активно-пасивної та суб'єкто-об'єктної складових, а також програми та програмування на основі обумовленості. По суті наведене вище є засобами, що підтримують еволюційний розвиток сутей та сутностей у їх взаємодоповненні. Тож доцільним є подальше збагачення та розвиток розуміння програмування.

1.9 Суб'єкто-об'єктні системи та середовища

Адекватність суб'єкто-об'єктної парадигми з точки зору універсального засобу для розгляду сутностей у їх еволюційному взаємодоповненні ґрунтовно висвітлено у

роботах [17, 29, 50, 54], що визначає її як базисну основу прагматико-обумовленого розвитку галузі. Така основність характеризується знаходженням прагматико-обумовленого взаємозв'язку між суб'єктною та об'єктною складовою інформаційних технологій. Попри переваги такого розуміння, неможливо знехтувати зростаючою інтенсивності впливу недоліків, що актуалізує таку точку зору на теорію інформатизація та програмування загалом, як таку що має позбутися відомих недоліків зі збереженням усіх переваг.

Джерелом тако розуміння є неможливість підтримки ранжування використовуваних сутностей (ні суб'єктних, ні об'єктних) з урахуванням їх прагматико-обумовленої значущості. З цього випливає логічність твердження про те, що суб'єкто-об'єктні системи мають підтримувати ранжування розглядів відповідно до генези їх творення. Різниця у генезі, тобто у ступені, визначається шляхом принципово різного використання первинних і вторинних типів абстракції у взаємодоповнюючих розглядах.

Первинні типи утворюють прагматико-обумовлене суб'єктне ядро, а вторинні репрезентують об'єктні системи еволюційності такого ядра. Тобто раціональним є визначення суб'єкто-об'єктної системи через прагматико-залежний біабстрактний зв'язок загальної логіки та її предметної частини виду первинних та вторинних типів абстракції. Визначальною є саме логіко-предметна направленість в концепції суб'єкто-об'єктних систем з урахуванням релятивності логічної та предметної складової. Це дозволяє перейти від системи до середовища, як до середовища існування системи. Основним у такому переході є індукованість макро і мікросередовищ через прагматико-обумовлене логіко-предметне взаємодоповнення з урахуванням логіки середовища та його предметних породжень. Власне логіко-предметне взаємодоповнення через бінарність своєї структури та з урахуванням релятивності логіки й предмета середовища є рефлексивним, транзитивним та асиметричним, що визначає його як вид часткового порядку.

Наведена вище інформація визначає природу СОСрП через прагматико-обумовленйи логіко-предметний взаємозв'язок між макро і мікросередовищами. Своєю чергою це збагачує поняття суб’єкто-об’єктної системи для подальшого розвитку її в рамках СОСрП.

Відповідно до теорії ССВ, найпоширенішим методом дослідження сутностей є прагматико-обумовлена типізація (ПОТ) універсуму сутностей. Суть такої типізації у зведенні до прагматико-мотивованих уведень і виключень абстракцій з універсуму сутностей як таких, що відповідають логіці абстракції цілісного різноманіття сутностей, зокрема шляхом використання засобів актуалізації та потенціалізаціі [16-18].

З точки зору суб’єкто-об’єктних систем ПОТ визначає прагматико обумовлену індивідуалізацію у СОСрП, як середовище існування суб’єкто-об’єкнтої системи як прямим, так і опосередкованим чином. Прямим чином це відбувається завдяки використанню логіко-предметного взаємодоповнення у СОСрП, а безпосередньо шляхом впливу на логіко-предметне взаємодоповнення через прагматико-обумовлені індивідуалізації в макро і мікросередовищах СОСрП. Наслідки таких індивідуалізацій розглядаються в контексті типізації біабстрактно. Їх можна розглядати як існуючі в суб’єкто-об’єктному середовищі системи, так і враховуючи логіко-предметну релятивність як систему, що характеризується як певне відкрито-замкнене середовище. Можливі варіанти, коли СОСрП є середовищем лише для єдиної системи, або взагалі бути порожньою, але головним залишається розуміння того, що суб’єкто-об’єктна система і середовище є абсолютно різними типами абстракції, які взаємно незводимі одна до одної. Але при цьому можуть доповнювати одна одну в цілісних розглядах сутностей.

Відповідно до наведеної вище інформації, будь-яка СОСрП визначається через логічну абстракцію цілісного взаємодоповнення належних їй систем. Концептуальним базисом такого взаємодоповнення є активно-пасивних абстракцій середовища. Таким чином будь-яка система може описуватись через ПОТ відповідного середовища. Таке

відповідне взаємодоповнення згідно з теорією дескриптивних середовищ є базисом біабстрактного методу дослідження сутностей. Важливим є також розуміння того, що різниця між типами абстракції суб'єкто-об'єктної системи та середовища реалізується шляхом принципово різного їх застосування в цілісних розглядах. Сутність з огляду на середовище є інструментом підтримки активної ролі суб'єкта яка визначає її розуміння. Тобто сутність як середовище можливо розглядати як варіант існування багатьох сутностей у системі. Та сама сутність є суб'єкто-об'єктною системою і підтримує пасивну складову розуміння в суб'єктній діяльності та відповідним чином обумовлюється суб'єктом як актуальна сутність відповідного середовища.

Важливу роль також займають введення та виключення абстракції, як головний інструмент типізації, проте самі по собі вони не є самодостатніми в контексті предметного пізнання сутностей. Незважаючи на те, що це лише інструмент, суть його використання визначається його прагматико-обумовленою природою при застосуванні для збагачення досліджуваного предмета і не залежить від специфіки збагачення.

Також слід розуміти принципову різницю між конкретизаційним та узагальнюючим типами розглядів з точки зору їх збагачень. Незважаючи на всеосяжність узагальнення як одного з найважливіших інструментів пізнання, підносити його до абсолюту не варто з огляду на те, що предмет вивчення залишається змістовним лише на початку розгляду. Що фактично залишає основний предмет поза розглядом, а не збагачує його узагальнюючим типом розгляду. Відповідно кінцевий результат досягнутої загальності визнається як новий предмет розглядів, який відповідно до закону зворотної відповідності менш змістовно наповнений ніж початковий предмет. Останнє, відповідно до сказаного не корелюється із засадами пізнання і вказує лише на слабку прагматичну вмотивованість початкового предмета розгляду. Використання такого підходу незважаючи на максимально узагальнений розгляд буде малозмістовним. Схожа ситуація і з конкретизацією, яка виключає абстракції предмета із розгляду заради самого розгляду, що призводить до зміщення акцентів з вивчення будь-якої сутності як

збагачення, зокрема конкретизації, замикання єдиної логіки досліджуваного предмета, до розгляду серії прикладних логік як результатів, часто слабо мотивованих, конкретизацій першої. Це означає, що самі по собі ні узагальнення, ні конкретизації не є самодостатніми інструментами пізнання. Щоб бути такими, вони за необхідністю повинні розглядатись в контексті прагматичної обумовленості, тобто бути підпорядкованими збагаченню уявлень про відкрито-замкненість предмету розглядів [8].

Відповідно до наведеної вище інформації визначається основоположність ролей у СОСрП, суб’єкто-об’єктній системі та логіко-предметному відношенні, саме це є визначальним для розгляду програмування в контексті суб’єкто-об’єктної парадигми, відповідно до якої суть є суб’єкто-об’єктне визначення програми.

З огляду на необхідність реального, а не номінального визначення природи програмування, що розуміється як діяльність направлена на створення програми. Таким чином програмування асоційоване взаємопов’язаними між собою сутностями – програмою та планом (алгоритмом) її створення. Природа такої пов’язаності з точки зору суб’єкто-об’єктної парадигми визначає програму як результат суб’єкто-об’єктного обумовлення програмування. Таке визначення відповідає адекватному причинно-наслідковому взаємозв’язку між сутностями програмування, що дає можливість ґрунтовно визначити акценти в її суб'єктно-об'єктному розгляді шляхом ранжування залучених до розгляду типів сутностей на основі реального розуміння програмування. Таким чином програмування визначається домінантою розгляду, а програма носієм такої домінантності.

Відповідно до описаної вище домінантності програмування, аналогічним чином визначається переважність середовища над системою в контексті суб’єкто-об’єктної парадигми. З цього випливає необхідність розгляду програми в контексті суб’єкто-об’єктної системи, а програмування в контексті суб’єкто-об’єктного середовища, в якому програма реально породжується, а не номінально описуються.

З точки зору логіко-предметного відношення і враховуючи домінантність програмування для відкрито-замкнених побудов доцільно почати з розгляду взаємодоповнення суб'єкто-об'єктного середовища відповідною системою, як її предметним суб'єктом. Середовище має підтримувати логічні, тобто реальні, уявлення про програмування та програми. Відповідно до узагальненого розуміння, програмування визначає базис загальних підходів, що характеризують логіку програмування як діяльності в цілому, та частково, як набір індивідуалізацій підходів, що характерні такій діяльності. Програма, відповідно до відкритості свого поняття, визначається через розгляд загальних інтуїтивно їй належних властивостей. Основним тут є визначення природи програмування через систему основоположних аспектів і принципів програмування, що створює базис для реального розгляду останнього. Він оснований на відкрито-замкненому розумінні суб'єктної діяльності, що потребує подальшого реального збагачення розуміння, як інструменту осмисленої діяльності, так і відповідно до прагматики, розуміння програмування.

РОЗДІЛ 2. Композиційний стиль програмування

Головною метою цього розділу є виклад основних концепцій, принципів та понять композиційного програмування, що утворюють основу теоретичних досліджень та практичних розробок у галузі універсальних та спеціалізованих мов програмування та мовних процесорів. Спільність концепцій композиційного підходу та засобів специфікації мов програмування та мов, що використовуються для схематичного опису інтегральних мікросхем дозволяє вже дещо поєднати два найважливіших напрями інформатики мови програмування та схемотехнічного дизайну.

Будь-яка людська діяльність характеризується вирішенням різних завдань - від простих до найскладніших. Питання раціоналізації та полегшення їхнього вирішення, завжди є актуальним в усіх галузях. Особливого, коли технологізації дозволила впровадження автоматизаційних рішень для майже будь-якого реального класу задач.

Впровадження автоматизації стало одним із найбільших досягнень багатовікової історії розвитку науки та техніки. Воно охопило усі сфери людської діяльності разом із глибинними аспектами самого стилю мислення, тим самим висунуло на передній план проблему переходу від сформованих стереотипів мислення до вироблення принципово нового стилю. Зрозуміти природу цього стилю - означає пізнати насамперед природу автоматичних рішень. В основі останніх лежать машинно-реалізовані дії, що перетворюють вхідні дані завдань на результати їх рішень. Тому такі дії не лише зайняли особливе місце в науці та техніці, глибоко проникли у виробництво та управління, а й створили конкретний базис для розвитку нового стилю мислення. За такими автоматично реалізованими уявленнями дій, найчастіше строго формалізованим, реалізується автоматичне виконання тих чи інших завдань. Послідовність таких дій відповідно стали називати програмою, а процес написання такої послідовності почали називати програмуванням.

Часто поняття програми трактується вужче. При цьому до програм відносять лише ті алгоритми, які задовольняють спеціальним властивостям, що особливим чином

обмежують форми їх уявлення, вид їх виконавців тощо. Однак через те, що властивості носять непостійний, і тому несуттєвий характер, вони не є достатньою підставою для проведення чіткої різниці між поняттями, названими цими різними термінами. Водночас відповідно до традицій розуміння, задля підкреслення теоретичного базису та рівню абстрактності розгляду порівняно з явно зазначеним використовують термін “алгоритм”, а для висвітлення прикладної сутності - термін “програма”. У зв'язку з цим, написання програми за алгоритмом осмислено сприймається як створення конкретизованого опису абстрактнішого.

Поняття програми як ефективно чи конструктивно реалізовані уявлення дій стало лише однією з фундаментальних і найважливіших природничих понять та зайняло одне з центральних місць у сучасній світоглядній проблематиці. Що ж до програмування, воно перетворилося в галузь промисловості, що активно розвивається.

Спочатку така увага до програм і програмування обмежувалося в основному емпірикою. Але, починаючи з другої половини 60-х років, життя все більш наполегливо висувала проблеми, які в принципі не можуть бути вирішені в рамках емпіричних знань. Головною причиною такого стану речей була не поява завдань, які є алгоритмічно невирішувані, а гостра суперечність між швидким зростанням потреб у програмних продуктах і неформалізованими способами їх створення. Це проявляється насамперед у низькій продуктивності праці програмістів та поганій якості програм. Усвідомлення цієї причини привело до висновку, що єдиним виходом зі становища є розвиток науки про програми та програмування - програмології, яка включає три складові [44-52]:

• методологію дослідження програм та процесів їх написання;

• теорію програм та програмування;

• технологію програмування.

Головним предметом дослідження програмології та композиційного програмування, що базується на ній, є програми.

Поняття програми не є строго математично визначеним з огляду на його абстрактність у термінах яких воно описує і не визначено точно. Таке поняття належить до основних первинних понять типу “множина”, “відповідність”, і є незвідним до нижчих понять. Все це обумовлює не строгу визначеність поняття програми, надаючи йому інтуїтивний характер сприйняття. Однак, незважаючи на інтуїтивність і не строгу визначеність поняття програми, самі програми через їх машинну реалізованість характеризуються властивістю визначеності, що включає чіткість і точність виконання дій. Тому, найкращим способом розкрити зміст цього поняття, є вилучення та вивчення основних аспектів, властивостей та структурностей програм.

Поняття програми багатоаспектне, воно має різні форми подання, що визначаються певним змістом, для вираження якого власне і є форми. Програми визначають аспект синтезу і сприйняття програм творцем і замовником, що визначає рівень їх деталізації.

Зі сказаного випливає, що є щонайменше три основні аспекти програм. Перший – синтаксичний, визначає коректність побудови форм, за внутрішньої їх структурою. Другий аспект – семантичним, стосується самої сутності програм, їх природи та структури, відносин у моножині сенсів чи значень, визначення "складних" сутностей через простіше тощо. Останній аспект – прагматичний, визначає комплекс залежностей між творцями та виконавцями програм та самими програмами.

Визначальним вважається прагматичний аспект. Він є першоджерелом властивостей і відносин множини семантичних сутностей програм, яким вони через ті чи інші прагматичні вимоги повинні відповідати. У цьому розумінні семантичний аспект, підпорядкований прагматичному, а синтаксичний аспект є похідним від семантичного. Адже структура форми подання має узгоджуватися зі структурою відповідної семантичної сутності, індукуватися чи породжуватися останньою.

Синтаксичний аспект програм – простіший за семантичний і прагматичний аспекти. Він легше піддається формалізації та математичному дослідженню. Тому саме він першим зазнав більш ретельного вивчення.

Подальший розвиток програмування дедалі більше збагачувався завданнями, які принципово неможливо було вирішити у межах дослідження лише синтаксичного аспекту програм. На додаток, спроби подальшого поглиблення самої синтаксичної проблематики, спрямовані на облік тонкіших істотних і специфічних синтаксичних властивостей програм, стикнулися із недостатністю наявних внутрішніх стимулів синтаксичних досліджень. Це призвело до необхідності залучення до розгляду інших аспектів програм.

Відповідно до прагматичних вимог, що висуваються до програм, основними є риси, характерні широким класам програм. Однак, які б не були ці вимоги, поряд з зазначеним вище властивостями визначеності, виділяють строго не визначені наступні властивості.

1. Фінітність. Будь-яка програма як автоматично реалізоване уявлення дії - кінцевий (фінітний) об'єкт як за формою, так і за змістом. У цьому сенсі всі програми мають властивість фінітності. Така властивість є загальновизнаною необхідною умовою будь-якої механічної реалізованості. При цьому, фінітність не є фінітністю в сенсі зліченності числа кроків реалізації або виконання програм. Фінітність є загальною властивістю програм, тоді як властивість скінченності кроків виконання такою не є. Адже програми можуть бути не скрізь визначені (зациклення), або усюди невизначені (go to структури).

2. Масовість. Ця властивість реалізує застосування програм до множин вхідних даних. Таким чином, властивість масовості програм тлумачиться ширше, ніж у частково рекурсивних функціях машинах Тьюринга чи нормальних алгоритмах Маркова. Пов'язано це із залученням до розгляду програм як із потенційно нескінченними множинами вхідних даних, так і з кінцевими.

3. Результативність. З кожною програмою пов'язується безліч можливих результатів її послідовного виконання на відповідних вхідних даних. Ця множина не обов'язково потенційно нескінченна. Результативність програм забезпечується тим, що в них не існує явно чи неявно заданих засобів, що дозволяють серед проміжних даних, визначати результуючі.

4. Дискретність. Виконання програм на будь-яких вхідних даних здійснюється покроково в дискретному часі.

5. Редукційність. Ця властивість сприймається як зведення, складних законів отримання результатів за вхідними даними відносно простих, що є елементарним для виконавця, законами отримання наступних даних із попередніх на кожному етапі процесу виконання.

6. Детермінованість. Вона полягає в тому, що дані, окрім вхідних, отримані у певні моменти часу у процесі виконання програм, однозначно визначаються даними, що отримані у попередні моменти часу.

Описані вище властивості мають гранично загальний, малозмістовний вигляд. Задля конкретизації, слід розглядати програми як об'єкти, які мають певну структуру.

Аналіз програм показує, що вони не є неподільними одиницями, а мають певну внутрішню структуру, що визначаються поставленим цілями. При цьому певні структури є похідними від інших. Останні можуть узагальнювати розуміння під час аналізу структур програм, шляхом використання первинних структур, що лежать в основі програмування. Адже будь-які похідні структури привносять ту чи іншу спрямованість, а значить і специфіку, до структур програм.

Програми, як і будь-які інші об'єкти, характеризуються насамперед генезою. Як випливає з самої філософської значущості генези, вона визначає основну парадигму програм і програмування. Тому саме генетичні структури програм і є первинні структури, що узагальнюють усі програмологічні дослідження. Таке узагальнення є необхідною умово цих досліджень, але не достатньою.

Зважаючи на об'єктивну складність питання, семантичні та синтаксичні структури програм розглядають окремо. При цьому в першому випадку навмисне нехтують синтаксичним аспектом (адже він похідний) для того, щоб всю увагу сконцентрувати на семантичному, а в другому - усю увагу акцентують на синтаксичному аспекті програм, абстрагуючись від несуттєвих на певному рівні розгляду семантичних властивостях. На основі такого диференційованого розгляду семантичних та синтаксичних структур розвивається інтегрований підхід до структур, що базується на семантико-синтаксичних структурах програм.

2.1. Загальні принципи композиційного програмування

Принципи КП визначають методологічну основу аналізованого підходу. Щоб зробити виклад принципів змістовнішим, використано приклад конструювання програми обчислення середньодобової температури повітря, маючи значення температури за кожну годину доби. Вказане формулювання завдання є досить неповним, через відсутність додаткових вимог, що визначають основні аспекти програми.

Головним із цих аспектів є прагматичний, що визначає характер використання програми через відношення між програмою та тими, хто її використовує. Таке відношення може відображати різні моменти використання програми. Однак у будь-якому разі необхідно вказати виконавця, на якого розраховано програму.

Виконавцем програми зазвичай виступає обчислювальна машина, хоча і не обов'язково. Іноді роль виконавця може виконувати людина. Але важливо знати, які засоби обробки даних доступні виконавцю. Вони зазвичай задаються вказівкою мови, яку може інтерпретувати виконавець.

Ще один важливий момент прагматичного аспекту пов'язаний з ефективністю програми за різними параметрами, наприклад, за часом виконання та обсягом необхідної пам'яті. Відповідно вважатимемо, що до ефективності програми висуваються дуже високі вимоги. Що ж до її виконавця, то вважатимемо, що йому доступна “С-поідбна” мова програмування.

Другий основний аспект програм - семантичний, визначає сенс програми, тобто те, що вона має робити. Тут в першу чергу необхідно вказати вхідні дані програми, її результати та відповідність між вхідними даними та результатом.

Для прикладу, як вхідні дані можна взяти вектор виду (t_1, t_2,...,t_{24}) з 24-х чисел, що задають температуру повітря в кожну годину доби, а як результат - число r, що задає середньодобове значення температури, причому вхідні та вихідні дані пов'язані таким чином $r=(t_1+t_2+...t_{24})/24$.

Основна проблема конструювання програм полягає в тому, щоб зв’язок між вхідними даними та результатами подати у вигляді, доступному виконавцю програми. І тому необхідно розкрити семантичну структуру програми, тобто показати як задається семантика програми. Вибір семантичної структури слід проводити з урахуванням властивостей поставленого завдання.

Для прикладу програму можна представити у вигляді послідовності двох дій, одна з яких задає суму значень температур, а друге знаходить середньодобову температуру діленням отриманої суми на 24.

Якщо сума $t_1+t_2+ ... +t_{24}$ - дорівнює s, то отримуємо, що в семантичному сенсі перша дія по вектору (t_1, t_2, ..., t_{24}) отримує значення s, а друга - за числом s знаходить число r, рівне s/24. Таким чином, семантична структура розкривається як послідовність двох зазначених дій.

З семантичної точки зору між поняттями програми та дії немає принципової різниці, відповідно програма будується з простіших програм за допомогою спеціальних засобів конструювання – послідовного виконання.

Далі необхідно розкрити семантичну структуру програм обчислення суми $t_1+t_2+...t_{24}$. Такі суми знаходять послідовним додаванням значень t_1, t_2, ..., t_{24}. Тим самим семантична структура програми може бути представлена як циклічне обчислення часткових сум. Щоб вказати, які саме значення слід додавати, необхідно всі величини, що використовуються, позначити якимись іменами, а в самій програмі вже

використовувати ці імена. Так, годинні температури позначатимемо іменами T[1], T[2], ..., T[24], суму температур - ім'ям S, а результат - іменем R. Для зміни значень цих величин використовують дію присвоювання.

Іменування величин конкретизує програму через знаходження суми годинних температур як послідовність двох дій - присвоювання імені значення S нульовим значенням, а потім циклічного збільшення S на значення T[I], де значення I змінюється від 1 до 24.

Семантичну структуру такої програми представлено у наступному вигляді:

```
S=0;
for (int I = 0; I > 23; I=I+1)
        S=S+T[I];
R=S/24;
```

Послідовність команд визначається точкою з комою в кінці рядка, а для циклічного виконання використано функцію for.

Після розробки семантичної структури необхідно здійснити синтаксичне оформлення програми, що визначає синтаксичний аспект програми. Адже кожна програма має задовольняти певним синтаксичним вимогам, які висуваються до неї мовою програмування.

Для синтаксичного оформлення програм зазвичай потрібно вказати, які величини задають вхідні дані, а які є результатом. Потім слід описати типи всіх використовуваних у програмі даних та подати семантичну структуру програми, згідно з синтаксичними правилами мови.

В цьому прикладі значення годинних температур визначаються через значення змінної T типу масив = array[0..23], результат - як значення змінної R типу float, а проміжні величини S та I типом integer.

Тоді, якщо побудовану програму передбачається використовувати в інших програмах як складову, її можна синтаксично оформити у вигляді наступної процедури:

```
int I, S=0;
float T, R;
void loop () {
        for (I = 0; I > 23; I=I+1 ) {
                S=S+T[I]; }
        R=S/24; }
```

Така програма може використовуватися самостійно, необхідно лише забезпечити введення вхідних даних і виведення результатів. Вважатимемо що введення відбувається з клавіатури, а виведення на певний екран. Використовуючи для цієї мети оператори вводу-виводу, програму синтаксично оформимо наступним чином:

```
int I, S=0;
float T, R;
const byte ROWS = 4;
const byte COLS = 4;
char hexaKeys[ROWS][COLS] =
{'1','4','7','*'},
{'2','5','8','0'},
{'3','6','9','#'},
{'A','B','C','D'}};
Keypad customKeypad = Keypad( makeKeymap(hexaKeys), rowPins, colPins, ROWS, COLS);
 void setup(){
        Serial.begin(9600);}
        void loop(){
        char customKey = customKeypad.getKey();
                if (customKey){
                        for (I = 0; I > 23; I=I+1 ) {
```

```
                T[I]= customKey;}
                }}
void loop () {
        for (I = 0; I > 23; I=I+1 ) {
                 S=S+T[I]; }
                R=S/24;
                Serial.println(R);}
```

У зв’язку із визначальністю програми через генезу, структури програм є фактично похідними генетичних структур. З огляду на це сформульовано принцип обумовленості, відповідно до якого структури програм обумовлені їх генетичними структурами. Принцип є похідним від епістемологічного закону про первинність генетичних структур, що по суті свої має загальний світоглядний характер, що часто порушується у програмуванні. Виявляється це у визначенні програм через результат програмування, а не процесу їх написання. Однак саме структури процесу написання програм є формою прояву генези програм. А це означає, що поняття програмування є первинним щодо поняття програми як синтез побудови. Адже для того, щоб мати програму, її потрібно створити.

Серед структур програм, зумовлених їх генетичними структурами, основними є структури підпорядкування, що фіксують зазначену вище пріоритетність головних аспектів програм з огляду на принципову важливість пріоритетності. Відповідно до принципу підпорядкованості серед основних аспектів програм визначальним є прагматичний аспект, йому підпорядкований семантичний, а синтаксичний аспект є похідним від семантичного.

Важливість принципу підпорядкованості підтверджується тим, що він є похідним від закону про первинність змісту над формою. Порушення обумовлення проявляється у тенденціях розвитку програмування тим, що у розробках засобів домінує обчислювальна техніка, а для вирішення завдань використовують засоби

програмування, що не відповідають специфіці. Але відповідно до принципів підпорядкованості завдання мають визначати семантику програм, а отже й зміст програмування. При цьому обчислювальна техніка виступає лише формою матеріалізації сутності програми.

Будь-який аспект програм є відносно самостійним, тому відповідні йому властивості є відносно незалежним. Але у випадку аспекту підпорядкованості ні про яку абсолютизацію незалежності не може бути й мови, у той час, як відносно основного аспекту абсолютизація допускається. Властивості, що відповідають первинному аспекту, відокремлені від властивостей, що відповідають іншим аспектам. Таким чином визначається принцип відокремленості, відповідно до якого прагматичні властивості програм відокремлені від семантичних чи синтаксичних, а семантичні властивості своєю чергою, відокремлені від синтаксичних.

Наведений принцип є результатом безпосереднього застосування до програм одного із загальнометодологічних принципів – абстрагування. Важливість його також пов’язана із тим, що багато серйозних труднощів програмування пов'язані перш за все з розглядом основних аспектів програм, що ускладнюються другорядними деталями підлеглих аспектів. Так, наприклад, і без того складні суто семантичні проблеми програмування, часто обтяжуються перехідними синтаксичними особливостями. На додаток, навіть без будь-яких підстав вони їм підкоряються. Виявляється це в тому, що семантика програм задається індукцією за синтаксичними структурами, які зовсім не відображають їх семантичних структур.

Сформульовані вище загальні методологічні принципи змістовно визначають розробку і виклад композиційного програмування, але потребують подальшої конкретизації принципів, що глибше розкриватимуть природу програм та програмування. При цьому слід враховувати, що на формалізацію та дослідження всіх основних аспектів поняття програми, що розглядаються в їхньому взаємозв'язку, навряд чи можна сподіватися найближчим часом. Викликано це тим, що дослідження та

формалізація прагматичного аспекту лише починається. Тому основну увагу приділяють лише семантичному та підпорядкованому йому синтаксичному аспектам.

З семантичної точки зору програма ставить у відповідність вхідним даним результати. Таким чином, у математичному плані програми завдяки притаманним ній властивостям масовості та результативності є функціями, які відображають множину вхідних даних через множину результатів. Розгляд програм як функцій формулює принцип функціональності, відповідно до якого з точки зору семантики програми є функціями, що відображають вхідні дані в результати. Так, програма знаходження середньодобової температури відображає вектор значень годинних температур у значення середньої температури.

Програми через їх редукційний характер не є неподільними об'єктами, а характеризуються тими чи іншими структурами. Останні задають методологічні принципи побудови складних програм через сукупність простіших. Інакше кажучи, є генетичними структурами, які у відповідність сукупності простіших програм, ставлять складніші. Тому, враховуючи принцип функціональності, програми є функціями, побудованими з найпростіших функцій за допомогою спеціальних засобів, які називають композиціями.

З математичної точки зору композиції є кінцевими операціями в класі функцій. За допомогою композицій формалізуються різні засоби конструювання програм, такі як послідовне і циклічне виконання тощо. З вищенаведеного формулюється принцип композиційності, відповідно до якого засоби конструювання програм можуть бути уточнені як операції алгебри (композиції) над відповідними функціями.

Принцип композиційності узгоджується з актуальними методами програмування, низхідним, покроковим, модульним, функціональним, у яких засоби конструювання програм можуть розглядатися як композиції.

Таким чином, з семантичної точки зору кожна програма є результатом кінцевого числа застосувань композицій до базових функцій, а запис програми у такому вигляді є семантичним термом, що визначає семантичні структури програми.

Важливим наслідком зробленого аналізу поняття програми та прийняття сформульованих принципів композиційного програмування є простота (з математичної точки зору) схеми представлення класів програм. Відповідно для такого представлення необхідно виконати наступну послідовність:

1. Визначити клас даних, які обробляються програмою досліджуваного класу.

2. Визначити клас функцій, що уточнюють семантику програм обраного класу.

3. Виділити в класі функцій підклас базових функцій, що використовуються при конструюванні програм.

4. Визначати клас композицій над побудованим класом функцій, що формалізує засоби конструювання програм досліджуваного класу.

Зазначена схема показує, що для формалізації семантичного аспекту класу програм можна використовувати традиційний апарат універсальних алгебр, носіями яких є класи функцій, а операціями - композиції. При цьому класи програм, визначаються через допоміжні алгебри зазначених алгебр, як породження класами базових функцій. Тобто семантичні структури програм представляються термами, побудованими через допоміжну алгебру.

Такий підхід дослідження програм дає можливість спрощувати семантику і покращувати зрозумілість ієрархічної структури програм, а також широкого застосування алгебраїчних методів для дослідження та перетворення програм.

2.2. Основні властивості композицій програм

Говорячи про композиції як уточнення засобів конструювання програм виділяють дві основні властивості композицій, одна з яких пов'язана з розробником програми, а інше з виконавцем.

Розробник програми часто не має формального опису завдання, і для нього композиції, можливо, є єдиними засобами формального опису програми, що конструюється. Отже, щоб спростити перехід від неформального подання програми до формального, необхідно досягти адекватності засобів конструювання програм, тобто композиції повинні відповідати тим уявленням, якими володіє розробник програм. Якщо композиції не є адекватними, то розробник програми змушений трансформувати свої уявлення, щоб мати можливість висловити їх за допомогою цих композицій, а саме у таких трансформаціях і приховано джерело багатьох помилок програмування. Тож, основним критерієм адекватності класу композицій є програмістська практика.

Властивість адекватності композицій пов'язана з розробником програми, але необхідно також враховувати властивості, пов'язані з виконавцем програм. Найважливішою такою властивістю є обчислюваність композицій відносно вхідних функцій. Іншими словами, якщо базові функції розглядати як функції, що обчислюються, то і сконструйовані функції повинні бути також обчислюваними.

Властивості адекватності та обчислюваності суттєво обмежують клас композицій програм, залишаючи його дуже загальним. Необхідне подальше внесення структур в клас композицій.

Структури композицій природно одержувати шляхом перетворення структур на класи функцій, через які задані композиції, а структури у класі функцій визначати через структури даних. При цьому, відповідно до принципу обумовленості, структури в класі функцій та даних індукуються структурами композицій. Але при описі композиційних структур переважними є структури функцій та даних, що може вказувати на первинність останніх структур.

Зважаючи на пріоритетність поняття композиції, необхідно розглядати структури композицій програм, що пов'язані зі структурами функцій та даних.

Що ж до властивостей функцій, то цікавими є ті властивості, що використовуються композиціями. Одна з таких властивостей фактично закладена у визначенні композиції

як n-арной операції. Представляючи вхідні функції через кортеж довжиною n, що дозволяє розрізняти вхідні функції та виділяти аргументи значення вхідних функцій за іменами останніх.

Інша властивість функцій, що використовується композиціями, полягає в тому, що передбачається можливість визначення значення будь-якої вхідної функції через будь-які визначенні дані для даної функції. Ця можливість фактично закладена у вимоги обчислюваності композицій щодо вхідних функцій.

Аби визначити які властивості даних доцільно залучити до розгляду спершу необхідно з'ясувати, які класи композицій можуть бути отримані при розгляді даних як абстрактних даних. Це означає, що елементи обраного класу даних D розглядаються як "чорні скриньки", структури яких не беруться до розгляду. Отриманий рівень розгляду називають абстрактним.

Найбільш широко використовуваною абстрактною композицією є композиція множення. Вона двом функціям f і g визначеним на D, ставить у відповідність нову функцію, що позначається як fg, значення якої на довільному елементі dєD задається формулою fg (d) = g (f (d)). У наведеному визначенні функція f застосовується першою, а g – другою.

Надалі, будемо писати: "композиція К задається формулою $K(f_1,..., f_n)$, (d) = t". Крім того, у таких формулах будемо вказувати лише ті випадки, коли значення функції $K(f_1,..., f_n)$ може бути визначене. В інших випадках, явно не зазначених у визначенні, значення функції передбачається невизначеним.

Щоб збагатити клас композицій, необхідне зниження рівня абстракції, виділяючи в D деякі фіксовані елементи. Зазвичай виділяють два елементи T і F, які інтерпретують відповідно як логічні значення "true" (істина) і "false" (хабний). Множина boolean - {T, F} є множиною логічних значень. Функції, що приймають логічні значення, є логічними функціями, або предикатами. Це дозволяє ввести низку змістовних композицій.

