

**MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
TARAS SHEVCHENKO NATIONAL UNIVERSITY OF KYIV**

**A. Yu. Doroshenko
Ie. V. Ivanov
M. S. Nikitchenko
O. A. Yatsenko
K. A. Zhereb**

FORMAL PROGRAM DEVELOPMENT METHODS

A textbook

Kyiv, 2021

Навчальне видання

ДОРОШЕНКО Анатолій Юхимович
ІВАНОВ Євген В'ячеславович
НІКІТЧЕНКО Микола Степанович
та ін.

ФОРМАЛЬНІ МЕТОДИ РОЗРОБКИ ПРОГРАМ

Навчальний посібник
(Англ. мовою)

Друкується за авторською редакцією

Технічний редактор *Л. П. Шевченко*

Оригінал-макет виготовлено ВПЦ "Київський університет"



UDC

004.42(075.8)

Authors:

A. Yu. Doroshenko, Ie. V. Ivanov, M. S. Nikitchenko,
O. A. Yatsenko, K. A. Zhereb

Reviewers: Dr.Sc., Prof., O.I. Rolik, Dr.Sc., Prof., V. M.
Tereshchenko, Ph.D., Associate Prof., T. V. Panchenko

Recommended for publication by academic council
of the information technologies faculty

Approved by scientific and methodological council by
Taras Shevchenko National University of Kyiv

FORMAL PROGRAM DEVELOPMENT METHODS : a text-
book / A. Yu. Doroshenko, Ie. V. Ivanov, M. S. Nikitchenko, O. A.
Yatsenko, K. A. Zhereb. – Kyiv: Publishing house of Taras
Shevchenko National University of Kyiv, 2021.

Theoretical basis of formal specification of sequential and parallel programs based on algorithm algebra and rewriting rules technique is outlined. In the context of development of V. M. Glushkov's ideas on formalization of programming languages, formal program specification methods are applied for solving software design and generation problems. Formalization and verification of programs are studied. The book is for students of software engineering specialty, graduate students and teachers of computer training faculties of higher education institutions and also for specialists that develop applied algorithms and software in different areas.

© Doroshenko A. Yu., Ivanov Ie. V.,
Nikitchenko M. S., Yatsenko O. A., Zhereb K. A. 2021

© Taras Shevchenko National University of Kyiv,
Publishing house of Taras Shevchenko National University of Kyiv,
2021

CONTENTS

LIST OF ABBREVIATIONS AND NOTATIONS.....	7
INTRODUCTION.....	9
PART I. THEORETICAL BASIS OF FORMAL PROGRAM SPECIFICATION METHODS.....	12
Chapter 1. Basic programming paradigms.....	13
1.1. Imperative programming	13
1.2. Functional and logic programming	16
1.3. Theoretical programming	19
1.4. Parallel programming.....	35
1.5. Object-oriented and component-oriented programming....	49
Control questions	53
Chapter 2. Algebras and specifications of algorithms	55
2.1. Dijkstra algebra	56
2.2. Algebra of flowgraphs	63
2.3. Glushkov system of algorithmic algebras and its modifications.....	67
2.4. Algebra of algorithmics	76
Control questions and exercises.....	81
Chapter 3. Algebraic programming based on rewriting rules	83
3.1. Aspects of rewriting: definitions and examples	83
3.2. Term rewriting	93
3.3. Existing software implementations	104
3.4. Applications of rewriting rules technique for working with program code	110
Control questions and exercises.....	121

Chapter 4. Generalization of Glushkov algorithmic algebra systems to the case of programs on hierarchical data..... 124

4.1. Different types of nominative data	124
4.2. Representation of data structures by nominative data	128
4.3. Algebras of nominative data.....	132
4.4. Operational semantics of operations on nominative data	141
4.5. Compositions of functions and predicates over nominative data.....	163
4.6. Generalized systems of Glushkov algorithmic algebras ..	167
4.7. Stability and monotonicity of programs over nominative data.....	169
Questions and exercises	181

PART II. APPLICATION OF FORMAL PROGRAM SPECIFICATION METHODS FOR SOFTWARE DEVELOPMENT AND VERIFICATION 182

Chapter 5. Formal design of algorithms and programs 183

5.1. Metarules of algorithm specification design.....	183
5.2. Algorithmic language SAA/1	188
5.3. Algebra-grammatical models for generation of algorithm specifications.....	192
5.4. Algebra-dynamic models of parallel programs	206
Control questions and exercises.....	221

Chapter 6. Software tools for design and synthesis of programs 224

6.1. The architecture of the integrated toolkit	225
6.2. Dialogue design of algorithm schemes	229
6.3. Generation of algorithms and programs	238
6.4. The rewriting rules system.....	251
Control questions and exercises.....	262

Chapter 7. Automated development of efficient parallel programs	265
7.1. Design and parallelization of programs for multicore processors.....	265
7.2. Transformation of coordination constructs in parallel programs.....	272
7.3. Development of parallel programs for graphics processing units	280
7.4. Formalization and verification of parallel programs using a proof assistant	288
Exercises	311
 REFERENCES.....	 313
 APPENDIX A. Models and source codes of examples.....	 332
A.1. Finding the quantity of prime numbers	332
A.2. Simulation of one-dimensional Brownian motion	334
A.3. Parallel summation of elements of a numeric vector	342
 APPENDIX B. The text of the formal proof of a theorem in Mizar language.....	 345

LIST OF ABBREVIATIONS AND NOTATIONS

AA — algebra of algorithmics;
ADT — abstract data type;
AHS — algebra of hyperschemes;
AP — address programming;
API — application programming interface;
AlgA — algebraic algorithmics;
ARS — abstract rewriting system;
CPU — central processing unit;
CUDA (Compute Unified Device Architecture) — a parallel computing platform and application programming interface (API) model created by Nvidia which allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general-purpose processing;
DA — Dijkstra algebra;
DSC-constructor — the dialogue constructor of syntactically correct programs (the component of the IDS toolkit);
DSC-method — the method of dialogue designing of syntactically correct programs;
DSL — domain-specific language;
GA — Glushkov algebra (Glushkov system of algorithmic algebras, SAA);
GPU — graphics processing unit;
IDS — an integrated toolkit for design and synthesis of programs;
IS — a set of states (an information set) of the operational automaton of the abstract automaton model of a computer;
KA — Kaluzhnin algebra (algebra of flowgraphs);
CNP — composition-nominative programming;
CP — composition programming;
MAS — many-sorted algebraic system;
Mizar — computer system for verification of mathematical proofs;
MPI — message passing interface;

$\bar{P}$ — a set of states (an information set) of the operational automaton of the abstract automaton model of the parameter-driven generator of texts;

PCS — parallel computing system;

PRS — parallel regular scheme;

RHS — regular hyperscheme;

RS — regular scheme;

SAA — Glushkov system of algorithmic algebras;

SAA-M — modified system of algorithmic algebras;

SDG — structured design grammar;

TermWare — symbolic processing system based on term rewriting paradigm;

UML — unified modeling language.

INTRODUCTION

The important challenge of modern programming is its mathematization, the development of formalized languages for designing algorithms and programs as well as their abstract models. The facilities for design, analysis, and implementation of algorithms are especially actual in connection with essential processes of computerization and automation of society activities. Such facilities are developed within the framework of algorithmics and algebraic programming, which are the directions of Ukrainian algebraic-cybernetic school originating from fundamental works of Academician V. M. Glushkov. The algebraic programming [88, 99] is based on term rewriting theory and provides the formal description of program design, algebraic transformations and proving of mathematical theorems. In algebraic programming, the terms represent the data, and the term rewriting systems, which are expressed with the help of systems of equalities, represent the computing algorithms. The algorithmics (which is also called the algebra of algorithmics) [5, 159] is the direction based on Glushkov system of algorithmic algebras (SAA) and is focused on solving the problems of formalization, substantiation of correctness and improvement of algorithms by selected criteria. Composition-nominative programming [127] develops V. M. Glushkov's ideas on programming language formalization based on the logic-algebraic approach, representing data as nominative data, programs as functions over nominative data, facilities for program development as operations (compositions) over such functions, and program analysis formalisms as composition-nominative program logics.

The problem of the development of efficient models and methods of parallel computing is very important within the framework algorithmics and algebraic programming since parallel computing on multiprocessor systems is the main source providing highly-productive computation at solving complex scientific and technical problems. The problem of designing efficient programs for multiprocessor architectures can be successfully solved by specialization to subject domains and deep coverage of all stages of program life cycle with the use of tools for automated designing and programming,

beginning from writing primary specifications to generating executable code. First of all, the basis for such automation is the high-level formalization of designing parallel programs and automation of formal transformations of programs with the purpose of optimization of their performance. The optimization of programs refers to the achievement of the required quality characteristics of a program at a given set of criteria, such as used memory, operating speed, loading of equipment, etc.

In this book, the theoretical basis of formal specification of sequential and parallel programs based on algorithm algebra and term rewriting technique is given. The developed methods and tools for software design and generation and their application for solving applied tasks are considered. Formalization and verification of programs are studied.

The aim of the first part of the book is to consider general theoretical concepts underlying formal models, methods and software tools for automated software design. The main concepts associated with formalized description of algorithmic processes by means of schemes in various algebras of algorithms are given: Dijkstra algebra associated with structured programming, Kaluzhnin algebra intended for graphical description of non-structured schemes of algorithms and programs, and also Glushkov algebra for description of structured and parallel schemes including the facilities of computation process prediction. The main definitions associated with algebraic programming based on rewriting rules and an overview of term rewriting systems are presented. In the context of composition-nominative programming, the generalization of Glushkov system of algorithm algebras for the case of programs over hierarchical data is outlined.

The second part is devoted to the consideration of formal methods and software tools of programming automation. We consider the metarules of scheme design and SAA/1 language intended for multilevel algorithm specification in a natural language form. The algebra-grammatical models for generation of algorithm schemes and methods of parameter-driven algorithm generation on the basis of higher-level schemes are described. Algorithmic algebra and rewriting rules technique are applied for developing algebra-dynamic

models of parallel programs intended for execution on graphics processing units. The models and methods form the basis of the developed software tools for automated software design, synthesis and transformation. The tools are applied for designing sequential and parallel programs in various subject domains. Formalization and verification of parallel programs with the help of the computer system for theorem-proof verification are considered.

PART I. THEORETICAL BASIS OF FORMAL PROGRAM SPECIFICATION METHODS

In this part, an overview of fundamental concepts and methods underlying the formal specification of sequential and parallel programs is given.

In Chapter 1, basic programming paradigms (imperative, functional, logic, theoretical, parallel and object-oriented) underlying the developed models, methods and software tools for automated program design and synthesis are considered.

Chapter 2 is devoted to algebraic facilities intended for formalized description of algorithmic processes with the help of structured and non-structured high-level models of algorithms and programs called schemes. The following algebras are considered: Dijkstra algebra associated with structured programming technology, Kaluzhnin algebra for a graphical description of non-structured algorithm schemes, and Glushkov algebra for description of structured schemes, including the facilities for computation process prediction and parallel algorithm design.

In Chapter 3, the concepts associated with programming based on rewriting rules and an overview of corresponding software tools are given.

In Chapter 4, the generalization of Glushkov system of algorithmic algebra for the case of programs over hierarchical data is considered.

Chapter 1

Basic programming paradigms

Programming paradigm is a generalized conceptual scheme, method of programming [89]. Together with a language formalizing it, a paradigm forms a programming style. It is the tool for grammatical description of facts, events, phenomena, and processes that may not exist simultaneously, but intuitively combine in a common concept. Thus, a paradigm represents and determines how a programmer sees a program implementation. For example, in object-oriented programming, a programmer examines the program as a set of interacting objects, while in the functional programming a program is given as a chain of function evaluations. A special role is played by paradigms of an application domain, for which program composition is developed.

In the following subsections, the following programming paradigms are considered: imperative, functional, logic, theoretical, parallel, object-oriented and component-oriented.

1.1. Imperative programming

Imperative programming paradigm represents the computing process by means of description of control flow of a program, i.e. as a sequence of separate commands that a computer must execute. Every command is an instruction that changes the state of the program. The imperative programming is based on Turing-Post machine, an abstract computing device which executes the sequence of program instructions (commands), and, thus, passes from one state to another. There is a concept of a “current execution step” and a “current status” changing over time. The imperative paradigm has developed on the basis of low-level languages (machine codes, Assembler) and is based on von Neumann architecture. Machine code is the most low-level example of implementation of this paradigm: the state of the program is determined by memory content and commands are determined by machine code instructions. As this paradigm is natural for human understanding and directly implemented at the hardware level, the majority of programming languages follow it. The imperative programming is the opposite of declarative pro-

gramming which describes what it is necessary to do, while the imperative specifies how it is to be accomplished. The examples of imperative programming styles are non-structured, structured, procedure-oriented, assembling and modular.

In *non-structured* programming, the whole code of a program is presented by a single continuous block. Passing to necessary sections of the program is performed by means of conditional and switch control statements (usually, goto or jump) without limitations. The examples of such languages are script languages, Fortran, Basic, Assembler.

Procedure-oriented or function-oriented programming sometimes is mentioned as a synonym for the imperative programming (a procedure state is determined by a value of a set of its variables at a point in time). Such programming is based on the concept of a call of a procedure (subroutine, function, method). Any procedure can be called at any point in the process of program execution. The examples of procedure-oriented languages are Algol, Ada, Basic, C, Cobol, Fortran, Pascal, PL/1, Matlab. Unlike the non-structured paradigm, the procedure-oriented facilitates software reuse, comprehensibility, and maintainability.

Assembling programming implies that a program is built by reuse of already existing fragments. This programming solves the problem of reusable and rapid application of already prepared details in the process of program development. The assembling can be done manually, or set by a certain assembling language, or extracted semi-automatically from a problem specific [158]. The assembling programming is closely linked to the method of reusing programming code, both source, and binary.

There are several varieties of assembling programming approaches that are largely determined by basic methodology:

- *modular*. This approach was historically the first and was based on procedures and functions of structured imperative programming methodology;
- *object-oriented*. The approach is based on object-oriented programming methodology and implies the sharing of class libraries as source code or packing of classes into a dynamically linked library;

- *component-oriented*. Basic ideas of the approach are sharing of classes in a binary form and providing access to class methods through certain interfaces, that allow avoiding the problem of incompatibility of compilers and provides the change of versions of classes without recompiling of applications that use them. There are certain technological approaches that support component-oriented assembling programming: COM (DCOM, COM+), CORBA, .NET;

- *aspect-oriented*. The concept of a component, in this case, is complemented by a concept of aspect-variant of implementation of procedures critical in terms of efficiency. Aspect-oriented assembling programming consists in a compilation of full-function applications from multi-aspect components that encapsulate different implementation variants.

Structured programming is a methodology of software development proposed in the 1970s by E. W. Dijkstra and further developed and complemented by N. Wirth [170]. In accordance with this methodology, any program is a structure built from the following three types of basic constructs:

- *sequential execution*, a single execution of operations in the order written in a program text;
- *branching*, one-time execution of one of two or more operations depending on the value of a defined condition;
- *loop*, a multi-pass execution of the same operation until some defined condition (loop termination condition) is met.

In a program, the basic constructs can be nested inside each other in an arbitrary manner, but no other facilities for managing a sequence of operation execution are implied. Reusable sections or logically coherent sections of a program can be arranged as so-called subroutines (procedures or functions). In this case, the mentioned program section is replaced with a subroutine call. In carrying out such instruction, the subroutine is executed, after which the execution of a program is continued from the instruction following the subroutine call. The examples of languages belonging to this paradigm are all modern imperative programming languages, in particular, Pascal, Ada, Fortran, Cobol, Basic, C.

In [159], structured and non-structured approaches were formalized and corresponding Dijkstra and Yanov algorithm algebras were developed. The algebras were applied for a description of programs by means of high-level schemes.

1.2. Functional and logic programming

Functional and logic programming belong to the declarative programming paradigm, under which a program describes what result must be received instead of a description of a sequence of its obtaining. This programming paradigm differs from the above considered imperative programming that requires a detailed description of an algorithm of obtaining results.

Functional programming is the paradigm where the execution of a program is the calculation of some expression that describes application of functions (in a mathematical sense) to input data [75]. Unlike the traditional programming approach (imperative programming), where execution of a program is considered as a sequential transition of states in computer memory (i.e., change of variables values), there is no concept of a variable and an assignment in functional programming, a function does not have the obvious internal state, but operates only on data. As a result, side effects are absent, a program becomes simpler to debug, and also implies more natural parallelization for multicore processors. In addition, the use of higher-order functions allows using functional abstraction to a greater extent, and automatic inference of types makes program text substantially more compact.

Basic features of functional languages are the following: conciseness and simplicity, strict typing, modularity, functions are values, cleanness (absence of side effects), call-by-need (“lazy”) evaluation.

Functional languages are classified as follows:

- pure, that use only a functional paradigm (for example, Haskell);
- hybrid, that have characteristics of both functional and imperative languages. Prime examples are Scala and Nemerle, which unite features of object-oriented and functional languages;
- non-strict, that support lazy evaluation, i.e. the arguments

of a function are calculated only when they are really necessary at the calculation of a function. The examples of unstrict languages are Haskell, F#, Gofer, Miranda;

- strict, that does not support lazy calculations (for example, Standard ML).

Some functional languages (Scala, Clojure, F#, Nemerle, SML.NET) are implemented above virtual machines (JVM, .NET), i.e. applications written in these languages can work in the runtime environment (JRE, CLR) and use built-in classes.

Below, a brief description of some functional programming languages is given.

LISP (“LIST Processor”) is considered to be the first functional programming language, untyped. Linked lists are one of Lisp’s major data structures, and Lisp source code is made of lists. Thus, Lisp programs can manipulate source code as a data structure, giving rise to the macro systems that allow programmers to create new syntax or new domain-specific languages embedded in Lisp. Lisp has changed since its early days, and many dialects have existed over its history. Today, the best-known general-purpose Lisp dialects are Clojure, Common Lisp, and Scheme.