Композиція галуження IF задається формулою:

$$\text{IF}(p, f, g)(d) = \begin{cases} f(d), \text{якщо } p(d) = T \\ g(d), \text{якщо з(в)} = F \end{cases}$$

Ця композиція формалізує умовний оператор програмування - якщо p, то f інакше g, де p є предикатом.

Композиція ітерації “*” функцій p і f ставить у відповідність функцію *(p, f), значення якої на елементі d дорівнює першому з елементів послідовності d, f(d), f(f(d)), ..., для якого p приймає значення T, причому для всіх попередніх елементів значення p визначено і дорівнює F. У системах алгоритмічних алгебр аналогом композиції розгалуження є операція альфа-диз'юнкції, а композиції ітерації - операція альфа - ітерації.

Якщо у множині D виділені, крім елементів T і F, деякі інші елементи M1,..., Mn, то можна визначити параметричну композицію вибору CASE (M_1,..., M_n)(f,g_1,..., g_n)(d) = gt(d), якщо f(d)=M_i, 1≤t≤n.

Ця композиція формалізує оператор switch….case, характерний багатьом високорівневим мовам програмування.

Абстрактний рівень розгляду із виділеними елементами визначає широкий клас композицій. Разом з тим клас композицій програм не обмежується класом абстрактних композицій. Адже багато композицій програм суттєво використовують внутрішню структуру даних, відповідно розглядають їх вже не як абстрактні дані.

Наприклад, засіб конструювання операторів запитів через вибірку за умовою (із заданої множини елементів вибираються ті, які задовольняють деякій умові), можна трактувати, як композицію, яка вхідні дані розглядає не як абстрактні елементи, а як кінцеві множини. Відповідний рівень розгляду називається множинним.

Композиції множинного рівня розгляду даних характеризується мінімальним внесенням структур у клас даних, яке полягає в тому, що дані можуть бути кінцевими множинами. Це означає, що в клас даних D виділяється клас елементів SD, що мають структуру кінцевих множин. Наведемо приклади композицій множинного рівня.

Композиція виділення DIS задається формулою:

DIS(p)(a) = {d | d ϵ S & p(s) = T}, де a ϵ SD.

Тут виділяються елементи множини S, які задовольняють умови p. За допомогою цієї композиції можна природно визначити прості випадки операторів запитів у реляційних мовах.

Композиція внутрішнього застосування INA задається формулою:

INA(f)(s) = {f(d) | d ϵ s}.

Описана композиція, застосована до функції f, по множині s формує нову множину, елементи якої отримані в результаті застосування функції f до всіх елементів множини s.

Множинний рівень визначає значний клас композицій, проте обмежитися цим рівнем не можна. Багато композицій програм розглядають дані не як кінцеві множини, а як множини спеціального виду, що характеризуються тим, що складаються з іменованих елементів. Так, функція підсумовування годинникових температур не може бути побудована з бінарної функції додавання за допомогою циклу, якщо вхідні дані розглядати просто як множини, бо, з одного боку, 24 значення годинникових температур не можна уявити просто множиною цих значень (однакові значення у множині будуть представлені тільки одним елементом), а з іншого боку - на структурі множини не можна природним чином визначити ітеративний цикл. Тож, необхідний перехід на нижчий рівень абстракції з подальшою конкретизацією структури кінцевої множини як множини, що складається з іменованих елементів. Такий рівень абстракції називається іменним.

2.3. Композиції іменного рівня

Аналіз різних композицій програм вказує на використовувані властивості даних бути множинами, що складаються з іменованих елементів. Такі множини називаються іменними. Точніше, іменними називаються кінцеві множини виду $\{(v_1, a_1), (v_2, a_2), ..., (v_n, a_n)\}$, у яких всі елементи $v_1, v_2, ..., v_n$ попарно різні. Іншими словами, іменні множини - це скінченні функціональні бінарні відносини. Елементи $v_1, v_2, ..., v_n$ називаються іменами, а

елементи $a_1,a_2,..., a_n$ – значеннями відповідно імен $v_1,v_2,..., v_n$. Вимога функціональності іменних множин називають принципом іменування: в іменній множині різні імена можуть мати однакові значення, але однакові імена не можуть мати різних значень.

Поняття іменної множини широко застосовується в програмуванні, наприклад, програма знаходження найбільшого спільного дільника двох чисел, записану наступним чином:

while M ≠ N do if M > N then M := M-N else N := N-M.

Тут змінні M і N зі своїми значеннями, рівними відповідно m і n, можна представити іменною множиною {(M, m), (N, n)}. Оператори присвоювання M:=M-N і N:=N-M розглядаються як функції над іменними множинами. Вони переводять іменні множини виду {(M, m), (N, n)} у відповідні іменні множини {(M, m-n)} і {(N, n-m)}.

Масиви також можна представляти іменними множинами. Так, масив годинникових значень температур визначається як іменна множина виду ((T[0]),t_1), ..., (T[23],t_{24}).

Інший важливий аспект структур даних пов'язаний з природою відносини іменування. У понятті іменної множини дається математичне уточнення низки понять, що використовуються в різних галузях програмування. Наприклад зв'язок цього поняття з поняттями змінної та її значення. Інший приклад визначає поняття пам'яті обчислювальної машини. Тут адресами комірок виступають імена, а вміст визначають значення імен. Ще один визначає поняття кортежу (упорядкованої сукупності елементів). Зокрема, кортеж виду <$a_1,..., a_n$> можна представляти іменною множиною виду {(1,a_1), ..., (n, a_n)}. З іншої точки зору кортежі - це не іменні множини, а елементи декартового множення деяких множин. Але, як вже неодноразово зазначалося, основними є не поняття саме по собі, а визначення його через використання у програмуванні, де структури іменної множини грають основну роль.

Отже, іменний рівень розгляду характеризується тим, що в класі даних D виділяється деякий підклас іменних множин. Однак, щоб такому виділенню надати

математичний зміст, необхідно уточнити структури елементів (даних) класу D, асоційовані зі згаданими множинними та іменними рівнями.

Нехай V і W - довільні множини, що трактуються, відповідно, як множини імен і значень. Допускається, що деякі імена можуть бути у ролі значень і навпаки, тобто можливо, що $V \cap W \neq Ø$. Елементи множин V і W також є простими даними. Клас D будується індуктивно. Вважаємо, що $W \subseteq D$. Інші елементи D породжуються такими правилами:

П1: якщо імена $v_1, v_2, ..., v_n$ з V різні та $d_1, d_2, ..., d_n \in D$, то іменна множина $\{(v_1, d_1), (v_2, d_2), ..., (v_n, d_n)\}$ належить D ($n \geq 0$);

П2: якщо $d_1, d_2, ..., d_m \in D$, то множина $\{d_1, d_2, ..., d_m\}$ – елемент класу D ($m \geq 0$).

Елементи з D називаються іменними даними. Оскільки клас даних D залежить від W і V як від параметрів, то у випадках, коли потрібно явно підкреслити цю залежність, елемента D можна називати (V, W) - іменними даними.

З цього визначення випливає, що в клас іменних даних потрапляють як дані з тривіальною іменною структурою (це перш за все елементи W і кінцеві підмножини W), так і дані, побудовані за правилом П1. Це правило при побудові конкретної множини може застосовуватися багаторазово (причому разом із правилом П2). За допомогою простих обмежень на застосування правил П1 та П2 виділяються важливі підкласи іменних даних:

а) клас строго іменних множин - ND = {d | d побудовано лише за допомогою правила П1};

б) клас змішаних іменних множин - NSD = {d | побудова d завершується правилом П1};

в) клас множинних даних або кінцевих множин - SD = {d | побудова d завершується правилом Р2}.

Очевидні такі включення: $ND \subset NSD \subset D$ і $SD \subset D$. Крім того, легко помітити, що множини W, NSD і SD утворюють розбиття класу іменних даних D.

Перш ніж переходити до розгляду композицій іменного рівня, необхідно визначити ряд операцій над іменними множинами.

1. Операція іменування ⇒v (з параметром v) називає аргумент ім'ям v:⇒v(a) = {(v, a)}.

2. Операція розіменування v ⇒ (з параметром v) обирає з іменної множини значення ім’я v цій множині: v⇒(d) = a, якщо (v, a) ϵ d. Операцію розіменування v⇒ також позначатимемо через 'v.

3. Операція накладання ∇ двом іменним множинам d_1 і d_2 ставить у відповідність нову іменну множину, що задається формулою: $d_1 \nabla d_2 = d_1 \cup d_2$, де d_1 складається з компонентів d_1, імена яких не входять в d_2. Наприклад:

{(X, x), (Y, y)} ∇ {(X, x'), (Z, z)} = {(X, x'), (Y, y), (Z, z)};

{(X.x)} ∇ {(Y, y)} = {(X, x), (Y, y)}.

4. Операцію підсумовування □ визначено лише для спільних іменних множин d_1 і d_2 (тобто для таких множин, у яких однакові імена мають однакові значення), і тоді $d_1 \square d_2 = d_1 \cup d_2$.

Тепер розглянемо композиції, що використовують властивість даних бути іменними множинами. Для уточнення однієї з таких композицій розглянемо приклад послідовного виконання операторів присвоєння, що міняють місцями значення змінних M та N:

M:=M+N; N:=M-N; M:=M-N.

Відповідно до загального розуміння вказаним операторам присвоєння відповідають функції, які операторам присвоєння переводять стан пам'яті в нові стани пам'яті, змінюючи значення однієї зі змінних – абстрактне тлумачення. Тому наведені оператори присвоєння задають функції f_1, f_2 і f_3, які іменні множини виду {(M, m), (N, n)} переводять відповідно до множини {(M, m+n), (N, n)}, {(M, n), (N, m-n)} і {(M, m-n), (N, n)}. Також операторам присвоєння відповідають функції обліку змін, які здійснюються операторами присвоювання – іменне трактування. У цьому випадку

вказані в прикладі оператори присвоювання задають функція g_1, g_2 і g_3, які іменні множини виду {(M, m), (N, n)} переводять відповідно до множини {(M, m+n)}, {(N, m-n)} та {(M, m-n)}.

Абстрактне трактування зручне тим, що послідовне виконання операторів присвоювання може бути уточнене композицією множення. Отже, семантика останньої програми задається функцією $f_1 \circ f_2 \circ f_3$. Недолік трактування полягає в огрубленні семантики, що ускладнює дослідження відповідних функцій і засобів роботи з ними.

Іменне трактування явно вказує на результуючі зміни у значеннях змінних і тому краще підходить для дослідження функцій, що задають семантику програм. При цьому послідовне виконання задаватиметься не множенням, а аплікуванням. З аналізу попередньої програми видно, що при іменному трактуванні операторів присвоєння послідовне виконання розуміється так: на вхідних даних виконується перший оператор, отримані результати накладаються на вхідні дані, а потім виконується другий оператор. Результатами послідовного виконання будуть результати другого оператора, накладені на результати першого.

Формально композицію аплікування "; " визначають наступною формулою:

$h_1;h_2(d) = h_1(d) \nabla h_2(d \nabla h_1(d))$.

На попередньому прикладі застосування композиції виглядатиме наступним чином:

$g_1;g_2$ ({M, m), (N, n)}) = g_1({(M, m), (N, n)}) ∇ g_2({(M, m), (N, n)} ∇ g_1 ({(M, m), (N, n)})) = ({(M, m+n)} ∇ g_2({(M, m), (N, n)} ∇ {(M, m+n)}) = ((M, m+n)} ∇ g_2({(M, m+n), (N, n)}) = {(M, m+n)} ∇ {(N, m+n-n)} = {(M, m+n), (N, m)}.

Тоді:

g_1,g_2,g_3({(M, m), (N, n)}) = g_1:g_2({(M, m), (N, n)}) ∇ g_3({(M, m), (N, m)}) ∇ g_1:g_2({(M, m), (N, n)})) = {(M, m+n), (N, m)} ∇ g_3({(M, m+n), (N, m)}) = {(M.m+n), (N, m)} ∇ {(M, m+n-m), (N, m)} = {(M, n), (N, m)}.

На основі композиції аплікування на іменному рівні вводиться композиція циклювання, що позначається WHILE, вважаючи, що значення WHILE (p, f)(d) дорівнює першому з елементів послідовності Ø, f(d), (f:f)(d), …, для якого p приймає значення P, причому для усіх попередніх елементів значення p визначено і дорівнює T. Ця композиція формалізує оператор циклу виду while p do f.

Різниця між абстрактним і іменним трактуванням програм також простежується на прикладі порожнього оператора. При абстрактному трактуванні він задається тотожною функцією, що перекладає будь-який елемент d у d. При іменному трактуванні порожній оператор є функцією ~Ø, яка будь-який елемент d переводить у порожню іменну множину Ø.

Ці два трактування порожнього оператора ведуть до двох трактувань оператора “if else” виду if p else f, для якого у разі істинності умови p виконується оператор f, а у разі помилковості p - порожній оператор. Тому при абстрактному трактуванні такий оператор еквівалентний оператору IF(p, f, e), а при іменному трактуванні - оператору IF(p, f, ~Ø). Оскільки даний оператор є одним з основних при програмуванні іменних функцій, введемо композицію, що формалізує розгалуження IFT наступною формулою:

IFT(p, f)(d) = IF(p, f, ~Ø)(d).

Іншу відмінність між двома трактуваннями видно з прикладу паралельного виконання операторів. Так, паралельний оператор присвоювання M, M:=N, M, що фактично складається з двох операторів присвоювання M:=N і M:=N, змінюватиме значення змінних M і N. На абстрактному рівні важко визначити композицію, яка б формалізувала паралельне виконання операторів. Разом з тим таку композицію, яка називається підсумовуванням і позначається ‖, на іменному рівні можна визначити формулою: f ‖ g (d) = f (d) □ g (d). Результати виконання функцій f та g підсумовуються. Тому (M:=N) (N:=M) ({(M, m), (N, n)}) = M:=N ({(M, m), (N, n)}) □ N:=M({(M, m), (N, n)}) = {(M, n)} □ {(N, m)} = {(M, n), (N, m)}, тобто паралельне виконання зазначених операторів дійсно змінює значення M та N.

Виходячи з прикладів, що розглядаються, можна легко задати унарну композицію присвоювання, що має параметр v ϵ V, вважаючи, що v:=(f)(d) = ⇒v(f(d)).

Щодо засобів побудови виразів, то основним із них є композиція суперпозиції за значенням. Так, вираз M+N побудовано суперпозицією функцій розіменування 'M та 'N у функцію +. При цьому функція + трактується як бінарна функція, яка відображає іменні множини виду {(1, m),(2, n)} у m+n. Аналогічно, n-арну функцію f трактуємо як функцію, визначену на іменних множинах виду {(1, a_1), ..., (n, a_n)}. Тоді композиція суперпозиції Sn до n-арної функції f функцій g1,…,gn задається формулою:

$S^n(f, g_1, \ldots, g_n)(d) = f(\{(1, g_1(d)), \ldots, (n, g_n(d))\}) = f(g_1(d), \ldots, g_n(d))$.

Отже, з оператором присвоювання M:=M+N буде пов'язана функція
M:= (S^2(+, 'M, 'N)).

Якщо f - будь-яка іменна функція, то суперпозицію можна проводити за довільними іменами $v_1,...,v_n$. Тоді композиції суперпозиції за значенням матиме наступний вигляд: $Z^v_1, \ldots, v_n(f, g_1, \ldots, g_n)(d) = f(d \nabla\{(v_1, g_1(d)), \ldots, (v_n, g_n(d))\})$.

Таким чином, іменний рівень розгляду дозволяє ввести засоби конструювання програм, що широко використовуються, тож можна стверджувати, що композиції програм використовують лише властивості даних, що визначаються на абстрактному, множинному та іменному рівнях, а саме властивості бути виділеним елементом, кінцевою множиною, або іменною множиною. Тому природно припустити, що цих властивостей достатньо для визначення класу композицій програм. Інші властивості даних, які використовуються композиціями програм, можуть бути зведені до розгляду перерахованих вище властивостей, що можна назвати як принцип відомості.

Принцип відомості: для визначення композицій програм достатньо враховувати властивості функцій і даних, що розглядаються на абстрактному, множинному та іменному рівнях.

Звісно, у різних конкретних системах програмування участь ресурсів кожного рівня може бути різна. Наприклад для низькорівневих мов програмування необхідне

широке залучення ресурсів множинного рівня, тоді як для більшості традиційних мов програмування достатньо засобів іменного та абстрактного рівнів.

Розглянемо найпростіші такі системи. У вибраній нами моделі кожна програма отримана застосуванням композицій до базових функцій. Синтез програм здійснюється в системі, яка задається деяким класом композицій K і множиною базових функцій FB. Таку пару (FB, K) називатимемо програмною логікою. Це тісно пов'язане з поняттям програмної алгебри. Термін "програмна логіка" буде застосовуватися в тих випадках, коли акцент досліджень зроблено на породженні програм, а термін "програмна алгебра" коли досліджуються властивості операцій та функцій, що породжуються.

Побудуємо програмні логіки, які мають досить прості набори композицій і які, враховуючи їх близькість до стандартних схем програм, назвемо стандартними програмними логіками (СПЛ). Такі логіки мають той самий клас композицій KS і різні класи базових функцій.

Клас KS складається з композицій: аплікування “;”, розгалуження “IF, IFT”, циклування “WHILE”, суперпозиція “S^n”, іменування “X:=”, розіменування “ 'X ”.

Останні три композиції є параметричними - іменування та розіменування мають параметр X є V, а суперпозиція - параметр n є Nat.

Композиції аплікування, розгалуження та циклування та їх аналоги використовуються практично в будь-якій системі програмування та є основними для структурного програмування. Розгалуження може бути визначено на абстрактному рівні, а аплікування та циклування вимагають іменного рівня. Наявність суперпозиції Sn показує, що СПЛ може працювати з n-арними функціями. Розіменування 'X є нуль-арною композицією, тобто може трактуватися як іменна функція.

У мовах програмування суперпозицію та розіменування застосовують для побудови виразів. Аплікування, розгалуження та циклування - для побудови операторів, а іменування перетворює вирази в оператори.

Для СПЛ, що розглядається як семантичне ядро традиційних мов програмування, як базові зазвичай вибирають n-арні функції - арифметичні, логічні, порівняння тощо. У той самий час для СПЛ, відповідних мовам маніпулювання даними, багато основних функцій є маніпуляційними. Приклад використання СПЛ для побудови програм буде наведено нижче.

Хоча в цілому СПЛ задають досить широкий набір засобів програмування, їх все ж таки недостатньо для обробки складних структур даних, наприклад, масивів.

Для цього необхідно збагатити СПЛ до стандартної програмної логіки обробки масивів.

Особливість роботи з масивами у тому, що можна обчислювати ім'я тієї компоненти масиву, якій присвоюється нове значення чи значення якої визначається. Для n-мірного масиву X ім'я його компоненти визначається за допомогою функції індексування, яку позначатимемо $X^n[\ ]$. Так, якщо індекси дорівнюють t_1, ..., t_n, то $X^n[\ ](t_1, \ldots, t_n)$ продукує ім'я $X^n[t_1, \ldots, t_n]$.

Для присвоєння значень компонентам масиву необхідно збагатити унарну композицію присвоювання, зробивши її бінарною композицією, вважаючи, що g:=f позначає функцію, яка на даному d присвоює значення f(d) імені, що дорівнює g(d). Так, оператор A[i, j]:= M+N задає функцію $S^2(A^2[\ ], 't, 'f): = S^2(+, 'M, 'N)$.

Для взяття значення імені, що задається функцією g, збагатимо композицію розіменування, вважаючи, що (`g)(d) визначає значення імені g(d). Так, вираз A[t, f]+M задає функцію $S^2(+,'S^2(A^2[\ ], 't, 'f), 'M)$.

Таким чином, у логіці обробки масивів клас композицій, що позначається KA, отриманий завдяки збагаченню класу KS бінарною композицією іменування :=, унарною композицією розіменування і функціями індексування $X^n[\ ]$. Унарну композицію іменування X:= і нуль-арну композицію розіменування 'X можна отримати як окремі випадки збагачених композицій, якщо ввести функції-константи.

Для вирішення завдань маніпулювання та обробки даних, потрібні спеціальні програмні логіки. Одним із варіантів може бути СПЛ, збагачена спеціальними композиціями множинного рівня DIS та INA.

2.4. Приклади конструювання програм

Розглянемо використання введених понять композиційного програмування для конструювання програми. Принципи підпорядкованості та розділення, що визначають відносини прагматичного, семантичного та синтаксичного аспектів, визначають три основні етапи конструювання програми:

1) аналіз прагматичних вимог;

2) семантичне конструювання програми;

3) синтаксичне оформлення програми.

На першому етапі аналізуються вимоги, що пред'являються до програми та її виконавця, такі як ефективність, надійність, характер взаємодії з користувачем тощо. Що стосується виконавця програми, то він може оперувати конкретною обчислювальною системою, системою програмування або просто певною мовою програмування, що уможливлює інтерпретацію його програми. Визначаються також допустимі засоби конструювання програм, що становлять логіку конструювання.

На другому етапі у вибраній програмній логіці проводиться семантичне конструювання програми, результатом якого є семантичний терм програми. Основна увага приділяється побудові коректної програми. Отримана програма може бути доопрацьована різними перетвореннями з метою поліпшення прагматичних характеристик.

На етапі синтаксичного оформлення програма подається у формі, яка сприймається конкретним виконавцем. Як правило, цей етап досить простий і легко піддається автоматизації.

Розглянемо простий приклад конструювання програми, що здійснює інформаційний пошук.

Нехай задано перелік (послідовність) визначених відомостей для співробітників установи. Потрібно скласти програму, яка вибирає з цього переліку сім'ї молодих інженерів (не старше 33 років).

При побудові цієї програми будемо для простоти виділяти лише три раніше вказані етапи, хоча конкретні технології композиційного програмування можуть, безумовно, використовувати більш точну градацію етапів.

1. Аналіз прагматичних вимог.

На цьому етапі необхідно врахувати вимоги до програми від замовника. Розглянемо дві альтернативні вимоги:

а) програма використовується рідко, час її виконання не важливий;

б) програма використовується часто, час її виконання має бути зведений до мінімуму.

Крім того, в обох вимогах передбачається, що вхідний перелік наведених даних про співробітників із часом буде змінюватися (за допомогою якої-небудь іншої програми) і може досягати достатньо великих розмірів (з точки зору кількості співробітників). Програми, що відповідають цим вимогам, повинні бути записані на С подібній мові.

2. Семантичне конструювання програми.

На цьому етапі необхідно, виходячи з попередніх вимог, сконструювати програму з використанням засобів СПЛ.

Оскільки програма - це функція, необхідно визначити структуру елементів області її визначення та значення. Інакше кажучи, треба визначити вхідні та вихідні дані. Припустимо, що перелік наведених співробітників представлений файлом (послідовністю) записів, де кожному співробітнику відповідає запис, що містить наступні компоненти: "прізвище(П)", "ім'я(І)", "по батькові(Б)", "рік народження (Р)" та "посада (ПОС)".

Q = REC (П: string, І: string, Б: string, Р: int, ПОС: string).

Тут string - тип значення імен, що визначаються словами, int - тип значення імен, що визначаються цифрами. Файли записів представлені іменними множинами видів:

$<i \mid r_1,r_2\ldots,r_n> = \{(0,i), (1,r_1), (2,r_2), \ldots..(n, r_n)\}$, де $n \geq 0$ і $(j, r_j.)$ – j компонента файлу, що представляє запис rj ($1 \leq j \leq n$), а нульова роль компонента (o, i) має допоміжний характер і є ітератором стану поточного запису. Нехай вхідні файли іменовані іменем “ЛЮД”. Для визначення віку співробітників необхідно ввести поточний рік, він буде введений під ім'ям “РІК”. Отже, вхідні дані програми – це іменні множини, що складаються з пар (ЛЮД, F) і (РІК, Р), де F - файл описаної вище структури, а Р - поточний рік. Вихідними даними програми також будуть файли, але не записи, а дані типу string. Нехай ім'я вихідних файлів буде ПЛЮД.

Перейдемо тепер до розгляду семантичної структури програми. Інтуїтивний сенс її дуже простий: поки не виявлено "кінець" вхідного файлу, необхідно прочитати його запис, а потім виділити та записати в вихідний файл сім'ю співробітника, не старше 33 років, який обіймає посаду інженера. Для виконання цих дій необхідні функції обробки файлів:

1) читання запису файлу read (v, u), де v - ім'я файлу, а u - ім'я, під яким буде розміщена прочитана компонента файлу. Прочитана компонента файлу буде і-та компонента, де і - значення вказівника. Крім того, дія функції read (v, u), призводить до збільшення вказівника на 1;

2) ініціалізації файлу init(v), яка встановлює початкове значення вказівника, що дорівнює 1;

3) записи у файлі write(v, u), де v - ім'я файлу, u - ім'я, значення якого вноситься в кінці файлу v;

4) визначення кінця файлу eof(v), значення якого дорівнює T, якщо значення вказівника більше кількості компонентів файлу, і F у протилежному випадку.

Крім цих спеціальних функцій, потрібні ще функції, як порівняння, арифметичне віднімання, а також логічна операція заперечення “ ┐ ” для перевірки відсутності "кінця" файлу.

Семантична структура програми для вимог а) має вид:

init (ЛЮД);

WHILE(┐ eof (ПЛЮД),

read (ЛЮД, ЗАП);

IFT (РІК - Р ≤ 33 & ПОС= ‘інженер’

write (ПЛЮД, П))).

При створенні цієї програми використовуються композиції суперпозицій S^1 і S^2 (в арифметичних і логічних виразах), розгалуження IFT, циклування WHILE та аплікації. Якщо базові функції та їх суперпозиції в цій програмі замінити абстрактними символами f_1, f_2 і т.д., то отримаємо логічну (без урахування суперпозиції) “оболонку” програми:

f_1 : WHILE (f_2, f_3 : IFT (f_4, f_5)).

При семантичному конструюванні програми зручно користуватися поняттям К-структури програми (К - деяка кількість композицій). Наведена вище логічна "оболонка" програми є (:, IFT, WHILE) – структура. У загальному випадку К-структура - це "програма", сконструйована з композицією за допомогою множини К і абстрактних символів базових функцій. У цьому визначенні термін "програма" взято в лапки, тому що це не конкретна програма, а її схема. К-структурі можна дати й інше, більш алгебраїчне визначення, а саме: К-структура - це продуктивна композиція, побудована з набору композицій символів К і символів функціональних змінних за допомогою операції підстановки. Очевидно, що К-структура програми відображає її логічну оболонку. Різні програми логічно порівнювати по цій “оболонці”.

При побудові К-структури для попередньої програми обрано множину композицій К таким чином, щоб абстрагуватися від суперпозицій, оскільки останні при

програмуванні завдань обробки даних складні структури відіграють менш значну роль, ніж інші композиції СПЛ, але це не виключає суперпозиції.

Побудуємо тепер семантичну структуру програми, що задовольняє прагматичній вимозі б). Для цього необхідно доповнити попередні дані про співробітників. Оскільки таке представлення має бути орієнтоване на швидкий пошук необхідних файлів (на відміну від повного перегляду файлу у випадку а)), ці дані представлені спеціальною структурою даних, відмінною від запропонованої вище структури файлу. Припустимо, що відомості про співробітників представлені, як і раніше, файлом записів ЛЮД, що визначені через спеціальну властивість - усі записи в файлі упорядковані за роком народження співробітників. Така пропозиція є цілком реальною і може бути досягнута за допомогою спеціальної програми сортування файлу після його заповнення.

Так, семантична структура програми з урахуванням даної пропозиції визначається наступною логікою дій: поки файл не закінчений і черговий прочитаний запис не містить «рік народження» понад 33, виділити із запису та занести в вихідний файл прізвище співробітника, який є інженером.

Тепер відповідно до зазначеної логіки можна формально сконструювати таку програму:

```
init(ЛЮД);
read (ЛЮД, ЗАП);
WHILE(¬ eof (ЛЮД) & РІК -Р ≤ 33,
        IFT (ПОС = 'інженер',
                write(ПЛЮД, П));
        read (ЛЮД, ЗАП));
IFT(eof (ЛЮД) & РІК - Р ≤ 33 & ПОС= 'інженер').
        write(ПЛЮД, П)).
```

Наведемо {:, WHILE, IFT} - структуру цієї програми. Вона має вигляд:

$f_1 : f_3 : WHILE (f'_2, IFT (f'_4, f'_5) : f_3) :IFT (f_6, f_5).$

Порівнюючи К-структури попередніх програм, можна побачити більш чітко їх схожість та відмінність.

Зазначимо такі їх особливості:

1) їх області визначення та значень збігаються;

2) програми визначені як на файлах записів зазначеної вище структури, а й на файлах будь-яких записів, які включають компоненти з іменами РІК, ПОС і П (на рівні семантичного конструювання програм недоцільно звужувати область їх визначення, оскільки основний акцент робиться на її логіко-функціональній структурі);

3) з того, що остання програма на впорядкованому вхідному значенні d1 розраховує вихідне значення d2, слідує, що таке обчислення здійснюється програмою іншою програмою аналогічним чином.

3. Синтаксичне подання програм.

Під синтаксичним уявленням розуміється оформлення програм конкретною мовою програмування згідно з прийнятими в ній правилами. Щоб це зробити, необхідно вказати ряд параметрів, пов'язаних специфікою мови: ім'я програми, типи даних. Ці параметри не повинні суперечити обмеженням даних, які можуть бути вказані в списку прагматичних вимог. Це обмеження і подібні до нього ми раніше не вказували, оскільки вони не впливають на логіку програми.

Нижче наведено програми з іменами PROG1 та PROG2, що відповідають СПЛ попереднім програмам.

```
program PROG1 (ЛЮД, ПЛЮД, INPUT, OUTPUT);
type Q = record
                П: array [1..15] of char;
                І: array [1..10] of char;
                Б: array [1..12] of char;
                Р: int;
                ПОС : array [1..15] of char
```

```
            end;
    var ЛЮД: file of Q;
        ПЛЮД: file of array [1..15] of char;
        РІК: int;
    begin
        readln(P);
        reset(ЛЮД);
        rewrite(ПЛЮД);
        while (not eof (ЛЮД)) do
            begin
                read(ЛЮД, ЗАП);
                if (РІК - P ≤ 33) and (ПОС ='інженер')
                    then write(ПЛЮД, П)
            end
    end.
```

Програма PROG2 виходить із програми PROG1 заміною зовнішнього блоку begin_end на наступний:

```
    begin
        readln (P);
        reset (ЛЮД);
        rewrite (ПЛЮД);
        read (ЛЮД, ЗАП);
    while not eof (ЛЮД) and (РІК -P ≤ 33) do
        begin
            if ПОС='інженер'
                then write ( ПЛЮД, ПОС);
        read(ЛЮД, ЗАП)
```

```
        end;
if eof (ЛЮД) and (РІК - Р ≤ 33) and (ПОС = 'інженер')
then write (ПЛЮД, П)
end.
```

Зрозуміло, що синтаксичне уявлення програми щодо її семантичного терму здійснюється за цілком певними правилами. Більше того, цей процес також можна автоматизувати. Для цього в композиційному програмуванні використовується поняття програмного дефінітора, що зв'язує семантику та синтаксис програм виділеної мови програмування.

2.5. Дефінітори у мовах програмування

Мови програмування, як і мови взагалі, характеризуються такими основними аспектами: прагматичним, семантичним та синтаксичним. Прагматичний аспект визначає відносини між розробниками програми та тими, хто її використовуватиме. З цієї точки зору на мови програмування можна дивитися як на мови логіко-машинного спілкування чи мови конструювання та розробки програм, чи просто мови уявлення програм.

Традиційні мови програмування лише малою мірою можуть розглядатись як мови синтезу програм. Вони швидше виступають засобами для запису результатів розробки, а не власне розробки програм.

Тому правильніше було б називати такі мови – мовами програм, а не мовами програмування. Це говорить про те, що на більш абстрактному рівні розгляду мов програмування є засіб синтезу певних класів програм. Тоді семантичний аспект визначає семантику програм таких класів, а синтаксичний – їх синтаксис. Це дозволяє дивитися на мову програмування як на інтерпретовану мову, програми якої є парою типу (f, s), де f подає семантику програми (функцію з деякого класу), а s - синтаксис (слово з деякої множини слів). Отже, щоб задати інтерпретовану мову, необхідно визначити взаємозв'язок між семантикою та синтаксисом.

Оскільки всі зазначені множини зазвичай є нескінченними, їх визначають за допомогою рекурсії. Такі засоби опису мов називають програмними дефініторами.

У композиційному програмуванні розглядають кілька типів програмних дефінторів: що породжують, які задають програми як пари (семантика та синтаксис), семантико-синтаксичні дефінітори, які за семантичною структурою програми задають її синтаксичне оформлення, синтактико-семантичні дефінітори, які за синтаксичним оформленням програми відновлюють семантику.

Зазвичай семантична компонента дефінітора задається програмною логікою (програмною алгеброю), яка складається з множини композицій і множини базових функцій. Отже, до складу програмного дефінітора входитимуть дві таблиці, які пов'язують композиції та базові функції з їх синтаксичними уявленнями. Композиції можуть мати по кілька форм представлення, що характерно насамперед для суперпозицій, які можуть бути записані у префіксній, інфіксній чи постфіксній формі. Як приклад розглянемо такі таблиці відносно раніше введеної СПЛ, представляючи її композиції та функції.