ML (“Meta Language”) is a general-purpose functional programming language which has roots in Lisp, and has been characterized as “Lisp with types”. Features of ML include a call-by-value evaluation strategy, first-class functions, automatic memory management through garbage collection, parametric polymorphism, static typing, type inference, algebraic data types, pattern matching, and exception handling. Its types and pattern matching make it well-suited and commonly used to operate on other formal languages, such as in compiler writing, automated theorem proving, and formal verification. Today there are several languages in the ML family; the three most prominent are Standard ML (SML), OCaml and F#.

Miranda is a purely functional programming language with lazy evaluation, polymorphic strong typing, and a powerful module system. It had a strong influence on the subsequent development of the field, influencing, in particular, the design of Haskell, to which it has many similarities.

Haskell is one of the most widespread non-strict languages.

It has a strong, static type system. The simplified dialect of Haskell is Gofer (GOod For Equational Reasoning), intended for learning functional programming.

Erlang is a functional programming language with dynamic typing, intended for developing distributed computing systems. This language contains facilities for generating parallel processes and their interaction by means of asynchronous messages. The program is translated to byte code that is executed by a virtual machine.

F# is a multi-paradigm programming language intended for executing on Microsoft .NET platform. It combines the expressiveness of functional languages, such as OCaml and Haskell, with capabilities and objective model of .NET.

Scala a multi-paradigm programming language designed for simple and rapid development of component applications for Java and .NET platforms and combines capabilities of functional and object-oriented programming.

In *logic programming*, a program consists of a theory and a proposition to be proved [31]. A theory is defined by means of axioms and inference rules (implications). A proposition statement that needs to be proved is entered into the program as a goal. The execution of a program consists in searching for proof of the proposition in the knowledge base which is a set of facts and rules. While in the functional programming functions are one-directed, i.e. they get arguments and return a result, in the logical programming a difference between input and output is conditional. It is possible to specify a desirable output and get an input that it will provide. Another important difference is nondeterminism of logical languages. A result is not necessarily determined explicitly. Thus, as an arbitrary proposition can be proved variously, a system implementing logic programming language (for example, Prolog) consistently suggests to renew an attempt to prove a goal in a different way. For proving propositions, unification (an algorithmic process of solving equations between symbolic expressions) and a resolution technique are used. In logic programming, as well as in functional, the low-level details of methods of calculations and sequences of elementary executions are unknown for a programmer. The greater part of responsibility for their efficiency depends upon the translator of the used programming

language. At present, there are several implementations of Prolog. Generally, a code generated by a translator of this language can be compared in terms of efficiency with code of a corresponding program in imperative language.

The first logic programming language was Planner, which had a possibility to automatically infer a result from data and a set of rules for searching variants called a plan. Planner was used in order to bring down requirements to hardware resources (by means of backtracking method) and provide the possibility to infer facts without active use of a stack. Further, Prolog was developed, that did not require a variants searching plan, and was a simplification of Planner in this sense. The logic programming languages originating from Planner are QA-4, Popler, Conniver, QLISP. Planner was a basis for several alternative logic programming languages that do not use backtracking method, for example, Ether. Derivatives from Prolog are Mercury, Visual Prolog, Oz, and Fril languages.

1.3. Theoretical programming

This subsection considers theoretical programming methods based on the concept of algebra. *Theoretical programming* [175] includes formal methods based on program specification and methods grounded on mathematical disciplines (logic and algebra) and provide the mathematical method of analysis and comprehension of problems of a subject domain, and also development of programs with mathematical symbolics, the correctness of which is to be proved in order to get necessary results on a computer. Further, the directions of algebraic programming being developed within the Ukrainian algebraic-cybernetic school are considered:

- algebraic and insertion programming [81, 88, 97];
- algebra-algorithmic programming [5, 41, 51, 159];
- composition-nominative programming [16, 96, 117–125, 127].

1.3.1. Algebraic and insertion programming. Algebraic programming is a special form of programming activity in which programs are constructed in terms of algebraic objects and their transformations. Objects of a subject domain and reasoning about these objects are represented by algebraic expressions (e.g. terms,

predicate formulas, algorithm schemes) in many-sorted algebra [5, 146]. The transformation of expressions is provided by application of equalities or rewriting rules. A number of systems are known, which support various forms of algebraic programming, such as Casl [25], Maude [105, 106, 169], APS [8, 88, 99, 100], etc. In particular, Casl (Common Algebraic Specification Language) is an expressive language for formal specification of functional requirements and modular design of software. It was designed to supersede many existing algebraic specification languages and provide a standard. Casl consists of several layers, including basic (unstructured) specifications, structured specifications and architectural specifications (the latter are used to specify the structure of implementations).

Algebraic programming is based on term rewriting theory [88] (see also Chapter 3). The terms represent data and systems of rewriting rules (expressed as systems of equalities) represent computation algorithms. The elementary step of computation includes pattern matching, verification of terms and substitution. Order of selection of rewriting rules and subterms of a given term for matching with left parts of equalities is defined by a rewriting strategy [38]. Strictly speaking, the strategy determines the result of a computation, a term. Rewriting strategy can be described in a lower-level programming paradigm, for example, procedural or functional, which results in the necessity of integration of the paradigms. The idea of integration of paradigms (procedural, functional, algebraic and logical) found the embodiment in the Algebraic Programming System (APS) [8, 88, 99, 100], which uses specialized data structures, the graph terms, for representation of data and knowledge about subject domains.

The concepts of algebraic programming underlie the insertion programming [81, 82] based on agent behavior models, transition systems and bisimulation equivalence notion [97]. The insertion programming considers a program as an algebraically defined transformation of a set of states of an information environment, where an active information environment, which has an observed behavior, is considered instead of the passive environment (a memory). The basis of the insertion programming is the model of behavior of agents in environments based on concepts of a transition system (a basic

standard in the behavioral theory of interactive processes) and the relation of bisimulation equivalence of agents with respect to an environment (the state of an agent is identical with its behavior). Unlike the agent programming, which pays more attention to the problems of intellectualization of agents, insertion programming embraces the behavioral aspects of agents.

The insertion programming allows defining behaviors of systems and their equivalence. In general, the transition systems can be components, programs and their specifications, objects that cooperate with each other and with the environment of their existence. The evolution of such a system is described by a history of system functioning, which can be finite or infinite and can include a survey part as a sequence of actions and a hidden part as a sequence of states. The history of functioning contains a successful completion of computation in the environment of a transition system, a deadlock state, when each of parallel parts of the system is in a state of waiting for events, and an uncertain state that arises up, for example, at execution of an algorithm with infinite loops.

The extension of the concept of a transition system is a set of final states with successful completion of functioning of the system and without uncertain states. The main invariant of a state of a transition system is a system behavior, which can be defined by expressions of the behavior algebra $F(A)$ on the set of operations of the algebra A . The operations include the prefixing operation $a \cdot u$, which defines the behavior u on the operation a , and the operation of the nondeterministic choice $u + v$ of one of the behaviors u and v . The finite behavior is specified by the constants Δ , $\perp$, 0 , which denote a state of successful completion, an uncertain and a deadlock state accordingly.

The transition systems are called *bisimulation equivalent*, if each state of one system is equivalent to a state of the other system. The new operations, which are used for construction of programs of agents, are defined on the set of behaviors. These operations are the sequential composition $(u;v)$ and the parallel composition $u \parallel v$.

The environment E is defined as an agent in the algebra of actions A and the insertion function $\text{Ins}(e, u) = e[u]$, where e is a

behavior of the environment, u is a behavior of the agent, which is inserted into this environment in a given state. The agent algebra is the parameter of the environment. The value of the insertion function is the new state of the environment.

Behaviors of agents are characterized by the state within a bisimulation and possibly a weak equivalence. Every agent is considered as a transition system with actions defining the nondeterministic choice and the sequential composition, i.e. primitive and compound actions. The interaction of agents can be of two types. The first type is expressed through the parallel composition of agents over the same set of actions. The other type is expressed through the function of agent insertion into some environment; the result of the transformation is a new environment. The development of new programming methods with the introduction of agents and environments allows interpreting elements of complex programs as independently interacting objects.

1.3.2. Algebra-algorithmic programming. Algebra of algorithmics (AA) [1, 5, 159] is a direction within the Ukrainian algebraic-cybernetic school rising from the fundamental works of Academician V. M. Glushkov [62, 64, 65] associated with the development of system of algorithmic algebras and the toolkit for automated program synthesis called MULTIPROCESSIST [112]. The specific feature of AA is a formalization of processes of design and synthesis of algorithms and programs. These objects are designed in terms of regular schemes, which are the representations in SAA. The development of facilities of equivalent transformations provided the possibility of improvement of objects being designed according to selected criteria (for example, used memory, execution time). The theory of clones [1, 5], which formalizes basic programming paradigms and aspects of forming of different subject domains, was proposed. The tools for automatization of programming were also developed [5, 40, 42–47, 49, 174].

The algebra of algorithmics is close to the directions of algebraic algorithmics [114] and intentional programming [33]. The algebraic algorithmics (AlgA) belongs to methods of top-down programming, whereas intentional programming belongs to bottom-up programming. AlgA is the formalized approach to description of

methods of processing mathematical (algebraic) objects. It combines different algebras and data processing algorithms, which are used for proving basic theorems in corresponding algebras. The purpose of intentional programming consists in creating various subject domains and their integration with the use of abstractions, biology, and ecology of programming. The abstraction is the analytical representation of knowledge related to a chosen subject domain (for example, formulas in corresponding algebras). The biology of programming consists in the availability of “genetic” connections between close subject domains. The ecology represents means of automation of construction of different subject domains and their integration.

Algebra of algorithmics serves to the solution of the same problems, as two mentioned approaches, i.e. the formalization of knowledge about subject domains with the help of algebraic means. The biological component of *AA* is reflected in the theory of clones, and the ecological one in a set of tools intended for automation of design and synthesis of software. However, in contrast to *AlgA* and intentional programming, *AA* uses three interdependent forms of algorithmic knowledge representation:

- analytical (in the form of an algebraic formula, which is called a regular scheme);
- natural-linguistic (the text in the conventional language of human communication);
- visual (flowgraphs).

Example 1.1. As illustration of the above-mentioned representation forms, consider the fragment of the bubble sort algorithm [5]. The analytical form of this fragment is specified as the following regular scheme:

$$ALT = ([l > r | Y_1] \text{Transp}(l, r | Y_1) * R(Y_1), R(Y_1)).$$

The main compound operator of the given scheme is called *ALT* and is a ternary operation of branching of the form “if then else”, which is denoted as $([u] A, B)$. The logical variable u of this operation is replaced by the predicate $l > r | Y_1$ and operator variables A and B are substituted by the operators $\text{Transp}(l, r | Y_1) * R(Y_1)$ and

$R(Y_1)$, accordingly. The mentioned predicate and operators are defined on the marked array M of data being processed:

$$M : H \ Y_1 \ a_1 \ \dots \ a_n \ K,$$

where H , K are markers fixing the beginning and the end of the array M , correspondingly; Y_1 is a pointer, which can move over the array with the help of the shift operator; l and r are the elements directly to the left and to the right of the pointer Y_1 , accordingly; $Transp(l, r | Y_1)$ is the operator of the transposition of the elements l and r ; $R(Y_1)$ is the operator shifting the pointer Y_1 over the array M by one element to the right.

The visual form of this fragment is given as a flowgraph in Fig. 1.1.

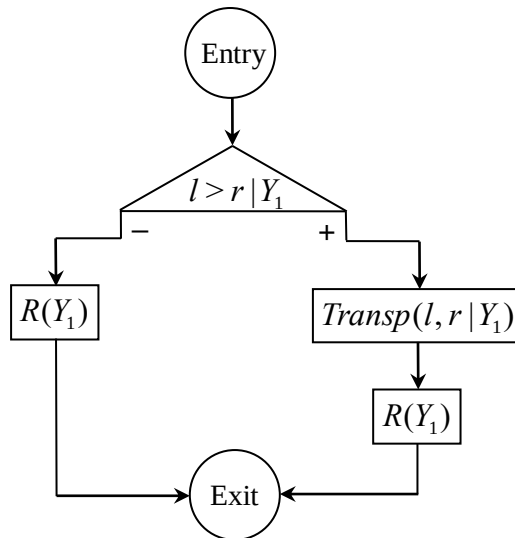


Fig. 1.1. The flowgraph of the compound operator ALT

The corresponding textual representation of the above fragment is the following:


```

ALT =
=== IF ' $l > r$  at  $Y(1)$  in  $(M)$ '
    THEN
        "Transpose  $l, r$  at  $Y(1)$  in  $(M)$ "
    THEN
        "Shift  $Y(1)$  by  $(1)$  in  $(M)$  to the right"
    ELSE
        "Shift  $Y(1)$  by  $(1)$  in  $(M)$  to the right"
    END IF

```

While noting the closeness of *AA* to the directions of AlgA and intentional programming, we would like to underline the conceptual integrity of the paradigm of *AA* (see Fig. 1.2), which largely determines the advantages of this paradigm in comparison with the mentioned approaches.

The main peculiar features of *AA* are the following:

- the system of algorithmic algebras are focused on the formalized design of objects (abstract data types) and algorithms (programs) in terms of high-level models (schemes), which can be represented in three forms: formula, text, and flowgraph;
- the schemes of algorithms can be modified by using the metarules of scheme design: a convolution (abstracting), an involution (detailed elaboration), a reorientation (convolution and involution) and a transformation (a conversion of a scheme by means of application of equalities);
- the algebra-algorithmic approach is supported by the developed software tools [5] providing an automated design and synthesis of programs. The process of program development is based on the fixation of basic notions (predicates and operators) associated with a chosen subject domain and use of system of algorithmic algebras. The process of design of objects assumes the integration of the developed tools with other programming automation means (for example, UML and Rational Rose [161]);
- the facilities of algebra of algorithmics were applied for designing programs in subject domains of symbolic processing (sort,

search, language processing) [5, 159, 173], meteorological forecasting [11] and other;

- operations of SAA, algorithm schemes, basic notions of subject domains and their interpretations in a target programming language form a knowledge base.

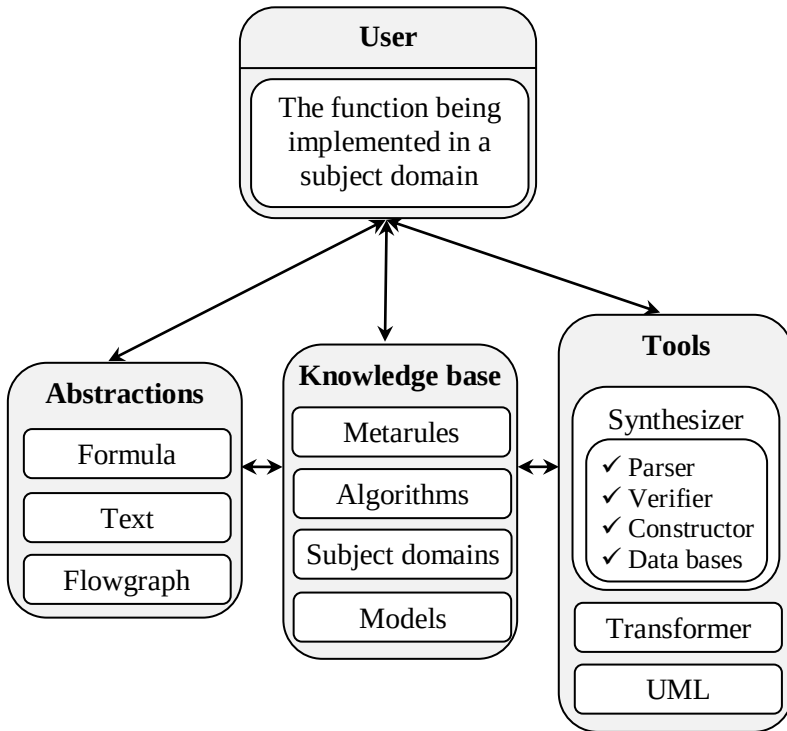


Fig. 1.2. The facilities of design and synthesis of programs used within the framework of the algebra of algorithmics

1.3.3. Composition-nominative programming. Composition-nominative programming (CNP) [96, 118–120, 122–125, 127] develops the ideas of V. M. Glushkov on formalization of programming language on the basis of the logic-algebraic approach. In CNP data are represented as nominative data, programs are represented as

functions on nominative data, means of program construction are operations (compositions) over such functions, and programs are studied using composition-nominative program logics.

CNP continues the ideas of address programming (AP) [66] and composition programming (CP) [144]. As the names suggest, address programming focuses on the study of programs over addressed data, and composition programming focuses on the study of program construction operations. Nominative data are generalizations of addressed data [127]. We focus on the consideration of the compositional and nominative aspects in semantics sense.

1.3.3.1. Main principles of CNP. The principles of CNP are described in [119, 127]. Its purpose is development of AP and CP in several directions. For all three approaches, the main objects of study are programs, but AP and CP are mainly concerned with formalization of structural, internal aspects of programs (most importantly, semantic and syntactic aspects), while CNP considers programs in a wider external context, related to the special activity of the subject. Such extension of context also requires an appropriate enrichment of the levels of consideration of the concepts under study. More specifically, the following levels are considered.

- general methodological (philosophical);
- scientific (professional);
- mathematical (formal).

These levels correspond to 3 classes of notions:

- categories (philosophical);
- scientific concepts that define the features of programming;
- formal notions that are mathematical formalizations of scientific concepts.

Categories are not the main subject of CNP, however, research in this field is impossible without understanding the system of categories. Examples of categories are: *abstract-concrete*, *quality-quantity-measure*, *element-part-whole*, *form-content*, *phenomenon-essence*, *singular-special-general* and many others. Scientific concepts often serve both as a certain development and a certain limitation (projection) of categories. Such a link between categories and notions is quite important, given the integrating aspect of CNP.

Principles of CNP are formulated in accordance with the levels of consideration. General methodological principles characterize the general laws of development.

Epistemological principle: clarification of the basic concepts of programming is carried out in accordance with the general laws (principles) of epistemology, the application of which is programmatic.

Epistemology should, in particular, provide general laws of refinement of the concepts of specific subject areas. CNP limits attention to principles relevant to programming.

The principle of universal interconnection: there are universal interconnection and interdependence of objects and phenomena of the world.