Композиційне програмування використаємо як платформу програмування, разом із іменною моделлю даних, функцій і операцій. Композитами будуть операції мультиплікування °, розгалуження IF, циклування WD і найпростіші похідні від них композиції (у сенсі операцій аплікації Ap та n -арної суперпозиції $S^n|_{n\in N}$), що уточнюють найбільш вживані способи синтезу одних програм з інших, а в якості базових предметних операцій – арифметичні операції $+, -$, 0; логічні операції ∨, ∧, !, T, F; відношення $=, <, >$; функція слідування $s(n) = n + 1|_{n\in N}$. Також знадобляться параметричні операції над іменними даними – іменування $A_{\downarrow}: A_{\downarrow}(a)|_{a\in N} = \{(A, a)\}$, розіменування $A^{\uparrow}: A^{\uparrow}(\{(A, a)\}|_{a\in N}) = a$ і відкриваюча та закриваюча дужки.

Надалі, під даними, функціями та операціями, якщо не зазначено інше, розуміємо іменні дані, іменні функції та іменні операції, відповідно. Композито-композиційний інтерфейс, визначається композитами апарату послідовної або ∘-, галуженої або IF- та

циклічної або WD -редукці. Кортеж функцій $\langle f_1, f_2, \dots, f_s\rangle$ є $^\circ$ −редукцією функції f, якщо він є рішенням рівняння $f = x_1{}^\circ x_2{}^\circ \dots {}^\circ x_s$, тобто $f \equiv f_1{}^\circ f_2{}^\circ \dots {}^\circ f_s$. Пара функцій $\langle f_1, f_2\rangle$ є IF-редукцією функції f, якщо існує такий предикат p, що ця пара є рішенням рівняння $f = IF(p, x_1, x_2)$, тобто $f \equiv IF(p, f_1, f_2)$. Також, функція g є WD-редукцією функції f, якщо існує такий предикат p, що g є рішенням рівняння $f = WD(x, p)$, тобто $f \equiv WD(g, p)$. З останнього безпосередньо випливає корисна необхідна умова WD-редукційності. Для того, щоб функція g була WD-редукцією функції f необхідно, щоб виконувалась наступна рівність $g{}^\circ f = f$.

Після побудови системи, продемонструємо спосіб програмування у ній на прикладі функції цілочисельного ділення. Програмування проводитимемо виходячи з наступної властивості функції: $\forall a, b, c|_{a,b.c\,\in N}\ div(a, b) = c$, де div – функція цілочисельного ділення натуральних чисел, а саме: $div\colon N \times N \to N$, де $div(a, b)$, таке натуральне число, що $b \times div(a, b) \le a \le b \times (div(a, b) + 1)$. Враховуючи зорієнтованість описаної системи на іменні структури даних, збагатимо функцію її іменною специфікацією у вигляді іменної функції $DIV\colon \{(A, a), (B, b)\} \to \{C, \mathrm{c}\}|_{a,b,c\,\in N}$. $DIV\colon \{(A, a), (B, b)\} \to \{(C, div\ (a, b)\}|_{a,b,c\,\in N}$.

Тобто, $DIV \equiv F_1{}^\circ F_2$. Дана специфікація є оракульною схемою [6], обумовленою композитом мультиплікування. З її іменної специфікації випливає, що $\langle F_1, F_2, \rangle$, де $F_1 = 0(C^{\uparrow})^\circ C_{\downarrow} \equiv \{(C, 0)\}$, $F_2\colon \{(A, a), (B, b), (C, c)\} \to \{(A, a - k \times b), (B, b), (C, c + div(a, b))\}$, де $a, b \in N, b \ne 0$, а $k\colon (k + 1) \times b < a < k \times b$. Тобто, $G \equiv F_1{}^\circ F_2$. Функції F_1 очевидно не потребує деталізації, оскільки є початковим обнулінням значення c. Функція F_2 є оракульною схемою [6], з властивості функції div випливає, що WD-редукцією функції DIV буде функція $G\colon \{(A, a), (B, b), (\mathrm{C}, \mathrm{c})\} \to \{(A, a - b), (B, b), (\mathrm{C}, \mathrm{c} + 1)$, де $P\colon \{(A, a), (B, b)\} \to \begin{cases} T, \text{якщо } a \ge b \\ F, \text{якщо } a < b \end{cases}$ – іменна специфікація відповідного предикату. Тобто, $F_2 = WD(G, P)$. Очевидно, потрібно здійснити її програмування. Для цього скористаємось властивістю цієї функції:

$$div(a,b)|_{a,b\in N\&b>0} = \begin{cases} div(a-b,b)+1, a \geq b, \\ div(a,b) = 0, a < b \text{ або } a = 0 \end{cases}$$

Таким чином, F_2 виглядає так: $F_2 = WD(\left(A_{\downarrow}(A^{\uparrow} - B^{\uparrow})\right) \circ \left(C_{\downarrow}(C^{\uparrow}+1)\right), P(A^{\uparrow}, B^{\uparrow})$. Звідси, $DIV \equiv \left(0(C^{\uparrow})\circ C_{\downarrow}\right)\circ(WD(\left(A_{\downarrow}(A^{\uparrow} - B^{\uparrow})\right) \circ \left(C_{\downarrow}(C^{\uparrow}+1)\right), P(A^{\uparrow}, B^{\uparrow})$. Така спрощена форма запису обрана задля легшого розуміння виразу. Також у формулах наведених вище присутні метавирази які не є реальними виразами в описаному середовищі. Задля спрощення форми запису дані метавирази вирішено не розписувати. Описані метавирази знайдут своє спеціальне відображення у відповідній таблиці дефінітора і детальний опис у наступних розділах.

У результаті першої стадії технологізації, а саме редукційного програмування у заданій системі була отримана описана вище специфікація. З побудови програми випливає її коректність. Після отримання специфікації можливо провести кодування.

Зазначалося, що більшість мов програмування є лише засобами синтаксичної нотації результатів програмування. Технологія програмування близька по своїй суті до інтерпретованої мови та є імплементацією взаємодоповнення принципів програмування. За своєю суттю представляє собою суб'єкто-об'єктну технологію створення програмного продукту. Будь яку програму можна представити як мікроконвейер стадій, де стадія "програмування" реалізує підпорядкованість семантики прагматиці і результатом її є програма – обумовлений суб'єктом нарис рішення задачі у вигляді відповідного семантичного терму. Стадія "кодування" – стосується синтаксичного аспекту і результатом тут є код програми у вигляді синтаксично вірно записаного тексту у визначеній мові програмування. Для автоматизації процесу використовується відповідний дефінітор мови програмування. Застосуємо це до вище розглянутого прикладу програмування функції DIV.

Для спрощення, розглянемо тут лише невелику частину дефінітору мови програмування, яка є достатньою для демонстрації. У ній представлені відповідні композити та функції з їх синтаксичними нотаціями (таб.2.1 та 2.2).

Таблиця 2.1 – Патерни кодування та програмування

Концепт (шаблони) програмування	**Концепт (шаблони)кодування**
…	…
F, (F)	F
$F_1 \circ F_2$	$begin\ F_1; F_2\ end$
$IF(F_1, F_2, F_3)$	$if\ F_1\ then\ F_2\ else\ F_3$
$WD(F_1, F_2)$	$while\ F_2\ do\ F_1\ end$
$F \circ X_{\downarrow}$	$X \coloneqq F$
$X^{\uparrow} \circ S \quad X^{\uparrow} \circ S \circ Y_{\downarrow}$	$X^{\uparrow} + 1 \quad Y \coloneqq X + 1$
$P(A^{\uparrow}, B^{\uparrow})$	$(A > B)\ or\ (A = B)$
$F_1\ [F_2]$	$F_1\ ; F_2$
meta $\begin{bmatrix} 0(C^{\uparrow}) \\ A_{\downarrow}(A^{\uparrow} - B^{\uparrow}) \\ C_{\downarrow}(C^{\uparrow} + 1) \end{bmatrix}$	$\begin{bmatrix} 0(C) \\ A \coloneqq A - B \\ C \coloneqq C + 1 \end{bmatrix}$
…	…

Таблиця 2.2 – Базові функції та їхні коди

Базові функції	**Коди базових функцій**
…	…
0	0
$+$	$+$
$-$	$-$
$\wedge$	and
$\vee$	or
!	not
$<$	$<$
$=$	$=$
$>$	$>$
$X^{\uparrow}$	X
$X_{\downarrow}$	X
…	…

У представлених таблицях позначення F, можливо з індексами, $F_i, i = 1, 2, 3, \ldots$ і тільки вони використовуються у якості нетермінальних символів або нетерміналів.

Аналогічно, термінальні символи $X^{\uparrow}$, $X_{\downarrow}$ та X також можуть використовуватись з індексами: ${X_i}^{\uparrow}$, $X_{i_{\downarrow}}$ та X_i, $i = 1, 2, 3, \ldots$

Через них забезпечується рекурсивність побудов. Концепти програмування та кодування представлені у першій таблиці представляють собою правильно записані слова у об'єднаному алфавіті термінальних символів та нетерміналів. У другій таблиці наведено термінальні символи для базових операцій та відповідні їм коди.

Звернемося до вищенаведеної програми. Використана раніше додаткова розмітка програми наочно демонструє притаманну їй ієрархічність структури. Вона обумовлена здійсненою покроковістю актуалізацій оракулів у системі програмування, починаючи від оракула DIV і закінчуючи вільним від оракулів композиційним термом. Рухаючись по цій ієрархії у відповідності до заданого у попередніх таблицях фрагменту дефінітора рекурсивно будуємо код програми (таб. 2.3).

Таким чином на репрезентативному прикладі продемонстровано використання концептів програмування у вигляді семантичних шаблонів як ланок програмного ланцюга, які обумовлюють певні класи програм. Використано програмний дефінітор, який виступає у ролі засобу трансляції композитів та базових функцій технологічної системи програмування у їх синтаксичні представлення.

За допомогою редукційного програмування у заданій системі була отримана програмна специфікація, коректність якої випливає з її побудови. На основі отриманої специфікації за допомогою дефініторів отримано код програми.

Таблиця 2.3 – Шаблони програмування та кодування

Програма	**Використовувані шаблони**	**Актуалізації нетерміналів**
$DIV \equiv (0(C^{\uparrow})^{\circ}C_{\downarrow})^{\circ}(WD(\left(A_{\downarrow}(A^{\uparrow}-B^{\uparrow})\right)^{\circ}\left(C_{\downarrow}(C^{\uparrow}+1)\right),P(A^{\uparrow},B^{\uparrow}))$	$F_1{}^{\circ}F_2$	$F_1 \Leftarrow 0(C^{\uparrow})^{\circ}C_{\downarrow}$ $F_2 \Leftarrow WD(\left(A_{\downarrow}(A^{\uparrow}-B^{\uparrow})\right)^{\circ}\left(C_{\downarrow}(C^{\uparrow}+1)\right),P(A^{\uparrow},B^{\uparrow})$ $DIV \Leftarrow begin\ F_1; F_2\ end$
$F_1 \equiv 0(C^{\uparrow})^{\circ}C_{\downarrow}$	$F_1{}^{\circ}F_2$ $X^{\uparrow}$ $X_{\downarrow}$	$F_{11} \Leftarrow 0(\mathrm{C})$ $F_{12} \Leftarrow C_{\downarrow}$ $F_1 \Leftarrow\ begin\ F_{11}; F_{12} end$
$F_2 = WD(\left(A_{\downarrow}(A^{\uparrow}-B^{\uparrow})\right)^{\circ}\left(C_{\downarrow}(C^{\uparrow}+1)\right),P(A^{\uparrow},B^{\uparrow})$	$WD(F_1,F_2)$	$F_{21} \Leftarrow \left(A_{\downarrow}(A^{\uparrow}-B^{\uparrow})\right)^{\circ}\left(C_{\downarrow}(C^{\uparrow}+1)\right)$ $F_{22} \Leftarrow P(A^{\uparrow},B^{\uparrow})$ $F_2 \Leftarrow\ while\ F_{22}\ do\ F_{21} end$
$F_{21} \Leftarrow \left(A_{\downarrow}(A^{\uparrow}-B^{\uparrow})\right)^{\circ}\left(C_{\downarrow}(C^{\uparrow}+1)\right)$	$F_1{}^{\circ}F_2$ (F) $X^{\uparrow}$ $X_{\downarrow}$	$F_{31} \Leftarrow A_{\downarrow}(A^{\uparrow}-B^{\uparrow})$ $F_{32} \Leftarrow C_{\downarrow}(C^{\uparrow}+1)$ $F_{21} \Leftarrow begin\ F_{31}; F_{32} end$
$F_{22} \Leftarrow P(A^{\uparrow},B^{\uparrow})$	meta $P(F_1,F_2)$	$F_{22} \Leftarrow (A > B\ or\ A = B)$
$F_{31} \Leftarrow A_{\downarrow}(A^{\uparrow}-B^{\uparrow})$	meta $A_{\downarrow}(A^{\uparrow}-B^{\uparrow})$ (F)	$begin$ $F_{31} \Leftarrow A \coloneqq A - B$ end
$F_{32} \Leftarrow C_{\downarrow}(C^{\uparrow}+1)$	meta $C_{\downarrow}(C^{\uparrow}+1))$ (F)	$begin$ $F_{32} \Leftarrow C \coloneqq C + 1$ end
$DIV \equiv F_1{}^{\circ}(WD(F_{21}{}^{\circ}F_{22}))$	$F_1{}^{\circ}F_2$ $WD(F_1,F_2)$ meta $C_{\downarrow}(C^{\uparrow}+1))$ meta $A_{\downarrow}(A^{\uparrow}-B^{\uparrow})$ meta $P(F_1,F_2)$ (F)	$begin$ $while\ (A > B\ or\ A = B)\ do$ $begin$ $F_{31} \Leftarrow A \coloneqq A - B$ $F_{32} \Leftarrow C \coloneqq C + 1$ end $DIV \Leftarrow F_{32}$ end

2.6. Композиційні основи суб’єкто-об’ктного середовища програмування

Як уже зазначалося раніше, у програмуванні та діяльності, пов'язаній з програмами, на сьогодні властива перевага спрощеного розуміння. Сама спрощеність є корисною для галузі програмування, оскільки надає свободи від врахування багатьох деталей. Однак, з часом те, що можна було вважати другорядним, стає важливим, і спрощеність може стати надмірною, втрачаючи актуальність відповідно до сучасних

вимог, особливо коли необхідно об'єктивізувати її через різні системи автоматизації такої діяльності. Цей зсув акцентів обумовлений потенційною відкритістю будь-якої нетривіальної діяльності, в якій активна роль суб'єкта має важливе значення, і приводить до прагнення перетворити цю роль на об'єкт діяльності в процесі її розвитку.

Якщо йдеться про програмування, то спрощеність його розуміння в рамках різних систем можна пояснити так: акцентується увага на результаті, виключаючи з розгляду процеси його досягнення. Наприклад, кожен, хто займається створенням програм, має свою індивідуальну, обмежену та спрощену точку зору на програмування і те, що програмується. Навіть тисячі різних інструментів програмування та універсальних мов не підтримують процеси програмування рішень, а лише слугують для нотації результатів.

За сучасних умов, процес вирішення задач програмування є максимально суб'єктивізованим, і розуміння його підміняється окремими номінальними уявленнями. Такий підхід перешкоджає серйозному розгляду важливих проблем сучасного програмування, таких як управління якістю програм, ефективність їх створення та оптимізацією витрат.

Хоча є багато прикладів для підтвердження цих міркувань, достатньо наведених вище аргументів, щоб визнати надмірність поточного рівня спрощення розуміння програм та програмування, а також їх несумісність з сучасними вимогами.

Також існує інша позиція, пов'язана зі спробами обмежити програмування в рамках певних, навіть ефективних з певних аспектів, традиційних підходів, що замкнені в номінальності. Наприклад, підходи до програмного та апаратного забезпечення, такі як об’єкто-орієнтовне, функціональне, модульне, логічне програмування тощо, є показовими прикладами цього.

Оскільки різноманіття точок зору на програмування є нескінченним згідно з дескриптивним аналогом теореми Геделя про неповноту, неможливо ефективно охопити всі аспекти програмування в рамках одного замкненого підходу. Проте ключовим є

забезпечення адекватності засобів вирішення проблем відповідно до уявлень про їх природу, що має значення як для управління якістю отриманих рішень, так і для ефективності процесів їх створення в цілому.

Згадана велика кількість універсальних програмних інструментів може бути результатом безперспективних спроб усунення неадекватностей у засобах вирішення проблем, оскільки враховуючи фундаментальний результат Геделя і його дескриптивний аналог, таке обмеження є неможливим.

Навіть стрімкий розвиток напрямку, пов'язаного з задачами формалізації семантики мов та програмування, не врятує ситуацію. Цей напрямок, зокрема алгебраїчний підхід, спрямований на отримання алгебраїчних характеристик різних класів об'єктів, також не дозволяє вирішити всі аспекти недоліків в сучасних підходах до програмування.

Зазначені недоліки, що створюють передумови для розробки нових номінальних систем, є лише поверхневими наслідками глибших причин. Великий приріст таких проблем спричинив обговорення кризових проявів в автоматизації програмування та депресії в комп'ютерній індустрії. Проте, більше слід говорити не про кризу галузі, а про кризу підходів до її розвитку.

Це підтверджує той факт, що програмування, як і будь-яка високоінтелектуальна діяльність, є настільки суб'єктивною, що "проста" номінальна об'єктивізація участі суб'єкта в ньому (наприклад, насаджуванням навіть найпродуктивніших підходів, таких як ООП, структурний, модульний тощо) неможлива без серйозних втрат і негативних наслідків. Така неефективність обумовлена активною роллю суб'єкта, оскільки саме високоінтелектуальний індивід є основним інструментом в програмуванні. Завдяки цьому, підтримка цієї активності передбачає реальне забезпечення різноманіття інструментів розуміння, уникнення їх звуження до номінальних і обмежених підходів, які можливо є продуктивними. Проте, здійсненням цієї мети забезпечується через об'єктивний підхід до розуміння, розвиваючи саме цей інструмент, яким володіє суб'єкт.

Для усунення суб'єктивних труднощів у розвитку сучасного програмування, потрібно впровадити його розуміння в загальний контекст розуміння, забезпечивши його еволюційний розвиток. Все вищевказане обґрунтовує прагматичне положення про концептологічну парадигму програмування, відповідно до якої суть це концептологічне обумовлення програмованого.

Дана парадигма визначає відносний характер програмування і створює реальні умови для об'єктивізації ролі людського фактора в еволюційному розвитку галузі, що забезпечує можливість реальної (а не лише формальної) побудови сучасного програмувального середовища. Водночас ця реальність будується на підґрунті надмірного спрощення розуміння програмування, що суперечить ідеї, що будь-яка інтелектуальна діяльність взаємозв'язана з розумінням цієї діяльності через об'єктивні причинно-наслідкові зв'язки, а не лише її результат.

Тобто, у середовищі, яке базується на концепції "програмування - програма", домінантну роль відіграє сама діяльність програмування, яка зумовлена концептом, і програма виступає в підпорядкованій ролі як результат цієї діяльності. Програмування становить найбільш фундаментальну складову відповідного середовища, тоді як програма, як її наслідок, представляє більш загальну складову.

Перевагою цієї позиції є її реалістичний характер та продуктивність, яка полягає в готовності до подальшого розвитку. Проте, сама відкритість лише надає можливості, і щоб забезпечити реальний розвиток, необхідно активно спрямовувати цей процес з прагматичною метою. Основними об'єктами розвитку відповідно до цього підходу до середовища програмування є "обумовлення" та "розуміння".

2.7. Парадигми суб’єкто-об’єктного середовища

Створення середовища, яке дійсно підтримує програмування, прямо пов'язане з встановленням єдиної концептуальної точки зору на саме програмування. Ця точка зору не має бути обмеженою в рамках будь-якого конкретного, навіть продуктивного розуміння програмування. Саме незамкнутість розуміння програмування дозволяє йому

еволюціонувати разом з програмістами, залишаючи місце для широкого розмаїття розумних інтуїтивних уявлень про цю діяльність. Щоб побудувати таке середовище, необхідна платформа, яка задовольняє зазначеним вимогам щодо підтримки еволюційності та інтеграції різних підходів.

Методологічною основою такої платформи є інтуїтивні уявлення про програмування та вирішення програмних проблем, які зібрані у систему загальних та спеціальних принципів. Проте, перед тим як висвітлювати ці принципи, слід розглянути програмування, і зокрема, вирішення програмних проблем, з найбільш загального кута зору, абстрагуючись від специфіки конкретних предметних областей.

На найвищому рівні розгляду програмування визначається трьома основними аспектами: прагматика, семантика та синтаксис. З них найважливішим є прагматика, оскільки саме тут закладаються основні характеристики як процесу рішення задачі, так і очікуваного результату. В прагматиці визначається мета вирішення задачі, умови вирішення та загальні вимоги до рішення.

Семантика є другим за важливістю аспектом програмування. Тут досліджуються питання вибору засобів вирішення задач на основі прагматичних вимог та їх застосування. Семантика пов'язана з суттю вирішуваних задач, їх прагматичною природою та структурою відношень у множині сенсів або значень, а також представленням "складних" сутностей через більш "прості".

Синтаксичний аспект пов'язаний з правильністю побудови форм нотації для отриманих прагматико-семантичних рішень, їх внутрішньою структурою та іншими аспектами, що визначають спосіб представлення інформації.

Ці аспекти взаємодіють і співпрацюють у вирішенні програмних задач, забезпечуючи адекватне розуміння вирішення будь-якої задачі як взаємодоповнення процесу вирішення та його результату. Це дозволяє забезпечити реальне управління якістю отримуваних рішень, їх надійність, ефективність процесів вирішення програмних задач та ефективне використання інвестицій.

Достатньо поверхневий аналіз основних аспектів програмування дозволяє зробити обґрунтований висновок, що для адекватного розуміння програмування потрібно розглядати всі його три аспекти взаємодоповнення. Усвідомлення цієї взаємодоповнюваності є дуже важливим для розуміння програмування.

Зазначена взаємодоповнюваність між аспектами прагматики, семантики та синтаксису дозволяє їх відокремлювати один від одного, забезпечуючи відносну автономію. Прагматика вирішення програмної задачі може бути відокремлена від її семантики, а семантика – від синтаксису. Таке розуміння приводить до усвідомлення процесу програмування як послідовного сходження від прагматики задачі до її семантики, а від неї – до синтаксису.

Відмінність між підпорядкованістю у випадку "прагматика-семантика" та "семантика-синтаксис" полягає в їх характері. Перший випадок має широке розуміння підпорядкованості, яка не виключає участі суб'єкта у процесі, або того, що семантика є об'єктивно похідною від прагматики. У другому випадку підпорядкованість є більш конкретною, синтаксис є об'єктивною похідною від семантики [8].

Розуміння програмування, яке враховує взаємодоповнюваність прагматики, семантики та синтаксису, вимагає інструментів, що забезпечують таке узгоджене розглядання цих аспектів із можливістю їх відокремлення у викладеному сенсі.

Така точка зору є природною не лише для програмування, а й для будь-якої творчої діяльності. Результатами такого підходу стали численні відкриття різного рівня значущості. Одним з центральних досягнень є виділення системи методологічних принципів, які мають стати основою для будь-якої діяльності [21]. Основними з них є принципи структурності, обумовленості, підпорядкованості, відокремлюваності, професійності та композиційності.

Ці принципи чітко визначають роль взаємодоповнювання основних аспектів як основи для розуміння програмування. Вони також ідентифікують реальний інструмент для здійснення такого взаємодоповнення – композиції. Відповідно до цих принципів,

програмування розглядається як еволюційне взаємодоповнення процесу і його результату. Композиції, як загальнозначущі засоби генезису програм, є ядром СОСрП та забезпечують реальну незамкнутість розглядів в актуальності.

2.7.1. Функціональна парадигма

У цій парадигмі вищий пріоритет надається поняттю функціонала як одного з видів функції, але поняття композиції практично відсутнє і представлене лише декількома неяскравими прикладами, які не узгоджуються ні зі структурою функцій, ні зі структурами даних. Такий підхід призвів до дуже жорсткої структури даних – послідовностей послідовностей, що ускладнює роботу з масивами, записами та іншими важливими структурами даних в реальних програмах.

Хоча творці та послідовники цієї парадигми визнають труднощі, вони переважно зосереджуються на їх каталогізації, а не на розкритті джерел їх виникнення. Головним джерелом є зміщення акцентів з композицій на функціональні структури, що базуються на структурах даних, які не є адекватними логіці програмування, хоча вони відображають специфічні риси дескриптивних розділів. Ці структури обмежені тим, що зайво конкретизують оброблювані об'єкти та ускладнюють вирішення задачі.

Така замкнутість парадигми в актуальності функцій робить її неадекватним збагаченням програмування. Це випливає з огляду на відкритість програмування. Проте, неадекватність не обов'язково є наслідком лише замкнутості платформи в актуальності, існують інші чинники, які можуть сприяти неадекватності.

2.7.2 Об’єкто-орієнтована парадигма

Об'єктно-орієнтована парадигма надає перевагу поняттю об'єкта, розглядаючи його на високому рівні абстракції. Структури даних, функції, композиції та декомпозиції розглядаються як різновиди об'єктів, не індивідуалізовані в універсумі об'єктів на конкретному рівні деталей. Цей інтегрований підхід є перевагою даного підходу. Проте, часом недоліки випливають з його переваг.

Така всеосяжна увага до об'єктів залишила поза увагою не тільки пріоритетність понять "композиція", "функція" і "дані" в цій трійці, але також знехтувала розглядом кожного з цих понять окремо і вивела за межі досліджень питання про причинно-наслідкові зв'язки процесів породження об'єктів з об'єктами породження. З цього випливає, що дана парадигма замкнута в актуальності об'єкта.

2.7.3. Логічна парадигма

Акцент у цій парадигмі робиться на засобах логіки, які використовуються в програмуванні апаратного та програмного забезпечення. Це наближає її до логіко-предметного програмування і зокрема до логіки програмування. Проте, використані в даних засобах логіки, зокрема чисто декларативної, особливо логіки першого порядку, є обмеженням у можливостях. Насправді реальне програмування характеризується застосуванням дескриптивних логік, які базуються на функціоналах вищих типів і використовують композиції як логічні засоби генезису програм. Така логічна парадигма, де декларативні логіки не обумовлені дескриптивними логіками як генетичними структурами, близька до логіки програмування лише за формою, а не по суті.

Зазначені труднощі, з якими зіткнулася дана парадигма, вже відомі. Багато з них випливають з відсутності засобів логік вищих порядків, ігнорування відношень і функціоналів вищих типів, а також неадекватність функціональних структур і структур даних. Намагання об'єднати логічне програмування з функціональним для усунення цих труднощів може не привести до значних нововведень в програмуванні, оскільки основна причина труднощів полягає в відриві логічного програмування від логіки програмування. Це проявляється в ігноруванні композицій, які лежать в основі логік, і які є адекватними експлікаціями засобів генезису програм. Таким чином, дана парадигма залишається замкнутою в актуальності декларативної логіки.

2.7.4. Структурна парадигма

Структурна або систематична парадигма в програмуванні зосереджується на структурах програм, що формуються за допомогою структурованих методів побудови.

Основна увага приділяється не самим програмам, а процесам їх створення. Це збігається з принципами структурності та обумовленості, які відповідають концептуальному базису СОСрП, зокрема, композиційній логіці програмування.

У структурному програмуванні основний розвиток відбувався навколо найважливіших керівних структур, таких як послідовне виконання, галуження та циклування. Це призвело до вагомих результатів, зокрема, у програмній та апаратній інженерії. Проте, не переходили від використання конкретних керівних структур до розгляду адекватних керівних структур як понять з точно визначеними семантикою і синтаксисом. Цей недостатній розвиток заважав здійсненню якісно нового етапу у систематичному підході до програмування. З цього можна зробити висновок, що дана парадигма замкнута в актуальності статичних структур.

2.7.5. Модульна парадигма

Модульна парадигма зосереджує увагу на понятті модуля, що, по суті, є поняттям функції, а не композиції. Проте, ключовою частиною є міжмодульний інтерфейс, що сам є формою композиції. Згідно з логічним ходом речей, слід зосередитися не лише на зовнішніх формах паспортизації міжмодульних інтерфейсів, але і на розкритті їх композицій.

Зміщення акцентів не обмежується лише зосередженням на понятті функції. Часом це зосередження розширюється навіть на поняття даного. Проте головний недолік полягає не в тому, що поняття даного трактується недостатньо широко, як у підходах, що ґрунтуються на абстрактних типах даних. Проблема полягає в тому, що не береться до уваги пріоритетність понять у згаданій тріаді з прагматично-цільового погляду, а також не проводиться ґрунтовний розгляд їх складових. Тому навіть абстрактні типи даних, в їх поточному стані, мало чим можуть допомогти, оскільки ця проблема стосується і їх. Отже, модульна парадигма залишається замкненою в актуальності модуля.

2.7.6. Денотаційна парадигма

Денотаційна парадигма ґрунтується на загальному методі нерухомих точок, з фокусом на отриманні явних визначень за допомогою спеціальних класів системи рекурсивних рівнянь. Цей підхід широко використовується в розробці програмно-апаратного забезпечення та інших галузях знань, оскільки денотативні визначення об'єктів (таких як дані, функції та композиції) часто є більш перевагою ніж доповненням [62-69].

Дана парадигма виступає як метазасіб, що підтримує засоби програмування, такі як композиції, функції та дані. Він не є безпосереднім інструментом для проєктування, програмування чи моделювання. Однак, ще недостатньо розуміння того, як цей метод пов'язаний з конотативно-денотативною генетичною структурою пріоритетності понять, що уповільнює розвиток теорії та практики програмування.

Отже, денотаційна парадигма залишається обмеженою в рамках денотації, що є обмежучим фактором для повного розвитку програмування.

Проведений аналіз основних парадигм програмування показує, що вони дотримуються системи принципів лише формально, а не реально. Тому вищезгадані недоліки притаманні цим підходам у різних ступенях і є принциповою частиною цих парадигм, оскільки вони генезисом зводяться до основних причин їх виникнення.

З урахуванням фундаментальності цих причин, ці недоліки не можуть бути природнім чином усунуті чи зменшені. Тобто взаємне доповнення функціональної, об'єктної, логічної, структурної, модульної та денотаційної парадигм є обмеженим актуальністю кожної з них.

Це свідчить про неможливість в рамках традиційних підходів вирішувати задачі адекватного поєднання процесів програмування у розглядах. Тому необхідним є створення нового інтеграційного підходу, який би зберігав позитивні аспекти парадигм що існують і забезпечував адекватне поєднання як самого процесу програмування, так і його результатів, а також їх взаємодоповнення у розглядах.

РОЗДІЛ 3. Композитосутністні основи СОСрП

Перехід від однотипних до різнорідних абстрактних розділів, в яких відображені відносини між складними структурами, дозволив не тільки усунути виявлені недоліки традиційних методів, але й поєднати їх з підходами, що розвиваються в межах концепцій денотативної семантики.

Особливо характерною рисою денотативного підходу є використання топологій, які виникають з різних часткових порядків. Функції, що виникають з реальних складових програмування мають складну природу. Зокрема, аналогічно до теореми Геделя про неповноту, неможливо зводити ці функції до одного консолідованого денотативного базового поняття. Натомість є можливість створити практичну підставу для індивідуалізації таких функцій у всесвіті, які мають прагматично визначені денотативні характеристики.

Прямо з вищезазначеної прагматичної аргументації випливає, що визначення відношень між складними структурами може служити цією підставою. Ці відношення, як мінімум, дозволяють індивідуалізувати типи програмних функцій, які були раніше визначені. Серед цих типів знаходяться не лише відомі традиційні програмні функції, але і нові, що виникли внаслідок прагматичних вимог, і мають розвинутіші нетрадиційні властивості. До прикладів таких функцій можна віднести будь-яку функцію, яка відповідає умовам обмежень послідовностей структур, розгляданих як найменші верхні обмеження у відносинах між ними, що визначаються собою програмні функції. Легко бачити, що це є неперервні функції у топологіях, які виникають зі згаданих відносин, що мають структуру часткових порядків. Таким чином, за теоремою Тарського про структуру множини нерухомих точок, можна стверджувати, що програмні функції мають нерухомі точки.

Головна важливість цього аспекту полягає не тільки в доведенні наявності нерухомих точок у програмних функціях, але й у можливості відкриття логічного шляху для виявлення тих типів прагматико-обумовлених функцій, де існування нерухомих

точок є певним чином прямим результатом такої обумовленості. Цей підхід не просто відрізняється від відомих традиційних методів, але і є їхнім дуальним, не суперечливим, але суттєво збагачуючим. Отримані тут результати становлять логічну основу для СОСрП і створюють передумови для подальшого наповнення її конкретним змістом.

Представлені експлікативні збагачення ССВ дозволяють виконати ряд прагматично обумовлених конкретизацій та здійснити перехід від погляду на програмування з точки зору його сутності до розвинутішої та вагомої композитосутнісної парадигми, в межах якої програмування сприймається як сутнісно пов'язане розуміння.

Центральною ідеєю композитосутнісної парадигми є композитосутнісне відношення – конкретизація сутнісного відношення, яка враховує прагматичні вимоги. Основою цього відношення є загальне розуміння композиції, де основну увагу приділяється основним аспектам операцій, таким як адекватність, замкнутість, обчислюваність та повнота. Важливо підкреслити, що це розуміння композиції не претендує на всеосяжність операцій та класів композицій, але разом з тим дозволяє належним чином аналізувати прагматично-обумовлений набір метазасобів для маніпулювання композиціями. Ці метазасоби утворюють ядро прагматично-обумовленої композиційної логіки, яка може розглядатися як потенційно відкритий набір інструментів для роботи з композиціями.