As a consequence, each subject has many aspects. They are considered in accordance with *the principle of development from abstract to concrete* (from simple to complex, from lower level to higher level, from old to new): programming notions are clarified in the process of their development. This process begins with the most abstract representations that reflect the general properties of programming, then moves on to more specific representations in their unity, which reflect specific and specific properties, thus gradually revealing the notion of programming in their richness and interrelations.

Such a process mainly occurs according to the basic schemes of development, among which the central place belongs to the scheme of triadic development: thesis - antithesis - synthesis (*principle of triadic development*).

The choice of the direction of development is determined by *the principle of unity of theory and practice*: the theory and practice of programming must be considered in their unity and interdependence. This principle, in fact, affirms the “equality” and interdependence of theory and practice.

Among the general scientific principles, the main principle is the *systematicity principle*, which states that one has to study objects and phenomena comprehensively, in the unity of the whole and parts, taking into account various relationships between parts of the system and the external environment. This principle can be regarded as a

certain “projection” to the general scientific level of the categories of entity, in particular, the categories “part-whole”.

The next principle is the principle of *unity of the intensional and extensional aspects* [123]: scientific concepts must be presented in the unity of their intensional and extensional aspects. Here intensional aspects describe the meaning of the concept, an extensional describe its scope. These aspects can be considered as projections of the categories “(all) general — special — single”. The intensional aspect plays the primary role in relation to the extensional.

Finally, one of the descriptological principles of CP is accepted – the principle of the leading role of the *semantic aspect* in relation to the syntactic one. Such aspects can be seen as projections of the “*content-form*” categories.

Among the specific scientific principles, there are two approach-specific ones: the compositionality principle and the nominativity principle. *The principle of compositionality*, which is the main principle of CP, emphasizes the need of study of compositions as the means of program construction, nominativity principle affirms the importance of naming relations in program construction and description.

The mentioned principles determine the directions and features of explications of the basic concepts of programming in CNP. A three-level explication scheme is adopted, the middle level of which defines scientific concepts. For these concepts, their relationships with categories (upper level) are established, as well as relationships with their formalization (lower level of consideration).

The connection of categories and concepts of programming can be demonstrated in the following example. An initial category (thesis) is the category “subject”. Its antithesis is the category “purpose”, and synthesis is the category “means”. The projection of the categorical triad “*subject — goal — means*” to the scientific (professional) level gives the triad the concepts of “*user — problem — program*”. Further development of this triad with the help of the triads “*user — program — execution process*” and “*problem — program — programming process*” leads to development of basic concepts of programming theory and their relations in the pentad “*us-*

er — problem — program — execution process — programming process” [119, 127].

The next step is the development of the concepts of the above-mentioned pentad. Here one unfolds the concept of a program in its descriptive aspect, which is given by the following pentad: *data — function — function name — composition — descriptive* [119, 127].

These notions are connected by a number of relations which are not mentioned here. The pentads allow one to formulate a rich system of essential aspects of programs which consists of two classes [120]:

- *external aspects: adequacy, pragmatics, computability, origin;*
- *internal (structural) aspects: semantics, syntax, denotation.*

Further definitions refine these notions and create a hierarchy (ontology) of the basic notions of programming. The development of such a system leads to the possibility of their formalization. For this purpose, data structures, functions and compositions should be formalized first.

1.3.3.2. Data structures in CNP. Traditionally, the formalization of data is done on the basis of the set-theoretic platform. In particular, this kind of formalization is done in AP and CP. However, the set-theoretic platform has certain limitations and is not always adequate for representing data usage in modern programming [123]. For this reason, a special classification of data at different levels of abstraction is developed in CNP. This classification is called *3x3 data typology* [120]. This classification is the result of projections of the categories *whole-part* and *abstract-concrete*.

The first pair of categories can be developed in accordance with the triad *whole (W) — part (P) — hierarchy (H)*, and the second one can be developed in accordance with the triad *abstract (A) — concrete (C) — synthetic (S)*. Thus we get 9 types of data. They can be characterized as follows.

Top-level data (D.W) are considered as a whole (integral, unstructured objects), which do not change unless this is stated explicitly (*identity law*). At this level there are three subtypes: abstract

(D.W.A, “*black box*” data), concrete (D.W.C, “*white box*” data), synthetic (D.W.S, “*black or white box*” data).

On the second level (D.P) data have parts. Data of this level are called *collections*. Informally, collections can be described as follows:

- each part of a collection is a certain *whole*;
- the law of identity holds for such parts, i.e. parts *do not change* until this is stated explicitly
- the parts exist *separately* from each other
- the parts are *independent*; such parts are called *elements*,
- the parts are “*accessible*”, i.e. each part can be obtained for processing,
- one can talk about the *exhaustive* processing of all parts.

All these properties are informal, their formalizations will define different types of collections.

Collections with abstract elements (D.P.A) are called *pre-sets*, collections with concrete elements (D.P.C) are *sets*, collections with synthetic elements (D.P.S) are called *nominats* (from Latin *nomen*, i.e. “name”).

The data of the third level (D.H) are hierarchical data, which are classified as hierarchical pre-sets (D.H.A), hierarchical sets (D.H.C), hierarchical nominats — nominative data (D.H.S).

The most important types of data (for programming) are nominates, which are constructed using the $name \mapsto value$ relation. Here a *name* is considered on the concrete level (“white box”), and a *value* is considered on the abstract level and can be a “black box”. For nominats we use the notation of the form $[v_1 \mapsto d_1, \dots, v_n \mapsto d_n]$, where $v_1, \dots, v_n$ are names from V , and $d_1, \dots, d_n$ are values. In contrast to CP, names in this sequence can coincide, meaning that *multi-valued naming* is allowed. Moreover, in CNP not only finite nominats are considered. *Infinite nominats* are possible. Nominats (as synthetic objects) have a dual nature: on the one hand, they are collections, and on the other hand, they are functions, since the $name \mapsto value$ relations have a functional connotation. It can be argued that the functional nature of nominats is im-

portant for defining operations on them, so in CNP this property of nominats is used (the *principle of theoretic-functional formalization*).

The notion of a nominat is at the heart of the definition of the universe of nominative data $NomD(V, W)$, which is built on the basis of elements of a set of names V and a pre-set of basic values W . More specifically, elements of W , and also the empty nominat $[]$, are objects (nominative data) of rank 0. Nominative data of the rank $i+1$ are nominats of the form $[v_1 \mapsto d_1, \dots, v_n \mapsto d_n]$, where $v_1, \dots, v_n$ are names from V and $d_1, \dots, d_n$ ($n \geq 0$) are nominative data of the rank less than or equal to i ($i \geq 0$).

The typology of nominative data is based on the properties of the fundamental relation $name \mapsto value$, and arises naturally as a classification of the components of this relation:

- *values* are classified as simple (unstructured) and complex (structured),
- similarly, *names* are classified as simple (unstructured) and complex (structured),
- *names* and *values* can be independent (there is only direct naming), or dependent (indirect naming is allowed).

These three binary parameters (three axes) give eight types of nominative data. Such a classification is called the *typological cube of nominative data*. Representation in the form of a cube is convenient, because the edges of the cube correspond to intensional inclusions of types.

In the next sections, nominative data of different types and basic operations on them will be considered in more detail.

1.3.3.3. Types of functions in CNP. The above-mentioned classification of data induces the corresponding classification of functions. Classes of functions (programs) over data of the abstract level D.W.A are, intensionally, very poor. Richer classes are the classes of functions over data of the synthetic level D.W.S (Boolean level). They allow one to define different types of predicates. The corresponding classes of predicates and compositions of predicates are described in [117–121, 124, 126]. The richest classes are the classes of functions over data of the nominative level (D.P.S and D.H.S). For programming, the most interesting class is the class of

functions of the type $NomD(V, W) \rightarrow NomD(V, W)$. Below are some examples of such functions (operations):

- *naming* $\Rightarrow v$ with parameter $v \in V$:

$$\Rightarrow v(d) = [v \mapsto d] \text{ for each } d \in NomD(V, W),$$

- *denaming* $v \Rightarrow$ with parameter $v \in V$:

$$v \Rightarrow (d) = d', \text{ if } d(v) = d',$$

- the name checking function $v!$ with a name parameter $v \in V$:

$$v!(d) = \begin{cases} [], & \text{if } d(v) \text{ is defined,} \\ d, & \text{if } d(v) \text{ is not defined,} \end{cases}$$

- the multivalued selection function $\bowtie$ which chooses between the empty and input data:

$$\bowtie(d) \text{ is either } \emptyset, \text{ or } d.$$

1.3.3.4. Compositions in CNP. Compositions are classified in accordance with the classifications of data and functions. The most important compositions of functions are (see [119]):

- the binary multiplication composition $\bullet$ (sequential execution);
- ternary branching composition Δ (*if_then_else* operator);
- the binary composition of the *conditional iteration* $*$ (*while_do* loop);
- the binary *overlapping* composition ∇ (in order to compute $f \nabla g (d)$ one needs to compute $f(d)$ and “write over” the obtained nominative data the result of $g(d)$).

Each of these compositions corresponds to a certain level of abstraction in the 3×3 data typology. Namely, the composition of the multiplication is defined on the abstract level of data D.W.A (data

are “black boxes”). Branching and conditional iteration compositions are defined on the level D.W.S, for which only one logical constant T (“white box”) is sufficient, while other data are “black boxes”. The overlapping composition is defined on the nominative level D.H.S, since it relies on nominative data structure.

CNP pays the most attention to the study of compositions that can be arranged according to the cubic typology of nominative data. Examples of such compositions are compositions based on indirect naming: the binary *assignment composition* $AS(f, g)$ and the unary *value composition* $VAL(f)$. These compositions are the respective analogues of such AP operators as $g \Rightarrow f$ and $'f$. Compositions and functions associated with complex names are also considered. The problems of independence of compositions and their computational completeness are investigated [128].

1.3.3.5. Program systems in CNP. Program systems are defined in accordance with the scheme of development of program concepts described by the program pentad. This scheme distinguishes the semantic, syntactic and denotational aspect. Each aspect is represented by the corresponding system, which fixes its relation to other aspects. Such systems are called, respectively, composition, description, and denotation systems. Composition systems define the means of construction of functions over a certain set of data. Description systems define descriptions of functions. Denotation systems define the meanings of descriptions. Depending on the level of abstraction, such systems can be simple or very complex.

In the simplest case, a *compositional system* is a pair of algebras: a data algebra and a function (program) algebra.

Description systems define descriptions inductively, using names of basic functions and predicates and the names of compositions. Usually, descriptions are terms in program algebras.

Program systems are defined as *composition-nominative systems*, which consist of composition, description, and denotation systems. It is important to note that the classes of predicates can also be defined using composition-nominative systems. This allows one to represent different predicate logics in the same way as program systems [118, 124].

1.3.3.6. Applications of CNP. CNP was first developed as an integrative project that aimed to integrate such basic programming disciplines as programming theory, mathematical logic, and computability theory. The central notion in CNP is the notion of a composition-nominative language, which (in semantic terms) is based on the notion of a composition-nominative algebra. The approach allows one to build a hierarchy of interconnected formal systems that serve as models of programming languages and specification languages, predicate logics of different types, and also define different types of abstract computability [54, 96, 110, 116–128].

1.4. Parallel programming

The development of modern software is characterized by dynamic expansion in construction and usage of parallel computing models, which became ubiquitous and permeate the most aspects of architecture and programming tools of computer systems. Network technologies and Internet facilities, operating systems and application software in present circumstances more or less are based on concepts of parallel and distributed computing. In this subsection, the main types of concurrency and formalized models and methods of parallel computations are considered.

1.4.1. Parallelism in computing systems. *Parallel computing system* (PCS) is the computing system providing the simultaneous (parallel) execution of operations of one or several programs. The main ideas for parallelizing operations in computer systems are the notion of *pipelining* of data flow computation and the concept of *true parallelism*, i.e. independent and simultaneous execution of operations on different devices of a computer. Using these two ideas in their pure form or in their union led to the following types of parallelism in computer architecture:

- pipeline processing;
- functional processing;
- vector-matrix processing;
- multiprocessing.

Pipeline parallelization of operations provides for several locations (stages) of processing, through which each element of a data flow being processed sequentially goes through. If the time of

processing of data element at each processing location is approximately the same, then the parallelism of computations is achieved by simultaneous processing of different elements of the input data flow at different stages of the pipeline. Thus, the longer the pipeline, i.e. the greater the number of its stages, and the longer the input data flow, the higher the degree of parallelization of operations is. Using this type of parallelism in arithmetic and control devices of a computer was chronologically the first and most common, especially in scientific and technical applications characterized by a high level of regularity of computations [77].

Functional parallelism provides the possibility of simultaneous execution of different processing functions (e.g. logic and arithmetic) by several independent functional units of a processor. An example of the use of functional parallelism is the presence of a coprocessor in the architecture of a personal computer to perform floating-point operations.

Vector-matrix processing combines the pipeline and functional parallelism. An important feature here is the presence of multiple identical processing elements with a single management system, which interact with each other through a shared memory area. An important milestone in the implementation of this type of parallelism was the first model of the supercomputer system Cray developed in 1976 [77].

Multiprocessing (or *true parallelism*), in contrast to the above-mentioned types of parallelism, is implemented by many full-featured processors, any of which is most often a full-fledged computer, except with specialized means of communication between processors and input/output. This parallelism, in contrast to the pipeline, does not require the organization of input data flow and is characterized by an independent and simultaneous execution of operations on different data. One of the earliest examples of this class of machines was a 64-processor system ILLIAC-IV (1972) with a capacity of over 50 Mflops. An example of national development was the multiprocessor system EC-1766 [143] created at V. M. Glushkov Institute of Cybernetics.

The main characteristics of parallel computing systems include:

- *the number and quality of processors*; systems with the number of processors $p > 100$ are called massively parallel. The parallel systems which consist of identical processors, are called homogeneous and the others are called heterogeneous;

- *memory type*, which describes the access of the processors to main memory: *shared* memory is global to the entire PCS and is equally accessible to all processors, *distributed* memory is split into private regions (according to the number of processors), the access to any one of which has only one processor, thus such memory is local;

- *the system of communication between the processors* characterizes the topology of communication of processors, such as static topology (linear, ring, star, matrix), dynamic topology (with switch connections), commutators, etc.;

- *control method: synchronous* (centralized), if commands in all processors of PCS are executed in accordance with a single signal from the central control unit, and an *asynchronous* (distributed), if the execution of commands in processors is not synchronized.

There are several classifications of PCS architectures in concordance with the above-mentioned characteristics. The most famous among them is the taxonomy proposed by M. J. Flynn [56, 57]. The four categories of architectures defined by Flynn are based upon the number of concurrent instruction (or control) streams and data streams available in the architecture:

- 1) SISD (Single Instruction Single Data). A sequential computer that exploits no parallelism in either the instruction or data streams. The single control unit fetches single instruction stream from memory. Examples of SISD architecture are the traditional uniprocessor machines like older personal computers and mainframe computers;

- 2) SIMD (Single Instruction Multiple Data). A computer that exploits multiple data streams against a single stream to perform operations which may be naturally parallelized. For example, an array processor or a graphics processing unit (GPU);

- 3) MISD (Multiple Instruction Single Data). Multiple instructions operate on one data stream. Uncommon architecture which

is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result;

4) MIMD (Multiple Instruction Multiple Data). Multiple autonomous processors simultaneously executing different instructions on different data. MIMD architectures include multicore superscalar processors, and distributed systems, using either one shared memory space or a distributed memory space.

As of 2006, the entire top 10 and most of the TOP500 supercomputers are based on MIMD architecture [57].

The Flynn's classification is useful, however, is not complete. In [57], it is supplemented, in particular, by Single Instruction, Multiple Threads (SIMT) category, an execution model where single instruction, multiple data (SIMD) is combined with multithreading.

1.4.2. Parallel computing models. The important models used in developing parallel applications [15, 23] are the shared-memory model (multithreading), the distributed-memory model (message passing model) and the hybrid model.

The *shared-memory model* allows multiple threads to exist within the context of one process. These threads share the process's resources, but are able to execute independently. Various mechanisms such as locks/semaphores are used to control access to the shared memory, resolve contentions and to prevent race conditions and deadlocks. From a programming perspective, threads implementations commonly comprise a library of subroutines that are called from within parallel source code and a set of compiler directives embedded in either sequential or parallel source code. The examples of implementation of this model are POSIX Threads [113], CUDA [130], OpenMP [133].

The *distributed-memory model* means that a set of tasks use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines. Tasks exchange data through communications by sending and receiving messages. Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation. From a programming perspective, message passing implementations usually comprise a library of subroutines. Calls to these subroutines are

embedded in source code. Message Passing Interface (MPI) [111] is the “de facto” industry standard for message passing. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran.

The *hybrid model* combines the shared-memory and distributed-memory programming models. Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with the threads model (OpenMP). Another similar and increasingly popular example of a hybrid model is using MPI with CPU-GPU (Graphics Processing Unit) programming.

The process of parallel program design includes four main stages: decomposition, communication, synchronization and mapping [15, 23]. The *decomposition* is the process of partitioning of the applied task and the data on which it operates into smaller subtasks. The *communication* focuses on the flow of information and coordination among the tasks that are created during the decomposition stage. The specific features of the applied problem and the decomposition method determine the communication pattern among these cooperative tasks. The four popular communication patterns commonly used in parallel programs are: local/global, structured/unstructured, static/dynamic, and synchronous/asynchronous. The *synchronization* is the coordination of parallel tasks in real time, which is generally associated with communications. It is often implemented by establishing a synchronization point within a program where a task may not proceed further until another task or tasks reach the same or logically equivalent point. The most common means of synchronization are critical sections, semaphores, barriers, mutexes, etc. *Mapping* is assigning each task to a processor such that it maximizes the utilization of system resources (such as CPU) while minimizing the communication costs. Mapping decisions can be taken statically (at compile-time/before program execution) or dynamically at runtime by load-balancing methods.

Formal models of parallel computing [46, 108] represent a convenient tool for solving the problems of formalized design, analysis, and transformation of parallel algorithms and programs. In work [46], these models are split into the following categories:

- models without limitation of parallelism forms;
- models with limited parallelism forms;
- algebra-algorithmic models of parallel computations.

The class of *models without limitation of parallelism forms* is divided into two categories:

- fully abstract models;
- partially abstract models.

Fully abstract models in their description do not use the facilities for explicit specification of parallelism, schemes of interaction and synchronization of parallel computation components. The most known models of this class are declarative computing models, such as functional and logic programming paradigms, as well as models based on graph and term rewriting technique, including algebraic programming paradigm [88, 99, 100]. Abstract models of parallel computing also include models that follow the paradigm of procedural programming. A well-known model of this type is UNITY [26], the model of nondeterministic parallel programs, which inherited the approach to parallelism based on E. W. Dijkstra's guarded commands [36]. A program in UNITY is a set of guarded commands of group assignment. A step of program execution consists in non-deterministic (but fair, in the sense of absence of never chosen command) selection of a command, calculation of the value of guard predicate and execution of the operator of the command, if the predicate was true. The semantics of the program is to calculate the fixed point of transformations defined by guarded commands, and then the program terminates. This group of models also includes works of Kyiv algebraic-cybernetic school of V. M. Glushkov: the algebra of data structures for synchronous parallel computing, transition systems, networks of algorithmic modules [88], and the model of controlling spaces [63]. The combination of these models completely covers the well-known triad of model types (functional, dynamic and structured), which were widely used to automate the design of computer systems, including parallel, and their software [65].

The basis of the theory of algebras of data structures is the concepts of algebra of algorithms, data structures and the data graph (the set of points with the set of shifts defined on it) [88]. The examples of typical data graphs are integer lattices used in most of the

tasks related to calculus mathematics. The information environment of programs, which are built using this model, includes scalar and structured variables. The primary means for transformation of the information environment of programs are the operators of structured assignment of the form:

$$r_1(c_1g_1):=E_1, \dots, r_n(c_ng_n):=E_n,$$

where $r_1, \dots, r_n$ are structured variables; $c_1, \dots, c_n$ are nodes of the data graph, and $g_1, \dots, g_n$ are shifts on them; $E_1, \dots, E_n$ are expressions in the data algebra, which in the general case are dependent on $r_1, \dots, r_n$. The semantics of this operator consists in simultaneous (parallel) assignment of values to a subset of scalar variables corresponding to structured variables $r_1, \dots, r_n$, and is described by periodically defined function over data structures. On the one hand, the model uses the algebraic means of a solution of functional equations, which makes it general and abstract, and on the other hand, it has the facilities for measuring the complexity of parallel algorithms. In work [88], the specific sets of basic structured functions generating the algebra of data structures are considered and the methodology for synthesis of classes of efficient parallel programs directly from functional specifications of the algebra of data structures is proposed.

The concept of controlling spaces proposed in [63] is an abstract model to adequately address all the necessary dynamic emerging relations between the individual independently executed activities of a complex recursively-parallel process. The controlling space consists of points and connections between them, identified with interaction channels. The concept of the point and the channel are abstractions of interaction and are not specified in the model. The points of the controlling space are associated with algorithmic modules, which are the owners of the points and the surrounding channels. The algorithmic module is a discrete transducer which has a control part and data being processed. The algorithmic language of modules is complemented by the language for the description of inter-module interaction through the channels of the controlling space. Both data and control can be transmitted. The modules can

create new points and channels, destroy them, as well as put the models, including the copies of themselves, into new points. Thus the inter-module recursion and dynamic change of the structure of the controlling space become possible. The basic structural-dynamic unit of the model is a process. The model of controlling spaces is the basis of the technology of parallel programming named PARUS [7], which is focused on the development of software systems for solving problems characterized by the recursive-parallel organization of computations, such as image processing, relational database operations, associative search, etc. The model of controlling spaces is very general and abstract, but does not have adequate means for expressing the performance of parallel algorithms

Thus, the parallelism of computations in fully abstract models is defined at a very high conceptual level, declaratively, and it gives them a property of maximum generality and abstractness. However, the models of this class have only high-level tools necessary to provide the measurability of performance of parallel computations. Therefore, these models are typically used for the purposes of specification, high-level simulation and design of parallel programs.

In the class of *partially abstract models*, the computation parallelism manifests itself more clearly, but uses some abstraction of interaction and synchronization. One of the earliest models of this class is the data flow model [48]. This model is similar to the functional model, as languages for programming of data flows have much in common with functional languages (the principle of single assignment, no side effect, etc.), although they are usually first-order languages. The main idea of the data flow model is the explicit definition of interactions of parallel computations through the variables so that the execution of each operation can proceed independently and asynchronously with execution of other operations, and depends only on the readiness of its input data. The data flow model is based on data flow graph, i.e. directed bipartite graph, which has two types of vertices called actors and relations. The nodes are connected by edges, which represent the environment of interaction used for transmitting the information in the form of markers. The actors are the operations to be executed and the relations contain and transmit

markers between the actors. Each actor is associated with the set of input and output relations. The presence of markers in all elements of the input set of the actor makes it possible to activate (launch) the actor for execution of the operation predefined for this actor. The result of operation execution is markers that appear on relations belonging to the output set of the actor. Thus, the operation of the actor can be executed (triggered) whenever its flow of operands is nonempty.

The main problem associated with the development of the data flow model consisted in the reduction of computational complexity through the parallelization and elimination of the processor-memory bottleneck and had not been successfully solved. However, the idea of data flows has proved useful in the theory and practice of Petri nets [95], which in the context of parallelization of control flows, similarly formalize and use the concept of an abstract asynchronous system in terms of events and conditions. Conditions-places and events-transitions are linked by the relation of direct dependence, represented by directed edges that pass markers between places and transitions. The main means of a static description of functioning of Petri nets is marking, which is the distribution of markers to places. At studying such uninterpreted networks of conditions and events, it was possible to investigate rather general properties of networks, such as reachability of network states (markings), agility of transitions, monotony, density, deadlock-freeness of networks, etc. These results were of great theoretical significance, but their effect on building practical models of parallel computation of wide application has been limited and reduced to a partial solution of tasks of design/simulation of parallel systems.

One of the most well-known theoretical models of parallel computing is the PRAM model of a parallel machine with a shared random-access memory [91] and its generalizations and improvements. The model generalizes the concept of von Neumann computer and consists of P parallel components (processors), having equal access to shared memory, the number of components is the parameter of the model. Although the model does not require it, most of the algorithms developed for this model suggest the synchronous execution of component operators. The PRAM model, although it is quite

general, is not abstract enough, as it requires the explicit definition of distribution of programs and data to processors, and also the explicit definition of variables for the interaction. However, the model has the property of measurability of performance of parallel algorithms. The measurability of performance in this model is limited because the estimation of complexity in the model does not account for time spent on exchanges, and in addition, the model ignores the costs associated with the limited scalability of shared memory. This was the reason for the appearance of the so-called practical models of parallel computing, which improve and complement the PRAM, such as the bulk-synchronous parallel (BSP) model [162] and the LogP model [101].

Thus, the parallelism of computations in the class of partially abstract models is defined at a highly abstract level, though with elements of detailing the architecture on which they are focused. Therefore, the properties of generality and abstraction are limited. But the models of this class have a much greater capacity and means to achieve measurable performance properties of parallel computing, and some of them, such as, BSP and LogP, are multidimensional and therefore can serve not only for the purposes of the specification or modeling, but also for developing parallel algorithms with predictable performance.

The main purpose of introduction of the *models with limited parallelism forms* is a specialization of computations for specific classes of applications and/or extension of the set of basic operations (which execution time are considered to be unit) with new (specialized) operations, which in the basic model are sufficiently complex procedures and in the new model have much improved estimation of complexity. These models are split into two categories:

- control parallelism models;
- data parallelism models.

Control parallelism models. In work [27], the means for structured management of parallel computation, which are called skeletons, are proposed. The skeleton is an abstraction of some computation scheme for a whole class of applications. The process of computing is a sequential-parallel structure, where the upper (sequential) level available to a programmer is the sequence of some

computational steps, each of which is implemented (in parallel) by a skeleton corresponding to the applied computational scheme, taking into account the architectural features of the target multiprocessor machine. Such approach to parallel computing is a common practice for computing paradigms, in which recursive-parallel generation of subtasks is performed (such as divide-and-conquer). Languages and models of functional programming are especially convenient for definition and use of the skeletons. For example, work [148] defines the computing skeletons as higher-order functions of some functional language, which have a direct implementation in the architecture. There are also works, where the functional interpretation of the skeletons is applied for defining the means of coordination sublanguage of parallel programming [24].

Data parallelism models compare favorably with control parallelism models, as they use very simple means of composition of operators (sequential concatenation for imperative computing paradigm and composition of functions for functional one), each of which may contain a parallel implementation in the form of application of the same operation to a set of various data. The substantial difference between these two types of models is that the means of data-parallel models are focused on a static structure of a program, while the control parallelism involves the dynamic behavior of computation processes. The simplest data structures for simultaneous execution of the same operation over the set of data elements are a list, an array or a vector of these elements [48]. Therefore it is not surprising that many parallelism models are based on operations over these data structures. The most general model based on vector operations, consists of a vector random access memory and a vector processor, and allows five types of operations over vectors, including scalar and scalar-vector operations, the parallel execution operations of the *map* type, specifying the simultaneous application of the same function to all elements of a vector, operations of permutation of vector elements and the operation *scan* for parallel computing of all partial sums of vector elements. In addition, the model has the means of segmenting the vectors, so that vector operations can be applied independently to different subvectors. In work [19], the vector model is supplemented by an important feature called nested parallelism,

which allows using data parallelism for programming recursive algorithms. The Bird-Meertens formalism (BMF) [17] is widely known among the developers of theoretical problems of parallel software and belongs to data parallelism models. It is a calculus for deriving programs from specifications by a process of equational reasoning. The formalism is based on categorical data types and their attendant operations. In particular, operations include a generalized *map* operation and a generalized *reduction* operation. The programs based on this formalism are single-threaded and data-parallel. BMF theories have been built for lists, trees, and arrays.

Algebra-algorithmic models of parallel computations (in particular, macro pipeline computations) are based on functions over data structures and regular programs represented in the algebra of algorithms [5], the basic operators and conditions of which are defined through these functions. The functions over the data structures are defined by means of functional equations of the form: $y = F(x, y)$, where $y = (y_1, \dots, y_s)$, $x = (x_1, \dots, x_r)$ are sets of data structures located in some deployment region C and taking the values in some data region D . The expression $F(x, y)$ is constructed by means of superposition of functions over the data structures, which are chosen from some set K of basic structured functions. The functions from K are assumed to be continuous. In this case, the equation $y = F(x, y)$ has the least solution $y = f(x)$, which can be obtained by means of successive approximations to a fixed point:

$$y = \bigcup_{k=0}^{\infty} y^{(k)}, \quad y^{(0)} = \perp, \quad y^{(k+1)} = F(x, y^{(k)}),$$

where $\perp$ designates a set of nowhere defined data structures. The difference $\text{Dom}(y^{(k+1)}) \setminus \text{Dom}(y^{(k)})$ of ranges of definition of two adjacent successive approximations is called the k -th computation wavefront. The size of this wavefront and geometry of its movement over the deployment region C with the change of k defines the possibilities of parallel computation of structured functions defined

by the initial equation. Namely, the values $y^{(k)}(c)$ can be simultaneously computed in all points of k -th computation wavefront.

The algebra-algorithmic models provide the possibility for analysis of organization of macro pipeline computations. The general method of computation of the function $y = f(x)$ on the multiprocessor system with p processors consists in splitting of the region $E \subseteq C$, which is supposed to be finite, into disjoint sets $E_1, \dots, E_p$ and distribution of work between the processors in such a manner that each processor computes the values $f(x)$ only in the points of the corresponding set. Computations are made step-by-step, where on k -th step processors simultaneously perform computations in the points of k -th computation wavefront and then exchange the necessary information. At such organization of computations, for estimation of the efficiency e_p of work of the multiprocessor system with p processors, the general formula was obtained [48]:

$$e_p = \frac{w}{u} \left(1 + \frac{m}{u} l \tau \right),$$

where w is the average amount of computations performed by one processor at one step, u is the maximum amount of calculations of one processor at one step, m is the maximum amount of exchanges of one processor at one step, l is the maximum value of a chromatic class of the graph of exchanges on one step (the nodes of the graph represent processors and the edges represent the necessity of exchanges); $\tau = \tau_1 / \tau_2$, where τ_1 is the time of transfer of one scalar data element, τ_2 is the average execution time of one operation over a scalar data element. Here it is supposed that all the mentioned sizes depend on the parameter n characterizing the volume of the region $E_i(n)$ in which the calculations are performed. It is supposed that $p \leq |E_i(n)|$, and the coefficients $\frac{w}{u}$, $\frac{m}{u}$ and l are called the coefficients of uniformity, macropipelining, and locality of splitting

$R(n) = \{E_1(n), \dots, E_p(n)\}$. The splitting $R(n)$ is called uniform, if the uniformity coefficient is bounded below by positive value which does not depend on p , and it is called macro pipeline or local, if the corresponding coefficients are bounded above by values which are not dependent on p . The fundamental result [102] consisted in the statement that if there is a pipeline, uniform and local splitting of the computation region, the organization of macro pipeline computations with *linear* speedup is possible. The general schemes of parallel macro pipeline computations were developed for a number of applications: static (wave, iterative and mixed) and dynamic macro pipelines [102].

The models of parallel macro pipeline computations have been further developed within the framework of the *algebra-dynamic* approach to construction of parallel programs. The set of algebra-dynamic models developed in [48] reflects the complex of aspects of parallelism models for multiprocessor systems: algorithmic, programmatic and coordination. The algebra-dynamic approach allows to integrate the information analysis of program operators into the operational semantics of parallel programming language and to implement the idea of asynchronosity of computations at intracomponential level or to display the obtained information in the coordination part of the model and implement the potential asynchronosity of computations at the level of intercomponential interactions. In the first case, at the specified depth of the information analysis defined by the efficiency requirements and possibilities of parallel system implementation, the approach allows to build the families of dynamic program models adapted to possibilities of implementation, and obtain parallel programs with the improved efficiency quality of the usage of a multiprocessor system. Furthermore, some classes of communication deadlocks can automatically be resolved. In the second case, fixing the results of information analysis in the form of coordination means (forcing expressions), it is possible to significantly improve the quality of the mechanisms of synchronization and exchange of components of macro pipeline programs, such as controlled buffering and pipelining of exchanges, and also to use this formalism for automation of transformations of pro-

grams for the purpose of minimization of losses of efficiency of parallelization, associated with references to slow memory levels of a multiprocessor system. The important consequence of application of such methods is the expansion of a range of processors on which the computation speedup of a parallel program linearly depends on the number of processors executing it. In Subsection 5.4 (see also works [6, 39, 46]) the algebra-dynamic approach is applied for formal design and automated transformation of parallel programs for graphics processing units.

1.5. Object-oriented and component-oriented programming

Object-oriented programming is a programming methodology based on the representation of a program as a set of objects. Each object is the implementation of some class (a type of special kind). Classes form a hierarchy that is based on inheritance principles. An object is characterized by all its properties and their current values and a set of actions possible for this object [78, 104]. Thus, basic notions of this programming paradigm are objects and classes.

The *object* is a set of state variables and methods (operations) associated with them. These methods define how the object interacts with the surrounding world. Possibility to manage the states of the object by calling methods determines the behavior of the object. The set of methods is often called the interface of an object.

The *class* is a group of data and methods (functions) to work with the data. Objects with identical properties, i.e. with identical sets of state variables and methods, form a class. The object is a certain implementation, a copy of the class.

The interaction of objects is carried out by means of messages. The *message* is information of event which took place and addressed to a certain object. An object that got a message must answer it with its actions by calling a method corresponding to the event.

Basic principles of object-oriented programming are encapsulation, inheritance, and polymorphism.

Encapsulation is bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components. It is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' di-

rect access to them. Publicly accessible methods are generally provided in the class to access the values, and other client classes call these methods to retrieve and modify the values within the object.

Inheritance is the mechanism of basing an object or class upon another object (prototype-based inheritance) or class (class-based inheritance), retaining similar implementation. Also defined as deriving new classes (subclasses) from existing ones (superclass or base class) and forming them into a hierarchy of classes. In most class-based object-oriented languages, an object created through inheritance (a “child object”) acquires all the properties and behaviors of the parent object (except constructors, destructor, overloaded operators and friend functions of the base class). Inheritance allows programmers to create classes that are built upon existing classes, to specify a new implementation while maintaining the same behaviors (implementing an interface), to reuse code and to independently extend original software via public classes and interfaces. The relationships of objects or classes through inheritance give rise to a directed graph.

Polymorphism allows using an object with the same interface without the information of the type and internal structure of an object. A programming language supports the polymorphism, if classes may have different implementation, for example, the class implementation can be changed in the process of inheritance.

Many of the most widely used programming languages (such as C++, Java, Python, etc.) are multi-paradigm and they support object-oriented programming to a greater or lesser degree, typically in combination with imperative, procedural programming. Significant object-oriented languages include Java, C++, C#, Python, PHP, JavaScript, Ruby, Perl, Object Pascal, Objective-C, Dart, Swift, Scala, Common Lisp, MATLAB, and Smalltalk.

Prototype-based programming is a style of object-oriented programming in which behavior reuse (known as inheritance) is performed via a process of reusing existing objects via delegation that serve as prototypes. This model can also be known as prototypal, prototype-oriented, classless, or instance-based programming. Delegation is a language feature that supports prototype-based programming. This programming uses generalized objects, which can then be

cloned and extended. The first prototype-oriented programming language was Self, Some current prototype-oriented languages are JavaScript (and other ECMAScript implementations such as JScript and Flash's ActionScript 1.0), Lua, Cecil, NewtonScript, Io, IoKe, MOO, REBOL, and AHK.

Component-oriented programming enables programs to be constructed from prebuilt software components, which are reusable, self-contained blocks of computer code. These components have to follow certain predefined standards including interface, connections, versioning, and deployment.

Components come in all shapes and sizes, ranging from small application components that can be traded online through component brokerages to the so-called large grain components that contain extensive functionality and include a company's business logic. In principle, every component is reusable independent of context, that is to say, it should be ready to use whatever from wherever.