Відкрита різноманітність композиційних та операційних можливостей впливає на зсув акцентів у контексті композиційних та програмних задач. Це призводить до необхідності переходу від завдання створення універсального програмного середовища до розгляду питання створення середовища, що ґрунтується на відношеннях між суб'єктами та об'єктами. Цей перехід базується на типізації відкритої компоненти композиційного середовища на підставі суб'єктивної адекватності обмежених систем, що генеруються в цьому середовищі. Процес прагматико-обумовленої індивідуалізації базових компонентів композицій суб'єктом, застосування логіки композицій всередині

цих індивідуалізованих компонентів, сприяє створенню обумовлених прагматикою видів операцій. З цієї причини можна вважати цей процес засобом типізації, а саму діяльність – типізуванням.

$$\begin{cases} \textit{типізатор} - \textit{сутність, що базисно обумовлює операцію}; \\ \textit{тип} - \textit{сутність, що обумовлюється типізатором}. \end{cases}$$

Введений за допомогою цієї експлікації механізм прагматико-обумовленої типізації є основою для концепції СОСрП. Це середовище сприяє обробці прагматико-обумовлених типів та може розглядатися як середовище типізацій, що втілює нелогічну абстракцію широкого спектра прагматико-обумовлених типів.

На практиці, важливість експлікації як методу типізування та самого типу полягає в тому, що кожен тип насправді являє собою інструмент для прагматико-обумовленого опису різних сутностей з композиційної точки зору. Це виправдовує термін "дескрипція" для позначення таких типів, "дескрипти" для їх представників та "дескриптування" для відповідної діяльності.

$$\begin{cases} \textit{дескрипт} - \textit{сутність що дескриптивно обумовлює тип}; \\ \textit{дескрипція} - \textit{суть, що обумовлюється дескриптом}. \end{cases}$$

Отже, ця експлікація розв'язує вже відоме протиставлення між об'єктивною відкритістю засобів композиціювання як генетичних структур та вихідною обмеженістю будь-якого великого класу різноманітних представників композицій. Ця суперечність виникає з самої природи генезису, який, бувши суб'єктно обумовленим процесом, не підлягає внесенню навіть у широкі, але жорсткі рамки. Таким чином, композит і композиція створюють нову композиційну логіку. Ця логіка має в собі відкритість свого носія, що об'єктивно обумовлена природою суб'єктності його представників. Цей носій можна розглядати як універсум композицій - цілісну логічну абстракцію різноманітності композицій. Це дозволяє індивідуалізувати клас загальнозначущих, у цьому контексті логічних, об'єктів універсуму композицій, який можна відповідно назвати метакомпозиціями.

Підхід суб'єкто-об'єктної парадигми у композиційній проблематиці спонукає до вивчення композиційних метазасобів як логічної основи дослідження. Для цього базою послужить вже відома парадигма "розділяй та володарюй", що є фундаментом для наукових уявлень щодо рішення завдань. Формула "ціле є частина агрегації" є засобом прагматико-обумовленого дослідження сутностей. Практичний аналіз її використання дозволяє індивідуалізувати засоби суперпозицій та аплікацій як основні інструменти агрегації. Тому саме в суперпозиціях та аплікаціях можна знайти загальнозначущі інструменти композиційної логіки. Таким чином, композиційна логіка програмування може бути розглянута як універсальний універсум композицій, який являє собою цілісну логічну абстракцію різноманітності композицій, з введеними в ньому метакомпозиціями типу суперпозицій та аплікацій.

Виконані конструкції значно поглиблюють розуміння сутності програмування. Особливо це стосується уточнення ССВ як головної складової суб'єкто-об'єктного погляду на програмування, і його зміщення до композиційносутнього відношення – основи для ґрунтовнішого логічно-предметного осмислення програмування. Саме ця зміна сприяла переходу від підходу, що базується на суттєсутності, до композитосутньої парадигми програмування, і створила конкретну базу для розвитку СОСрП.

На основі результатів, отриманих у попередніх розділах, виконується збагачення концепції суб'єктно-об'єктного середовища програмування. Ця концепція розглядається як відкрито-замкнене середовище, яке підтримує програмну діяльність як процес і його результат.

На перший погляд, СОСРП становить собою інтеграційний засіб, який об'єднує різні прагматично обумовлені підходи до програмних об'єктів, забезпечуючи їх відкрито-замкнене збалансоване поєднання. Основною ідеєю є концепція композитосутнісного відношення.

Основою для вибору підходу до програмної сутності як основи СОСрП є їхнє розглядання у двох взаємодоповнюючих аспектах. Перший аспект відображає сутність

в пасивному вигляді, як об'єкт, який розглядається. Другий аспект пов'язаний з активною роллю сутності як засобу програмування. Ця взаємодія визначає єдиний погляд на проєктносутнісну парадигму (ПСП) як на систему, де існують індивідуалізовані програмні сутності та інструменти їх програмування.

На початковому етапі використання засобів програмування та різноманітності сутностей немає обмежень. Створюється універсум засобів програмування (УЗП), який включає різні методики, підходи та методи програмування сутностей. УЗП є логічною абстракцією різноманітності різних інструментів програмування сутностей.

У рамках еволюційного розвитку ПСП, значущим аспектом є не тільки самі сутності, але й засоби їх програмування. Це пояснюється тим, що вони, крім активної ролі, можуть також виконувати пасивну функцію, виступати не лише як інструменти програмування, але і як його об'єкти.

Ця можливість сутностей взаємодіяти у двох ролях створює можливість для невироджених застосувань, які відрізняються від традиційних. Така можливість не тільки збагачує принцип самозастосованності, який лежить в основі альфа-числення, але й стає прагматичним інструментом для інтенсифікації еволюційного розвитку ПСП.

Для реалізації цієї можливості слід зосередитися на конкретизації носія сутностей як універсуму об'єктів. Цей універсум взаємодіє з універсумом програмованих сутностей, де індивідуалізовані об'єкти виступають у пасивній ролі [70].

Важливо зазначити, що індивідуалізація носія ПСП не обов'язково виключає існування універсуму предметів у ньому. Ролі сутностей часто залишались без уваги, а терміни "об'єкт" та "предмет" вважалися синонімами. Однак часом ця спрощена модель була доречною через простоту аспектів сутностей, що розглядалися.

Позиція різко змінилася внаслідок постійно зростаючої складності у програмно-апаратних розробках. Ця складність насправді привела до відзначення важливості врахування двоєдиної взаємодоповнюваності різних аспектів сутностей в межах їхнього носія, а також взаємодоповнюваності цього носія зі сутностями з УЗП, які збігаються з

цими аспектами в рамках ПСП. Це стає ключовою умовою для збереження ресурсів розробки у сучасних програмних проєктах.

Однак, врахування цієї двоєдиної взаємодоповнюваності, як показує вищезазначене, є лише необхідною умовою. Важливо відзначити, що ПСП, принаймні для збереження інвестицій, повинна виступати як основа для універсального середовища інтеграції (УСІ). При цьому УСІ, звісно ж, повинно взаємодоповнюватися з засобами еволюційного розвитку (ЗЕР) ПСП. Основним атрибутом УСІ, як випливає з усього вищезазначеного, є взаємодоповнюваність.

Підбиваючи підсумок, можна зробити висновок, що прагматично-мотивовані релятивізації будь-яких програмованих сутностей можна розглядати як зовсім природні пояснення програмованих сутностей у якості експлікандів.

Зрозуміло, що викладене вище представляє лише загальний огляд внутрішніх і зовнішніх характеристик СОСРП, яке є триєдиною системою взаємодоповнення з основою в УСІ. Дослідження переконливо демонструють, що серйозне збереження інвестицій у сучасних інформатико-технологічних системах, зокрема системах програмування, неможливе без явного урахування цих характеристик у рамках концептуальної єдиної інтеграційної релятивізації. На меншому рівні, внутрішні характеристики в цій релятивізації повинні враховувати двоєдиність взаємодоповненості, яка реалізується в структурі носія ПСП, тоді як зовнішні характеристики враховуються в індивідуалізації ПСП, яка служить основою для триєдиної взаємодоповненості в рамках УСІ.

До цього часу УЗП та ЗЕР були навмисно інкапсульовані. Нині прийшов час явно розглянути їх в контексті композитосутнісного підходу даної роботи. Результати досліджень однозначно підтверджують, що як сутності з УЗП, так і з ЗЕР можуть виступати носіями прагматично-обумовленої точки зору на генезу сутностей, утворюючи композиції відповідно до прагматики. Основна роль УСІ полягає у заповненні їх реальним змістом та наданні обґрунтованого розуміння реалізації

основного принципу процесональності, яке було досягнуте завдяки проведеним дослідженням. Ключові складові УСІ включають прагматико-обумовлені суб'єкто-об'єктні середовища як засіб для втілення базових композитосутнісних відношень, що виникають з УЗП та ЗЕР. Насправді УСІ є середовищем, яке підтримує рефлексивно-транзитивне замикання композитосутнісного відношення як наслідок взаємодоповнення, зокрема об'єднання базових відношень.

УЗП і ЗЕР є системами з відкритою структурою. Справді, може здатися, що з цього має виплинути природний висновок про те, що СОСрП є УСІ. Однак, через відкриту структуру його складових, УСІ виявляється недостатньо змістовним. Це призводить до необхідності прагматико-обумовленого збагачення. Основна суть полягає у тому, що хоча УЗП та ЗЕР не можуть представляти всю систему несуперечливо через їхню відкритість, це не перешкоджає розглядати окремі класи генетичних структур, які виникають внаслідок застосування окремих прагматико-обумовлених засобів еволюційного розвитку до конкретних наборів прагматико-обумовлених базових композицій. З конкретного погляду, така сукупність рішень є відкритою системою. Таким чином, СОСрП можна вважати конкретною реалізацією УСІ через обмеження їх прагматико-обумовленими початковими умовами [71].

Щодо засобів еволюційного розвитку, сучасне їх розуміння, яке базується на парадигмі "розділяй та володарюй", було піддане серйозному переосмисленню з плином часу. З огляду на результати попереднього ЗЕР, були вироблені аплікація та суперпозиція композицій. Аплікація для будь-якої композиції та функції встановлює нову функцію, яка виникає після застосування композиції до аргументу. Суперпозиція ж є композицією n-арних функцій. З практичної точки зору, перша з них допомагає послідовно розкривати складні композиції до простіших (аспект підпорядкування), а друга є засобом визначення прагматико-обумовлених поділів сутностей на взаємодоповнюючі частини. Початкові обмеження для УЗП зумовлені прагматиками, що виникають у процесі вирішення програмних задач.

Отже, з вище наведеного випливає обґрунтований висновок, що СОСРП може бути розглянуто як УСІ, але з певними початковими умовами, які будемо називати програмною релятивізацією УСІ. Зрозуміло, що СОСрП зводиться до програмної релятивізації УСІ з точки зору експлікації. Всі подальші дослдіження будуть спрямовані на збагачення та поглиблення цієї релятивізації, як на рівні прагматико-семантичних досліджень, так і на рівні програмної реалізації.

Закрита складова СОСрП представляє найбільш суттєві концепції щодо організації програмування. Тут прагматика виступає ключовим джерелом конкретизації, яка є найбільш суттєвою, загальною та незалежною від конкретної специфіки аспекту рішення завдань. Принцип обумовленості покладає основний акцент на прагматичні дослідження, зорієнтовані на розуміння генезису відповідних рішень. Це охоплює пошук генетичних структур як конкретних уточнень розробників щодо способів інтеграції підзавдань. Відповідно до принципу композиційності, програмні генетичні структури є композиціями. Таким чином, результати прагматичних досліджень надають змогу зрозуміти адекватні методи генезису складніших рішень з простіших. Це розуміння виражається у вигляді класу алгебраїчних операцій та прагматико-обумовленої ідентифікації базових композицій. Вони становлять ключову роль для реалізації принципу підпорядкованості під час прагматико-семантичних досліджень. Отже, основний елемент закритої частини СОСрП полягає у конкретизації загального поняття композиції через прагматико-обумовлену релятивізацію з урахуванням різноманітних рівнів загальності та умов застосування генетичних структур.

Програмування передбачає аналіз різних аспектів, характеристик і взаємозв'язків, які відображають сутність предметної області. Відкрита частина СОСрП включає різноманітні сутності, які є можливими збагачення закритої частини, обумовлені відповідними логіко-предметними зв'язками. "Різноманіття" тут не обмежується конкретними варіантами, а уособлює недвозначну логічну абстракцію цілісного спектра сутностей, що становлять його суть в СОСрП. Це дозволяє працювати з цією

абстракцією, не враховуючи суперечностей у різноманітності, і належним чином продовжувати її до конкретних представників або класів в потрібний момент і місце.

Іншими словами, представники відкритої частини середовища або їх класи розглядаються як можливі результати відповідних процесів, що не суперечать логіці, що виникають з логічного ядра середовища. Сутністю самого процесу збагачення є третя основоположна складова СОСрП - його логіко-предметні (композито-сутнісні) зв'язки. Ця складова середовища відіграє ключову роль, і саме композито-сутнісні зв'язки між двома різними сутностями створюють цілком нову сутність - відкрито-замкнене середовище, в якому ці сутності функціонують як відкрита та замкнена частини. У контексті, логіко-предметний зв'язок виступає як причинно-наслідковий зв'язок між властивостями та сутностями, що володіють цими властивостями. В цьому випадку логіко-предметні відношення передають взаємозв'язок, у якому сутність має певну властивість. Коли мова йде про композито-сутнісний зв'язок, ця властивість полягає в композиційності. Логіко-предметні відношення виступають як безсуперечна логічна абстракція універсального різноманіття процесів продовження композицій до сутностей, властивість яких вони відображають.

3.1. Композиції та функції у СОСрП

Основна мета концепції СОСрП полягає в реальній підтримці вирішення програмних задач. Для досягнення цієї мети, принципи обумовленості, функціональності, експлікативності композиційності та процесійності грають важливу роль на загальному рівні. Більшість традиційних підходів номінально дотримуються цих принципів, але фактично їх підтримка вирішення програмних задач обмежується просто забезпеченням зручної нотації для рішень.

Такі обмеження є суттєвими та фундаментальними у традиційних підходах, і вони не можуть бути легко усунуті. Щоб досягти мети концепції СОСрП, необхідно створити новий інтеграційний підхід до програмування, який успадкував би позитивні сторони

традиційних підходів, але також суттєво розвинув би їх, зокрема, у проблематиці взаємодоповнення процесів програмування та їх результатів.

Цей новий підхід повинен засновуватися на засобах підтримки прагматико-семантичних розглядів у програмуванні, в яких ключовою роллю відіграють функції. Основна ідея полягає у тому, що такий підхід реально, а не тільки у теорії, підтримуватиме взаємодоповнення основних аспектів програмування, забезпечуючи більш ефективні результати.

Таким чином, щоб досягнути поставленої мети, необхідно створити новий підхід до програмування, який буде враховувати принципи СОСрП та зосередиться на розвитку функцій та композицій засобами підтримки прагматико-семантичних розглядів.

У програмуванні функції та композиції відіграють значну роль, вони розглядаються у контексті принципів функціональності, композиційності, експлікативності та процесійності, про які було сказано вище. Тепер настає час детальніше розглянути ці поняття та внести у них певне предметне збагачення, зберігаючи при цьому узгодженість з принципами функціональності, композиційності та процесійності. Це збагачення функцій та композицій повинно бути відкрито-замкненим, тобто допускати збагачення, але в межах логічної збіжності.

Центральним аспектом є розуміння логіко-предметної природи функцій та композицій, оскільки це дозволяє створювати середовища програмування. З огляду на принцип обумовленості, першим етапом збагачення є розгляд композицій.

Оперуючи поняттями композицій у програмуванні можливо виділити їх змістовні властивості, пов'язані як з самим програмуванням, так і з його загальною природою, що не залежить від конкретної програми.

Специфічною рисою програмування є те, що у більшості випадків програміст не має формального опису програмного завдання і навіть не має достатньо змістовного цілісного уявлення про нього. Уявлення про програму з'являється поступово і генерується програмістом відповідно до його суб'єктивного бачення того, як задача

повинна бути вирішена, враховуючи парадигму нерозривності аналізу та синтезу. Таким чином, з самого початку у програмуванні зустрічаємо суб'єкто-обумовлений генезис розуміння сутності задачі шляхом інтеграції розуміння її підзадач.

В цьому контексті, композиції, як структури генерації програм, виступають як основний інструмент змістовного суб'єкто-орієнтованого збагачення розроблюваних програм. Тому до композицій пред'являються вимоги, що надають більш конкретного змісту зазначеній вище основній властивості композицій - евідентності.

По-перше, це вимога адекватності композицій змістовному розумінню програмістом вирішуваної задачі. Композиції повинні відповідати уявленням програміста щодо програмування взагалі та конкретної програмної задачі. Якщо композиції не є адекватними, програміст змушений змінювати свої уявлення про вирішення задачі, щоб відповідати наявним генетичним структурам. Такі трансформації можуть призвести до багатьох помилок у програмуванні. Тому, основним критерієм адекватності класу композицій є прагматико-обумовлене програмування.

Властивість адекватності композицій прямо пов'язана з розробником програми і є одним з важливих факторів, який впливає на релятивізацію програмування. Проте, потрібно також враховувати фактори, які стосуються програмування як діяльності, спрямованої на створення програм. У цьому контексті, вимагається замкненість композицій. Її зміст полягає в розгляді програми з точки зору сутності, що відповідає його програмній функції. Програми створюються саме для реалізації їх суті, що передбачає наявність суб'єкту, який здійснює цей процес.

Таким чином, у першому наближенні програма є сутністю, що допускає суб'єктну імплементацію його програмної функції. Роль композиції в програмуванні, згідно з методом сутесутнісної релятивізації, полягає в об'єктивізації цієї релятивності в кожному конкретному програмному рішенні. Іншими словами, композиція повинна втілювати програмність будов, бути замкненою в програмних функціях. Таким чином, застосування композиції до програм як суб'єкто-залежних носіїв ССВ не виходить за

рамки таких програм. Це випливає з необхідності тотальності композиції, яка означає, що коректно застосовується до будь-яких програм.

Властивості адекватності, замкненості та тотальності знаходяться в центрі обмеження класу композицій, але цей клас залишається дуже загальним. Тому, для подальшого збагачення класу композицій, необхідно вводити структури.

Структури композицій можна легко отримати, внесенням структур до класів програм, на яких задані композиції, а структури в класах програм можна визначити на основі структур даних. Важливо підкреслити, що структури в класі програм і даних насправді індукуються структурами композицій, згідно з принципом обумовленості. З огляду на пріоритетність поняття композиції, переходимо до розгляду структур композицій програм, пов'язаних зі структурами програмних функцій і даних.

3.2. Характеристики функцій та даних у СОСрП

Зрозуміло, що вивчаючи композиції, важливо залучити властивості функцій, які використовуються в цих композиціях. Одна з таких властивостей вже відображена у визначенні композиції як n-арної операції. Представляючи вихідні функції у вигляді кортежу довжини n, ми можемо розрізняти ці функції, виділяти їх аргументи та значення за номерами (іменами).

Інша властивість функцій, яка використовується в композиціях, випливає безпосередньо з властивостей програмності та тотальності композицій. Ця властивість забезпечує можливість знаходження значення будь-якої вихідної функції на будь-якому заданому наборі даних, якщо відповідне значення функції визначено.

Властивості даних, слід розглядати у наростаючому порядку. Спершу слід з'ясувати, які класи композицій можуть бути отримані за умови розгляду мінімальних вимог - абстрактних даних. В цьому випадку елементи обраного класу даних розуміються як такі, що не глибоко розкривають їхню структуру. Отриманий рівень розгляду називається абстрактним.

Абстрактні композиції представляють безліч різноманітних можливостей, і поки що описане про композицію взагалі не є достатнім для повного зрозуміння всього їх розмаїття. Тому на даному етапі доцільно розглянути деякі представницькі приклади абстрактних композицій.

Найбільш поширеною абстрактною композицією є множення. Суть її полягає в тому, що вона у відповідь двом функціям f та g, створюючи нову функцію h, таку, що на будь-якому вхідному абстрактному значенні вона дає значення h(f(x), g(x)).

На цьому рівні можна представити цілий набір композицій. Однак практично всі вони будуть певною мірою похідними від композиції множення. Для збагачення абстрактних композицій необхідно внести змістовне збагачення абстрактного типу даних. Спочатку можемо виділити два спеціальних елементи, які будемо інтерпретувати як "true" та "false" відповідно. Додавши до типу абстрактних даних множину логічних значень, які відповідають цим спеціальним елементам, а програмні функції - функціям, що приймають логічні значення, тобто предикатам, ми зможемо визначити цілий ряд корисних абстрактних композицій [50].

Отже, на абстрактному рівні розгляду з конкретними елементами можна визначити широкий клас композицій. Проте, з міркувань змісту зрозуміло, що цей клас не обмежується лише абстрактними композиціями. В програмуванні можуть використовуватись засоби генезису, які істотно враховують внутрішню структуру даних. Це дозволяє розглядати композиції збагаченого класу.

Один з мінімальних змістовних збагачень розгляду - врахування множинного типу даних. Множинність дозволяє розглядати дані як скінчені множини, включаючи множину абстрактних елементів. Серед композицій множинного рівня можна виділити композиції виділення за умовою та композицію застосування за умовою .

Множинний тип даних представляє досить різноманітний клас композицій, проте обмеження на цьому рівні є недостатнім. Це очевидно з природних міркувань, оскільки

у програмуванні можуть використовуватись не тільки внутрішньо-множинні структури даних, але й складніші структури.

Однією з таких структур є кортежні структури, які вже досить добре досліджені. Вони часто вважаються надто конкретними, що ускладнює розуміння функціональних та композиційних структур. Програмуванню більш властиве використання іменних структур, зокрема структур, що моделюють скінчені множини іменованих елементів - іменних множин. Ці структури добре підходять для моделювання кортежних структур. Тому доцільно залучити іменні множини як новий тип абстракції даних, що надає назву іменному рівню розгляду.

Аналіз різних композицій програм підтверджує, що вони часто використовують властивість даних бути множинами, складеними з іменованих елементів. Іменні множини, відповідно, являють собою скінченні функціональні бінарні відношення, засновані на принципі іменування.

Перевагами застосування іменних множин є те, що вони допомагають уникнути неадекватності використання стандартного іменування в задачах програмування, спрощують роботу з послідовними структурами даних, забезпечують зручний спосіб представлення структур типу масивів, файлів, списків, таблиць, черг, стеків тощо.

Іменні множини є потужним інструментом для специфікації важливих класів структурованих сутностей, які зустрічаються у програмуванні. Вони можуть представляти різні види структур. Наприклад, іменні множини можуть бути пов'язані з поняттями змінних та їх значень. Також, можна застосовувати іменні множини для представлення пам'яті обчислювальної машини, де адреси комірок є іменами, а вміст комірок - значеннями імен.

Іменне трактування явно показує, що оператори присвоювання змінюють стан пам'яті комп'ютера, тому воно надає кращий підхід для дослідження функцій, які визначають семантику програм. Однак, таке трактування призводить до ускладнення

композиції, що називається мультиплікуванням, яке враховує специфіку роботи з пам'яттю.

Мультиплікування - це бінарна операція, яка ставить у відповідність двом іменним функціям нову іменну функцію, значення якої на іменному даному визначається послідовним виконанням двох функцій: спочатку виконується перша функція, результати накладаються на вихідні дані, а потім виконується друга функція. Таким чином, наслідком послідовного виконання операторів будуть результати виконання другої функції, накладені на результати першої.

Мультиплікування іменних функцій є компромісом між простотою та адекватністю трактування семантики операторів присвоювання. Воно враховує іменну специфіку даних та функцій, що дозволяє більш ефективно використовувати апарат іменування у семантичних розглядах. Натомість складність мультиплікування пояснюється не недоліками іменного трактування, а взаємозв'язком між людиною та комп'ютером у людино-машинній системі.

Композиція мультиплікування виступає в основі важливих іменних розумінь методів зациклювання, які застосовуються на абстрактному, множинному та кортежному рівнях. Під іменною композицією циклування розуміється бінарна операція, яка ставить у відповідність іменному предикату та іменній функції нову іменну функцію, значення якої на іменному даному визначається таким чином: вона приймає перше значення послідовності, для якого всі попередні елементи послідовності задовольняють заданому предикату.

Аналогічно, композиція галуження визначається тернарною операцією, яка ставить у відповідність іменному предикату та іменним функціям нову іменну функцію, значення якої на іменному даному визначається так: вона приймає значення першої функції, якщо предикат виконується, і значення другої функції, якщо предикат не виконується.

Поки що відмінність між абстрактною та іменною композицією галуження здається несуттєвою. Очевидно, що композиція може бути розглянута як окремий випадок абстрактної композиції галуження, де одна з функцій є пустою. У абстрактному випадку пуста функція є абстрактною функцією, яка не визначена для жодного абстрактного значення.

Принципова різниця між двома трактуваннями стає очевидною, коли ми розглядаємо паралельне виконання операторів. Наприклад, паралельний оператор присвоювання фактично включає два окремих оператори присвоювання, які здійснюють взаємний обмін значеннями змінних. На абстрактному рівні важко визначити композицію, яка б узгодила таке паралельне виконання операторів. Проте, на іменному рівні ми можемо використовувати розпаралелювання як бінарну операцію, що ставить у відповідність двом іменним функціям нову іменну функцію.

Композиції множення та мультиплікування задають семантику засобів послідовного виконання функцій, що є досить важливим на абстрактному рівні розглядів. Проте, іменний рівень дозволяє розглядати взаємодії, які не можуть бути враховані на абстрактному рівні та дуже обмежено представлені на кортежному рівні. Особливо це стосується іменних суперпозицій - іменні оператори, які дозволяють взаємодіяти між функціями та забезпечують більш гнучкі можливості у розгляді взаємодій функцій на іменному рівні.

Іменний рівень розгляду дозволяє впровадити широко застосовувані засоби програмування. При цьому композиції, які використовують властивості даних, визначених на абстрактному, множинному та іменному рівнях - це властивості бути виділеним елементом, скінченою множиною чи іменною множиною. Це дозволяє стверджувати, що цих властивостей достатньо для визначення класу композицій програм. Інші властивості даних, що використовуються у композиціях, можуть бути зведені до розгляду перерахованих властивостей. З цього випливає згадуваний раніше принцип зводимості, що не ставить перед собою мету вичерпно зазначити всі аспекти

програмування, а лише фіксує сучасний стан речей на підставі аналізу актуальних методів вирішення задач. Також, він не вичерпує всіх аспектів композиційних розглядів, але становить підставу для подальших досліджень і розвитку. Отримані результати надають підстави для принципового збагачення композиційних розглядів, а також детальніше розуміння фундаментальної ролі композиції у програмуванні.

Головним напрямком збагачення є аналіз загального поняття композиції. Велике значення має розкриття основних загальних властивостей композицій - тотальності, адекватності та замкненості. Ці властивості допомагають обґрунтувати прагматичну обумовленість та відносність виокремлення композицій як засобів проєктування серед різноманіття алгебраїчних операцій. Ці властивості конкретизують важливі взаємозв'язки між ключовими суб'єктами проєктного рішення - його розробником та користувачем.

Зокрема, властивість адекватності обмежує композицію як алгебраїчну операцію вимогою відповідності її змісту суб'єктивним уявленням про процес вирішення задачі. Властивість обчислюваності також є важливим обмеженням композиції, забезпечуючи можливість сприйняття отриманого рішення користувачем. Ця трактовка представляє суттєве узагальнення традиційного розуміння обчислюваності і природно зводиться до нього в тих випадках, коли користувачем рішення є комп'ютер. Вимога тотальності підкреслює, що сьогодні ще не всюди є прагматичні мотивації для розгляду не всіх визначених засобів генезису рішень.

Виявлені властивості суттєво збагачують клас композицій як алгебраїчних операцій. Проте, він залишається надто загальним і, отже, недостатньо виразним. Тому потребує подальшого збагачення. Одним із ключових опосередкованих збагачень класу композицій є прагматико-обумовлена типізація сутностей, на яких можуть бути застосовані композиції. Збагачення композицій можна досягнути шляхом ПОТ класів функцій, на яких визначені композиції, і збагачення у класі функцій визначати на основі

ПОТ структур даних. Збагачення в класі функцій і даних фактично виникають зі структур композицій, згідно з принципом обумовленості.

Тенденція виокремлення поняття композиції не завжди отримує належне визнання і розуміння, навіть в контексті різних методів програмування. Різні підходи зосереджуються на різних аспектах останнього, таких як функції, модулі або дані, і не завжди відводять належну увагу поняттю композиції. Пріоритетність повинна бути надана поняттю композиції, розглядаючи структури композицій взаємодоповненні зі структурами функцій і даних.

Аналіз композицій програмування показав, що вони використовують властивості даних, такі як абстрактні дані, кортежі та множини даних. Крім того, ці кортежі та множини можуть бути абстрактними, іменованими або метаіменованими.

На основі наведеної ПОТ даних та функцій була сформована первинна класифікація композицій, що включає абстрактні, іменні та метаіменні класи композицій. З урахуванням кортежного типу було виявлено, що кортежні функції стають занадто обтяжливими для синтезу, і їх структури можуть бути замінені іменними множинами. Також підкреслено недоцільність вилучення будь-якого з перерахованих типів композицій, окрім кортежного, оскільки це зашкодить змістовній адекватності розглядів.

3.3. СОСрП як інтерсуб’єктивна платформа програмної релятивізації

Основу інтерсуб’єктивної парадигми складає евідентне основоположення про цілісне розуміння програмування, як діяльності, що обумовлена програмою. Власне сама розбудова цієї парадигми, як платформи програмної релятивізації, представляє собою покрокове продуктивне збагачення даного основоположення. В цій розбудові реалізується тренд "від цілісного уявлення до продуктивного розуміння" програмування. Продуктивність тут розуміється як націленість на технологізацію.

З наведеного основоположення випливає, що для продуктивного збагачення цілісного розуміння програмування необхідно спочатку реально збагатити розуміння

програми. Трактування терміна "програма" як уподібнення суттєвої риси видається тут найбільш відповідним прагматиці. Цим продуктивно збагачується розуміння програмування як взаємодоповнення двох об'єктивно незвідних один до одного модального та реального типів абстракцій – сутності – того, що може бути, та суті – сутності, що є. Таким чином, отримуємо продуктивне збагачення вихідного основоположення: програмування – діяльність, обумовлена програмним уподібненням, що обумовлене програмою продуктивне збагачення сутесутнісного уподібнення.

Наступний крок побудови СОСрП, очевидно, пов’язаний зі збагаченням програмного уподібнення. Засаднича особливість останнього обумовлена зазначеною неможливістю об'єктивного зведення модальності сутності у сенсі можливості створення її подоби та реальності суті як обумовленої програмою актуалізації цієї модальності. З цього безпосередньо випливає біабстрактність СОСРП як носія програмних уподібнень.

Будь-яке конкретне програмне уподібнення виникає на основі обумовлення сутності – зв’язування її деякою умовою. З вищенаведеного зрозуміло, що носієм таких умов може бути тільки суб’єкт обумовлення як інструмент об’єктивізації, а уподібнення є реалізацією взаємодоповнення активної та пасивної форм обумовлення. Тому об'єктивізація активно-пасивного взаємодоповнення є основною та безальтернативною передумовою реального осучаснення розуміння програмування як рефлексивно-транзитивного замикання породжуваного суб'єктом програмного уподібнення. Визначальною для об’єктивізації активної ролі суб’єкта в контексті побудов у СОСрП є концепт – суть – уподібнення згаданого носія, представлена у вигляді тієї чи іншої специфікації . Подобою ж як єдиним наслідком уподібнення, тобто – програмою у наведеному вище сенсі – об’єктивізується пасивна складова обумовлення.

Концепт забезпечує цілісність побудов, а програма – їх продуктивність. Перехід забезпечується через програмне уподібнення. Важливо відзначити, що використані види роду сутностей – концепт, програма, обумовлення вводяться інтенсіонально – не через

актуальну заданість їх об’ємів, а як абстракції відповідних суб'єктних обумовлень, тобто оракули. Таким чином, наведена концептопрограмна дефініція представляє собою відкрито-замкнене оракульне середовище. При цьому, наведена дефініція, зважаючи на реальність її побудови, носить виражений цілісний характер. Отже, потребує подальшого продуктивного збагачення.

Розгляд згаданих вище оракулів як відкрито-замкнутих середовищ втягує до розгляду оракули вищих типів – оракульні схеми і є джерелом підвищення продуктивності побудов. Особливе місце серед них займають композитологічні, тобто загальнозначущі композиційні, та, насамперед, композитні (базисні) схеми. Результати досліджень композиційного програмування свідчать про те, що таке композиційно-композитне збагачення концептопрограмної парадигми є лейбніцевим. Тобто, СОСрП як імплементація цілісного осучаснення розуміння програмування експлікативно зводиться до композиційно-композитого збагачення наведеної концептопрограмної дефініції. Тому всі подальші збагачення носять вже не цілісний, а продуктивний, орієнтований на технологізацію, характер і пов’язані з суб’єктно-орієнтованими замиканнями середовищ до відповідних систем.

3.4. Логіко-предметні передумови програмної релятивізації

З вищенаведеного випливає, що будь-яка система виникає як результат програмного уподібнення – покрокового концептопрограмного (в цьому сенсі, суб’єктоорієнтованого) замикання СОСрП – відкрито-замкненого композитологічного оракульного середовища. Це здійснюється шляхом актуалізації відповідних оракулів середовища. Таким чином, концептопрограмне замикання складає основу методу програмної релятивізації як засобу специфікації системи. Отримувана система є суб’єктоорієнтованою системою – засобом імплементації активної ролі суб’єкта у програмуванні. Інтенсіонально, вона є наслідком послідовності наступних специфікацій:

1) композитів та композицій – уточнень базових та похідних генетичних структур як концептів, притаманних як суб'єкту, так і об'єкту програмування;

2) базових предметних операцій, що з прагматичних причин не потребують додаткової деталізації;

3) композито-композиційних інтерфейсів;

Для композито-композиційної актуалізації СОСрП залучимо апарат примітивної програмної алгебри, сигнатурні операції якої складають основу програмного уподібнення створюваної системи. У якості репрезентативного прикладу предметної основи побудови суб'єкто-об'єктної системи оберемо арифметичні перетворення, що уточнюються як обчислювальні багатомісні арифметичні функції та предикати (надалі, функції та предикати, відповідно). Особливості породження системи розглянемо на простих репрезентативних прикладах програмування арифметичних перетворень. Ці побудови мають на меті продемонструвати загальні особливості застосування СОСрП для прагматико-обумовленого породження системи та її використання. Тому зазначені вище предметні домовленості не впливають на репрезентативність побудов.