Component-oriented programming is to develop software by assembling components. While object-oriented programming emphasizes classes and objects, component-oriented programming emphasizes interfaces and composition. In this sense, it could be said that component-oriented programming is interface-based programming. Clients in component-oriented programming do not need any knowledge of how a component implements its interfaces. As long as interfaces remain unchanged, clients are not affected by changes in interface implementations.

An individual *software component* is a software package, a web service, a web resource, or a module that encapsulates a set of related functions (or data).

All system processes are placed into separate components so that all of the data and functions inside each component are semantically related (just as with the contents of classes). Because of this principle, it is often said that components are modular and cohesive.

With regard to system-wide co-ordination, components communicate with each other via *interfaces*. When a component offers services to the rest of the system, it adopts a *provided* interface that specifies the services that other components can utilize, and how they can do so. This interface can be seen as a signature of the com-

ponent — the client does not need to know about the inner workings of the component (implementation) in order to make use of it. This principle results in components referred to as *encapsulated*.

However, when a component needs to use another component in order to function, it adopts a *used* interface that specifies the services that it needs. Another important attribute of components is that they are *substitutable*, so that a component can replace another (at design time or run-time), if the successor component meets the requirements of the initial component (expressed via the interfaces). Consequently, components can be replaced with either an updated version or an alternative without breaking the system in which the component operates.

Software components often take the form of objects (not classes) or collections of objects (from object-oriented programming), in some binary or textual form, adhering to some interface description language so that the component may exist autonomously from other components in a computer. In other words, a component acts without changing its source code. Although, the behavior of the component's source code may change based on the application's extensibility, provided by its writer.

When a component is to be accessed or shared across execution contexts or network links, techniques such as *serialization* or *marshalling* are often employed to deliver the component to its destination.

Reusability is an important characteristic of a high-quality software component. Programmers should design and implement software components in such a way that many different programs can reuse them. Furthermore, component-based usability testing should be considered when software components directly interact with users.

A *component model* is a definition of properties that components must satisfy, methods and mechanisms for the composition of components. Examples of component models are Enterprise JavaBeans (EJB) model, Component Object Model (COM) model, .NET model, X-MAN component model, and Common Object Request Broker Architecture (CORBA) component model.

Software engineering practitioners regard components as part of the starting platform for service-orientation. Components play this role, for example, in web services, and more recently, in service-oriented architectures, whereby a component is converted by the web service into service and subsequently inherits further characteristics beyond that of an ordinary component.

Control questions

1. What is a programming paradigm?
2. What is the imperative programming paradigm? Give examples of imperative programming styles and corresponding programming languages.
3. What is the difference between structured and non-structured programming? Describe the basic constructs of structured programming.
4. What distinguishes the declarative programming paradigm from the imperative one?
5. Name the types and examples of functional programming languages.
6. What is logic programming? What are the differences between logic and functional programming?
7. What is theoretical programming?
8. What are the main features of algebraic and insertion programming?
9. What problems are solved within the framework of the algebra of algorithmics? What facilities are used for design and modification of algorithms in the algebra of algorithmics? What are the differences and similarities between the algebra of algorithmics, algebraic algorithmics, and intentional programming?
10. What is composition-nominative programming? What are the main principles underlying CNP?
11. What are the main CNP data types. What is nominative data?
12. Give examples of CNP compositions. List CNP compositions by indicating their notations.
13. What are program systems in CNP?

14. What is a parallel computing system? What are the main categories of parallel architectures according to the taxonomy proposed by Flynn?
15. Describe the basic parallel computing models.
16. What are the main concepts and principles of object-oriented programming?
17. What is component-oriented programming?

Chapter 2

Algebras and specifications of algorithms

In this chapter, algebras of algorithms underlying the algebra-algorithmic programming are considered. They are intended for formalized description of structured and non-structured schemes represented in analytic, natural-linguistic and flowgraph forms.

The algebras considered in this chapter are based on the concept of the abstract automaton model of a computer [64], which is the interaction of a control automaton Ψ and an operational automaton Φ . The automata Ψ and Φ are also called control and operational structures, correspondingly. The control structure Ψ is a system of control terminals $\Psi = \{\Psi_q \mid q \in I\}$. The operational structure Φ is the composition $R(\Phi) = \{\Phi_q \mid q \in I\}$ of nonintersecting substructures Φ_q associated with corresponding control terminals Ψ_q . The set of states $IS = \{m_i \mid i \in I\}$ of the automaton Φ will be called further as an *information set*. On the set IS , the set of operators $Op = \{A_j \mid j \in I\}$ and the set of logical conditions $Pr = \{u_k \mid k \in I\}$ are defined. The interaction of the automata Ψ and Φ is carried out according to a feedback principle (Fig. 2.1).

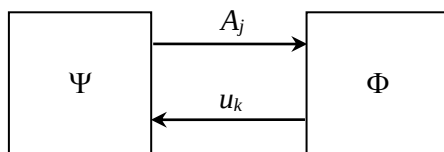


Fig. 2.1. An abstract automaton model of a computer

The input of the automaton Ψ is a set of logical conditions u_k , which characterize the current state of the operational automaton Φ . According to the mentioned set, the control automaton produces an operator A_j , which transforms the current state of the automaton Φ . The considered cycle is repeated until the automaton Ψ transits

to a final state. The composition of control and operational automata is also called a discrete transducer.

Algorithms of functioning of the discrete transducer in the given concept of an abstract automaton model are represented by schemes in corresponding algebras of algorithms.

In the following subsections, the following algebras are considered: Dijkstra algebra associated with the technology of structured programming, Kaluzhnin algebra for graphical description of non-structured schemes of algorithms, Glushkov algebra for description of structured schemes, including the facilities for computation process prediction and design of parallel algorithms, and the algebra of algorithmics, which is based on the mentioned algebras. The signature of each algebra consists of predicate and operator constructs corresponding to a specific method of algorithm design, i.e. structured, non-structured and other.

2.1. Dijkstra algebra

In 1968, E. W. Dijkstra proposed the idea of construction and research of the algebra of programs, which is based on well-known programmer constructs: composition, branching and a loop.

The *Dijkstra algebra* defined in [159] is the two-sorted algebraic system intended for analytical description and transformation of structured algorithms:

$$DA = \langle \{Pr, Op\}; \Omega_{DA} \rangle,$$

which consists of the set of conditions (predicates) Pr and the set of operators Op , both defined on the information set IS . The set IS is a set of all data (input, output and intermediate), being processed by algorithms. The signature Ω_{DA} includes the Boolean operations taking values in Pr and generating compound conditions, and also the operations taking the values in Op and generating compound operators.

Operators from the set Op map the information set IS to itself. Predicates from the set Pr are functions that map elements of

the set IS to elements of the set $E_2 = \{0, 1\}$, where 0 is for false, 1 is for true.

The compound predicates from the set Pr are constructed from basic predicates by using the Boolean operations:

- disjunction $u \vee u'$, where $u \in Pr$, $u' \in Pr$ are logic variables;
- conjunction $u \wedge u'$;
- negation $\bar{u}$.

The compound operators from the set Op are built from elementary ones by means of the following operations:

- the composition $A * B$, which is the sequential execution of operators $A \in Op$ and $B \in Op$;
- the alternative (the branching) $([u] A, B)$, which executes the operator A if the condition u is true, and applies operator B otherwise;
- the loop operation $\{[u] A\}$, which executes the operator A repeatedly until the condition u is true.

The operator variables $A = \{A_1, A_2, \dots, A_n\}$ and logic variables $U = \{u_1, u_2, \dots, u_m\}$ are used as elementary (basic) operators and logic conditions, accordingly.

Example 2.1. The following is the compound scheme P constructed from the above elementary constructs:

$$\begin{aligned}
 P &= \{[u_1] A_1\}, \\
 A_1 &= \{[u_2] A_2 * D\}, \\
 A_2 &= A_3 * C, \\
 A_3 &= ([u] A, B), \\
 u &= \bar{u}_2 \wedge u_3.
 \end{aligned}$$

It specifies the structure of algorithms belonging to some class. After the superposition, i.e. the convolution of the given process of step-by-step development of the scheme P , we obtain the following formula:

$$P = \{[u_1] \{[u_2] ([\bar{u}_2 \wedge u_3] A, B) * C\} * D\}.$$

The P scheme is an example of a non-interpreted scheme. The transition from non-interpreted schemes to algorithms is concerned with the interpretation of the operator and logic variables included in a non-interpreted scheme. The construction of interpretations is associated with the concept of *marking* of data being processed [1, 5, 159]. The marking is carried out by means of the special markers fixing certain positions in data sequences, and the pointers moving over the processed sequences in defined directions. On the marked sequences of data, the interpretations of operator and logic variables are defined.

Example 2.2. Consider the marked sequence of symbols

$$M : H Y_1 a_1 a_2 \dots a_n K,$$

where H and K are the markers fixing the beginning and the end of the sequence M ; a_i is the element of the sequence, $i=1, \dots, n$; Y_1 is the pointer moving over the M in a direction from left to right. On the sequence M , the predicate $d(Y_1, K)$ is defined, which takes the true value, if Y_1 reached the marker K , and false otherwise, and also the following operators:

- $R(Y_1)$, which is the shift of the pointer Y_1 over the sequence M one symbol to the right;
- $SET(Y_1, H)$, which is the operator placing the pointer Y_1 in a position directly to the right of the marker H .

Let us introduce the following partial interpretation of the variables of the scheme P :

$$u_2 \rightarrow d(Y_1, K), \quad C \Rightarrow R(Y_1), \quad D \Rightarrow SET(Y_1, H),$$

where $\rightarrow$ and $\Rightarrow$ designate the interpretation of logic and operator variables, accordingly.

The result of the interpretation of the scheme P is the partially interpreted scheme

$$SP = \{[u_1] \{[d(Y_1, K)] (\overline{[d(Y_1, K)]} \wedge u_3] A, B) * R(Y_1)\} * SET(Y_1, H)\}.$$

The essence of SP scheme (under the corresponding interpretation of operator and logic variables) consists in cyclic scanning over the sequence M in a direction from left to right up to the truth of the condition $d(Y_1, K)$ with subsequent return of Y_1 to the marker H and an exit to an external loop. The described cyclic process proceeds until the condition u_1 is true. During such multiphase scanning, depending on the value of the condition u_3 , the operator A or the operator B processes a current symbol of the sequence M .

Partially interpreted schemes are called *strategies* of symbolical processing. The above-given strategy SP is the multiphase (bubble) processing of the sequence M . Depending on the interpretation of the variables included in a strategy, algorithms for solving various applied problems can be designed.

Example 2.3. Let the above-considered sequence M represent a numerical array

$$M : H \ Y_1 \ a_1 \ a_2 \ \dots \ a_n \ K.$$

The following interpretations of the variables included in the SP strategy (see Example 2.2) are introduced:

$$u_1 \rightarrow Sorted(M), \ u_3 \rightarrow l > r \mid Y_1, \ A \Rightarrow Transp(l, r \mid Y_1), \ B \Rightarrow E,$$

where $Sorted(M)$ is the predicate, which takes the true value, if the array M is sorted and the false value otherwise; l and r are the array elements located directly to the left and, accordingly, to the right of the pointer Y_1 ; $l > r \mid Y_1$ is the predicate, which takes the true value if the specified relation holds for the elements, located to the left and to the right of the pointer Y_1 ; E is an identity operator,

which is not changing a state of the processed sequence, i.e. $E(M) = M$; $Transp(l, r | Y_1)$ is the operator of transposition of the elements adjacent to the pointer Y_1 . As a result, the algorithm *Bubble'* of bubble sorting of the numerical array M is obtained:

$$Bubble' = \{[Sorted(M)] \{[d(Y_1, K)] (\overline{[d(Y_1, K)]} \wedge (l > r | Y_1)) [Transp(l, r | Y_1), E] * R(Y_1)\} * SET(Y_1, H)\}.$$

Thus, the introduction of missing interpretations in non-interpreted and partially interpreted schemes leads to a construction of specific algorithms solving the problems concerned with a chosen subject domain.

For Dijkstra algebra, the transformation facilities based on usage of properties of operations included in its signature was developed. For instance, the connection between branching and a loop is presented by the following equality:

$$\{[u] A\} = ([u] E, A * \{[u] A\}). \quad (2.1)$$

Besides the basic operations included in the signature of the algebra, the derivative operations are used, which can be obtained as a result of the superposition of basic operations and constants of the algebra under consideration. For example, in the algebra of numbers the multiplication operation is a derivative of an addition; in the Boolean algebra the disjunction operation is the derivative of conjunction and negation.

One of the derivative operations of *DA* is the filtration operation $\varphi(u) = ([u] E, W)$, obtained as a result of the superposition of an identity operator E , an uncertain operator W and a branching operation. Filters provide flexible control over computation in algorithm schemes. The computation in branches with truth values of the filter condition u proceeds, whereas computation in other branches breaks off.

Another example of derivative operation in *DA* is the operation of the generalized loop, which represents the superposition of

composition and loop operations and implements an exit from a loop by the condition located in the middle of the loop:

$$\{A [u] B\} = A * \{[u] B * A\}.$$

At $A = E$ we have a basic loop: $\{E [u] B\} = \{[u] B\}$, and at $B = E$ we obtain a do-while loop in which the condition is checked after the application of the body of the loop:

$$\{A [u] E\} = \{A [u]\}.$$

Alternative and loop operations satisfy the following properties:

$$\{[u] A\} = \{[u] \varphi(\bar{u}) * A\}, \quad (2.2)$$

$$\{[u] \varphi(\bar{u}) * ([\bar{u} \wedge u'] A, B) * C\} = \{[u] \varphi(\bar{u}) * ([u'] A, B) * C\}. \quad (2.3)$$

Consider the modification of the bubble sort algorithm *Bubble'* given in Example 2.3. The initial scheme of the algorithm is the following:

$$\begin{aligned} \text{Bubble}' = \{ & [\text{Sorted}(M)] \{ [d(Y_1, K)] (\overline{d(Y_1, K)} \wedge (l > r \mid Y_1)] \\ & \text{Transp}(l, r \mid Y_1, E) * R(Y_1) \} * \text{SET}(Y_1, H) \}. \end{aligned}$$

Let us replace the interpretations of operators and conditions of the *Bubble'* algorithm with corresponding variables:

$$\begin{aligned} \text{Sorted}(M) &\rightarrow u_1, \quad d(Y_1, K) \rightarrow u_2, \quad \overline{d(Y_1, K)} \wedge (l > r \mid Y_1) \rightarrow u_3, \\ \text{Transp}(l, r \mid Y_1) &\Rightarrow A, \quad E \Rightarrow B, \quad R(Y_1) \Rightarrow C, \quad \text{SET}(Y_1, H) \Rightarrow D. \end{aligned}$$

As a result, the non-interpreted scheme P is obtained:

$$P = \{[u_1] \{[u_2] ([\bar{u}_2 \wedge u_3] A, B) * C\} * D\}.$$

Consider the process of transformation of the P scheme into a more compact form by means of application of equalities (2.2) and (2.3):

1) equality (2.2) is applied, forming the filter $\varphi(\bar{u})$ before the enclosed branching:

$$P = \{[u_1] \{[u_2] \varphi(\bar{u}_2) * ([\bar{u}_2 \wedge u_3] A, B) * C\} * D\};$$

2) on the basis of equality (2.3), the first conjunctive factor of the condition of the enclosed branching is absorbed:

$$P = \{[u_1] \{[u_2] \varphi(\bar{u}_2) * ([u_3] A, B) * C\} * D\};$$

3) equality (2.2) is applied in the reverse direction:

$$P = \{[u_1] \{[u_2] ([u_3] A, B) * C\} * D\} = P'.$$

The applied transformations can be considered also as formalized substantiation (derivation) of the identity:

$$\{[u] ([\bar{u} \wedge u'] A, B) * C\} = \{[u] ([u'] A, B) * C\}.$$

As a result of the interpretation of variables of the scheme P' , a more compact representation of the bubble sort algorithm is obtained:

$$\begin{aligned} \text{Bubble} = & \{[\text{Sorted}(M)] \{[d(Y_1, K)] ([l > r \mid Y_1] \\ & \text{Transp}(l, r \mid Y_1), E) * R(Y_1)\} * \text{SET}(Y_1, H)\}. \end{aligned}$$

Notice that the illustrated process of transformations is a powerful tool for constructing of new algorithmic knowledge.

2.2. Algebra of flowgraphs

In this subsection, the basic definitions concerning flowgraphs of algorithms are given, and the algebra flowgraphs (Kaluzhnin algebra) [5, 159] is considered.

Let us fix two sets of variables $\tilde{U} = \{U_j \mid j = 1, 2, \dots, m\}$ and $\tilde{A} = \{A_i \mid i = 1, 2, \dots, n\}$, where U_j are logic variables; A_i are operator variables. Let G be a directed graph having the set of vertices (nodes) $V = \tilde{U} \cup \tilde{A}$ with the binary relation R defined on V and associated with the set of edges of the graph G . Each vertex of the graph is marked by some variable $v \in V$. The node marked with the logic variable $U_j \in \tilde{U}$ is called a *recognizer* (or a *u-node*) and the node marked with the operator variable $A_i \in \tilde{A}$ is called an *operator* (or an *A-node*). Besides the vertices, G contains edges of two types: ordinary and logic. Logic edges are supplied by plus and minus marks. Each *A-node* has only one outgoing edge, whereas the recognizer has two outgoing edges, one of which is signed with a plus and another with a minus. Let us notice also that different *A-nodes* (*u-nodes*) can be marked with the same operator variable (accordingly, logic variable).

The graph G also contains two additional nodes with the marks “Entry” and “Exit”. The “Entry” node has no incoming edges, and the “Exit” node has no outgoing edges. Other nodes can be marked by labels of two types: operator labels contained in rectangles and predicate labels contained in triangles. The transition to the “Exit” node completes the work of an algorithm.

The considered connected graph G is called a *flowgraph* of an algorithm.

Let us introduce the concept of a flowgraph interpretation. Let G be some flowgraph with nodes marked with logic variables $U_j \in \tilde{U}$ and operator variables $A_i \in \tilde{A}$. Let IS be information set on which the set of operators Op and the set of predicates Pr is defined. Operators are functions taking values from IS , and predicates are logic functions taking values from the set of logic values

$E_2 = \{0, 1\}$, where 0 is a false value, 1 is a true value. To each logic variable $U_j \in \tilde{U}$ in the flowgraph G , the basic predicate $u \in Pr$ is assigned and to each operator variable $A_j \in \tilde{A}$ some basic operator $B \in Op$ is assigned. The considered assignment is called the *interpretation* of the flowgraph G . The flowgraph G is called *interpreted* and is designated as $G/\hat{I}$, where $\hat{I} \subset Pr \cup Op$.

Let us describe the process of moving along the nodes of the interpreted flowgraph $G/\hat{I}$ during the processing of data, arriving at the input of the given flowgraph. Let $m \in IS$ be the element arriving at the entry node of the interpreted flowgraph $G/\hat{I}$. At passing the nodes of this flowgraph, according to a direction of edges, the element m is changed under the influence of operators corresponding to A -nodes. Let on some step the element m is transformed to $m' \in IS$. We will consider the following two cases.

1. The element m arrives at the input of A -node assigned with the operator $A \in \hat{I}$ accordingly to the given interpretation. Then the element $m'' = A(m')$ continues to move along one edge outgoing from the given A -node, $m'' \in IS$.

2. The element m arrives at the input of a recognizer assigned with the predicate $u \in \hat{I}$ according to selected interpretation. In this case, m moves without changes along one of the edges. If $u(m') = 1$, then it moves along the plus edge, if $u(m') = 0$, then along the minus edge.

If during the functioning of the flowgraph $G/\hat{I}$, the element $m \in IS$ arrived at the entry node, was transformed to $\tilde{m} \in IS$ and reached the exit node, then $\tilde{m}$ is regarded as the result of application of the algorithm represented by the interpreted flowgraph $G/\hat{I}$. This fact is designated as $\tilde{m} = G/\hat{I}(m)$.

Example 2.4. Let us illustrate the process of functioning of an interpreted flowgraph on the bubble sorting of an array. As the interpretation, the elementary (basic) operators and predicates de-

finer earlier in Examples 2.2, 2.3 are chosen. The interpreted flowgraph *Bubble* is shown in Fig. 2.2.

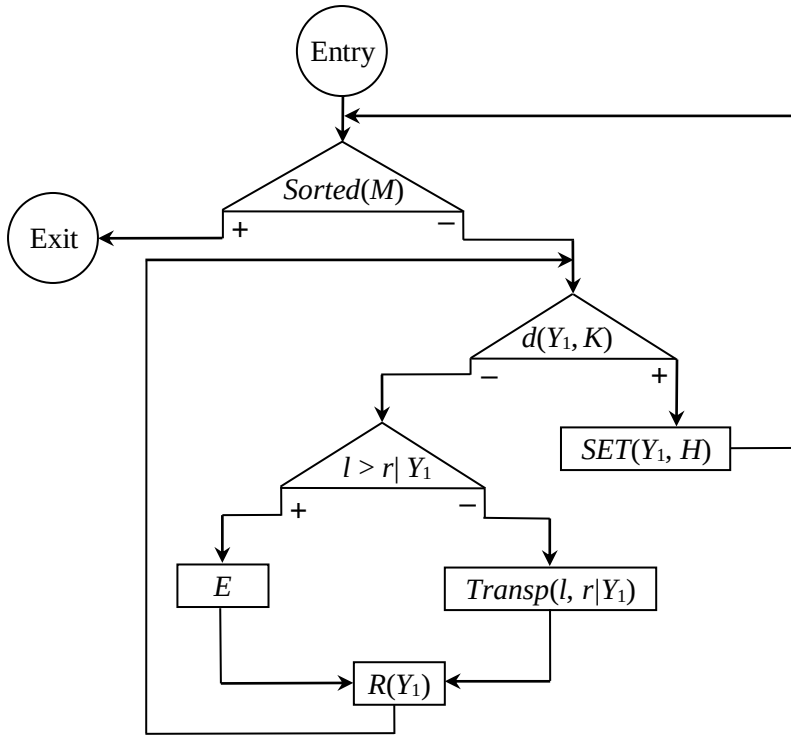


Fig. 2.2. The flowgraph of bubble sort algorithm

Let us describe moving along the nodes of the given flowgraph at sorting the array

$M : H \ Y_1 \ 5 \ 2 \ 3 \ 1 \ 4 \ K.$

The given array arrives at the input of the flowgraph and then the value of the predicate *Sorted(M)* is checked. Since the array is unsorted (the value of the condition *Sorted(M)* is false), it

moves along the minus edge to the recognizer of the condition $d(Y_1, K)$. Because $d(Y_1, K) = 0$, the M array comes along the minus edge to the recognizer which checks if the pair of elements l and r next to Y_1 are ordered. Since the marker H is less by definition than any element of the array, the array M moves along the minus edge of the recognizer $l > r | Y_1$ and arrives at the operator node E with the subsequent shift $R(Y_1)$. Then on a cyclic edge, the predicate $d(Y_1, K)$ is checked again and further, the value of the predicate $l > r | Y_1$ is checked also. Because the value of the predicate $l > r | Y_1$ is true ($5 > 1$), the operator $Transp(l, r | Y_1)$ is executed, which transposes the elements l and r , and then the operator $R(Y_1)$ shifts the pointer Y_1 over the array by one element to the right. The nested loop is executed until Y_1 comes to the marker K , which means that the maximum element 5 has “floated up” and the first phase of sorting is completed.

The state of the array after the first phase is the following: $M: H \ 2 \ 3 \ 1 \ 4 \ 5 \ Y_1 \ K$. After the application of the operator $SET(Y_1, H)$, the pointer Y_1 comes back to the beginning with the transition to the next phase of scanning, on which the element 3 “floats up”. Further, we pass to a finishing phase of sorting, where element 2 “floats up”: $M: H \ 1 \ 2 \ 3 \ 4 \ 5 \ Y_1 \ K$. Finally, after the return of Y_1 to the beginning, the sorted file is obtained: $M: H \ Y_1 \ 1 \ 2 \ 3 \ 4 \ 5 \ K$.

Let the flowgraphs G and G' be defined on the same set of operator and predicate variables $V = \tilde{U} \cup \tilde{A}$. We fix in the flowgraph G all the occurrences of one of the operators $A_j \in \tilde{A}$ and substitute the flowgraph G' instead of each such occurrence. As a result, the new flowgraph $G'' = G(A_j \Rightarrow G')$ is obtained, which is the *superposition* of flowgraphs G and G' . In [159], the concept of a generalized flowgraph was introduced and the algebra of such flowgraphs, with signature consisting of operation of their superposition, was constructed.

Kaluzhnin algebra is the two-sorted algebra

$$KA = \langle \{Pr, Op\}; \Omega_{KA} \rangle,$$

where Pr is the set of logic conditions represented in graph form; Op is the set of operator flowgraphs; Ω_{KA} is the signature of operations, which includes: Boolean operations; operation of sequential execution of operators, represented by means of an edge connecting operator nodes of a flowgraph; unary operation of flowgraph restructuring, which consists in switching of some edges, outcoming from an operator node or a recognizer, to some other node of the flowgraph with the subsequent elimination of nodes unreachable from a root. The operation of the restructuring of a graph corresponds to the jump operation included in the signature of the Yanov algebra [159], focused on analytic representation of non-structured schemes of algorithms.

2.3. Glushkov system of algorithmic algebras and its modifications

The inclusion of prediction operation into the signature of Dijkstra algebra (see Subsection 2.1) leads to the Glushkov algorithmic algebras [5]. Like algebraic specifications in general, Glushkov SAA are focused on an analytical form of representation of algorithms and formalized transformation of these representations, in particular, for the purpose of optimization of algorithms by given criteria.

Glushkov SAA (or Glushkov algebra, GA) is the two-sorted algebra

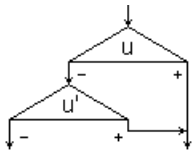
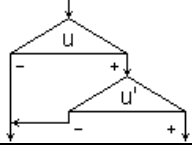
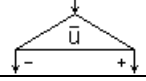
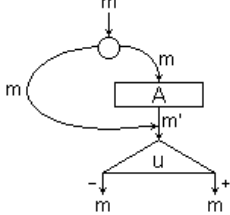
$$GA = \langle \{Pr, Op\}; \Omega_{GA} \rangle,$$

where sorts Pr and Op are sets of operators and logic conditions, accordingly, defined on the information set IS ; Ω_{GA} is the signature of operations. Operators represent mappings (probably, partial) of the information set to themselves, logic conditions are predicates, which

are defined on the set IS and take the values of three-valued logic $E_3 = \{0, 1, \mu\}$, where 0 is for false, 1 is for true and μ is for unknown. The signature $\Omega_{GA} = \Omega_1 \cup \Omega_2$ consists of the system Ω_1 of logic operations taking values in the set Pr , and the system Ω_2 of operations taking values in the set of operators Op .

The operations included in the signature of SAA and their modifications focused on multiprocessing, are given in Table 2.1.

Table 2.1. The main operations of SAA and their modifications

Type	Operation name	Representation form		
		Analytic	Natural language	Flowgraph
Logical	Disjunction	$u \vee u'$ $u + u'$	u OR u'	
	Conjunction	$u \wedge u'$ $u \cdot u'$	u AND u'	
	Negation	$\bar{u}$	NOT(u)	
	Prediction	$A \cdot u$	AFTER A CONDITION u	

Continuation of Table 2.1

Type	Operation name	Representation form		
		Analytic	Natural language	Flowgraph
Operator	Composition	$A * B$	A THEN B $A ; B$	
	Alternative (branching)	$([u] A, B)$	IF u THEN A ELSE B END IF	
	Loop	$\{[u] A\}$	WHILE NOT u LOOP A END OF LOOP	
	Filtration	$\phi(u)$	FILTER(u)	
	Asynchronous disjunction	$A // B$	A PARALLEL B	
	Control point	$CP(u)$	CP u	
	Synchronizer	$S(u)$	WAIT u	

In the table, $u \in Pr$, $u' \in Pr$ are logic, $A \in Op$, $B \in Op$ are operator variables; $E \in Op$ is the identity operator; $W \in Op$ is an uncertain operator.

In SAA, the system of generators representing a finite functionally complete set of operators and logic conditions, is fixed. By means of this set and superpositions of operations included in Ω_{GA} , any operators and logic conditions from the sets Pr and Op are generated.

Logic operations of the system $\Omega_1 \subset \Omega_{GA}$ are generalized Boolean operations: the disjunction $u \vee u'$, the conjunction $u \wedge u'$, the negation $\bar{u}$, defined by the truth tables (see Table 2.2), and also the prediction operation $A \cdot u$ (left multiplication of the condition u by operator A), which consists in checking the value of the predicate u after the execution of operator A .

Table 2.2. The truth tables for disjunction $u \vee u'$, conjunction $u \wedge u'$ and negation $\bar{u}$.

$\vee$	0	μ	1
0	0	μ	1
μ	μ	μ	1
1	1	1	1

$\wedge$	0	μ	1
0	0	0	0
μ	0	μ	μ
1	0	μ	1

u	$\bar{u}$
0	1
μ	μ
1	0

Consider the operations included in the signature $\Omega_2 \subset \Omega_{GA}$, which generate operators from the set Op in SAA.

The *composition* $A * B$ is a binary operation defined on the set Op , consisting in sequential application of operators $A \in Op$, $B \in Op$.

The *alternative (branching)* $([u] A, B)$ is a ternary operation, which depends on the condition $u \in Pr$ and the operators $A \in Op$, $B \in Op$. This operation generates the operator $C = ([u] A, B)$, such that

$$C(m) = \begin{cases} A, & \text{if } u(m) = 1, \\ B, & \text{if } u(m) = 0, \\ W, & \text{if } u(m) = \mu. \end{cases}$$

for any $m \in IS$.

The *loop* $\{[u] A\}$ is a binary operation, which depends on the condition $u \in Pr$ and the operator $A \in Op$. The essence of the given operation consists in cyclic application of the operator A at false value of u until the condition u is true.

Consider also some derivative operations included in the signature $\Omega_2 \subset \Omega_{GA}$.

The *for loop* operation $\text{For}(var, start_expr, end_expr)\{A\}$ is applied to iterate over a range of values of the variable var ; $start_expr$ to end_expr are numerical expressions, which define the initial and final values of the variable var . This operation sequentially executes the operator $A \in Op$ for all the values of var in the interval $[start_expr, end_expr]$.

The *switch* operation $\text{SELECT}(u_1 \rightarrow A_1, \dots, u_n \rightarrow A_n)$ executes one of the operators A_i ($i = 1, \dots, n$) if the value of the corresponding condition u_i is true, and then breaks computation without checking the values of other conditions.

Let us fix the basis $Z \subset Pr \cup Op$ of SAA.

The superposition of operations included in the signature Ω_{GA} and the elements of basis Z , representing the compound operator (algorithm) $F/Z \in Op$ in SAA is called an interpreted *regular scheme* (RS) F/Z .

Theorem 2.1. Any algorithm A (including a program or a microprogram) can be represented by means of interpreted RS F/Z in GA , $A = F/Z$ [5].

Thus, GA coincides by a generic capacity with well-known algorithmic systems.

Example 2.5. Let us illustrate the essence of the prediction operation $A \cdot u$ at computing the condition $Sorted(M)$ included in the algorithm scheme *Bubble* (see Subsection 2.1, Example 2.3). Such checking of the value of the predicate $Sorted(M)$ has to be done in the process of array sorting, without introducing additional pointers. For this purpose, we use the prediction operation in the scheme of the predicate:

$$Sorted(M) = Scan(M) \cdot [d(Y_1, K)],$$

where $Scan(M) = \{[d(Y_1, K) \vee (l > r | Y_1)] R(Y_1)\}$.

When the array M enters the predicate $Sorted(M)$, the elements of the array are scanned by moving Y_1 in the direction from left to right until Y_1 reaches the marker K or an unordered pair l, r is found. After the completion of this loop, the value of the predicate $d(Y_1, K)$ is checked up and it is assigned to the predicate $Sorted(M)$, and then Y_1 goes back to the marker H .

The *modified SAA* (SAA-M) [64] with an extended signature of operations Ω'_{GA} is focused on formalization of parallel computations. Besides the above-listed constructs included in the signature Ω_{GA} , the signature Ω'_{GA} contains the following operations.

The *filtration* $\varphi(u)$ is a unary operation, which depends on the condition $u \in Pr$ and generates the filter operators $\varphi(m) \in Op$, such that

$$\varphi(m) = \begin{cases} E, & \text{if } u(m) = 1; \\ W, & \text{otherwise,} \end{cases}$$

for any state $m \in IS$. Filters provide flexible control of computing process. Thus computation at branches with true filtering conditions proceed, whereas other branches of computation break.

The *synchronous disjunction* $A \vee B$ is a binary operation on the set Op , consisting in simultaneous application of operators A and B .

The *asynchronous disjunction* $A // B$ is the operation consisting in parallel execution of operators A and B . For synchronization of parallel processes, control points and synchronizers are used.

The *control points* $CP(u)$ are fixed positions between operators in an algorithm scheme [5]. Each control point $CP(u)$ is associated with the condition u , which is false while a computation process has not yet reached the point $CP(u)$, and true from the moment of achievement of the given point and uncertain in the presence of emergency stops on a path which conducts to the point $CP(u)$. The condition u is called a *synchronization condition* associated with the point $CP(u)$.

The delay of computation in schemes of algorithms is implemented by means of *synchronizers*. The synchronizer is defined by means of the loop $S(u) = \{[u]E\}$, where u is a logic function which depends on synchronization conditions associated with certain control points of a scheme. The synchronizer located in a certain place of a regular scheme carries out a delay of computation in the given place of the scheme up to the moment when its synchronization condition u is true.

A representation of an algorithm in SAA-M, which specifies asynchronous operator interactions, is called a *parallel regular scheme* (PRS).

Example 2.6. Consider the parallel algorithm of two-way bubble sort $PBubble$, obtained as a result of parallelization of the corresponding sequential sort scheme $Bubble$ (see Subsection 2.1, Example 2.3). Marking of the array to be sorted is the following:

$$M : H \ Y_1 \ a_1 \ a_2 \ \dots \ a_n \ Y_2 \ K,$$

where H and K are the markers fixing the beginning and the end of the array M ; Y_1 and Y_2 are the pointers moving over M towards each other. The PRS of the algorithm is the following:

$$PBubble = SET(Y_1, H) * SET(Y_2, K) * \\ * \{[Sorted(M)] INIT_CP * (Bubble(1) // Bubble(2))\},$$

$$Bubble(1) = \{[d(Y_1, K)] TRANSP_L\} * SET(Y_1, H) * \\ * CP(PROC_FIN(1)) * S(PROC_FIN(2)),$$

$$Bubble(2) = \{[d(Y_2, H)] TRANSP_R\} * \\ * ([d(Y_1, Y_2) = 1] S(d(Y_1, Y_2) = 0), E)\} * \\ * CP(PROC_FIN(2)) * S(PROC_FIN(1)),$$

$$TRANSP_L = ([l > r | Y_1] Transp(l, r | Y_1), E) * R(Y_1),$$

$$TRANSP_R = ([l > r | Y_2] Transp(l, r | Y_2), E) * L(Y_2).$$

Besides the basic operators and predicates, introduced in the previous examples, the scheme *PBubble* contains the following additional basic and compound elements:

- *SET*(Y_2, K) is placing the pointer Y_2 in a position directly to the left of the marker K ;
- *INIT_CP* is the operator initializing the synchronization conditions *PROC_FIN*(1) and *PROC_FIN*(2) with false value;
- *Bubble*(1) and *Bubble*(2) are compound operators representing the contradirectional processes of parallel bubble sort;
- *TRANSP_L* is the compound operator, which transposes the elements adjacent to the pointer Y_1 in case they are unordered and shifts Y_1 by one element to the right;
- *CP*(*PROC_FIN*(i)), where i is the control point fixing the moment of completion of computation in the branch and setting the synchronization condition to true value;

- $S(PROC_FIN(i))$, where $i=1,2$, is the synchronizer implementing the delay of computation up to the moment of completion of computation in the branch ;
- $PROC_FIN(i)$, where $i=1,2$, is the predicate, which takes the true value if the branch $Bubble(i)$ completed processing, and false otherwise;
- $d(Y_2, H)$ is the predicate, which takes the true value if Y_2 reached the marker H , and false otherwise;
- $TRANSP_R$ is the compound operator, which transposes the elements adjacent to the pointer Y_2 in case they are unordered and shifts Y_2 by one element to the left;
- $l > r | Y_2$ is the predicate, which takes the true value if the array element located directly to the left of the pointer Y_2 is greater than the element located directly to the right of Y_2 ;
- $Transp(l, r | Y_2)$ is the operator of transposition of elements adjacent to the pointer Y_2 ;
- $L(Y_2)$ is the shift of the pointer Y_2 by one element to the right over the array;
- $d(Y_1, Y_2)=1$ is the predicate, which takes the true value if there is one array element between the pointers Y_1 and Y_2 ;
- $S(d(Y_1, Y_2)=0)$ is the synchronizer implementing the delay of computation until the condition $d(Y_1, Y_2)=0$ (see below) is true;
- $d(Y_1, Y_2)=0$ is the predicate, which takes the true value if Y_1 reached the pointer Y_2 , and false otherwise.