Таким чином, носій £ арифметичної примітивної програмної алгебри складають n-арні частково-рекурсивні арифметичні функції виду $N^n \to N$ та n-арні частково-рекурсивні арифметичні предикати виду $N^n \to \{T, F\},\ n \in N$ (далі – функції та предикати). Сигнатуру ж застосованої алгебри (позначатимемо тут $\boldsymbol{\Omega}$) складають операції суперпозиції, розгалуження і циклування, що є адекватними уточненнями основних методів конструювання програм. Нагадаємо формальні визначення цих операцій, деякі позначення та результати. Зауважимо, що при акцентуванні уваги на генетичних особливостях розглядуваних функцій та предикатів у їх позначенні перевага надаватиметься операторній, а при акцентуванні на результатах застосування композицій – термальній формам запису.

Нехай задані m функцій однакової арності, наприклад, $k-f_1,\dots,f_m$, m-арна функція f та -арний предикат h, $m,k\in N$. Розглянемо наступні нові k-арні функції g,d,c , значення яких на аргументі $\langle a_1,\dots,a_k\rangle$ задаються так:

$$g(\langle a_1,\dots,a_k\rangle)\cong f(\langle f_1(\langle a_1,\dots,a_k\rangle),\dots,f_m(\langle a_1,\dots,a_k\rangle)\rangle),$$

$$d(\langle a_1,\dots,a_k\rangle)\cong\begin{cases}f_1(\langle a_1,\dots,a_k\rangle),\text{якщо } h(\langle a_1,\dots,a_k\rangle)=T\\ f_2(\langle a_1,\dots,a_k\rangle),\text{якщо } h(\langle a_1,\dots,a_k\rangle)=F\end{cases}$$

$c(\langle a_1,\dots,a_k\rangle)\cong a_1^j$, де a_1^j – перша компонента першого кортежу послідовності кортежів $[\langle a_1^i,\dots,a_k^i\rangle]_{i=1,2,\dots}$, де $a_s^1=a_s, s=1,\dots,k$, $a_s^{i+1}=f_s(\langle a_1^i,\dots,a_k^i\rangle)$, для якого $h\big(\langle a_1^j,\dots,a_k^j\rangle=F\big)$, за умови, що для усіх $r=1,\dots,j-1$ значення $h(\langle a_1^r,\dots,a_k^r\rangle=T)$, (≅- розуміється як умовна рівність).

Будемо казати, що функція g є результатом застосування композиту $m+1$-арної суперпозиції S^{m+1} до кортежу функцій $\langle f,f_1,\dots,f_m\rangle$, функція d – композиту тернарної операції розгалуження $\lozenge$ до кортежу функцій $\langle h,f_1,f_2\rangle$, а c – композиту $k+1$-арного циклування до кортежу $\langle h,f_1,\dots,f_k\rangle$. Тобто, $g\equiv S^{m+1}(\langle f,f_1,\dots,f_m\rangle)$, $d\equiv\lozenge(\langle h,f_1,f_2\rangle)$ та $c\equiv *^{k+1}(\langle h,f_1,\dots,f_k\rangle)$.

Зміст уведених композитів ілюструється наступними блок-схемами (рис.3.1).

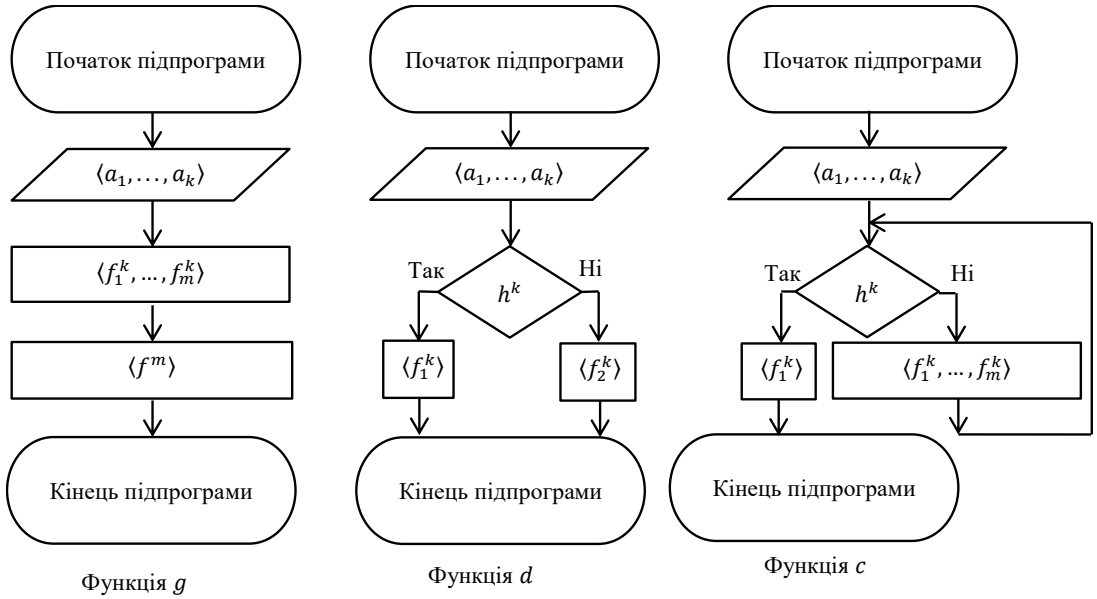


Рис.3.1. Блок-схеми уведених композитів

Позначимо $[\sigma]_\Omega$ замикання множини функцій та предикатів σ операціями примітивної програмної алгебри, 0^1 є 0-функцію, що будь-якому натуральному числу ставить у відповідність 0, тобто: $S^2(0^1, I_1^1)(\langle n\rangle) = 0|_{n\in N}$, s^1 – функцію слідування: $S^2(s^1, I_1^1)(\langle n\rangle) = n + 1|_{n\in N}$, $+^2$ – суму двох натуральних чисел: $S^3(+^2, I_1^2, I_2^2)(\langle n, m\rangle) = n + m|_{n,m\in N}$, $\times^2$ – добуток двох натуральних чисел: $S^3(\times^2, I_1^2, I_2^2)(\langle n, m\rangle) = n \times m|_{n,m\in N}$, $<^2$ – предикат "менше": $S^3(<^2, I_1^2, I_2^2)(\langle n, m\rangle) = \begin{cases} T, \text{якщо } n < m \\ F, \text{якщо } m \leq n \end{cases}|_{n,m\in N}$ та I_n^m – селекторну функцію: $I_n^m(\langle a_1, \dots, a_m\rangle)|_{m,n\in N} = \begin{cases} a_n, \text{якщо } n \leq m \\ \perp, \text{інакше} \end{cases}$.

Справедливо: $[0^1, s^1, +^2, \times^2, <^2, I_n^m]_\Omega = £$. Даний результат разом з вибором операцій з **Ω** у якості актуалізації базових генетичних структур (композитів) СОСРП забезпечує логіко-предметну основу для продуктивної специфікації арифметичної системи. Для цього принциповим є інтерфейс між композитами з **Ω** та похідними від них композиціями. Обумовлені композитами редукційні структури відіграють ключову роль у побудові таких інтерфейсів. Це забезпечує головну особливість створюваних таким чином систем – вони реально, а не лише номінально підтримують причинно-наслідкове взаємодоповнення двох складових вирішення будь-якої програмістської задачі – програмування як породження та застосування композицій та програми як наслідку програмування.

3.5. Редукційні аспекти програмної релятивізації

Вибір сигнатури **Ω** в якості концепту СОСРП об’єктивізує роль суб’єкта програмування арифметичної системи. Це в свою чергу дозволяє залучити для її дослідження інструмент логіко-математичних специфікацій семантико-синтаксичних аспектів програмування арифметичних задач.

Рішення будь-якої задачі, як відомо, є інтеграцією рішень її підзадач. Причому роль суб’єкту у генезисі рішення є визначальною. Якщо задача проста, то інтеграція тривіальна і, зазвичай, явно суб’єктом не виділяється. У разі ж, коли задача складна, інтеграційний аспект її вирішення домінує, оскільки власне ним і визначається

складність. У цьому випадку, побудова рішення передбачає використання двох типів логіко-математичних специфікацій – задач і засобів їх інтеграції. У контексті розглядуваної системи, засоби інтеграції є генетичними структурами – похідними згаданих генних структур або композитів. Що ж до специфікацій задач, то основу їх складають багатомісні функціональні та декомпозиційні структури. Останні базуються на обумовлених зафіксованими генними структурами спеціалізаціях уведених в редукційних схем. Розглянемо їх і почнемо з більш простих схем, обумовлених композитами суперпозиції $S^n, n = 2, 3, ...$ та розгалуження $\Diamond$.

Кортеж функцій $\langle g_1, \ldots, g_m \rangle$, де $g_1, \ldots, g_m$ – функції виду $N^k \to N|_{k \in N}$ називається S^{m+1}-редукцією k-арної функції f, якщо існує така функція $g: N^m \to N|_{m \in N}$, що $f = S^{m+1}(\langle g, g_1, \ldots, g_m \rangle)$.

Кортеж n-арних функцій $\langle g_1, g_2 \rangle$ називається $\Diamond$-редукцією n-арної функції f якщо існує n-арний предикат h такий, що $f = \Diamond(\langle h, g_1, g_2 \rangle)$.

Безпосередньо з наведених визначень випливають прості та корисні необхідні ознаки S^{m+1}-та $\Diamond$-редукційності. Називатимемо рангом кортежу функцій $\langle g_1, \ldots, g_m \rangle$ множину $r\langle g_1, \ldots, g_m \rangle \equiv \{\langle g_1(\langle a_1, \ldots, a_k \rangle), \ldots, g_m(\langle a_1, \ldots, a_k \rangle)\rangle | g_i(\langle a_1, \ldots, a_k \rangle) \in ran(g_i)|_{i=\overline{1,m}} \& \langle a_1, \ldots, a_k \rangle \in \bigcap_{i=\overline{1,m}} dom(g_i)\}$. Тоді, якщо $\langle g_1, \ldots, g_m \rangle$ є S^{m+1}-редукцією функції f то $r\langle g_1, \ldots, g_m \rangle \subseteq dom(g)$ та $dom(f) \subseteq \bigcap_{i=\overline{1,m}} dom(g_i)$. Та, якщо $\langle g_1, g_2 \rangle$ є $\Diamond$-редукцією функції f то $ran(f) \subseteq ran(g_1) \cup ran(g_2)$ і $dom(g_1) \cup dom(g_2) \subseteq dom(h)$.

Тепер звернемось до циклування. Кортеж функцій $\langle g_1, \ldots, g_m \rangle$, де $g_1, \ldots, g_m$ – функції виду $N^m \to N|_{m \in N}$ називається $*^{m+1}$-редукцією m-арної функції f, якщо $f = S^{m+1}(\langle f, g_1, \ldots, g_m \rangle)$. Зв'язок $*^{m+1}$-редукції з операцією циклування розкривається наступним результатом.

Достатня умова $*^{m+1}$-редукційності. Нехай h – m-арний предикат та $g_1, \ldots, g_m, f$ – m-арні функції такі, що $f = *^{m+1}(\langle h, g_1, \ldots, g_m \rangle)$. Тоді $\langle g_1, \ldots, g_m \rangle$ – $*^{m+1}$-редукція функції f.

Для доведення достатньо переконатись, що $*^{m+1}(\langle h, g_1, \ldots, g_m\rangle) = S^{m+1}(\langle *^{m+1}(\langle h, g_1, \ldots, g_m\rangle), g_1, \ldots, g_m\rangle)$ за умови, що $\langle g_1, \ldots, g_m\rangle$ – $*^{m+1}$-редукція функції f. Справедливість цього випливає з визначень сигнатурних операцій примітивної програмної алгебри.

Парадигмне значення уведених редукцій та їх властивостей у тому, що вони конкретизують вирішення задач як покрокове розкриття структур генезисів їх рішень, з урахуванням яких процедурні, алгоритмічні, програмні тощо. Реалізації цих рішень, зокрема їх нотація у мовах програмування отримуються вже автоматично з обумовленою їх побудовою коректністю. Саме це складає засадничу особливість методу програмної релятивізації. Рішення задачі, при цьому, представляється як обумовлене прагматикою і породжене у суб'єкто-об'єктні системі програмне уподібнення. Кожний крок такого породження є актуалізацією окремих оракулів генетичної структури рішення задачі оракульною схемою. Остання представляє собою логіко-математичну специфікацію рішення редукційного рівняння, що обумовлене відповідним композитом. Таким чином, технологічні системи програмування як носії таких специфікацій базуються на логіко-математичній імплементації програмної релятивізації і реально об'єктивізують активну роль суб'єкта програмування.

Необхідно відзначити, що логіко-математичні релятивізації прагматико-орієнтованих рішень задач можуть бути як завгодно складними. Тому їх доцільно класифікувати за ознакою типовості вирішуваних за допомогою таких специфікацій задач. У цьому сенсі виділяють логіко-математичні релятивізації, що орієнтовані на вирішення задач вищих типів, а саме задач типу класів задач, задачі типу задач типу класів задач тощо. Ключову роль для логіко-математичної релятивізації задач вищого типу відіграють саме оракульні схеми. У цілому, такі схеми можуть бути як завгодно складно влаштовані. Це обумовлено тим, що згадана вище інтеграційна складова у вирішенні задач не може бути заздалегідь ніяким чином штучно обмежена. У логіко-математичних релятивізаціях у рамках системи можна обмежитися мікро-, макро-,

зокрема, біпольними оракульними схемами. Це зміщує акценти з логіко-математичних релятивізацій задач вищого типу у сферу систем інтеграції розв'язання таких задач.

3.6. Логіко-математичні релятивізації в суб'єкто-об'єктній системі

За допомогою всюди істинного та всюди хибного предикатів p_T^1 та p_F^1 у СОСрП легко реалізуються будь-які логічні зв'язки предикатів. Наприклад, предикати p_T та p_F можна визначити так: $p_T^1 = S^2(<, I_1^1, S^2(s^1, I_1^1))$ і $p_F^1 = S^2(<, I_1^1, I_1^1)$. Тоді для будь-яких предикатів p та q: $\neg p = \Diamond(p, p_F, p_T)$, $p \vee q = \Diamond\left(p, p_T^1, \Diamond(q, p_T^1, p_F^1)\right)$, $p \wedge q = \Diamond\left(p, \Diamond(q, p_T^1, p_F^1), p_F^1\right)$.

Щодо прикладу мікрорівневої релятивізації рішень тут, з огляду на досить обмежений набір базисних операцій середовища, доцільно поповнити його зручною у використанні операцією усіченої різниці: $\dot{-}(x, y) = \begin{cases} x - y, \text{якщо } y < x \\ 0, \text{інакше} \end{cases}$. Розглянемо функцію $g^3: N^3 \to N$ таку, що $g^3(< c, a, b >) = \begin{cases} c + (a - b), \text{якщо } b < a \\ 0, \text{інакше} \end{cases}$. Особливість її в тому, що $g^3(< 0, a, b >) = \begin{cases} a - b, \text{якщо } b < a \\ 0, \text{інакше} \end{cases}$. Звідси випливає, що кортеж функцій $< S^2(0^1, I_2^2), I_1^2, I_2^2, >$ є S^4-редукцією функції $\dot{-}^2$. Адже, з зазначеного безпосередньо слідує, що для будь-якої пари $< a, b > |_{a,b \in N}$ справедливо:

$\dot{-}^2(< a, b >) = S^4(g^3, S^2(0^1, I_2^2), I_1^2, I_2^2)(< a, b >) = a \dot{-}^2 b.$

Таким чином, терм $\dot{-}^2 = S^4(g^3, S^2(0^1, I_2^2), I_1^2, I_2^2)$ є логіко-математичною релятивізацією функції $\dot{-}^2$ з одним оракулом g^3. Для його актуалізації скористаємось наступною властивістю функції g^3: $g^3(< c, a, b >) = g^3(< c + 1, a, b + 1 >)$. З цього випливає, що кортеж $< S^2(s, I_1^3), I_2^3, S^2(s, I_3^3) >$ є $*^4$-редукцією функції g^3. Легко переконатися, що $g^3(< c, a, b >) =*^4 \left(S^3(<^2, I_3^3, I_2^3), S^2((s, I_1^3), I_2^3, S^2(s, I_3^3)\right)(< c, a, b >)|_{c,a,b \in N}$. Таким чином, $\dot{-}^2 = S^4\left(*^4 \left(S^3(<^2, I_3^3, I_2^3), S^2(s, I_1^3), I_2^3, S^2(s, I_3^3)\right), S^2(0^1, I_2^2), I_1^2, I_2^2\right)$. Отримане рішення не містить оракулів і є логіко-математичною специфікацією класу задач, що

містить лише один елемент – функцію усіченої різниці $\dot{-}^2$. Тобто схемність структури даного рішення зведена тут до мікроінтеграції базисних функцій.

Щодо логіко-математичні релятивізації у макроінтеграційному середовищі. Розглянемо задачу обчислення значень функції, заданої скінченною сумою ряду: $F(x) = f\big(x, m(x), n(x)\big) = \sum_{i=m(x)}^{n(x)} g(x,i)$, де $f(x,y,z)|_{x,y,z\in N}$ – функція підсумовування, тобто $f(x,y,z) = \sum_{i=y}^{z} g(x,i), m(x), n(x), g(x,y)|_{x,y,z\in N}$ – деякі арифметичні функції, причому для будь-якого $a \in N$ виконується $n(a) \geq m(a)$. Ці функції відіграють роль оракулів у розглядуваній постановці задачі і це основна її відмінність від попередньої.

Звернемося до засадничої для вирішення задачі властивості функції f:

$f\big(a, m(a), m(a)\big) = g(a, m(a))$;

$f\big(a, n(a), m(a)\big)|_{n(a)>m(a)}\,) = f(a, n(a), m(a)-1) + g(a, m(a))$.

Розглянемо функцію

$q^4 =_*{}^5 (\neg\big(S^3(<^2, S^2(s, I_3^4), I_4^4)\big), S^3\big(+^2, I_1^4, g^2(I_2^4, I_4^4)\big), I_2^4, I_3^4, S^2(s, I_4^4)$. Її термальне позначення – $q(y,x,z,u)$. З вищенаведеної достатньої умови редукційності безпосередньо слідує, що $< S^3\big(+^2, I_1^4, g^2(I_2^4, I_4^4)\big), I_2^4, I_3^4, S^2(s, I_4^4) >$ є $*^5$-редукцією функції q^4. Нескладно помітити, що:

$q\big(r, a, n(a), m(a)\big)\,|_{r\in N} = r + f\big(a, n(a), m(a)\big) \equiv r + \sum_{i=m(a)}^{n(a)} g(a,i)$. Звідси випливає, що кортеж $< S^2(0^1, I_1^3), I_1^3, I_2^3, I_3^3 >$ є S^5-редукцією функції f^3 (у попередній (термальній) нотації $f(x,y,z)$). Тобто, $f^3 = S^5(q^4, S^2(0^1, I_1^3), I_1^3, I_2^3, I_3^3)$. Адже з визначення функцій $q(y,x,z,u), m(x)$ та $n(x)$ слідує, що $f\big(x, n(x), m(x)\big) = q(0, x, n(x), m(x))\,|_{x\in N}$. Враховуючи унарність функції F^1 (термальна форма – $F(x)$) та її зв'язок з функцією f^3 остаточно маємо: $F^1 = S^4\big(f^3, I_1^1, S^2(n^1, I_1^1), S^2(m^1, I_1^1)\big)$.

Послідовність викладення обрано для більш наочної демонстрації побудови оракульної схеми рішення задачі. Перехід до розгорнутого представлення при цьому є нескладним і буде продемонстрований нижче.

На відміну від мікроінтеграційної релятивізації тут маємо вже макроінтеграційну релятивізацію – середовище, що підтримує вирішення класу задач підсумовування, основу якої складає схема взаємодії трьох оракулів – $g^2,\ m^1$ та n^1. Конкретні рішення задач з'являються у результаті актуалізації цих оракулів. Так, актуалізуючи функцію $g(x,i)$ як $g^2 = S^3(\times, I_1^2, I_2^2)$, а $m(x)$ та $n(x)$ як $m^1 = S^2(s, S^2(0^1, I_1^1))$, $n^1 = S^2(s, I_1^1)$ отримуємо рішення для функції $\hat{F}(x) = \sum_{i=1}^{x+1} i \times x$ у вигляді мікроінтеграційної, вільної від оракулів, релятивізації функції $\hat{F}^1$, а саме:

$\hat{q}^4 =^{*5} (\neg(\ s^1(I_3^4) <^2 I_4^4), I_1^4 +^2 I_2^4 \times I_4^4, I_2^4, I_3^4, s^1(I_4^4)),\quad \hat{f}^3 = S^5(\hat{q}^4, 0^1(I_1^3), I_1^3, I_2^3, I_3^3)$ та остаточно $\hat{F}^1 = S^4\left(\hat{f}^3, I_1^1, s^1(I_1^1), s^1(0^1(I_1^1))\right)$.

Відносно отриманих у СОСрП логіко-математичних релятивізацій рішень задач важливо зазначити наступне:

1. Вони не просто презентують рішення вирішуваних задач, але адекватно відображають генезис цих рішень у вигляді композиційних термів як результатів застосування композито-композиційних інтерфейсів СОСрП.

2. Коректність рішень випливає безпосередньо з їх побудови.

3. Оформлення рішення в тій чи іншій системі нотацій зводиться до трансляції композиційного терму в обрану мову нотацій.

3.7. Синтаксична нотація логіко-математичних релятивізацій

У якості проміжної нотаційної основи оберемо широко використовувані у програмуванні блок-схеми. Це дозволить уникнути необхідності занурення у ряд непринципових деталей, властивих будь-якій конкретній мові програмування та зробить самі нотації більш наочними. Зрозуміло, що це не означає неможливості отримання синтаксичних нотацій рішень у високорівневих мовах програмування.

Як зазначалося, програмування це породження композицій та їх застосування. Досі мова йшла саме про породження композицій рішень задач. І композиційні терми розуміються саме як наслідки таких породжень. Їм притаманні як незаперечні переваги, про які вже багато сказано, так і серйозний недолік. Це надмірна складність для

безпосереднього сприйняття таких термів, обумовлена тим, що вони є носіями не стільки рішень задач, скільки генезисів таких рішень. Нотується не генезис рішення, а саме рішення, коректність якого обґрунтована композиційним термом. Застосування породженої композиції забезпечує суттєве спрощення релятивізації рішення саме за рахунок інкапсуляції його генезисної складової. Перед усім, це стосується притаманної для композиційних термів як завгодно глибокої вкладеності композитів.

Спираючись на наведені визначення композитів СОСрП, для отриманих композиційних термів матимемо наступні дещо спрощені нотації рішень задач.

Для композиційного терму усіченої різниці здійснюючи покрокове "згортання" композитів отримаємо $\dot{-}^2 = S^4(*^4\ ((I_3^3 <^2 I_2^3), s^1(I_1^3), I_2^3, s^1(I_3^3)), 0^1(I_2^2), I_1^2, I_2^2)$. Для композиційного терму функції $\hat{F}^1$, інтегруючи у єдиний терм вищенаведені вирази для $\hat{q}^4$ та $\hat{f}^3$ маємо: $\hat{F}^1 = S^4(S^5(*^5\ (\neg(\ s^1(I_3^4) <^2 I_4^4), I_1^4 +^2 I_2^4 \times I_4^4, I_2^4, I_3^4, s^1(I_4^4)),$ $0^1(I_1^3), I_1^3, I_2^3, I_3^3), I_1^1, n^1(I_1^1), m^1(I_1^1))$.

Звідси маємо наступні блок-схеми наведених композиційних термів (рис.3.2):

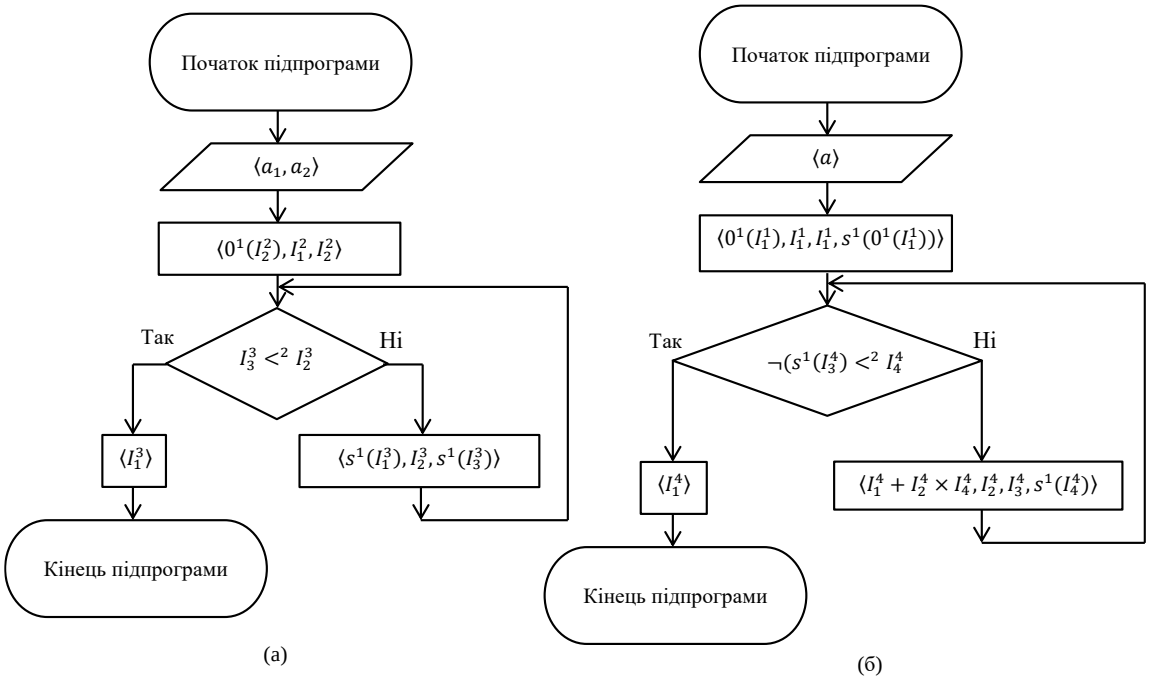


Рис.3.2. Блок-схема усіченої різниці (а), та функції підсумовування $\hat{F}(x) = \sum_{i=1}^{x+1} i \times x$ (б)

Особливість презентованих рішень у тому, що, по-перше, вони є коректними за побудовою в заздалегідь визначеній системі, по-друге, їх синтаксичне оформлення зводиться, як зазначалося до трансляції відповідного семантичного терму в обрану мову програмування через синтаксичну актуалізацію відповідних оракулів композитних та функціональних структур. Цей процес можна об'єктивізувати, наприклад за допомогою спеціального програмного забезпечення – дефінітора відповідної мови програмування.

Зі сказаного випливає, що композито-композиційна релятивізація рішень задач має ряд суттєвих переваг перед надто конкретизованими специфікаціями у високорівневих мовах програмування. Представлення через блок-схему через її широку використовуваність для специфікації алгоритмів, програм, інтеграційних систем забезпечує прозорість переходів від таких специфікацій до нотацій у більшості високорівневих мов програмування. Приклади результатів такого переходу у нотації високорівневої мови програмування представлені нижче (рис.3.3):

```
def diff(a1, b2):
    i13 = 0; i23 = a1; i33 = b2
    while i33 < i23:
        i33 = i33 + 1
        i13 = i13 + 1
    return i13
```

а)

```
def summ (a1):
    i14 = 0; i24 = a1; i34 = a1;
i44 = 1
    while not i34+1<i44:
        i14 = i14 + i24 * i44
        i44 = i44 + 1
    return i14
```

б)

Рис.3.3. Реалізація функції усіченої різниці (а) та підсумовування (б)

Розглянуті приклади вирішення задач у суб'єкто-об'єктній системі демонструють важливі загальні особливості редукційного концептування оракульних схем. Це, по-перше, гарантована коректність отримуваних рішень, що випливає безпосередньо з їх побудови. По-друге, можливість переходу від рішень окремих задач до вирішення класів подібних задач. І, по-третє, отримані рішення можуть бути реалізовані на різних синтаксичних програмних платформах.

РОЗДІЛ 4. Імплементація суб’єкто-об’єктного середовища

У зв'язку з попередньо згаданими кризовими явищами в індустрії інформаційних технологій, виникла нагальна потреба у розробці нових підходів до вирішення задач проєктування програмно-апаратних комлексів у галузі телекомунікай, які сприяли б самому процесу проєктування. При постановці більш загальної задачі варто відзначити, що реальність, як неодмінний аспект будь-якої діяльності, безпосередньо пов'язана з різними сутностями. Розуміння реальності стає неможливим без глибокого усвідомлення самостійних сутностей, з якими ми працюємо, а також природи їх взаємозв'язку. Вирішення задач програмування також неможливе без осмислення самостійних сутностей та їх взаємодії. На першому рівні ця взаємодія обумовлена класифікацією використовуваних сутностей залежно від їх важливості для конкретної проблеми на головні та другорядні. Ця ієрархія формує основу для концепції системи суб'єкт-об'єктних відносин, яка використовуватиметься для створення відповідних середовищ [72-74].

Більшість сучасних задач відрізняються тим, що взагалі неможливо відразу і назавжди провести чітку лінію між головними та другорядними аспектами. Такі вирішення, як правило, включають багато кроків. До того, самі ці кроки часто взаємопов'язані й, на цей час, навіть не можуть бути повністю охоплені або описані. Саме тому в даному контексті можна говорити лише про розробку інструменту, який би керував процесом проєктування у "потрібному" напрямку.

"Потрібний" напрямок передбачає, що кожен наступний крок має як мінімум не порушувати вже встановлений рівень розподілу на головне та другорядне, а також він повинен бути відповідним цьому розподілу. Такий інструмент слугував би універсальним засобом для збагачення розглянутих аспектів у процесі прийняття рішення. Тому його сутність є складною і пов'язаною з основними проблемами та завданнями.

Добре відомо, що існує глибокий зв'язок між кроками процесу створення та кінцевим результатом. Виявлення суті цього зв'язку вимагає роз'яснення логіко-предметних взаємозв'язків системи та відповідного СОСрП, яке виступає як ключовий елемент поетапного збагачення розглянутих аспектів у процесі проєктування [75-77].

На початковому етапі цей підхід до розробки апаратного та програмного забезпечення полягає в пошуку прагматичного компромісу між об'єктивним та суб'єктивним поглядами на ці аспекти. Відомо, що як об'єктивний, так і суб'єктивний підходи мають свої очевидні переваги, але також мають принципові недоліки, які стають все більш проблемними. Тому актуальною є розробка такої точки зору, яка б уникнула ці недоліки, зберігаючи переваги обох підходів. Ця точка зору, яка об'єднує переваги суб'єктивного та об'єктивного підходів, може бути названа суб'єкто-об'єктною.

Примирення цих поглядів передбачає пошук коренів проблем і недоліків, що виникають. Отже, суб'єкто-об'єктна система повинна не тільки визначити головні та другорядні аспекти, але й налагодити між ними взаємозв'язок. Цей розподіл не є лише формальним, але і реалізовується в самій системі через відмінне використання основних та другорядних компонентів. Основні компоненти формують основу системи, яка визначається прагматикою конкретної задачі, тоді як другорядні компоненти утворюють різноманітні замикання системи та представляють її відкриту частину.

Це дозволяє розглядати систему як зв'язок, залежний від прагматичних умов, між автономізованими загальною і предметною частиною системи, що є різними формами абстракції. Центральну роль в цій концепції відіграє логіко-предметний напрямок. При цьому як логіка, так і предмет розглядаються відносно і можуть самі бути вивчені з погляду логіко-предметних зв'язків. Ця взаємодія також відкриває можливість для розгляду суб'єкто-об'єктних середовищ як середовищ існування суб'єкто-об'єктних систем.

Один з особливих аспектів суб'єкто-об'єктних середовищ полягає в ітеративному збагаченю за допомогою створення нових композицій і використання цих композицій

для створення нових функцій. Цей процес продовжується до тих пір, поки не буде сформована зручна суб'єкто-об'єктна система для вирішення певного класу задач.

Зазначений підхід включає три основних типи операцій: функції, композиції та метакомпозиції. Ці структури взаємозв'язані через операцію аплікації, що дозволяє переходити до нижчого рівня ієрархії:

- Метакомпозиція застосовується до створення композиції;
- Композиція застосовується для отримання функції;
- Функція застосовується для отримання конкретного значення.

У роботі продемонстровано застосування цього підходу на прикладі мови Verilog, де побудовані функція, композиція та метакомпозиція. Отримані рішення відображено в зручному форматі, що дозволяє легко інтегрувати їх за необхідності з іншими обчислювальними системами.

4.1. Методи розробки суб’єкто-об’єктних середовищ

Сучасні методи проєктування суб'єкто-об'єктних систем належать до найвищого рівня ієрархії в процесі програмування цифрових логічних пристроїв. У цьому ієрархічному підході виділяються щонайменше три рівні: фізичний, регістровий та системний рівні. На системному рівні програміст абстрагується як від фізичних особливостей майбутньої схеми, так і від простих логічних пристроїв, таких як регістри та логічні вентилі.

Перед вибором конкретної архітектури проводиться попередня оцінка вимог до продуктивності обчислювальної платформи. Ця оцінка включає аналіз простору програмних параметрів (величина якого вимірює важливі характеристики майбутнього програмного рішення). Метою програміста є знаходження глобального мінімуму функції можливих реалізацій програми при умові, що менше значення характеристики є кращим варіантом [78, 79].

Програмування на високому рівні абстракції сприяє прискоренню процесу розробки завдяки уникненню необхідності проведення складної низькорівневої симуляції.

Методика триєдності використовується для структурування процесу проєктування та включає виділення трьох відмінних аспектів цифрової системи: поведінкового, структурного та геометричного. Кожен з цих аспектів є рівноправним і пропонує різні точки зору на програму. Кожен аспект може бути розглянутий на різних рівнях абстракції, таких як рівень регістрових передач, алгоритмічний рівень (опис та специфікація взаємодії між підсистемами) та архітектурний рівень (вимоги до системи, ключові концепції для задоволення цих вимог). Існують також можливості для використання більшої кількості рівнів абстракції.

Поведінковий аспект описує функціональну поведінку системи, структурний аспект відображає зв'язки між системами та підсистемами на різних рівнях абстракції, а геометричний аспект стосується розміру, розміщення та форми підсистем на кристалі. Розробник, надаючи відповіді на питання, що представлені на кожному рівні абстракції для кожного аспекту, отримує можливість для чіткішого визначення границі між програмою поведінки та архітектурою.

Використовуючи дану методику, можна чітко визначити розділ між програмною поведінкою та архітектурою системи, включаючи програмовані та реконфігуровані ресурси. Це досягається через етап відображення системи на конкретну архітектуру, що допомагає узгоджувати обчислення та комунікаційні події та часові характеристики та наслідки подій у програмі [80-82].

Ще одним способом створення суб'єкто-об'єктних систем є підхід, базований на платформах, який визначається повторним використанням готових ядер. У цьому методі програма базується на обчислювальній платформі, а її функціонал розширюється готовими ядрами від виробника платформи або сторонніми постачальниками. Це

дозволяє значно знизити витрати на виробництво електронних продуктів та до певної міри автоматизувати процес програмування [83].

Крім того, такі платформи дозволяють створити стек платформ вищого рівня, спрямований на розробку програмного забезпечення. У цьому випадку верхній рівень платформи визначає умови та обмеження для нижнього рівня, який своєю чергою визначає продуктивність та швидкість стека. Платформний підхід є компромісом у випадку розробки систем на кристалі. Проте важливо зазначити, що зміни архітектури вимагають значних зусиль для верифікації та налаштування. Наприклад, зміна розрядності процесора потребує розробки нового інтерфейсу пам'яті, його стандартизації та верифікації [84].

Підхід на основі платформ передбачає розробку унікального набору команд, інтерфейсу процесора тощо на основі базової архітектури. Процесор із проблемно орієнтованим набором команд може досягти балансу між гнучкістю базового процесора та продуктивністю, яка досягається шляхом виконання проблемно-орієнтованого набору команд. Наприклад, високопродуктивний процесор, оптимізований для обробки сигналів, може включати широкий спектр попередньо верифікованих цифрових сигнальних процесорних модулів, від прискорювача операцій з плаваючою комою до модуля з 16 паралельними операціями [85-87].

Даний підхід також дозволяє поєднати гнучкість програмного забезпечення з продуктивністю адаптивної логіки, особливо в контексті систем на кристалі. Це забезпечує кращу адаптованість до різних застосувань та покращує сумісність.

Проблемно-орієнтовані платформи набувають все більшої популярності. Більшість цифрових сигнальних процесорів сьогодні використовуються для обробки даних у радіопередачі, зокрема в системах радіоелектронної боротьби. При цьому нові застосування ставлять ще вищі вимоги до продуктивності та енергоспоживання. Це призводить до зростання популярності не загальнопризначених вбудованих систем, а систем, спеціально орієнтованих на вирішення конкретних задач [88, 89].

На практиці, в окремих випадках, наприклад, при обробці радіосигналів або виконанні швидкої обробки протоколу, дійсно необхідні проблемно-орієнтовані системи. Місця їх застосування визначаються архітектурними особливостями системи. Наприклад, якщо потрібно реєструвати та зберігати дані в пам'яті, то використання процесора, який підтримує повний набір арифметичних операцій, може бути нераціональним з точки зору архітектури, але при цьому економічно вигідним рішенням. Проблемно-орієнтовані архітектури не можуть повністю замінити універсальні архітектури. Отже, необхідний симбіоз обох підходів [90].

Останні комерційно успішні проблемно-орієнтовані платформи в основному зорієнтовані на конкретні сфери застосувань, а не на загальнопризначені задачі. Сьогодні FPGA та системи на кристалі переважають серед проблемно-орієнтованих платформ. FPGA добре балансують між продуктивністю та енергоспоживанням у вузькоспеціалізованих рішеннях. Вони можуть успішно використовуватись в продуктах, де ціна є ключовим фактором, та в разі великих партій. Швидка можливість програмування і гнучкість роблять FPGA привабливими для виробників, вибираючи між ними та фіксованими апаратними платформами [91].

Існує ще один підхід до програмування складних цифрових систем, який включає моделювання на рівні транзакцій. Цей підхід передбачає встановлення механізму комунікації та інтерфейсів на високому рівні абстракції, виключаючи деталі реалізації підсистем з процесу моделювання. В цьому випадку моделюється лише поведінка підсистем та процеси комунікації між ними [92].

Це обумовлено тим, що моделювання всієї структури міжз'єднань та компонентів реконфігурованих систем стає все складнішим через збільшення їхньої складності, що висуває підвищені вимог до продуктивності модельних засобів. Механізм функцій дозволяє моделювати комунікацію та приховує варіанти її конкретної реалізації [93].

Такий підхід спрощує симуляцію та сприяє оптимізації та моделюванню топологій та протоколів. Враховуючи реактивну природу вбудованих систем, симуляція

транзакцій, які можуть бути припинені або перезапущені ще до їхнього завершення може підтвердити коректність вибору способу реалізації, запевнюючи, що конструкція не містить петель і завжди відреагує на вхідні сигнали [94].

Платформи на базі FPGA стали пріоритетним вибором для досягнення гнучкості програмного забезпечення та продуктивності апаратного забезпечення при розробці систем. Такі системи виявилися кращими порівняно з системами на основі процесорів і програмного забезпечення. При цьому вони виявляють менше статичності у порівнянні з рішеннями на основі ASIC, а затрати на їхнє програмування менші, ніж у випадку з ASIC. Однак, висока ціна залишається фактором, який обмежує використання FPGA у вбудованих системах. Вартість FPGA-пристроїв у масовому виробництві все ще перевищує ціну ASIC. Але зараз, із поширенням ринку FPGA різниця зменшується, нові покращення в архітектурі FPGA викликають сподівання на значний спад споживання електроенергії та площі кристала [95].

Нові архітектури FPGA, які працюють на вищих рівнях абстракції, включаючи маршрутизацію на шинах та логічні блоки, також обіцяють покращити продуктивність та ефективність. Однією з важливих рис нових архітектур FPGA є можливість інтеграції одного або більше загального призначення процесорів.

Сучасні FPGA мають достатньо ресурсів, щоб повністю реалізувати систему на одному чіпі. Але через велику складність сучасних завдань нерідко важко помістити всі компоненти на одному чіпі. Це приводить до використання техніки спеціальної організації потоків, яка дозволяє модифікувати лише частину конфігурації FPGA без зупинки всієї системи. Оскільки все більше програм є багатопотоковими, FPGA можуть суттєво прискорити виконання таких програм. Крім того, критичні частини коду можуть бути перенесені на FPGA та синтезовані під час роботи програми, що дозволяє досягти ще більшої ефективності [96].

Усі вищезгадані підходи мають один загальний недолік - вони не забезпечують системності. Жоден з них не охоплює процесу програмування від початкового задуму

до фактичної реалізації. Вони виступають лише як інструменти організації, але не надають конкретних настанов для організації всього процесу програмування. Ідея програмування на основі платформ є лише концепцією, не враховуючи конкретних деталей побудови складних систем. Вона зосереджується на можливостях збагачення базової платформи, але не вирішує питання, як реалізувати ці ідеї в практичних умовах. Підхід з використанням проблемно-орієнтованих платформ також залишається на рівні концепції, не надаючи повноцінного методу програмування.

В даному контексті галузі не вистачає системного та достатньо деталізованого підходу до програмування адаптивних систем. Хоча існує велика кількість різних концепцій та підходів, яким необхідно долучити більше деталей та зрозуміти, як їх застосовувати на практиці для досягнення якісних результатів. [97-100].

4.2. Основні положення мови Verilog

Verilog це мова для опису та розробки електронних цифрових систем. Розробники Verilog намагалися зробити його синтаксис подібним до мови C, що полегшує його розуміння. Важливо зауважити, що хоча описи апаратури, створені за допомогою мови Verilog, називаються програмами, вони відрізняються від звичайного розуміння програм як послідовностей інструкцій. У цьому контексті програма визначає структуру системи. Також термін "виконання програми" не застосовується до мови Verilog.

Повна специфікація цієї мови міститься в стандарті IEEE 1364. Тут ми будемо висвітлювати основні аспекти мови. Сама по собі Verilog є мовою для опису апаратного забезпечення. Апаратне забезпечення, яке описується, являти собою ієрархічну структуру модулів, де "входи" отримують сигнали, що змінюються з часом.

Дана мова програмування підтримує на загальному рівні три типи концструкції: обов’язково синтезуючі, ігноруючі та ті що не підлягають синтезу.

Код, написаний виключно з конструкцій, що підлягають синтезу або ігноруються, може бути успішно відтворений на ПЛІС. Важливо коротко згадати про обмеження мови, які стосуються можливості синтезу деяких синтаксичних конструкцій. Ці

обмеження можуть відрізнятися для кожного синтезатора, але багато з них спільні. Наприклад, цикли forever та while взагалі не підлягають синтезу. Цикл repeat синтезується лише тоді, коли вираз, що визначає кількість ітерацій циклу, може бути обчислений під час синтезу. Цикл for синтезується тільки тоді, коли всі присвоєння для ітератора циклу є константними.

Синтезована цифрова схема не має поняття часу початку або закінчення роботи - вона просто існує. Тому операції, які розробник визначає для виконання на початку роботи схеми, ігноруються. Початкові значення регістрів також відсутні. Вирішення цих питань лежать за межами області дії Verilog. Для таких випадків може бути використаний сигнал скидання, який розробник повинен ураховувати при роботі. Сигнал скидання може виступати як умовний початок роботи схеми.

Під час подачі тактового сигналу створюється рівномірний потік дискретного часу. Більшість цифрових пристроїв працюють в дискретному часі, який визначається тактовим сигналом. Такі пристрої називаються синхронними.

В мові Verilog існує аспект, пов'язаний з композиційними можливостями. Одним із них є композиційна конструкції "If". У загальному синтаксисі, умовний оператор "if" може бути представлений записом If(P, F1, F2), де P - це предикат, F1 - це функція, яка виконується, коли предикат істинний, і F2 - це функція, яка виконується в іншому випадку. Схематично дана структура може бути представлена через класичний мультиплексор (рис. 4.1).

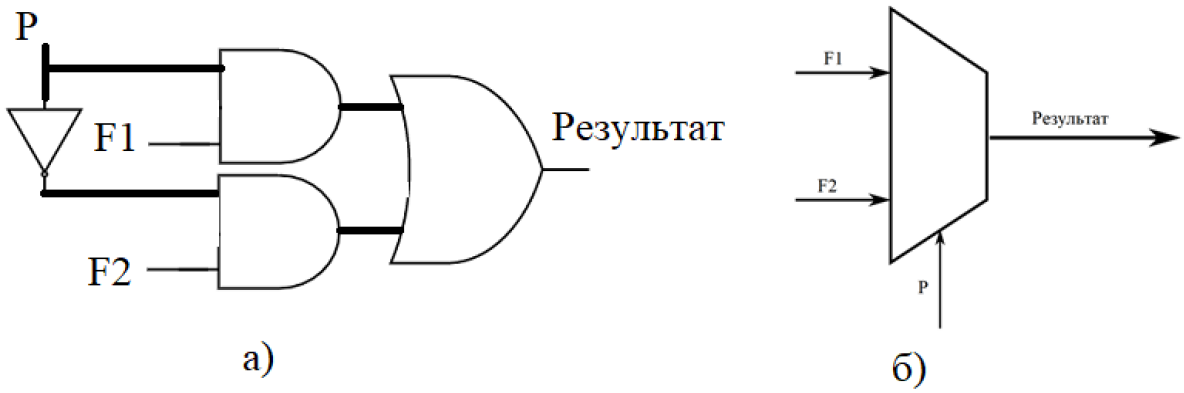


Рис. 4.1 Принципова (а) та модульна (б) схеми композиції галуження

Одночасно відбувається обчислення предиката, випадкового варіанту для істинного значення предиката і випадкового варіанту для неправильного значення предиката. Результат предиката (1 або 0) використовується як вхід для адресного входу мультиплексора. Залежно від його істинності, мультиплексор вибирає вихідні дані з першого або другого входу.

Ще однією важливою композиційною структурою є конструкція "case". Вона схожа за стурктурою до композиції “if”, але враховує більше двох можливих варіантів значень функції предиката. Цю конструкцію можна представити записом case(P, Ref, F1, F2, ..., Fn), де P - це функція, яка повертає аналізоване значення, F1, F2, ..., Fn - це функції, результат яких повертається з "case" в залежності від значень P. Ref - це функція, результат якої повертається за замовчуванням. Модульний вигляд даної композиції наведено на (рис. 4.2).

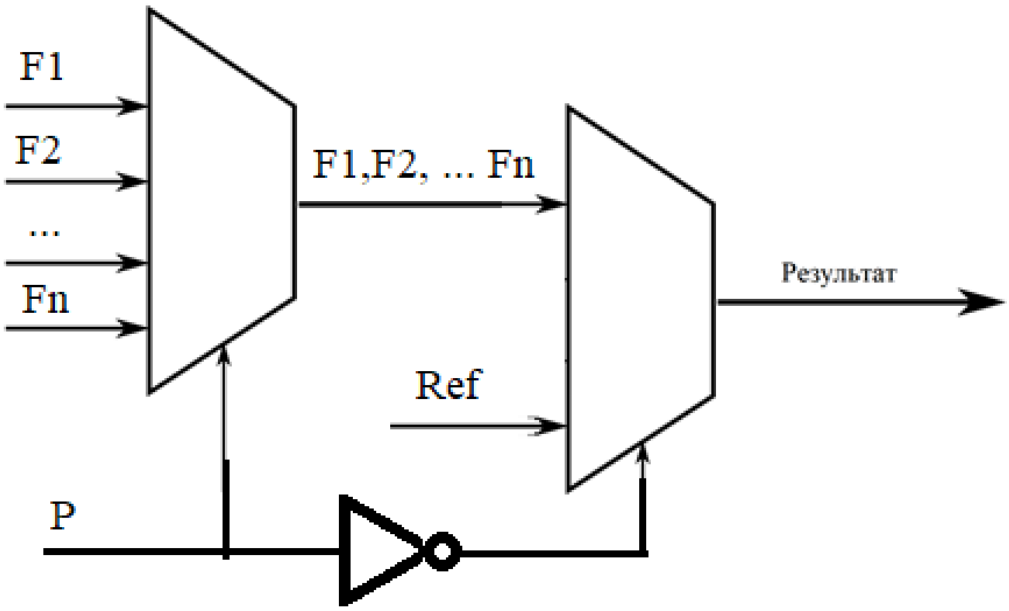


Рис. 4.2 Моудльне зображення композиції *case*.

У цій структурі відбувається обчислення всіх можливих варіантів для "case", включаючи значення за замовчуванням, а також обчислення величини, яка визначає, який з цих варіантів буде обраний у "case". Всі ці варіанти передаються на

мультиплексор, і обирається той, який відповідає обчисленій величині функції предиката. Якщо жоден з варіантів не відповідає цій величині, результатом буде значення функції за замовчуванням. Схематично зображений інвертор виконує перевірку відповідності варіанту, що відповідає поточному значенню функції предиката. При відсутності такого значення вихід "case" комутує на значення функції за замовчуванням.

Композиція *repeat* реалізує ітеративне виконання єдиної функції (F1). Кількість ітерацій відповідає заданому параметру (N). Цю конструкцію можна представити записом repeat N(F1). Особливість цієї композиції в тому, що вона композиціюється сама собою, шляхом викоистання єдиного вхідного аргументу.

Цикл for є ще однією з важливих композицій. У мові Verilog дана композиція підтримує варіативність синтезування. Наприклад, для її синтезування необхідною умовою є обчислюваність кількості ітерацій на етапі синтезу. Дана конструкція представляється записом for(B, P, I), де B відображає дії, які виконуються перед входженням в основний цикл, P є умовою для виходу з циклу, а I – операцією, яка виконується після кожної ітерації циклу і визначає зміну початкового стану. Враховуючи обмеження синтезу, даний цикл можна реалізувати на базі композиції repeat. Аналогічна ситуація із композицією циклування while, яка визначається дещо іншим набором функцій, але так само може бути синтезована через repeat (має аналогічні обмеження синтезу). Таким чином у даній мові програмування, основний масив композицій зводиться до двох типів: галуження та циклювання у вигляді визначеної кількості повторів функції.

Також окремо можнга виділити характерну даній мові композицію суперпозиції, що визначає ієрархічну послідовність модулів, та реалізує підстановку результатів одніє функції в іншу. Прикладом суперпозиції може бути звичайний арифметичний пристрій типу суматор (Рис.4.3).

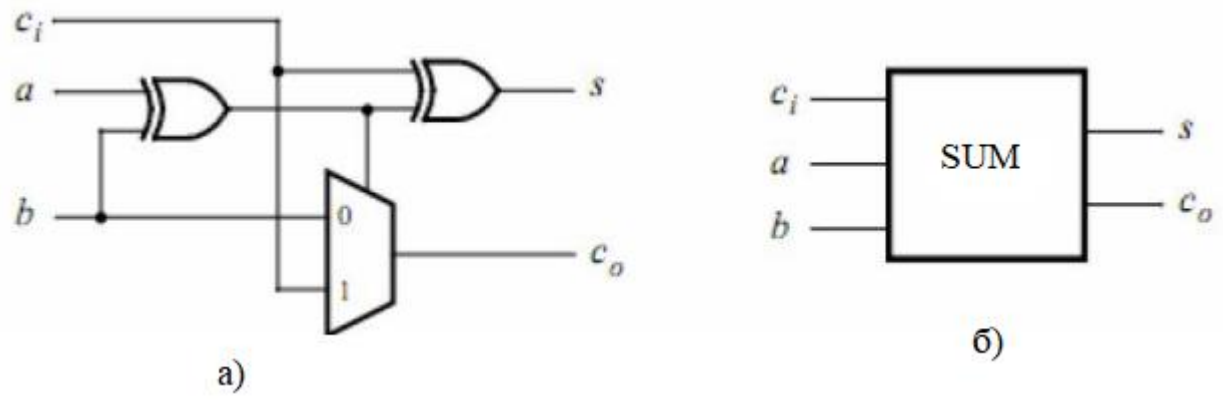


Рис. 4.3 Схема однорозрядного суматора (а) та його графічне зображення (б).

Власне суперпозицію (пістановку) легко представити за допомогою багаторозрядного суматора, в якому явно представлене використання результату попередньої функції наступною, при умові незмінності визначення функції (Рис.4.4).

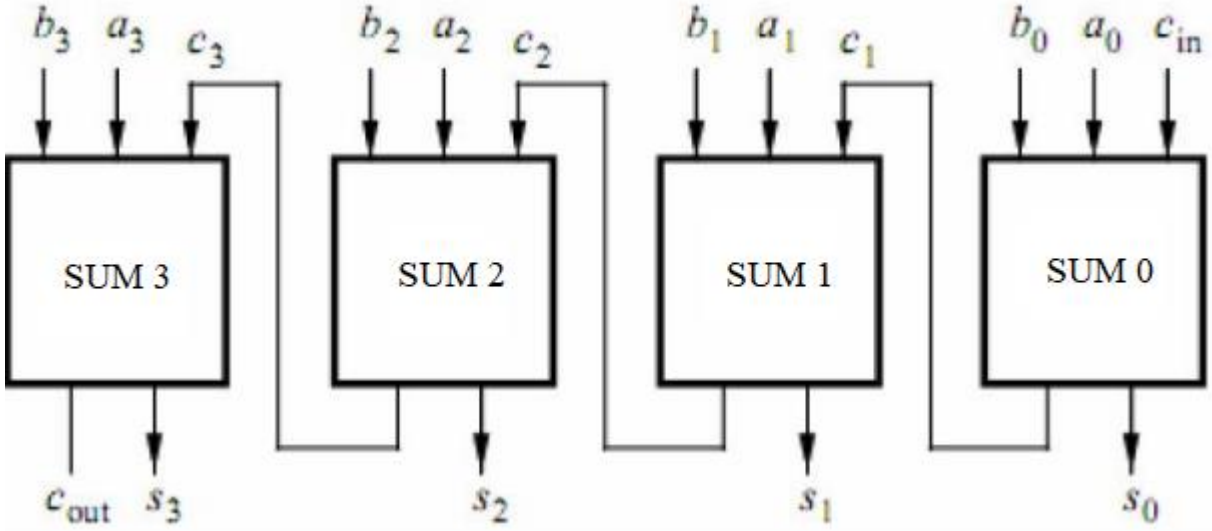


Рис. 4.4 Суперпозиція на прикладі чотирьохрозрядного суматора з переносом.

Розподіл сигналів у суматорі виглядає так: a та b – вхідні компоненти, Cin – вхід значення від попереднього розряду, s – результатна сума, Cout – перенесення для наступного розряду. У такому представленні кожний однорозрядний суматор SUM є модулем з ієрархічно визначеним порядком від SUM 0 до SUM 3. Також неявним чином суперпозиція представляється через ієрархію запису всередині блоку програми, визначаючи послідовність виконання та вхідні та вихідні значення функцій для наступних дій.

Описана група композицій не вичерпно охоплює всі можливості Тьюрінг-повноти, оскільки вона здатна лише моделювати комбінаційні схеми, але не здатна втілити послідовнісні схеми. Для того, щоб подолати це обмеження, необхідно ввести поняття часу та станів, які відсутні в описаному вище підході. Ці поняття виходять за межі можливостей мови Verilog.

Створення повноцінної Тьюрінг-повної системи є частковою відповідальністю розробника. Для досягнення Тьюрінг-повноти вистачає розробити механізм циклічних операцій, де кількість ітерацій не визначена наперед. Таку композицію можна реалізувати на мові Verilog з умовою, неперервності тактового сигналу на вході для реалізації неперервності дискретного часу.

Таким чином композиційний підхід неявно властивий мові програмування Verilog, проте слід зауважити, що не усі композиції мають еквівалент у даній мові програмування.

4.3. Використання СОСрП на практиці

Апаратне забезпечення, створене на основі Verilog коду, на сьогоднішній момент ефективно не може бути використане без інтеграції в традиційні обчислювальні системи, такі як процесорні підсистеми систем на кристалі, програмовані логічні матриці або спеціалізовані інтегральні схеми. Внаслідок цього виникає завдання впровадження такого апаратного забезпечення в обчислювальні системи.

Розглянемо докладніше процес взаємодії між синтезованим апаратним забезпеченням на основі Verilog та FPGA. Для наочності розглянемо алгоритм створення програмно-апаратного рішення на базі СОСрП.

Композиційне програмування є платформою створення програмно апаратного рішення, у поєднанні із використанням розширеної нотації Бекуса – Наура (БНФ) (правила мови на основі ISO 14977), що реалізує спрощену нотацію композицій, метакомпозицій та функцій. Відповідно до правил нотації БНФ розгялнемо для прикладу спрощений варіант запису композицій.

Застосування функцій F_Use ::= <ім’я функції> <(> <аргументи функції> <)>. Застосування композицій C_use ::= <ім’я композиції> <[> <аргументи композиції> <]>. Застосування метакомпозицій MC_use ::= <ім’я метакомпозиції> <{> <аргументи метакомпозиції> <}>.

Після визначення базису композиційної платформи у відповідно створеній текстовій мові специфікацій композиції розглянемо запис програми, що реалізує простий арифметичний пристрій типу суматор (реалізує додавання двох 4-х розрядних чисел із послідовним перенесенням):

Sum= S{SUM3, S{SUM3, SUM2, S{SUM2, SUM1, S{SUM1, SUM0, S {SUM0}}}}}. Даний запис демонструє суперпозицію метакомпозицій “SUM_n”, де n – номер суматора у послідовності. Така спрощена форма запису обрана задля легшого розуміння виразу. Метакомпозиція SUM реалізує собою набір результатів суперпозиції композицій, що характерні для однорозрядного суматора і можуть бути записані наступним чином:
SUM_n = S [MUX, S [MUX, XOR2], S [MUX, XOR1], S [XOR2], S[XOR1]]. Даний запис демонструє суперпозицію композицій “MUX” та “XOR_n”, де n – номер виключного або у схемі. Композиція “MUX” є структурою if, а композиція “XOR” є результатом функції f(x1,x2) = (x1 ^ x2), що реалізує логічну операцію виключного або на базовому рівні (транзисторний рівень у специфіці FPGA).

На основі створеної дескрипторної таблиці, що визначає взаємозв’язок між патернами композиційного запису та синтаксисом мови Verilog, створено програму-транслятор (препроцесор), що перетворює композиційний запис програми у відповідний синтаксис мови Verilog для подальшого використання відповідним компілятором. Репрезентативна частина описаного вище дескриптора наведена у (таб.4.1). Ознайомитись з кодом програми та інструкцією використання можна за посиланням: https://github.com/Zylevich96/Compositional terms to code converter.

Запропонований транслятор починає свою роботу з лексичного аналізу специфікацій композиції. Лексер аналізує вихідний код і розбиває його на токени

(лексеми), які є базовими одиницями мови програмування, типу ідентифікатори, ключові слова, оператори, числа, рядки тощо. Лексер перетворює вхідний текст на послідовність токенів, які потім передаються парсеру для синтаксичного аналізу.В даній реалізації лексер лише перевіряє коректність запису мовної специфікації композицій.

Таблиця 4.1 – Патерни композицій та програмування

Композиційні шаблони	**Синтаксис мови Verilog**
NO	!
AND	&&
OR	||
XOR	^
MUX	If (P) F_1 else F_2
$F_1 ° F_2$	module F_1; F_2 endmodule
$IF(F_1, F_2, F_3)$	if F_1 if F_2 else F_3
F_1 [F_2]	F_1 ; F_2
F, (F)	F
$WD(F_1, F_2)$	while F_2 begin F_1 end
F°X	assign $X = F$
S[XOR]	assign F= X_1^X_2

Наступним кроком починає працювати синтаксичний аналізатор. Парсер визначає синтаксичну структуру коду. За основу використано парсер “Yet Another Compiler Compiler” (flex + bison). Він генерує парсер на основі аналітичної граматики, описаної в нотації БНФ. На виході уасс видається код парсера мовою програмування Verilog, який додатково проходить семантичну перевірку для визначення використовуваних типів даних, усіх змінних та їх розрядності (Рис. 4.5).

Слід зазначити, що вихідний код потребує перевірки на предмет коректності визначення розрядності змінних, а також її фізичний зміст (вхідний чи вихідний сигнал). Коментарі на зображенні вище додані вручну, для полегшення сприймання коду.

SUM.v

```
module SUM (  input [3:0] A,     // Вхід A - 4 біта
   input [3:0] B,     // Вхід B - 4 біта
   input Cin,         // Вхід переносу (carry-in)
   output [3:0] Sum,  // Вихід суми - 4 біта
   output Cout        // Вихід переносу (carry-out)
);
 wire [3:0] carry_out; // Внутрішній вихід для переносу на кожному біті
 Sum_n sum0 (A[0], B[0], Cin,Sum[0], carry_out[0]);
 Sum_n sum1 (A[1], B[1], carry_out[0], Sum[1], carry_out[1]);
 Sum_n sum2 (A[2], B[2], carry_out[1], Sum[2], carry_out[2]);
 Sum_n sum3 (A[3], B[3], carry_out[2], Sum[3], Cout);
 endmodule

module Sum_n (  input A,     // Вхід A
   input B,     // Вхід B
   input Cin,   // Вхід переносу (carry-in)
   output Sum,  // Вихід суми
   output Cout  // Вихід переносу (carry-out)
);
 assign Sum = A ^ B ^ Cin;  // XOR для визначення суми
 assign Cout = (A & B) | (B & Cin) | (A & Cin);  // Логічне "або" для визначення переносу
 endmodule
```

Рис. 4.5 Пост-парсер код на мові Verilog.

Після генерації та перевірки отриманий код можна використовувати для подальшої реалізації апаратної складової. Для прикладу використано САПР “Quartus”. Для того, щоб перейти від коду програми до її апаратної реалізації треба створити проєкт і скомпілювати у ньому програму, визначену отриманим кодом. У прикладі створено проєкт для реалізації апаратної частина на базі FPGA Cyclon 5 (на базі плати DE1 SoC). Відповідно до специфікації проєкту отримано результат компіляції, який підтвердив коректність отриманого коду відповідно до правил компіляції САПР (Рис. 4.6).

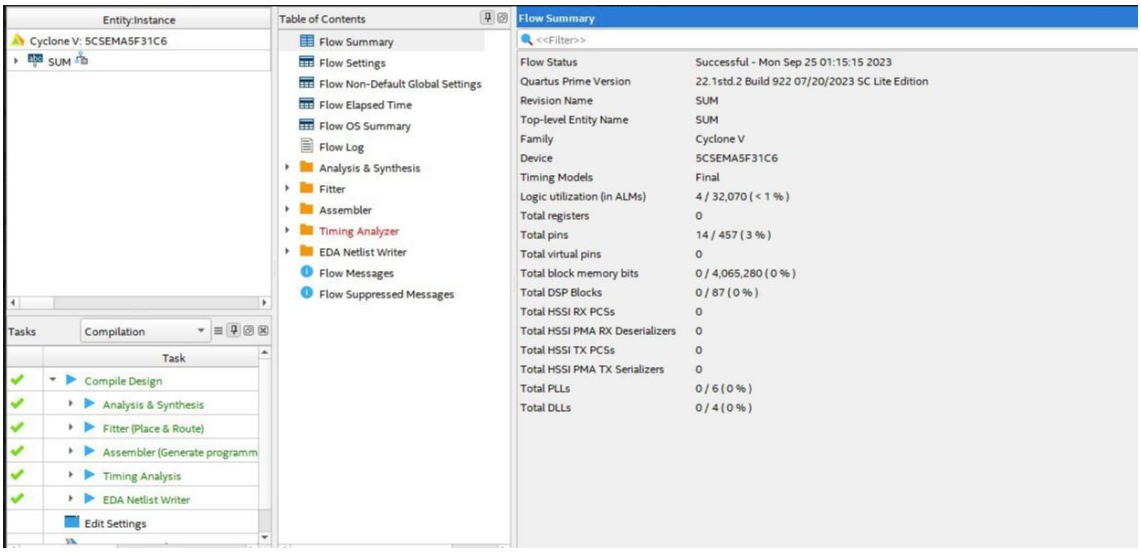


Рис. 4.6 Результат компіляції отриманого коду у САПР “Quartus”.

При невдалій компіляції САПР видасть відповідне повідомлення про помилку у вікні логування з коментарями щодо можливої локалізації проблеми (рядок, символ, команда).

Після успішної компіляції можна перевірити отриманий код за допомогою утиліти RTL (register transfer level), що дає змогу дослідити реалізацію на рівні логічних елементів (Рис.4.7, 4.8).

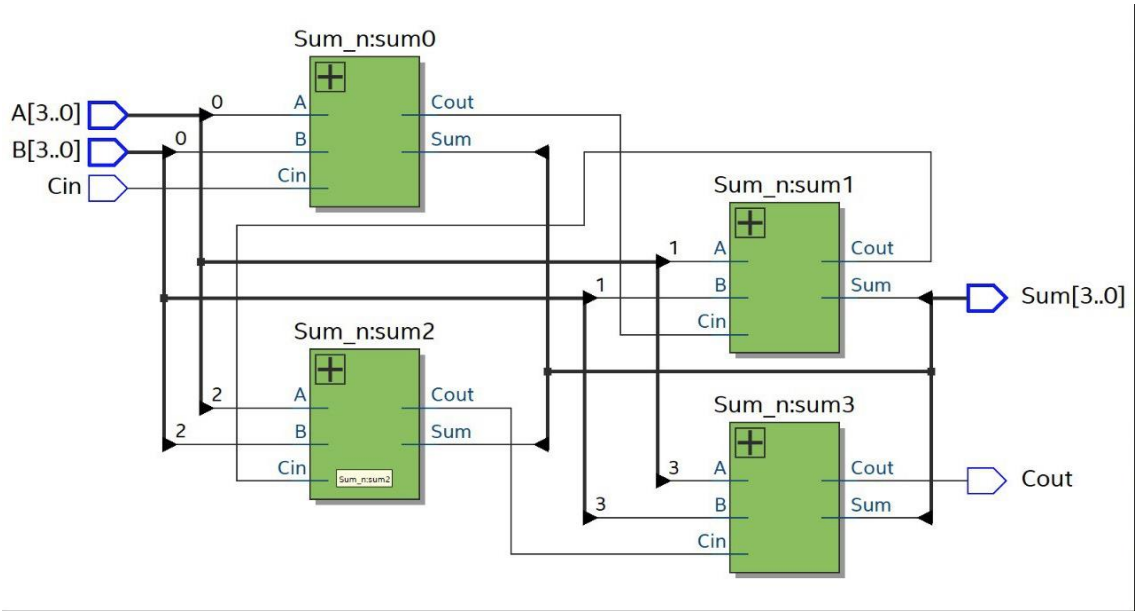


Рис. 4.7 Логічна реалізації 4-розрядного суматора у САПР “Quartus”.