The essence of the scheme consists in conjoint functioning of contradirectional processes and of bubble sort, which process the array at adverse sides. The compound operators implement the conditional transposition of adjacent elements of the array and shift of the pointers and to the right and to the left, correspondingly. When the pointers reach the critical state (the condition is true) the process turns into a waiting state up to the moment when the pointers merge

(the condition is true), whereas continues to work. Then both processes reach the opposite ends of the array. The described loop repeats again up to the fulfillment of the condition of the orderliness of the array, and after the pointers and reach the opposite ends of the array, the algorithm completes its execution.

2.4. Algebra of algorithmics

The *algebra of algorithmics* (AA) [5, 159] is based on the above considered Dijkstra, Glushkov and Kaluzhnin algebras, and is focused on solving the problems of formalization, substantiation of correctness and transformation of algorithms in various subject domains.

AA is the two-level system. The theory of clones [1, 5] is the basis of the first (top) level. The algorithmic clone is a two-sorted system

$$C_A = \langle \{L(2), Op\}; SUPER \rangle,$$

where $L(2)$ is a set of all Boolean functions; Op is the operator base consisting of a set of noninterpreted operator schemes; $SUPER$ is the signature which contains only the operation of superposition of operations.

The selection of the system of generators of C_A determines the system of algorithmic constructions typical for a technique of designing algorithms and programs. So, for the structured programming method, such a system consists of constructions included in the signature of operations of Dijkstra algebra (see Subsection 2.1).

At the top level of AA , the construction of the demanded algebra of algorithms from the family specified by the corresponding clone is carried out depending on a problem being solved, the chosen method of development of algorithm classes, the technological environment of programming. The signature of the constructed algebra satisfies the theorem of functional completeness for the clone and is the basis of the clone. The constructed algebra of algorithms can be analyzed for the purpose of axiomatization, the development of op-

timizing transformations, canonization of analytical representations, etc.

At the second (lower) level of *AA*, the development of quite specific applied algebra of algorithms, focused on representation of algorithmic knowledge of the chosen subject domain, is carried out. The scheme of development of the applied algebra of algorithms is based on interpretation of elementary operator and predicate variables for the algebra of algorithms constructed at the top level. In turn, the development of sets of interpretations is concerned with construction of many-sorted algebraic systems, associated with abstract data types, classification and inheritance mechanisms.

A many-sorted algebraic system (MAS) is the system

$$S_A = \langle \{D_i | i \in I\}; \Omega_{pr}, \Omega_{op} \rangle,$$

where D_i are bases; Ω_{pr} and Ω_{op} are sets of predicates and operations (correspondingly), defined on the bases D_i .

If $\Omega_{op} = \emptyset$ ($\Omega_{pr} = \emptyset$), we come to the definition of the many-sorted model (algebra, correspondingly). Thus, MAS is a generalization of concepts of a model and an algebra. If bases D_i are interpreted as sets of data being processed, then MAS is a formalization of a concept of an abstract data type (ADT), which is widespread in connection with development of object-oriented programming methods and tools.

MAS can be used for interpretation of logical schemes of algorithms, given at a top level in an algebraic system or some of its subalgebra.

Let $DA = \langle \{Pr, Op\}; \Omega_{DA} \rangle$ be Dijkstra algebra with the system of generators $\Sigma = \Sigma_{pr} \cup \Sigma_{op}$, where $\Sigma_{pr} \subset Pr$ is the set of basic predicates, $\Sigma_{op} \subset Op$ is the set of basic operators.

We introduce the mapping $g: V_{pr} \rightarrow \Omega_{pr}$ and $f: V_{op} \rightarrow \Omega_{op}$, where $V_{pr} = \{A_i | i = 1, \dots, n\}$ are predicate variables; $V_{op} = \{u_j | j = 1, \dots, m\}$ are operator variables; Ω_{pr} and Ω_{op} are com-

pound components of the signature $\Omega_{S_A} = \Omega_{Pr} \cup \Omega_{Op}$ of some MAS $S_A = \langle \{D_i | i \in I\}; \Omega_{S_A} \rangle$.

Example 2.7. We fix the two-sorted algebraic system

$$MAS1 = \langle \{arr, sarr\}; \Omega_{Pr}, \Omega_{Op} \rangle,$$

where $arr = \{M_q \mid q \in 1, \dots, p\}$ is a set of marked arrays of the type

$$M_q : H \ Y_1 \ a_1 \ a_2 \ \dots \ a_n \ K,$$

$sarr = \{R_t \mid t = 1, \dots, r\}$ is a set of marked arrays of the type

$$R_t : H \ Y_1 \ s_1 \ s_2 \ \dots \ s_k \ K,$$

where $s_i \leq s_{i+1}$ for any $i = 1, 2, \dots, k-1$.

The subsystem Ω_{Pr} includes the predicate $d(Y_1, K)$, which takes the true value, if the pointer Y_1 reached the marker K (which fixes the end of the array M_q), and takes the false value otherwise. The subsystem Ω_{Op} includes the following operator operations: $SET(Y_1, H)$ is the operator placing the pointer Y_1 at the position directly to the right of the marker H (which fixes the beginning of the array); $R(Y_1)$ is a shift of the pointer Y_1 by one element to the right in the array $M_q \in arr$.

Consider the scheme P_1 , which is represented in DA and depends on predicate variables u_1, u_2 and operator variables A_1, A_2, A_3 :

$$P_1 = \{[u_1] ([u_2] A_1 * A_2, A_3)\}.$$

We build the mappings g_1 and f_1 , such that

$$g_1(u_1) = d(Y_1, K); \quad f_1(A_2) = SET(Y_1, H); \quad f_1(A_2) = R(Y_1).$$

As a result of application of the given mappings to the scheme P_1 , we obtain the scheme

$$MP = \{[d(Y_1, K)] ([u_2] A_1 * SET(Y_1, H), R(Y_1))\},$$

which presents the multiphase processing of the arrays $M_q \in arr$.

The construction of mappings g and f is carried out level by level in accordance with the conception of the top-down design of algorithms and their classes. Thus, continuing the construction of interpreting functions g_1 and f_1 , on the basis of the strategy MP , we can obtain, for example, an algorithm of array sorting.

Example 2.8. Into the signature MAS , we include the predicate $(l > r | Y_1) \in \Omega_{pr}$ which takes the true value if the condition $l > r$ is satisfied for elements of the array $M_q \in arr$, which are directly to the left and to the right of the pointer Y_1 , and also the operator $Transp(l, r | Y_1) \in \Omega_{op}$, which carries out the transposition of the elements l and r at the pointer Y_1 in the array M_q . By continuing the definition of the mappings g_1 and f_1 , considered in Example 2.7, so that

$$g_1(u_2) = l > r | Y_1; \quad f_1(A_1) = Transp(l, r | Y_1),$$

we obtain the algorithm

$$Solute = \{[d(Y_1, K)] ([l > r | Y_1] Transp(l, r | Y_1) * SET(Y_1, H), R(Y_1))\},$$

which implements sorting of the arrays $M_q \in arr$ using the *Solute* method [159].

Thus, in Examples 2.7 and 2.9 we illustrated the process of level-by-level specification of the scheme P_1 for the purpose of obtaining the strategy of multiphase processing MP and then the sort algorithm. By switching from one algorithm to a family of algorithms, related to a selected subject domain, we extend the system of generators due to successive addition of operators and predicates. As a result of such extension, the monotone increasing sequence of subalgebras SDA_i of the algebra DA is obtained. In the process of extension of the system of generators, we continue to define the interpreting functions g and f , and thus fixing the values of operator and predicate variables, included in the system of generators DA . As a result, we obtain the monotone decreasing sequence of subalgebras SA_i , such that $SA_i \supset SDA_i$ for each $i=1, 2, \dots, k$. We will call the subalgebra SA_i a superclass, and a subalgebra SDA_i will be called a subclass at designing the many-sorted ADT.

Theorem 2.2. Let $MAS1$ be a many-sorted algebraic system, which represents the designed ADT and $\{(SA_i, SDA_i) \mid i=1, 2, \dots, k\}$ is a set of pairs (superclass, subclass) associated with the process of ADT designing, then

$$SA_1 \supset SA_2 \supset \dots \supset SA_k; SDA_1 \supset SDA_2 \supset \dots \supset SDA_k$$

and

$$MAS = \bigcap_{i=1}^k SA_i = \bigcup_{i=1}^k SDA_i.$$

The considered conception of the two-level algebraic system corresponds to the process of a level-by-level designing of classes of algorithms and programs. The basis of the mentioned system is the apparatus of schemes of algorithms and MAS, which is polymorphic and allows flexible reorientation depending on a selected subject domain.

The transition from the superclass SA_i to the subalgebra $SDA_i \subset SA_i$ of lower level at designing the monotone decreasing se-

quences of subalgebras consists in redefining the interpreting functions g and f . In the process of such a transition, the subclass SDA_i inherits the interpretations of the superclass SA_i .

The presence of the transformation facilities developed within the framework of the algebra of algorithmics [159] gives the possibility of conversion of algorithms and programs, in particular, their optimization by selected criteria.

The algebra of algorithmics is the basis of the algebra-grammatical and algebra-dynamic models and methods considered in Chapter 5 and also the toolkit for design and synthesis of programs considered in Chapter 6.

Control questions and exercises

1. Name the operations included in the signature of Dijkstra algebra.
2. What operator representations in Dijkstra algebra are called strategies?
3. Build the representation of the sort algorithm *Solute* (see Subsection 2.4, Example 2.8) in Dijkstra algebra using the do-while operation $\{A[u]\}$ instead of the loop $\{[u]A\}$.
4. What are the main types of flowgraph nodes in Kaluzhnin algebra?
5. Build the flowgraph of the sort algorithm *Solute* (see Subsection 2.4, Example 2.8).
Describe moving along the nodes of the flowgraph at sorting the array

$M : H Y_1 5 2 3 1 4 K.$

6. Build the scheme of an algorithm finding a maximum number in a marked integer array in Dijkstra algebra and draw a flowgraph of the algorithm.
7. What is the difference between Dijkstra and Glushkov algebra?
8. Name the operations included into the SAA-M signature. What differs SAA and SAA-M?

9. Build the regular scheme in SAA checking the presence of some number in a numerical array using the prediction operation.
10. What is the difference between synchronous and asynchronous processing?
11. What facilities are used for synchronization of parallel processes in SAA-M?
12. Build flowgraphs for each of the compound operators of the parallel sort algorithm *PBubble* (see Subsection 2.3, Example 2.6).
13. Demonstrate the process of sorting the array

$$M : H \ Y_1 \ 10 \ 9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1 \ Y_2 \ K$$

with the parallel sort algorithm *PBubble* (see Subsection 2.3, Example 2.6).

Chapter 3

Algebraic programming based on rewriting rules

One of the directions of algebraic programming is usage of rewriting rules technique. This direction formalizes the transformational aspects of programming, which allows to describe transformations of some formal objects and to research properties of such transformations. The rewriting rules technique is both a powerful formal tool for transformation of formal systems and a practical tool for programming, which allows implementing transformations of complex objects.

3.1. Aspects of rewriting: definitions and examples

A considerable quantity of formal systems using rewriting technique was proposed [13]. In this subsection, we will consider several such systems demonstrating various rewriting aspects.

3.1.1. Abstract rewriting systems. In this subsection, we use some definitions from the theory of sets and algebraic systems [13, 103].

Definition 3.1. The *binary relation* R on the sets A and B is a subset of the Cartesian product $R \subseteq A \times B$, i.e. the set of pairs $\{(a, b) \in R \mid a \in A, b \in B\}$. The fact that two elements $a \in A$ and $b \in B$ are in a relation R , we will denote as $aRb \sim (a, b) \in R$.

The binary relation on the set A is a subset of the Cartesian square of the set A , i.e. $R \subseteq A \times A$.

Since relations are sets, usual operations over sets are defined for them: a union $R \cup S$, an intersection $R \cap S$, a difference $R \setminus S$, a subset $R \subseteq S$.

The *product* (or *composition*) of relations $R \subseteq A \times B$ and $S \subseteq B \times C$ is the relation $R \circ S \subseteq A \times C$ such that $R \circ S = \{(a, c) \mid a \in A, c \in C, \exists b \in B : aRb \wedge bSc\}$.

The *identity relation* on the set A is the relation $I_A \subseteq A \times A$, such that $I_A = \{(a, a) \mid a \in A\}$.

The *inverse relation* for the relation $R \subseteq A \times B$ is the relation $R' \subseteq B \times A$, such that $bR'a \sim aRb$.

The *power of relation* $R \subseteq A \times A$ for any $n \in \mathbb{N}_0$ is defined as follows:

- 1) $R^0 = I$;
- 2) $R^1 = R$;
- 3) $\forall n \in \mathbb{N}: R^{n+1} = R^n \circ R$.

The relation $R \subseteq A \times A$ is called

- 1) *reflexive*, if $I \subseteq R$ (i.e. xRx);
- 2) *symmetric*, if $R' = R$ (i.e. yRx follows from xRy);
- 3) *transitive*, if $R^2 \subseteq R$ (i.e. xRz follows from xRy and yRz).

The *transitive closure* of the relation $R \subseteq A \times A$ is the relation $R^+ = \bigcup_{i \in \mathbb{N}} R^i$, i.e. the union of all positive powers of the relation.

The *reflexive transitive closure* of the relation $R \subseteq A \times A$ is the relation $R^* = I \cup R^+ = \bigcup_{i \geq 0} R^i$, i.e. the union of all non-negative powers of the relation.

The *symmetric closure* of the relation $R \subseteq A \times A$ is the relation $R^{sym} = R \cup R'$.

Generally, the *closure* of the set S over the property P is an inclusion-wise minimal set C , such that $S \subseteq C$ and for the C property P is satisfied. It is easy to check that the constructed transitive, reflexive transitive and symmetric closures satisfy this definition.

Any formal system implementing a rewriting paradigm includes the basic operation of rewriting, i.e. the transition between objects by certain rules. Formally this operation is described as follows.

Definition 3.2. The pair $\langle A, \rightarrow \rangle$, where A is any set and $\rightarrow \subseteq A \times A$ is any binary relation on this set is called an *abstract rewriting system*.

We are interested in transitions between the elements of the set A according to the relation $\rightarrow$, i.e. the chains of the form $a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n \rightarrow \dots$, such that $\forall i: a_i \rightarrow a_{i+1}$. With the usage of

designations from Definition 3.1, the chain can be written in the form $a_1 \rightarrow^* a_n$. ARS can be considered as a generalization of a function: in the case of a function, for each value of an argument, the transition to a single value of a function can be made, while in the case of ARS different transitions for a given initial element are possible (the indeterminacy of ARS).

The abstract rewriting system (ARS) can be considered in two aspects: static and dynamic. In the static aspect the elements of the set A are represented in the form of points of some space, and the chains $a \rightarrow^* b$ describe the paths between these points. In the dynamic aspect, the elements of the set A are the states of some system, and the chains $a \rightarrow^* b$ describe transitions between the states. (The dynamic aspect of ARS is also called transition systems.) Both aspects are not mutually exclusive, it is possible to speak, for example, about a path between two states, or about transitions from a point to a point.

For the paths, it is natural to raise the question about “destination”, i.e. a final element of the path. In particular, the existence and uniqueness are of interest. To formalize these questions, we will introduce additional properties of separate elements of ARS as well as ARS as a whole.

Definition 3.3. The element b is called a *successor* of the element a , if $a \rightarrow b$. The element a is called *reducible* if it has a successor: $\exists x \in A: a \rightarrow x$. The element which is not reducible is called *irreducible*, or an *element in normal form*. The element b is called a *normal form* of the element a , if $a \rightarrow^* b$ and b is irreducible. If the element a has a unique normal form, it is designated $a \downarrow$.

At first, consider a question of the existence of a normal form.

Definition 3.4. ARS is called *normalizing* if for each element there is a normal form (at least one). ARS is called *terminating*, or *Noetherian*, if there are no infinite ways $a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n \rightarrow \dots$ in it.

It is obvious that terminating ARS is always normalizing. The reverse is incorrect: we will consider, for example, the system with two elements $A = \{a, b\}$, such that $a \rightarrow a$ and $a \rightarrow b$. In this case $a \Downarrow b$ and $b \Downarrow b$, i.e. both elements have a unique normal form. At the same time, there is an infinite chain $a \rightarrow a \rightarrow \dots \rightarrow a \rightarrow \dots$.

To guarantee that transitions terminate, it is necessary to forbid a part of transitions. In some cases, it can be made by finding and eliminating loops in transitions. We will consider a function of similarity $f : A \rightarrow A$, such that $f(x) \rightarrow f(y)$ follows from $x \rightarrow y$. An example of such function is $f(x) = x$, and also $f(x) = \text{setp}(a, p, x)$, i.e. some construct containing x as one of its parts (see Subsection 3.1.3). In this case, if for some element there is a path to a similar element: $\exists a \in A, \exists n > 0 : a \rightarrow^n f(a)$, then the system is not terminating: since $f(a) \rightarrow^n f(f(a))$, then $a \rightarrow^{2n} f(f(a))$ and in general $\forall k > 0 : a \rightarrow^{kn} f^k(a)$. At a practical implementation, the algorithm for searching a normal form should remember the “history”, i.e. all previously considered elements, and when a loop is found, it should stop the search in this direction.