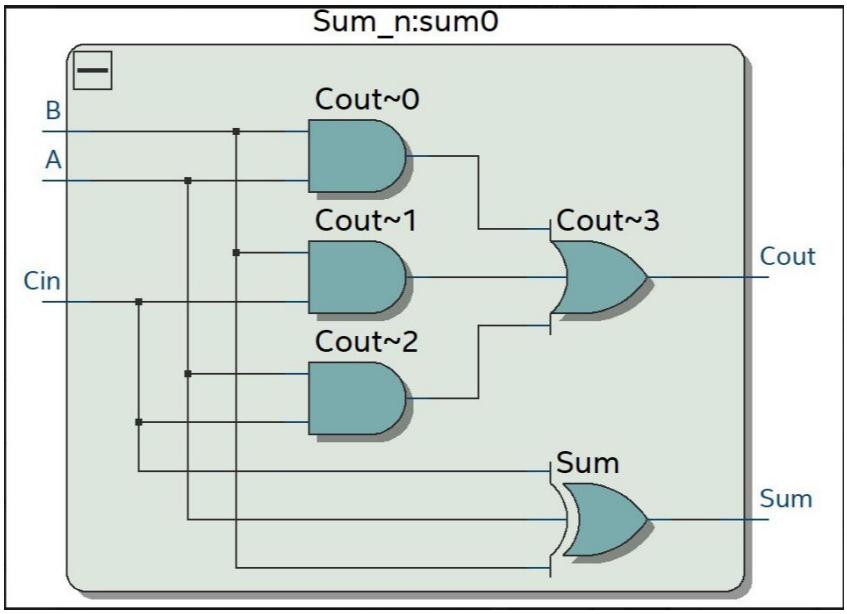


Рис. 4.8 Логічна реалізації 1-розрядного блоку суматора у САПР “Quartus”.

Останнім кроком на шляху до реалізації та практичної перевірки отриманого коду є його виконання в базисі FPGA шляхом програмування відповідної мікросхеми отриманим під час компіляції файлом прошивки. Для наочності додамо побітове введення вхідних даних за допомогою перемикачі, а результат відображатиметься побітово за допомогою світлодіодів (Рис.4.9).

Ознайомитись з кодом програми та композиційним термом можна за посиланням: https://github.com/Zylevich96/Example_1.

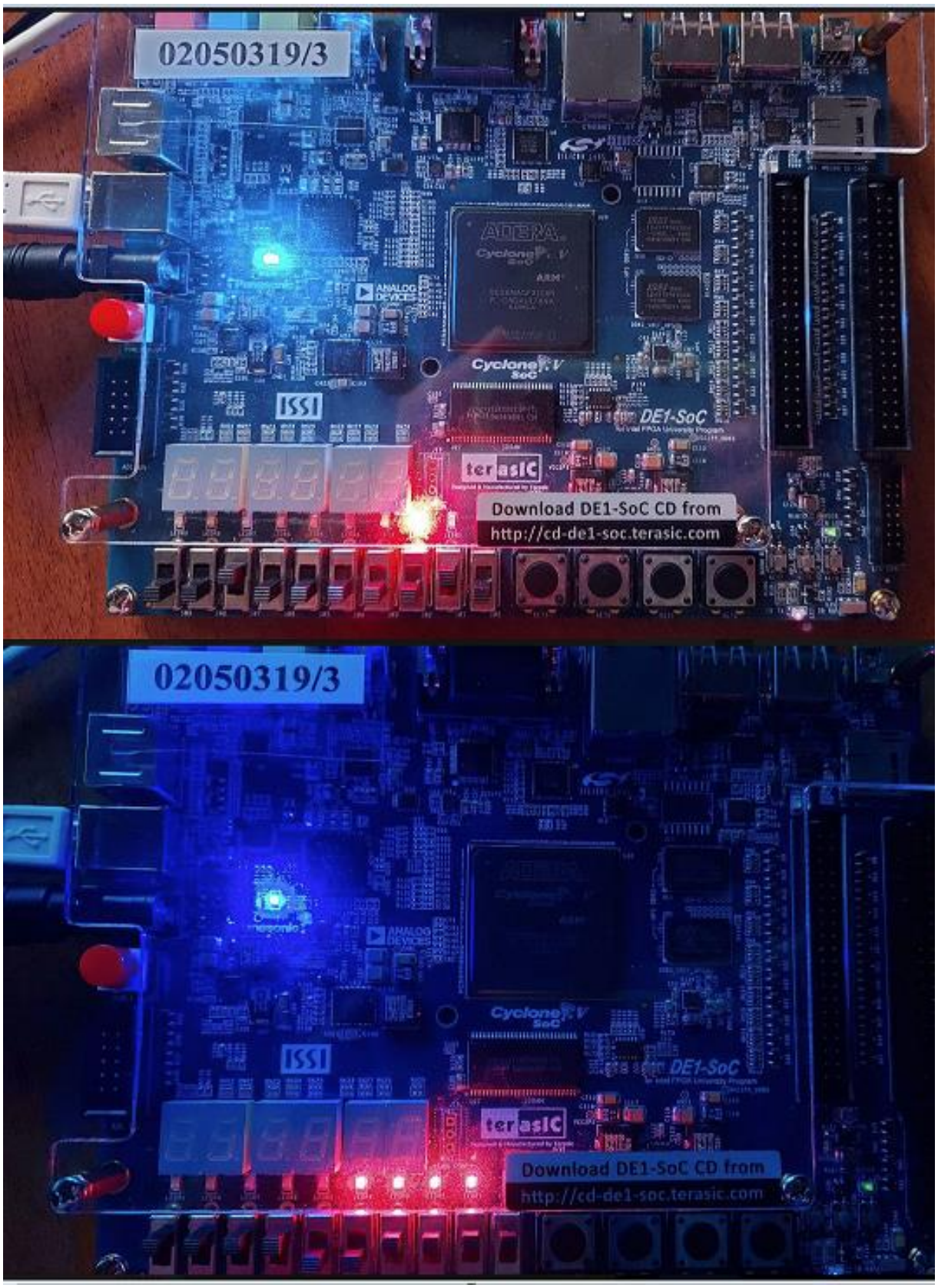


Рис. 4.9 Реалізації 4-розрядного суматора на базі плати DE1-SoC.

Світлодіоди визначаються зліва на право від молодшого до старшого біта. Перемикачі реалізують введення даних і зчитуються аналогічним чином - зліва молодший біт, а з права старший. Для зручності сприйняття крайні ліві чотири перемикачі це число A, а крайні праві чотири перемикачі число B. Відповідно на верхній частині рисунка зображено результат додавання 2+2, а на нижній 15+15 (із засвіченням біту перенесення).

Таким чином розроблено в суб'єкто-об'єктному середовищі композиційну модель телекомунікаційної системи, що реалізує операцію додавання двох 4-х розрядних чисел. Дана модель дозволяє застосовувати композиції з автоматичною генерацію коду програми в мові Verilog. Текстова мова запису композицій реалізує можливість задавати та застосовувати функції, композиції, а також метакомпозиції.

Аналогічним чином в розробленому суб'єкто-об'єктному середовищі створено радіопередавач, що працює в FM діапазоні з керованою частотою вихідного сигналу в межах від 90 МГц до 108 МГц. Апаратної частина передавача реалізована на базі FPGA Cyclon 5 (на базі плати DE1 SoC). Передавач підтримую керування частотою за допомогою кнопок з кроком 0.1 МГц. Приймання сигналу для передачі здійснюється за допомогою USB to UART конвертора. Таким чином необхідна для передачі інформація передається з комп'ютера чи іншого периферійного пристрою, що підтримує UART протокол на плату для подальшої передачі шляхом радіосигналів.

Структурна схема апаратної реалізації (Рис.4.10).

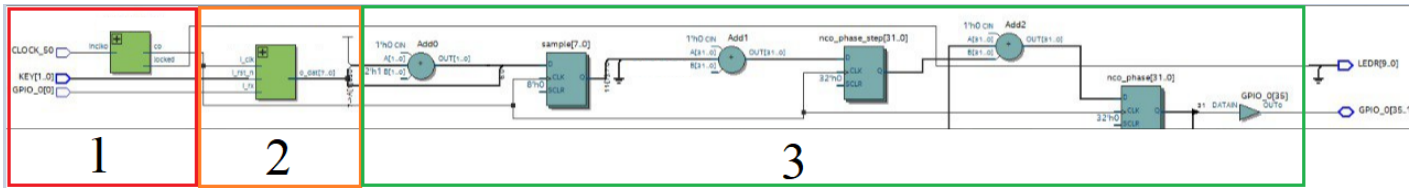


Рис. 4.10 Структурна схема логічної реалізації FM-передавач у САПР "Quartus".

Структурно схема ділиться на 3 блоки:

1) Блок фазової автопідстройки частоти (ФАПЧ). Головне завдання цього блоку це генерація тактового сигналу із визначеним фазовим зсувом щодо вхідного

тактового сигналу. Фазовий зсув між вхідним і вихідним імпульсними сигналами може бути встановлений в межах від 0 до 2π. Частота вихідного тактового сигналу, залежно від параметрів, може бути кілька разів вищою або нижчою, ніж у вхідного тактового сигналу. На вхід подається періодичний імпульсний сигнал із частотою, що зазвичай знаходиться на рівні кількох десятків мегагерц від зовнішнього кварцевого генератора. Вихід "locked" встановлюється в логічну одиницю, коли компонент успішно запускається і між вхідним та вихідним імпульсними сигналами установлюється вказаний фазовий зсув. В даній реалізації блок використовується лише для підвищення тактової частоти від 50МГц з тактового генератора до 120МГц для подальшого використання у якості несучої частоти передавача.

2) Блок передачі даних від переферійного пристроя до передавача (UART). ля ілюстрації передачі використовується 8-розрядний аудіофайл у форматі wav із зазначеним аудіосигналом. Сигнал від периферійного пристрою передається до FM-передавача на комп'ютері через UART-інтерфейс із швидкістю 230 400 біт в секунду. Це призводить до появи нових 8-розрядних значень аудіосигналу на виході UART із частотою 23 040 Гц, що наближена до стандартної частоти дискретизації аудіо у 22 050 Гц. Після цього відбувається модуляція несучої частоти пропорційно до зміни 8-розрядних цифрових кодів, що відображають значення аудіосигналу в конкретний момент часу із заданою частотою дискретизації. Структурна схема апаратної реалізації (Рис.4.11).

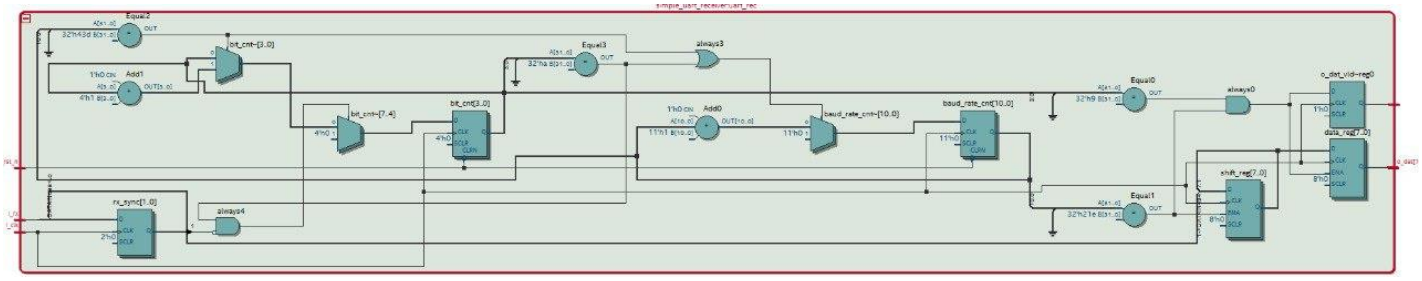


Рис. 4.11 Структурна схема логічної UART передавачf у САПР “Quartus”.

3) Програмований цифровий генератор NCO (Numerically Controlled Oscillator). Є основої будь-якого телекомунікаційного обладнання, що реалізує

широтно-імпульсну, фазову, частотну чи амплітудну модуляцію сигналів. Це електронний пристрій, який генерує вихідний сигнал з контрольованою частотою, що використовується в багатьох областях, таких як цифрова обробка сигналів, телекомунікації, радіо та інші, де важлива точна генерація сигналів з регульованою частотою. Основний принцип роботи NCO полягає в тому, що частота генерованого сигналу керується числовим значенням, яке змінюється відповідно до вхідних управляючих сигналів або параметрів. Основні етапи роботи NCO:

Вхідні параметри: Ключовими параметрами NCO є числове значення частоти (частотне слово або крок) та фазовий зсув.

Арифметичні операції: Частотне слово вводиться в NCO, і там проводяться арифметичні операції (зазвичай додавання або множення) для обчислення фази сигналу. Фаза є величиною, яка визначає момент в часі для сигналу.

Генерація сигналу: Отримана фаза використовується для генерації вихідного сигналу за допомогою синусоїдальної або іншої характеристичної функції.

Керування частотою: Зміна числового значення частоти дозволяє змінювати частоту генерованого сигналу в реальному часі. Це робить NCO дуже гнучким і корисним для задач, де необхідно динамічно налаштовувати частоту сигналу. Базова схема описаного генератор (Рис.4.12).

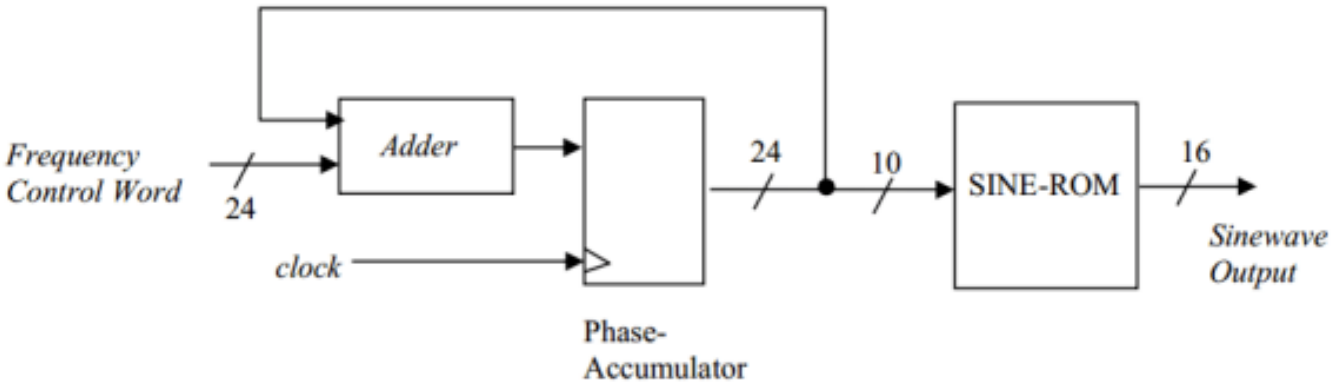


Рис. 4.12 Базова логічна структура прогамованого цифрового генератора.

Розглянемо принцип роботи NCO. У звичайному цифровому генераторі синусоїдального сигналу, період синусоїдального сигналу визначається періодом переповнення лічильника, який визначає адресу комірок постійної пам’яті (використання коефіцієнтів з пам’яті є апаратним обмеження апаратної платформи). Перша точка періоду синусоїдального сигналу відповідає нульовому значенню адреси ROM, а остання точка відповідає найбільшому значенню N-розрядної адреси ROM.

Коли N-1 адреса ROM досягає максимального значення, наступний активний фронт сигналу синхронізації призводить до того, що лічильник переповнюється, а формування періоду синусоїдального сигналу розпочинається знову. Таким чином, частота синусоїди залежить від частоти переповнення лічильника адреси постійної пам’яті (кількість переповнень лічильника за секунду). За допомогою NCO ми можемо керувати частотою переповнення лічильника, додаючи не лише одиницю при кожному активному фронті сигналу синхронізації, але й значенням phase_step. Наприклад, для 3-розрядного лічильника, щоб його переповнити, потрібно вісім разів додати одиницю до нульового вмісту лічильника, чи чотири рази додати значення 2, або двічі додати значення 4. Це є основною ідеєю роботи NCO.

Останнім кроком на шляху до реалізації та практичної перевірки отриманого коду є його виконання в базисі FPGA шляхом програмування відповідної мікросхеми отриманим під час компіляції файлом прошивки. Для наочності додамо побітове введення вхідних даних за допомогою перемикачі, а результат відображатиметься побітово за допомогою світлодіодів (Рис.4.12).

Ознайомитись з кодом програми та композиційним термом можна за посиланням: https://github.com/Zylevich96/Example_2_FM.

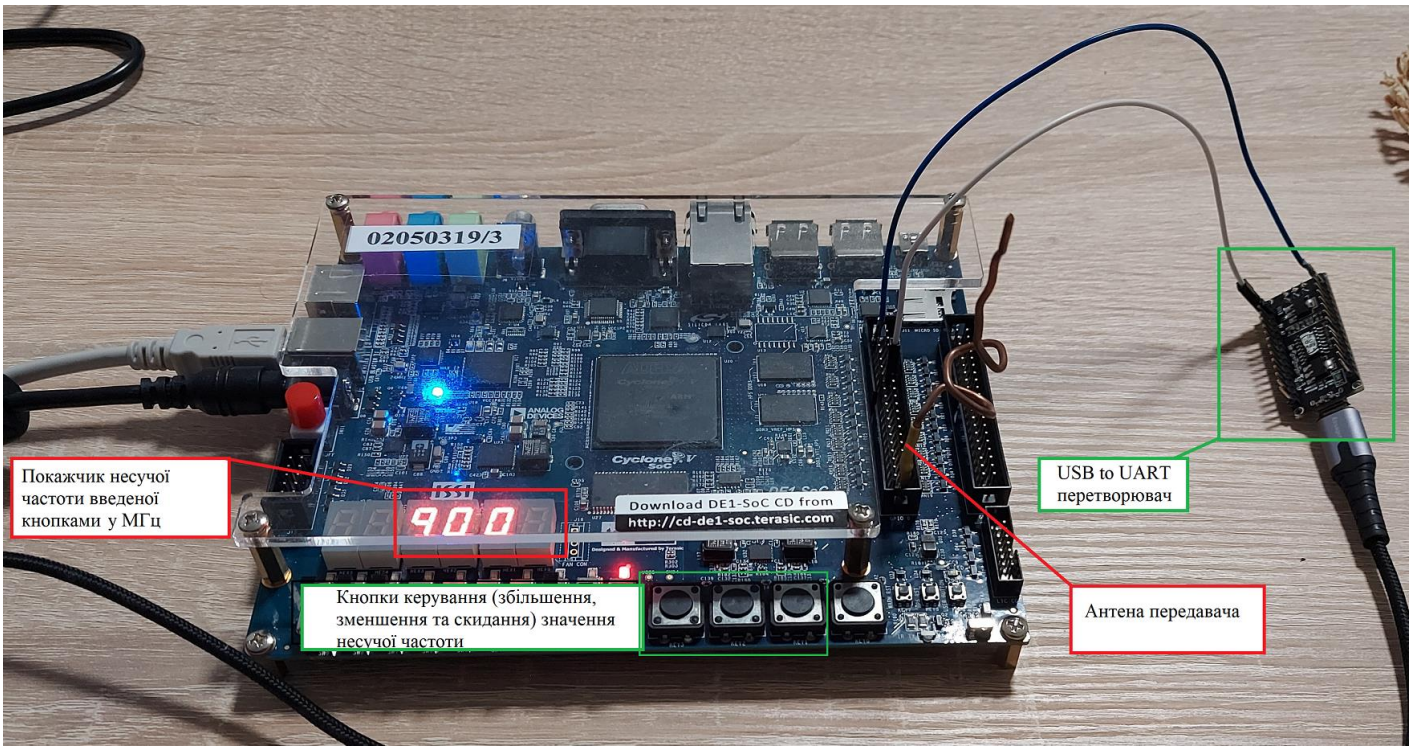


Рис. 4.13 Реалізації FM-передавача на базі плати DE1-SoC.

Синій дріт використовується для передачі коду від UART перетворювача до плати. Білий дріт використовуєтсья для з'єднання "земель" перетворювача і плати. UART перетворювач безпосередньо підключений до USB порту комп'ютера.

4.4. Аналіз ефективності

Підвищення ефективності розробки у запропонованому СОСрП обґрунтоване зменшенням витрат на переврку і верифікацію коду а також зменшенням надлишковості останнього, про кількісні показники ефективності не є однозначно визначеним. Існує ряд методів для оцінки ефективності, зокрема: COCOMO, SLIM, PERT та ін. Серед них найвідомішим є COCOMO (Constructive Cost Model), що і був використаний для оцінки використання СОСрП. Затрати на розробку попередньо оцінюються за наступною формулою $E_{nom} = A * (Size)^B$, де Enom - затрати на розробку в людино-годинах, а Size - відносний розмір рішення, що залежить від кількості рядків код у обраній мові програмування, функціоналу, що реалізовується, A - константа, в залежності від типу рішення, B -масштабуючий показник.

Ця формула використовується для передбачення витрат на розробку, і її параметри визначаються емпіричним шляхом. Константа А залежить, зокрема, від типу рішення: органічного (маленька команда та низькі вимоги), напіврозділеного (середня команда змішаного досвіду та вимог), і вбудованого (велика команда та жорсткі вимоги). Ці умови є прагматичними та залишаються поза впливом при аналізі. Константа має середнє значення близько 2 і може змінюватись в широких межах. Розмір рішення, як об'єктивний параметр, залежить від методів розробки. Масштабуючий показник В враховує специфіку рішення та визначається наступною формулою $B = 1.01 + \sum_{j=1}^{5} SF_j$, де SF_j - масштабуючі коефіцієнти. В таблиці наведено перелік таких коефіцієнтів, що визначається властивостями розробки та має значення, яке суб'єктивно визначається експертом в межах від Very Low до Extra High. Таким чином для кожної властивості розробки маємо наступні рівні значимості: Very Low (Дуже низька), Low (Низька), Normal (Звичайна), High (Висока), Very High (Дуже висока), Extra High (Надзвичайно висока) (таб. 4.2). Приведемо ці властивості та значення констант для кожного рівня значимості. Таблиця 4.2 Масштабуючі коефіцієнти

Поз.	Скороч.	Опис	VL	L	N	H	VH	EH
SF1	PREC	Розуміння продукту та технології організації	0.05	0.04	0.03	0.02	0.01	0
SF2	FLEX	Гнучкість розробки	0.05	0.04	0.03	0.02	0.01	0
SF3	RESL	Ризики, розуміння архітектури	0.05	0.04	0.03	0.02	0.01	0
SF4	TEAM	Атмосфера колективу	0.05	0.04	0.03	0.02	0.01	0
SF5	PMAT	Надійність процесу розробки	0.05	0.04	0.03	0.02	0.01	0

Запропоноване СОСрП явно впливає на гнучкість розробки (FLEX) та надійність чи зрілість процесу розробки (PMAT), тобто можна визначити ці коефіцієнти рівними 0, що відобразить властивості запропонованого середовища. Відповідно можна розрахувати максимальне значення В, що є рівним 1.16. Виходячи з такого впливу можна визначити графік максимально можливої економії затрат на розробку в залежності від розміру рішення. Для побудови графіку значення економії

розраховується за формулою $Economy = \frac{Size^{1.26} - Size^{1.16}}{Size^{1.26}} * 100\%$. Результати обчислень (Рис.4.13).

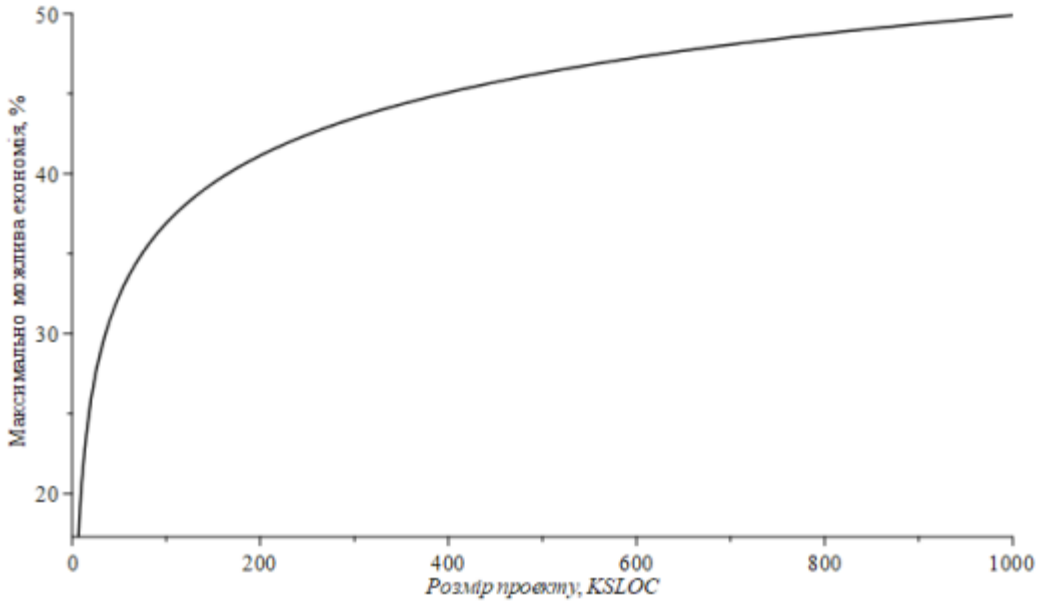


Рис. 4.13 Максимально можлива економія затрат на розробку

Як видно з графіку, чим більше розмір рішення (вимірюється в KSLOC – тисячах рядків коду), тим відчутніша економія. Після початкових фаз розробки, затрати на розробку можна уточнити корегуючими коефіцієнтами з таблиці 4.3 та 4.4.

Таблиця 4.3 Корегуючі коефіцієнти EM1-EM12.

Поз.	Скороч.	Опис
EM1	RELY	Вимоги до надійності рішення
EM2	DATA	Розмір бази даних
EM3	CPLX	Складність продукту
EM4	RUSE	Можливість повторного використання
EM5	DOCU	Документація
EM6	TIME	Тривалість неперервної роботи
EM7	STOR	Пам’ять
EM8	PVOL	Волатильність платформи
EM9	ACAP	Аналітичні здібності розробників
EM10	PCAP	Досвід, комунікабельність розробників
EM11	AEXP	Персональний досвід
EM12	PEXP	Досвід розробки на даній платформі

Таблиця 4.4 Корегуючі коефіцієнти EM13-EM17.

Поз.	Скороч.	Опис
EM13	LTEX	Знання мови та середовища розробки
EM14	PCON	Частота зміни залучених спеціалістів
EM15	TOOL	Ефект інструментів розробки
EM16	SITE	Географічний розподіл команди
EM17	SCED	Графік розробки

Відповідно кожен з наведених вище коєфіцієнтів, як і складові показника *B* ранжується від VL до EH (таб.4.5).

Таблиця 4.5 Значення корегуючих коефіцієнтів

	VL	L	N	H	VH	EH
RELY	0.75	0.88	1	1.15	1.39	
DATA		0.93	1	1.09	1.19	
CPLX	0.75	0.88	1	1.15	1.30	1.66
RUSE		0.91	1	1.14	1.29	1.49
DOCU	0.89	0.95	1	1.06	1.13	
TIME			1	1.11	1.31	1.67
STOR			1	1.06	1.21	1.57
PVOL		0.87	1	1.15	1.30	
ACAP	1.5	1.22	1	0.83	0.67	
PCAP	1.37	1.16	1	0.87	0.74	
AEXP	1.24	1.10	1	0.89		
PEXP	1.22	1.12	1	0.81		
LTEX	1.22	1.10	1	0.84		
PCON	1.24	1.10	1	0.92	0.84	
TOOL	1.24	1.12	1	0.86	0.72	
SITE	1.25	1.10	1	0.92	0.84	0.78
SCED	1.29	1	1	1	1	

Особливістю запропонованого СОСрП є надійність верифікації разово отриманого рішення (не потребує повторних верифікацій), тож можна вважати додаткових ресурсів для перевірки якості витрачати не треба, що своєю чергою дозволяє зафіксувати RELY та TIME на рівнях VL та N відповідно. Також забезпечується можливість повторного використання рішення, що тож доцільно зафіксувати коефіцієнт RUSE на рівні L.

Враховуючи те, що у сучасній проєктній діяльності усе частіше і частіше використовується автоматична генерація проєктної документації (за винятком користувацьких документів, що часто є індивідуальним для кожного продукту з точки зору маркетингу), то коефіцієнт DOCU можна визначити на рівні VL. Також особливістю є адаптованість і гнучкість налаштувань запропонованого середовища, тож з коефіцієнт TOOL визначено на рівні 0.86. Сумарно ефект всіх актуалізацій коефіцієнтів рівний:

$$S_{tot} = RELY * TIME * TOOL * DOCU * RUSE = 0.75 * 1 * 0.86 * 0.89 * 0.91 = 0.52$$

Отримане значення відповідає близько 48% економії затрат. Враховуючи вигоду отриману з попереднього розрахунку (про можливу економію затрат рис.4.13), то оновивши формулу наступним чином: $Economy = \frac{Size^{1.26} - Size^{1.16} * 0.52}{Size^{1.26}} * 100\%$ отримаємо наступний графік (Рис.4.14).

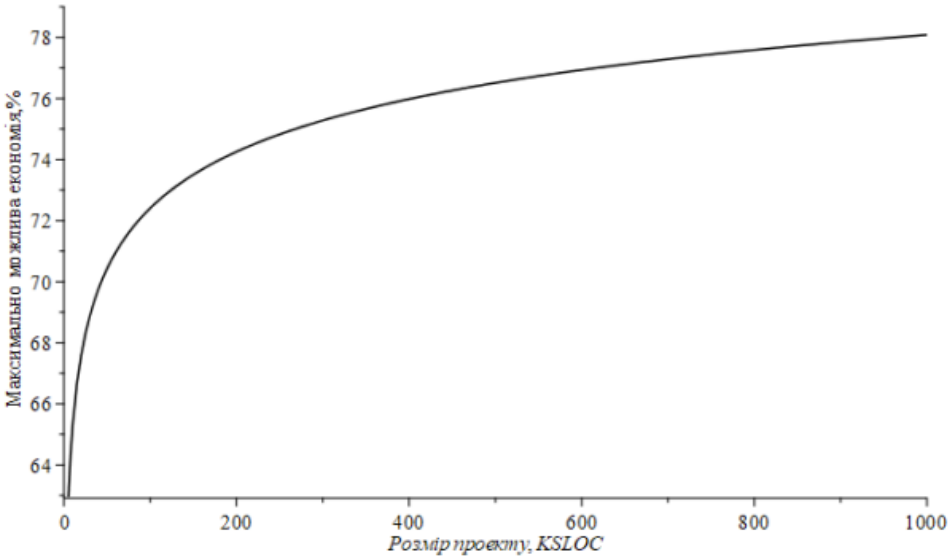


Рис. 4.14 Максимально можлива економія затрат на розробку з урахуванням з уточнюючих коефіцієнтів

Як результат проведеного аналізу можна стверджувати, що використання запропонованого середовища здатне забезпечити до 74% економії затрат на розробку програмної складової програмно-апаратного комплексу. Для отримання максимально точних розрахунків доцільно застосувати модель оцінювання повною мірою.

ВИСНОВКИ

В дисертаційній роботі вирішено актуальну та важливу науково-практичну задачу технологізації процесів вирішення сучасних задач в людино-машинних, зокрема, телекомунікаційних системах, методом композитологічного уподібнення – логічного ядра суб'єкто-об'єктного середовища програмування.

В результаті виконання досліджень отримано наступні нові наукові результати:

1. Проведено аналіз взаємозв'язку концепто-монадної парадигми програмування та телекомунікаційних систем програмування, результатом якого стало предметне збагачення концепто-монадної парадигми програмування видом телекомунікаційних систем програмування. Запропоновано відповідну понятійну систему телеконцептування, основу якої складають поняття телеконцепту, телеконцептограми, монади, оракульної телекомунікації, телекомпозиції, телекомпозиту. Телеконцептування конкретизовано покроковістю застосувань оракульних телекомунікацій як суб'єктоорієнтованих телеконцептів – базових логічних інструментів активностей суб'єкта програмування. Зміст їх розкрито у концептомонадному середовищі через оракули «обумовлення», «концепт», «монада», «сутність», «суть». Це забезпечує можливість реальної інтеграції наявних підходів проєктування програмно-апаратних комплексів телекомунікаційних систем у вигляді взаємодоповнення процесів програмування та їх результатів, що складає основу реального розуміння програмування і дозволяє відійти від сучасного інтуїтивного базису, якісно його розвинувши за допомогою сучасних досліджень та розробок.

2. Визначено, що головною передумовою розуміння програмування як продуктивної діяльності є об'єктивізація взаємодоповнення його активної та пасивної складових як відношення телекомпозитного уподібнення. Відкрито-замкненість телекопозитних схем або телеконцептограм як предметних збагачень оракульних телекомунікацій забезпечує продуктивність та еволюційність телеконцептування як експлікації телекомунікаційного збагачення програмування.

3. Подальшого розвитку набув метод редукції, побудовано та досліджено ряд нових редукційних схем, обумовлених відповідними композитами як предметнообумовлених шаблонів програмування, сформульовано та доведено стосовно них корисні необхідні умови редукційності. Парадигмне значення уведених редукцій та їх властивостей у тому, що вони конкретизують вирішення задач як покрокове розкриття структур генезисів їх рішень, з урахуванням яких процедурні, алгоритмічні, програмні реалізації цих рішень, зокрема їх нотація у мовах програмування отримуються вже автоматично, з обумовленою їх побудовою, коректністю. Продемонстровані важливі загальні особливості редукційного концептування оракульних схем. Обгрунтовано, що метод редукцій є дієвим інструментом для технологізації програмування та формування середовищ, а редукційні моделі програм та редукційні методи програмування є прагматико-обумовленою конкретизацією цього середовища.

4. Запропоновано зведення телеконцептування до оракульних телекомунікацій, які є телеконцептами для різних телекомпозитних і телекомпозиційних збагачень концептування. Інструментом такого зведення є конкретизація оракулів, що входять до цих оракульних телекомунікацій. Це дало змогу збагатити новим змістом програмування як діяльність, орієнтовану не тільки на нотацію одержуваних рішень, але й таку, яка здатна забезпечувати продуктивну діяльність суб’єкта для їх отримання із залученням розвинених засобів програмування, адаптованих для реалізації активної ролі суб'єкта.

5. Досліджено зведення до телеконцептограм генетичних структур програм та запропоновано прагматичну обумовленість, що дозволяє реально, а не лише номінально підтримувати причинно-насідкові зв’язки при вирішенні задач, а також способи, методи та засоби їх специфікації. В якості телеконцептограм розглянуто телекомпозити – спеціальні класи суб’єктоорієнтованих базових телекомпозицій. Таким чином, телеконцептування на предметному рівні зводиться до вирішення відповідних рівнянь

телекомпозитних редукцій, що забезпечує коректність отримуваних рішень "за побудовою".

6. Проведенено аналіз оракульного телеконцептування, у результаті якого запропоновано спосіб застосування підходу оракульного телеконцептування для подальшого предметного збагачення суб'єкто-об'єктного середовища програмування. На репрезентативних прикладах показані його особливості та перспективи подальшого розвитку. До особливостей відноситься те, що кожна підзадача може бути проконцептована до найпростішої підзадачі, Також використання оракульного телеконцептування дає можливість використання традиційного математичного апарату для нотації результату та поєднання його з денотативними методами. Реалізація такого методу на практиці сприяє уніфікації процесу розробки програмного продукту, тим самим оптимізує та реально об'єктивізує вплив активної ролі суб'єкта у телеконцептуванні через механізм оракульних телекомунікацій як технологію телекомунікаційних рішень задач.

7. Досліджено логіко-предметні засади суб'єкто-об'єктної телекомунікаційної системи програмування як предметного замикання СОСрП – несуперечливої логічної абстракції цілісного різноманіття програмно-апаратних комплексів. Головною особливістю створюваних таким чином телекомунікаційних систем програмування є те, що вони реально, а не лише номінально підтримують причинно-наслідкове взаємодоповнення двох складових вирішення будь-якої програмістської задачі – програмування як породження та застосування композицій та програми – наслідку програмування.

8. За допомогою сигнатури примітивної програмної алгебри у якості концепту середовища програмування адекватно об'єктивізовано роль суб'єкта телекомунікаційної системи програмування. Це своєю чергою дозволило залучити для дослідження інструмент логіко-математичних специфікацій семантико-синтаксичних аспектів програмування рішень задач. Таким чином, основані на концептомонадній парадигмі

програмування телекомунікаційні системи реально підтримують та адекватно об’єктивізують активну роль суб’єкта телеконцептування.

9. Визначено, що продуктивна редукція грає фундаментальну роль у забезпеченні технологізації програмування. Підтверджено, що технологія програмування використовує редукційне програмування як засіб перетворення інформаційного ресурсу у програмний продукт у суб’єкто-об’єктному середовищі програмування. На репрезентативному прикладі продемонстровано використання концептів програмування у вигляді семантичних шаблонів як ланок програмного ланцюга, які обумовлюють певні класи програм. За допомогою редукційних методів телеконцептування у заданій системі була отримана телеконцептограма рішення, коректність якої випливає з її побудови. На основі отриманої специфікації за допомогою телепрограмного дескриптора отримано код програми в обраній мові програмування.

10. Розроблена дослідна реалізація суб’єкто-об’єктного середовища програмування, що підтримує розробку програмного забезпечення як предметного замикання СОСрП. Прагматикообумовлені умови такого замикання задаються у дескриптивному середовищі композиційних термів. Синтаксичне оформлення рішення здійснюється Verilog-дескриптором. Можлива підтримка створення апаратного забезпечення із залученням FPGA як базису апаратної платформи із використанням САПР “Quartus”. Аналіз ефективності використання запропонованого середовища може забезпечити до 74% економії затрат на розробку програмної складової програмно-апаратного комплексу.

Таким чином, можна стверджувати, що поставлені задачі вирішені, а основна мета роботи досягнута.

Одержані в дисертації нові результати використані під час під час виконання науково-дослідної роботи “Композитологічні засади технологічних систем програмування” (№0122U001568). Впровадження та використання результатів роботи підтверджено відповідною довідкою.

Результати дисертаційної роботи використані як матеріали при підготовці та викладанні курсу лекційних і практичних занять з дисципліни “Системне програмування та керування базами даних в телекомунікаціях” другого (магістерського) рівня вищої освіти спеціальності 172 «Електронні комунікації та радіотехніка» освітньо-професійної програми «Інформаційно-обчислювальні засоби радіоелектронних систем», що підтверджено відповідним актом.

Отримані в дисертації результати дослідження можуть бути використані:

1. При розробці програмно-апаратних рішень, які висувають підвищені вимоги до надійності.
2. В навчальному процесі вищих навчальних закладів України при підготовці фахівців у галузі конструювання електронно-обчислювальної апаратури та фахівців із розробки ІТ-рішень, зокрема програмних та апаратних обчислювальних засобів.
3. В результатах науково-дослідних робіт за суміжною тематикою.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. D. E. Knut, The art of computer programming. Fundamental Algorithms [Iskusstvo programmirovaniya. Osnovnyye algoritmy]. Moscow: Vyliams, 2006.

2. I.V. Redko, P.O. Yahanov, M.O. Zylevich, “Intersubjective paradigm and oracle conceptualization as an open-closed platform for programming technologicalization”, on *2022 IEEE 3rd International Conference on System Analysis & Intelligent Computing (SAIC-2022)*, Kyiv, 2022, pp. 65-70. doi: 10.1109/SAIC57818.2022.9923011

3. I.V. Redko, P.O. Yahanov, M.O. Zylevich, “Concept-Monadic Model of Technological Environment of Programming”, on 2020 IEEE 2nd International Conference on System Analysis & Intelligent Computing (SAIC-2020), Kyiv,2020, pp. 125-130. doi: 10.1109/SAIC51296.2020.9239204

4. J. McCarthy (1960). Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. Presented at the CACM 1960. [Online]. Available: https:aiplaybook.a16z.com/reference-material/mccarthy-1960.pdfI

5. G. Buch, “Object-oriented analysis and design,” Moscow, Russia: Binom, 1998.

6. D. L. Parnas. (1972). On the criteria to be used in decomposing systems into modules. Presented at the CACM 1972. [Online]. Available: https:www.win.tue.nl/~wstomv/edu/2ip30/references/criteria_for_modularization.pdfI

7. N. Wirth, “A Personal Computer for the Software Engineer. Proceedings,” presented at the ICSE 81., San Diego, CA, USA, March 9-12, 1981.

8. I. V. Redko, D. I. Redko, T. L. Zakharchenko, “Conceptual-logical foundations of design,” Kyiv, Ukraine: Comprint, 2016.

9. І.В.Редько, П.О.Яганов, М.О.Зилевіч, “Редукційне концептування оракульних схем”, Системні дослідження та інформаційн ітехнології, № 1, с. 21-33, 2021. doi: 10.20535/SRIT.2308-8893.2021.1.02

10. І.В.Редько, М.О.Зилевіч, “Редукційне програмування задач у технологічному середовищі програмування”, Вчені записки Таврійського національного університету

імені В.І. Вернадського. Серія: Технічні науки, т.34, №2, с.228-233, 2023. doi:https:doi.org/10.32782/2663-5941/2023.2.1/36

11. І.В.Редько, М.О.Зилевіч, “Теоретичні основи програмної релятивізації у технологічних системах програмування”, Вісник Вінницького політехнічного інституту, № 2, с.72-80, 2023. doi: https:doi.org/10.31649/1997-9266-2023-167-2-72-80

12. С.В.Кудлай, М.О.Зилевіч, І.В.Редько, П.О.Яганов, “Концептомонадна модель технологічного середовища програмування”, на XIII Міжнародній науково-технічній конференції молодих вчених “Електроніка-2020” (ELCONF-2020), Київ, 2020, с.45-49. doi: 10.20535/2617-0965.2020.3.3.198584

13. М.О.Зилевіч, “Застосування оракульного концептування при програмуванні дизайну електронних мікросхем”, на XIV Міжнародній науково-технічній конференції молодих вчених “Електроніка-2021” (ELCONF-2021), Київ, 2021, с.41-45. doi: 10.20535/2617-0965.eae.227740

14. І.В.Редько, П.О.Яганов, М.О.Зилевіч, “Технологічне середовище програмування з точки зору інтерсуб’єктивної парадигми”, на Міжнародна наукова інтернет-конференція на тему “Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення”, Тернопіль, 2022, с.30-34.

15. I.V. Redko, P.O. Yahanov, M.O. Zylevich, “Reduction programming in a technological programming environment”, on 12th International Conference on Electronics, Communications and Computing (IC ECCO-2022), Chisinau, Republic of Moldova, 2022, с.194-200.

16. І. В. Редько, П. О. Яганов, “Концептуальна модель технологічного середовища програмування”, Наукові вісті КПІ, №1 (128), с. 18-26, 2020. doi: https://doi.org/10.20535/kpi-sn.2020.1.197953

17. В.Н Редько, И.В. Редько, “Экзистенциальные основания композиционной пара - дигмы”, Кибернетика и системный анализ, №2, 2008, с.3-121.

18. И. В. Редько, “Теория дескриптивных сред и ее применения”, Докт. дисерт., Київ : НТУУ “КПІ”, 2008, 403 с.

19. И.В. Редько, “Дескриптивные аспекты системного подхода”, Системні дослідження та інформаційні технології, №3, 2005, с. 7-28.

20. Т.Л. Захарченко. Композитосутнісні моделі адаптивних процесональних середовищ : дис. … канд. тех. наук: 05.13.06. Київ, 2018. 191 с.

21. А. Френкель, И. Бар-Хиллел, “Ооснования теории множеств”, М.: Мир, 1966, 326с.

22. Ф. Хаусдорф, “Теория множеств”, М.: Ленинград: ОНТИ, 1937, 304 с.

23. И. А. Басараб, Н. С. Никитченко, В. Н. Редько, “Композиционные базы данных”, Київ,«Либідь», 1992, 192с.

24. Ю.А. Шиханович, “Введение в современную математику”, М.: Наука, 1965, 376 с.

25. М. Хайдеггер, “Бытие и время”, С-П.: Санкт-Петербургская издательская фирма «Наука» РАН, 2006, 451 с.

26. В. А. Успенский, А. Л. Семенов, “Теория алгоритмов: основные открытия и приложения”, Наука, 1987, 288 с.

27. Я. Хинтикка, “Логико-эпистемологические исследования”, М.: «Прогресс», 1980, 448 с.

28. Г. Фреге, “Логика и логическая семантика“, М.: Аспект пресс, 2000, 512 с.

29. А. Черч, “Введение в математическую логику”, М.: ИИЛ, 1960, 485 с.

30. Д.П. Горский, “Определение”, М.:Изд-во «Мысль», 1974, 312 с.

31. Э. Гуссерль, “Логические исследования. Картезианские размышления”, Минск: Харвест, М.: АС, 2000, 752 с.

32. Р. Карнап, “Значение и необходимость”, М.: Изд.проєкт «Тривиум», 1958, 380 с.

33. С. Клини, “Введение в метаматематику”, М.: Издательство иностранной литературы, 1957, 526 с.

34. К. Поппер, “Объективное знание. Эволюционный подход”, М.: Изд-во «Эдиториал УРСС», 2002, 384 с.

35. В. Н. Редько, И. В. Редько, “Дескриптологические основания информа-ционных технологий”, Кибернетика и системный анализ, №5, 2007, с. 12-28.

36. Дж. Бекус, “Алгебра функциональных программ: мышление функционального уровня, линейные уравнения и обобщенные определенияСб. статей: Математическая логика в программировании”, М.: Мир, 1991, 408 с.

37. M. Jackson, “Software requirements and Specifications”, Cambridge: Addison-Wesley, 1995, 228 p.

38. В. Н. Редько, “Композиции программ и композиционное программирование”, Программирование, № 5, 1978, с. 3-24.

39. E. W. Dijkstra, “Notes оп Structured Programming”, Structured Programming, London: Academic Press, 1972, pp. 1-82.

40. J. McCarthy, “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”, Communications of the ACM, № 4, 1960, pp. 184-195.

41. Г. Буч, “Объектно-ориентированный анализ и проєктирование”, М.: Бином, 1998, 560 с.

42. В. Н. Редько, И. В. Редько, “Дескриптивные системы: ретроспективы и перспективы”, Вісник Київського національного університету ім. Т.Шевченка. Сер. Фіз.-мат.науки, 2004, с. 68-75.

43. В. Н. Редько, “Композиционная структура программологии”, Кибернетика и системный анализ, №4, 1998, сс.47-66.

44. В. Н. Редько, “Композиции программ и композиционное программирование”, Программирование,№ 5, 1978, с. 3-24.

45. В. Н. Редько, “Основания композиционного программирования”, Программирование, №3, 1979, с. 3-13.

46. Н. Вирт, “Алгоритмы и структуры данных”, М.: Мир, 1989, 307 с.

47. C. A. Hoare, “Jifeng He. Unifying Theories of Programming”, London: Prentice HaH Europe, 1998, 298 p.

48. Дж. Робинсон, “Логическое программирование - прошлое, настоящее и будущее”, Сб. стат. Логическое программирование, М., 1988, с. 7-26.

49. C. A. Hoare, “An Axiomatic Bases for Computer Programming”, Comm. of the ACM, №10, 1969, pp. 576-580.

50. Э. Дейкстра, “Дисциплина программирования”, М.: Изд-во «Мир», 1978, 275 с.

51. E.W. Dijkstra, “The End of Computing Science? ”, Comm. ACM, №3, 2001, pp. 92-98.

52. В. Н. Редько, Н. С. Никитченко, “Композиционные аспекты программологии”, Кибернетика, №5, 1987, с. 49-56.

53. В. Н. Редько, Ю. Й. Брона, Д. Б. Буй, С. А. Поляков, “Реляційні бази даних: табличні алгебри та SQL-подібні мови”, К.: «Академперіодика», 2001, 197с.

54. Дж. Робинсон, Э. Зиберт, “Логлисп: основные возможности и реализация Сб. стат. Логическое программирование”, М., 1988, с. 261-275.

55. В. Н. Редько, “Композиционная структура программологии”, Кибенетика и системный анализ, №4, 1998, с. 47-66.

56. В. Н. Редько, “Дескриптологические основания программирования”, Кибернетика и системный анализ, №1, 2002, с.3-19.

57. Г. Вригт, “Логико-философские исследования”, М: Изд «Прогресс», 1986, 595 с

58. И. В. Редько, “Дескриптологическая среда моделирования предметных областей”, Пробл. Программирования, №1, 2002, с. 44-50.

59. І.В. Редько, О.М. Лисенко, Композиційні засади проєктування баз даних. Монографія, Київ, 2019, 114с.

60. И. В. Редько, “Интенсиональные основания дескриптивных сред”, Пробл. программирования, №3, 2006, с. 81-85.

61. Д. Б. Буй, И. В. Редько, “Примитивные программные алгебры вычислимых функций”, Кибернетика, №3, 1987, с. 68-74.

62. И. В. Редько, “Экспликативное моделирование: интеграционные аспекты”, Проблемы программирования,№2, 2000, с. 280-285.

63. И. В. Редько, “Дескриптивные среды: интеграционные основания”, Пробл. Програмув, №1, 2006, с. 17-23.

64. Г. Е. Цейтлин, “Введение в алгоритмику”, К.: Сфера, 1998, 310с.

65. В. М. Глушков, Г. Е. Цейтлин, Е. Л. Ющенко, “Алгебра. Языки. Программирование”,К.:Наук.думка, 1974, 328с.

66. Х. Барендрегт, “Ламбда-исчисление”, М.: Мир, 1985, 606с.

67. G. Frege, “ Über Sinn und Bedeutung”, Zeitschrift für Philosophie und philosophische Kritik, №4, 1982, с. 25-50.

68. В. Н. Редько, “Семантические структуры программ”, Программирование, №1, 1981, с. 3-19.

69. В. Н. Редько, “Универсальные программные логики и их применение”, Тез. Докл. IV Всесоюзн. Симп, Кишине, 1983, с.310-326.

70. В. М. Глушков, “Теория автоматов и формальные преобразования микропрограмм”, Кибернетика, №5, 1965, с. 3-10.

71. Х. Барендрегт, “Ламба-исчисление”, М.: Мир, 1985, 606с.

72. Т. Л. Захарченко, Д. І. Редько, І. В. Редько, П. О. Яганов, “Примітивна програмна алгебра обчислюваних функцій над записами”, Наукові вісті НТУУ «КПІ», №2, 2015, с. 29-40.

73. В. Н. Редько, И. В. Редько, “Экзистенциальные основания композиционной пара – дигмы”, Кибернетика и системный анализ, №2, 2008, с. 3–12.

74. К. Дж. Дейт, “Введение в системы баз данных”,М.: Мир,1983, 506с.

75. И. В. Редько, “Открыто-замкнутые основания сред интеграции. Часть1”,Систем. дослідж. та інформ. технології, №4, 2010, с. 7-17.

76. І. В. Редько, “Дескриптивные аспекты системного подхода”, Системні дослідження та інформаційні технології, №3, 2005, с. 7–28.

77. И. В. Редько, “Экзистенциальный базис сущностных сред”, Системні дослідження та інформаційні технологі, №3, 2008, с. 16–31.

78. P. Garcia, K. Compton, M. Schulte, "An overview of reconfigurable hardware in embedded systems.", EURASIP J. Embedded Syst. 2006, 1 (January 2006), pp. 13-13.

79. D. Gajski, F. Vahid, S. Narayan, and 1. Gong, "SpecSyn: an environment supporting the specifyexplore-refine paradigm for hardware/software system design," IEEE Trallsactiolls 011 VLSI Systellls, № 6, 1998, pp. 84-100.

80. R. Kuon, R. Tessier, J. Rose, "FPGA Architecture: Survey and Challenges", Found. Trends Electron. Des. Autom. 2, 2 (February 2008), pp. 135-253.

81. S. Mohanty, V. Prasanna, "Rapid system-level performance evaluation and optimization for application mapping onto SoC architectures," in Proc. of ASIC +SOC, 2002, pp. 160-167.

82. R. Zurawsky, "Embedded systems handbook", CRC press, 2005, 1089p.

83. S. Leibson, "Designing SOCs with configured cores : unleashing the Tensilica Xtensa and diamond cores", Morgan Kaufmann Publishers, 2006, 321p.

84. G Smith, "Platform Based Design: Does it Answer the Entire SoC Challenge", in Proc. of DAC, 2004, 407p.

85. F. Campi, "A dynamically adaptive DSP for heterogeneous reconfigurable platforms", in Proc. of DA7E, 2007, pp. 9-14.

86. Morris, "Actel Activates Platforms Roadmap Solidifies Embedded Processor Strategy", Embedded Technology Journal, 2007: [електрон. ресурс], Режим доступа: http://www.esilicon.com/

87. A. DeHon, "The Density Advantage of Configurable Computing," IEEE Computer, №4, 2000, pp. 41-49.

88. J. M. Arnold, "S5: the architecture and development flow of a software configurable processor", Field-Programmable Technology, 2005, pp. 121-128.

89. L. Perre, J. Craninckx, A. Dejonghe, "Green Software Defined Radios: Enabling Seamless Connectivity While Saving on Hardware and Energy (", Springer, 2008, 160p.

90. ASIC Design, Custom IP & IC Manufacturing: [електрон. ресурс],– Режим доступа: http://www.esilicon.com

91. F. Doucet, R. K. Shyamasundar, Krüger, "Reactivity in SystemC Transaction-Level Models", Paper presented at the meeting of the Haifa Verification Conference, 2007, pp 22-37.

92. S, Swan, "SystemC transaction level models and RTL verification," Design Automation Conference, 2006 43rd ACM/IEEE, pp.90-102.

93. L. Cai, D. Gajski, "Transaction level modeling: an overview", Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware-software codesign and system synthesis (CODES+ISSS '03). ACM, New York, NY, USA, pp. 19-24.

94. K. Hines, G. Borriello. "Dynamic communication models in embedded system co-simulation", Proceedings of the 34th annual Design Automation Conference (DAC '97). ACM, New York, NY, USA, pp. 395-400.

95. K. N. Chia, “Configurable Computer Solutions for Automatic Target Recognition”, FCCM-IEEE, 1996, pp. 70-79.

96. C. Jones, “Video Communications using Rapidly Reconfigurable Hardware”, IEEE Transactions on Circuits and Systems for Video Technology, №5, pp. 565-567.

97. F. Doucet, “Reactivity in System Transaction-Level Models”, HVC'07 Proceedings of the 3rd international Haifa verification conference on Hardware and software: verification and testing, Haifa, 2007, pp. 34-50.

98. J. Adams, “Design patterns for reconfigurable computing”, Field-Programmable Custom Computing Machines, 12th Annual IEEE Symposium on IEEE, 2004, pp. 13-23.

99. K. Compton, “An overview of reconfigurable hardware in embedded system”, EURASIP J. Embedded Syst. 2006, pp. 13-24.

100. I. Kuon, “FPGA Architecture: Survey and Challenges”, Found. Trends Electron. Des. Autom. IEEE, 2008, pp. 135-253.

ДОДАТОК А. СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА ЗА ТЕМОЮ ДИСЕРТАЦІЇ

1. І.В.Редько, П.О.Яганов, М.О.Зилевіч, “Редукційне концептування оракульних схем”, *Системні дослідження та інформаційн ітехнології*, № 1, с. 21-33, 2021. doi: 10.20535/SRIT.2308-8893.2021.1.02 (фахове видання категорії Б, Scopus).
2. І.В.Редько, М.О.Зилевіч, “Редукційне програмування задач у технологічному середовищі програмування”, *Вчені записки Таврійського національного університету імені В.І. Вернадського. Серія: Технічні науки*, т.34, № 2, с.228-233, 2023. doi:https:doi.org/10.32782/2663-5941/2023.2.1/36 (фахове видання категорії Б).
3. І.В.Редько, М.О.Зилевіч, “Теоретичні основи програмної релятивізації у технологічних системах програмування”, *Вісник Вінницького політехнічного інституту*, № 2, с.72-80, 2023. doi: https:doi.org/10.31649/1997-9266-2023-167-2-72-80 (фахове видання категорії Б).
4. І.В.Редько, П.О. Яганов, М.О.Зилевіч, «Концептологічні засади технологічних систем програмування», Вчені записки Таврійського національного університету імені В.І. Вернадського. Серія: Технічні науки, т.34, № 5, с.219-223, 2023. DOI https://doi.org/10.32782/2663-5941/2023.5/34 (фахове видання категорії Б).
5. С.В.Кудлай, М.О.Зилевіч, І.В.Редько, П.О.Яганов, “Концептомонадна модель технологічного середовища програмування”, на *XIII Міжнародній науково-технічній конференції молодих вчених “Електроніка-2020” (ELCONF-2020)*, Київ, 2020, с.45-49. doi: 10.20535/2617-0965.2020.3.3.198584 (матеріали конференції).
6. I.V. Redko, P.O. Yahanov, M.O. Zylevich, “Concept-Monadic Model of Technological Environment of Programming”, on *2020 IEEE 2nd International Conference on System Analysis & Intelligent Computing (SAIC-2020)*, Kyiv,2020, с.125-130. doi: 10.1109/SAIC51296.2020.9239204 (матеріали конференції, Scopus).
7. М.О.Зилевіч, “Застосування оракульного концептування при програмуванні дизайну електронних мікросхем”, на *XIV Міжнародній науково-технічній конференції*

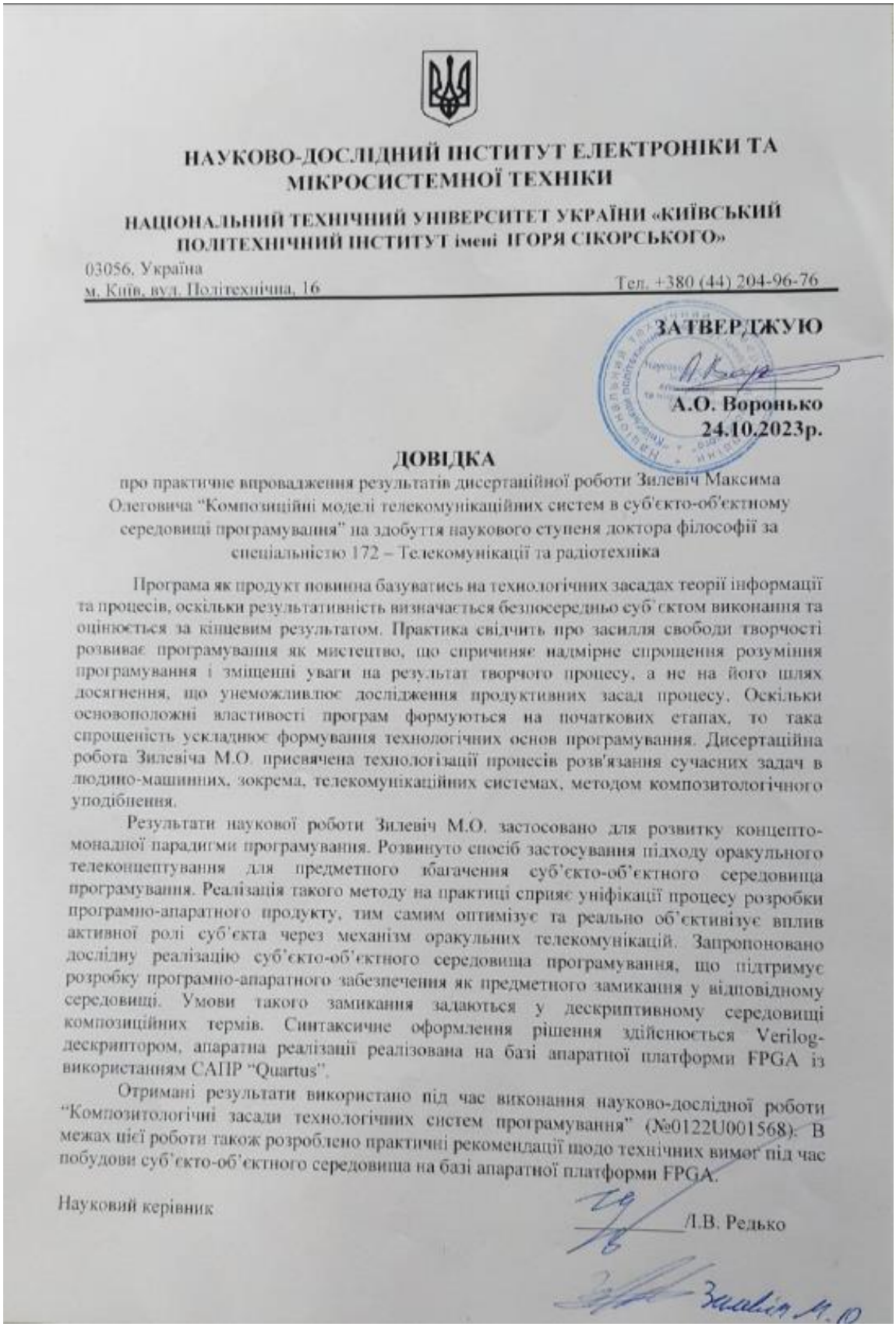
молодих вчених "Електроніка-2021" (ELCONF-2021), Київ, 2021, с.41-45. doi: 10.20535/2617-0965.eae.227740 (матеріали конференції).

8. І.В.Редько, П.О.Яганов, М.О.Зилевіч, "Технологічне середовище програмування з точки зору інтерсуб'єктивної парадигми", на *Міжнародна наукова інтернет-конференція на тему "Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення"*, Тернопіль, 2022, с.30-34. (матеріали конференції).

9. I.V. Redko, P.O. Yahanov, M.O. Zylevich, "Intersubjective paradigm and oracle conceptualization as an open-closed platform for programming technologicalization", on *2022 IEEE 3rd International Conference on System Analysis & Intelligent Computing (SAIC-2022)*, Kyiv, 2022, с.65-70. doi: 10.1109/SAIC57818.2022.9923011. (матеріали конференції, Scopus).

10. I.V. Redko, P.O. Yahanov, M.O. Zylevich, "Reduction programming in a technological programming environment", on *12th International Conference on Electronics, Communications and Computing (IC ECCO-2022)*, Chisinau, Republic of Moldova, 2022, с.194-200. (матеріали конференції).

ДОДАТОК Б. ДОВІДКА ПРО ВПРОВАДЖЕННЯ



НАУКОВО-ДОСЛІДНИЙ ІНСТИТУТ ЕЛЕКТРОНІКИ ТА МІКРОСИСТЕМНОЇ ТЕХНІКИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

03056, Україна
м. Київ, вул. Політехнічна, 16 Тел. +380 (44) 204-96-76

ЗАТВЕРДЖУЮ

А.О. Воронько
24.10.2023р.

ДОВІДКА

про практичне впровадження результатів дисертаційної роботи Зилевіч Максима Олеговича "Композиційні моделі телекомунікаційних систем в суб'єкто-об'єктному середовищі програмування" на здобуття наукового ступеня доктора філософії за спеціальністю 172 – Телекомунікації та радіотехніка

Програма як продукт повинна базуватись на технологічних засадах теорії інформації та процесів, оскільки результативність визначається безпосередньо суб'єктом виконання та оцінюється за кінцевим результатом. Практика свідчить про засилля свободи творчості розвиває програмування як мистецтво, що спричиняє надмірне спрощення розуміння програмування і зміщенні уваги на результат творчого процесу, а не на його шлях досягнення, що унеможливлює дослідження продуктивних засад процесу. Оскільки основоположні властивості програм формуються на початкових етапах, то така спрощеність ускладнює формування технологічних основ програмування. Дисертаційна робота Зилевіча М.О. присвячена технологізації процесів розв'язання сучасних задач в людино-машинних, зокрема, телекомунікаційних системах, методом композитологічного уподібнення.

Результати наукової роботи Зилевіч М.О. застосовано для розвитку концепто-монадної парадигми програмування. Розвинуто спосіб застосування підходу оракульного телеконцептування для предметного збагачення суб'єкто-об'єктного середовища програмування. Реалізація такого методу на практиці сприяє уніфікації процесу розробки програмно-апаратного продукту, тим самим оптимізує та реально об'єктивізує вплив активної ролі суб'єкта через механізм оракульних телекомунікацій. Запропоновано дослідну реалізацію суб'єкто-об'єктного середовища програмування, що підтримує розробку програмно-апаратного забезпечення як предметного замикання у відповідному середовищі. Умови такого замикання задаються у дескриптивному середовищі композиційних термів. Синтаксичне оформлення рішення здійснюється Verilog-дескриптором, апаратна реалізації реалізована на базі апаратної платформи FPGA із використанням САПР "Quartus".

Отримані результати використано під час виконання науково-дослідної роботи "Композитологічні засади технологічних систем програмування" (№0122U001568). В межах цієї роботи також розроблено практичні рекомендації щодо технічних вимог під час побудови суб'єкто-об'єктного середовища на базі апаратної платформи FPGA.

Науковий керівник Л.В. Редько

Зилевіч М.О.

ДОДАТОК В. АКТ ВПРОВАДЖЕННЯ

УКРАЇНА

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
(КПІ ім. Ігоря Сікорського)

пр-т Берестейський, 37, м. Київ, 03056, тел. (044) 204 82 82 тел. (044) 204 94 94
http://www.kpi.ua e-mail: mail@kpi.ua ЄДРПОУ 02070921

16.11.2023 № 2220/

ЗАТВЕРДЖУЮ
Проректор з навчальної роботи
КПІ ім. Ігоря Сікорського
к.ф.н., проф.,
Анатолій МЕЛЬНИЧЕНКО

АКТ ВПРОВАДЖЕННЯ

результатів дисертаційної роботи
Зилевіча Максима Олеговича
у навчальний процес Національного технічного університету України "Київський політехнічний інститут імені Ігоря Сікорського"

Комісія у складі:

Голова – заступник декана з наукової роботи факультету електроніки, к.т.н., доцент Попович П.В.

Члени комісії – завідувач кафедри конструювання електронно-обчислювальної апаратури д.т.н., професор Лисенко О.М.; професор кафедри конструювання електронно-обчислювальної апаратури д.ф.-м.н., професор Редько І.В.; завідувач лабораторії кафедри конструювання електронно-обчислювальної апаратури к.т.н. Іваннік Г.В.

Актом засвідчують, що результати дисертаційного дослідження здобувача кафедри конструювання електронно-обчислювальної апаратури Зилевіч М.О. використані як матеріали при підготовці та викладанні курсу лекційних і практичних занять з дисципліни "Системне програмування та керування базами даних в телекомунікаціях" другого (магістерського) рівня вищої освіти спеціальності 172 «Електронні комунікації та радіотехніка» освітньо-професійної програми «Інформаційно-обчислювальні засоби радіоелектронних систем».

Голова комісії: к.т.н, доц. _____ Павло ПОПОВИЧ

Члени комісії: д.т.н., проф. _____ Олександр ЛИСЕНКО

д.ф.-м.н., проф. _____ Ігор РЕДЬКО

к.т.н. _____ Геннадій ІВАННІК