Another method for ensuring that transitions terminate is the selection of a proper rewriting strategy (see Subsection 3.2.5).

The verification of the termination property of a system is an algorithmically unsolvable problem. However, the methods for proving the termination property for certain classes of systems were developed [13, 35] (see also Example 3.1, a).

Further, consider a question of the uniqueness of normal forms. The more general statement is the following question: if two paths beginning from one element have diverged, then do they converge later?

For research of this question, it is necessary to consider transitions not only in direct but also in reverse directions. For the relation $\rightarrow$ we will designate the inverse relation as $\leftarrow \Rightarrow'$. We will designate the symmetric closure $\leftrightarrow \Rightarrow \rightarrow \cup \leftarrow$. Then the paths with transitions in any directions are defined by the relation $\leftrightarrow^*$ which is a symmetric reflexive transitive closure of the relation $\rightarrow$. We will

designate $C \Rightarrow^* \circ \Leftarrow^*$ (convergent paths) and $D \Leftarrow^* \circ \rightarrow^*$ (diverging paths). We will consider also two relations defining special cases of diverging paths: $D_s \Leftarrow \circ \rightarrow^*$ and $D_l \Leftarrow \circ \rightarrow$. It is obvious that

$$C \subseteq \Leftrightarrow^* \quad (3.1)$$

$$D_l \subseteq D_s \subseteq D \subseteq \Leftrightarrow^* \quad (3.2)$$

Generally, the subset relations are strict, as an example $a \rightarrow b \Leftarrow c \rightarrow d$ shows: here aCc , bDd and $a \Leftrightarrow^* d$, but not aCd and not aDd .

Definition 3.5. ARS has a *Church-Rosser property*, if $\Leftrightarrow^* \subseteq C$ (from (3.1) follows that this property is equivalent to $\Leftrightarrow^* = C$). Otherwise, this property can be written as $\forall n, p: n \Leftrightarrow^* p \Rightarrow \exists q: n \rightarrow^* q \Leftarrow^* p$.

ARS is called *confluent*, if $D \subseteq C$, or $\forall m, n, p: n \Leftarrow^* m \rightarrow^* p \Rightarrow \exists q: n \rightarrow^* q \Leftarrow^* p$.

ARS is called *semi-confluent*, if $D_s \subseteq C$, or $\forall m, n, p: n \Leftarrow m \rightarrow^* p \Rightarrow \exists q: n \rightarrow^* q \Leftarrow^* p$.

ARS is called *locally confluent*, if $D_l \subseteq C$, or $\forall m, n, p: n \Leftarrow m \rightarrow p \Rightarrow \exists q: n \rightarrow^* q \Leftarrow^* p$.

ARS is called *convergent*, if it is terminating and confluent.

Theorem 3.1. The following statements are equivalent:

- 1) ARS has a Church-Rosser property;
- 2) ARS is confluent;
- 3) ARS is semi-confluent.

The proof. We will prove the equivalence according to the scheme $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1$. We will notice that from (3.2) follows $1 \Rightarrow 2 \Rightarrow 3$, therefore it is necessary to prove only $3 \Rightarrow 1$, i.e. $D_s \subseteq C \Rightarrow \Leftrightarrow^* \subseteq C$.

For notation simplification, we will designate $\rightarrow = R, \Leftarrow = L, \Leftrightarrow = B$, then $B = L \cup R, C = R^* L^*, D_s = LR^*$. Let $D_s \subseteq C$. We will prove by induction on n that $\forall n \geq 0: B^n \subseteq C$.

The induction basis: it is obvious that at $n=0$ the $B^0 = I \subseteq C$ is satisfied.

The induction step: let $B^k \subseteq C$. We will notice that $B^{k+1} = B^k B = B^k (L \cup R) = B^k L \cup B^k R$.

Let us prove that $B^k L \subseteq C$. Indeed, $B^k L \subseteq CL = R^* L^* L \subseteq R^* L^* = C$.

Let us prove that $B^k R \subseteq C$, in this case, it will be easier to prove the equivalent inclusion for inverse relations $(B^k R)' \subseteq C' = C$. So,

$$\begin{aligned} (B^k R)' &= R'(B^k)' = LB^k \subseteq LC = LR^* L^* = \\ &= D_s L^* \subseteq CL^* = R^* L^* L^* = R^* L^* = C. \end{aligned}$$

From the proved inclusions $B^k L \subseteq C$ and $B^k R \subseteq C$ follows $B^k L \cup B^k R \subseteq C$, as it was required. The theorem is proved.

Due to Theorem 3.1, further we will speak only about confluent ARS. The following statement holds.

Theorem 3.2. If ARS is normalizing and confluent, there is a unique normal form for each element.

The proof. The existence of the normal form follows from that ARS is normalizing. We will admit that for the element a there are two normal forms b, c . Then from $a \rightarrow^* b$ and $a \rightarrow^* c$ follows bDc , from which bCc follows due to the confluence. But as b, c are not reducible, from here follows $b = c$. The uniqueness of the normal form is proved.

Thus, the confluence allows calculating the normal form, not caring about a choice of successors on each step: irrespectively of this choice, the unique normal form will be achieved. Therefore this property is important enough for rewriting systems. Unfortunately, it is difficult enough to check: for all the set of pairs of elements which are in relation $\leftrightarrow^*$ (either D , or D_s) it is necessary to check up that corresponding paths converge.

It is easier to verify a property of local confluence, as it unites only relations $\rightarrow$ and $\leftarrow$, but not their transitive closures. Unfortunately, in general case, this property is not equivalent to confluence as the examples $\{0 \leftrightarrow 1, 0 \rightarrow a, 1 \rightarrow b\}$ or $\{2n \rightarrow a, 2n+1 \rightarrow b, n \rightarrow n+1 | n \geq 0\}$ show (Fig. 3.1).

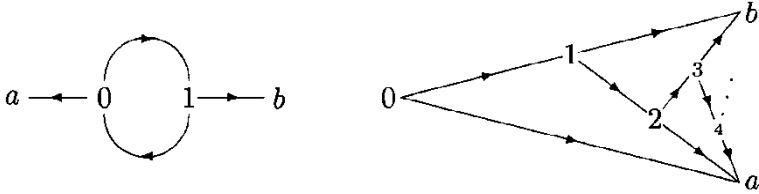


Fig. 3.1. The confluence does not follow from the local confluence [13].

However, in the case of a terminating system, the following statement hold.

Theorem 3.3 (Newman's lemma). If ARS is terminating and locally confluent it is confluent.

See the proof in [13, 79].

Below, some examples of ARSs are given.

Example 3.1. a) Let the relation $\rightarrow$ be the relation of a strict partial order, i.e. the properties of transitivity and antisymmetry ($\rightarrow \cap \leftarrow = \emptyset$) are satisfied for it. Then the existence of a normal form for the element $a \in A$ corresponds to the existence of a minimum of the set $\{x \in A : a \rightarrow^* x\}$. In particular, if A is a well-ordered set, the minimum (i.e. the unique normal form) always exists.

Order relations can be used for the proof of a termination property of a system. For this purpose, the given system $(A, \rightarrow)$ is immersed into the well-ordered set $(B, >)$ by means of monotonous function $f : A \rightarrow B$, such that from $x \rightarrow y$ follows $f(x) > f(y)$. If such function exists, from the existence of the infinite chain $a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n \rightarrow \dots$, the infinite chain $f(a_1) > f(a_2) > \dots > f(a_n) > \dots$ follows, which contradicts the defini-

tion of the system $(B, >)$. Some strategies of construction of such immersing are given in [13, 35].

b) Let the relation $\rightarrow$ be the equivalence relation (symmetric, reflexive and transitive). Then the confluence is satisfied, but the system is not terminating, the existence of infinite chains $a \rightarrow a \rightarrow \dots \rightarrow a \rightarrow \dots$ follows from the reflexivity.

c) We will consider the case when the relation of equivalence $\approx$ is given, but for its definition, the transitive closure of some other relation is used. For example, the set of equalities $a_i \approx b_i$ is given, and equivalent to the given element are those elements, which can be obtained from it by sequential application of any number of equalities. This is a statement of the equivalence problem, which is algorithmically unsolvable in general. However, if it will be possible to construct the convergent ARS $(A, \rightarrow)$, such that $a_i \leftrightarrow^* b_i$, then the means for checking the equivalence will be obtained: for the given elements it is necessary to calculate normal forms, and elements will be equivalent in the only case when their normal forms coincide. For construction of such ARS, it is possible to use, for example, various forms of Knuth-Bendix algorithm [94]. At the same time, in general case such algorithms do not guarantee the construction of demanded ARS, owing to algorithmic unsolvability of the equivalence problem.

d) Let the relation $\rightarrow$ describe the execution of some program, in this case the set of states of the program and admissible transitions in this set are considered. Then the termination property of the system provides a program stop and convergence provides the existence of well-defined unique result (the determinacy). The normal form for the given input data is the result of calculation.

3.1.2. Rewriting rules. In ARS, the binary relation $\rightarrow$ generally contains an infinite number of element pairs. Therefore its representation in a tabular form is inconvenient. We will consider a method of representation of the relation by means of a small number of rules.

Definition 3.6. Let A be a set of elements and S be a set of substitutions. Consider the following functions: $match: A \times A \rightarrow S$

and $subst: A \times S \rightarrow A$, and also the relation $ismatch \subseteq A \times A$. The pair of elements $(lhs_i, rhs_i) \in A \times A$ is called the *rule*.

Then the ARS $(A, \rightarrow)$ is defined as follows

$$a \rightarrow b \sim ismatch(a, lhs_i) \wedge b = subst(rhs_i, match(a, lhs_i)).$$

Thus, the relation is defined in three stages:

- 1) at first, the compatibility of the given element a and the left part of the rule lhs_i is checked;
- 2) then the substitution $s = match(a, lhs_i)$, which combines a and lhs_i is computed;
- 3) at last, the same substitution is used for the right part of the rule, and the element $b = subst(rhs_i, s)$ is computed.

In this case, the assigning of a finite number of rules allows defining an infinite relation.

Further, the rules will be designated in the form $lhs_i \rightarrow rhs_i$, when it is clear from the context that we deal with rules, not the relation $\rightarrow$ in ARS.

Example 3.2. a) The change of states of the Turing machine is performed according to a set of rules. The set of states consists of the pairs (q, s) , where q is the current state of the Turing machine, s is a current symbol on a tape, which is read out by a head. The substitution consists of the triples (q, s, a) , where q and s are new values of a state and a symbol on tape and a is one of the actions: to move to the right, to move to the left, to stop. The function *match* finds in the table the necessary substitution for the given state. The function *subst* modifies a state according to the substitution: changes a symbol on tape and an internal state, and also moves a head on tape and accordingly changes a visible symbol. Thus, the infinite relation of transitions between different states can be presented as a final set of rules which in turn can be considered as input data for other programs (for example, the universal Turing machine).

b) In a similar way, it is possible to describe the other models of programs, such as various versions of automata, Petri nets, etc.

c) Logic languages, such as Prolog, use systems of implications as rules for derivation of new facts.

d) In many functional languages (ML, Haskell) there is a concept of algebraic data type, for example, $Maybe\ a = Some\ a \mid Nothing$ or $Either\ a\ b = Left\ a \mid Right\ b$. For dealing with such types, a set of rules is specified for each possible value of a type.

e) In contemporary programming languages (both object-oriented and functional), it is possible to consider the polymorphism as usage of a rule set for establishing the conformity between a name of a function (a method) and a called implementation.

f) Various inference systems, for example, used in expert systems, are based on rule sets IF ... THEN.

3.1.3. Rewriting of fragments. The notation of a rewriting relation in the form of rules works well for the sets consisting of rather simple elements which will be transformed completely. However, in many cases, ARS contains complex objects and rewriting touches only their separate fragments. For this case we will define an application of rewriting rules to fragments of elements.

Definition 3.7. Let A be a set of elements and P be a set of paths, or identifiers of fragments. Consider the following functions: $getp: A \times P \rightarrow A$ (obtaining fragment on a given path), $setp: A \times P \times A \rightarrow A$ (fragment replacement on the given path) and $listp: A \rightarrow P^*$ (the list of all paths to fragments of the given element). Let also the ARS $\langle A, \rightarrow \rangle$ be defined on the set A . Then it is possible to consider the ARS $\langle A, \rightarrow_p \rangle$ defined as follows:

$$\begin{aligned} a \rightarrow_p b &\sim \exists p \in P, \exists a_p, b_p \in A: \\ p \in listp(a) \wedge a_p &= getp(a, p) \wedge b_p = getp(b, p) \wedge b = \\ &= setp(a, p, b_p) \wedge a_p \rightarrow b_p. \end{aligned}$$

Thus, the relation $\rightarrow$ is applied to a fragment of the given element, and the result of rewriting is substituted instead of this fragment.

Notice that fragments can intersect. In particular, there is an empty path $\varepsilon \in P$ such that $getp(a, \varepsilon) = a$ and $setp(a, \varepsilon, b) = b$. Therefore $\rightarrow \subseteq \rightarrow_p$.

The choice of a fragment for rewriting also defines the non-determination of the transition.

Example 3.3. a) In the case of rewriting of strings, $A = \text{String}$, $P = \{a, b \in \mathbf{N}_0 : a \leq b\}$ is the position of a substring.

b) The special case of rewriting of strings is Markov normal algorithms.

c) One more special case is grammars used for description of languages.

d) Data storage systems, such as relational and non-relational databases, file systems, repositories of the type “key-value”, etc. can be considered in the form of rewriting systems. In this case, access to information is carried out through the operation $getp$, and addition, updating and removal through the operation $setp$.

3.2. Term rewriting

In this subsection, the formal model of a rewriting term system is considered.

3.2.1. The alphabet of the system. The system alphabet consists of the following symbols.

1. The set of constant symbols $\Sigma_c = \{c_i\}$. Constants belong to one of the primitive types

$$\Sigma_{pt} = \{\text{CHAR}, \text{INT}, \text{DECIMAL}, \text{BOOL}, \text{STRING}, \text{ATOM}\}.$$

The types CHAR, INT, DECIMAL, BOOL, STRING correspond to standard types of programming languages (symbols, integer values, decimal fractions, logic values, strings). The ATOM type contains atomic, not interpreted values. One of the atomic constants, NIL, has a special value in a system and designates an “empty” term.

2. The set of terminal symbols $\Sigma_t = \{t_i\}$.

3. The set of substitutable symbols (variable symbols) $\Sigma_x = \{x_i\}$.

4. The delimiter symbols: brackets “(, ” and a comma “,”.

3.2.2. Terms. The terms are basic objects, which the system deals with. Informally, terms can be defined as constructs of the form $f(t_1, \dots, t_n)$. At formal definition, we will distinguish two kinds of terms: specific (not containing variables or substitutable symbols) and substitutable (containing variables).

The set of specific terms T_c is defined recursively as follows:

- 1) $\forall c \in \Sigma_c, c \in T_c$;
- 2) $\forall f \in \Sigma_f, \forall t_1, \dots, t_n \in T_c, f(t_1, \dots, t_n) \in T_c$.

The set of substitutable terms T_v is defined similarly, with the addition of variable symbols:

- 1) $\forall c \in \Sigma_c, c \in T_v$;
- 2) $\forall x \in \Sigma_x, x \in T_v$;
- 3) $\forall f \in \Sigma_f, \forall t_1, \dots, t_n \in T_v, f(t_1, \dots, t_n) \in T_v$.

It can be seen from the definition that the set of substitutable terms extends the set of specific terms: $T_c \subset T_v$. The set $T_{fv} = T_v \setminus T_c$ will be called the set of terms with free variables.

3.2.3. Operations on terms. Below we introduce some natural operations for the terms defined above. We will formulate the definitions of operations for the set of substitutable terms T_v , similar definitions in some cases are also applicable for specific terms from the subset $T_c \subset T_v$.

At first, we will define the elementary properties of terms: a name, an arity and a subterm.

1. The name: $name : T_v \rightarrow STRING$

- a) $\forall c \in \Sigma_c, name(c) = String(c)$;
- b) $\forall x \in \Sigma_x, name(x) = String(x)$;
- c) $\forall f \in \Sigma_f, \forall t_1, \dots, t_n \in T_v, name(f(t_1, \dots, t_n)) = String(f)$.

2. The arity: $arity : T_v \rightarrow INT$

- a) $\forall c \in \Sigma_c, arity(c) = 0;$
- b) $\forall x \in \Sigma_x, arity(x) = 0;$
- c) $\forall f \in \Sigma_t, \forall t_1, \dots, t_n \in T_v, arity(f(t_1, \dots, t_n)) = n.$

3. The subterm (of the first level): $subterm : T_v \times INT \rightarrow T_v$

- a) $\forall c \in \Sigma_c, \forall j \in INT, subterm(c, j) = NIL;$
- b) $\forall x \in \Sigma_x, \forall j \in INT, subterm(x, j) = NIL;$
 $\forall f \in \Sigma_t, \forall t_1, \dots, t_n \in T_v, \forall j \in INT,$
- c) $subterm(f(t_1, \dots, t_n)) = \begin{cases} 0, & j < 1; \\ t_j, & 1 \leq j \leq n; \\ 0, & j > n. \end{cases}$

Besides the subterms of the first level, the subterms of arbitrary levels will also be needed. Such subterms are defined by the path, i.e. the sequence of integers — the positions of subterms inside the term of higher level.

4. The set of paths: $subterm_paths : T_v \rightarrow INT^{**}$

- a) $\forall c \in \Sigma_c, subterm_paths(c) = \{\epsilon\}$ (only the empty sequence);
- b) $\forall x \in \Sigma_x, subterm_paths(x) = \{\epsilon\};$
 $\forall f \in \Sigma_t, \forall t_1, \dots, t_n \in T_v,$
- c) $subterm_paths(f(t_1, \dots, t_n)) =$
 $= \{\epsilon\} \cup \bigcup_{i=1}^n i \wedge subterm_paths(t_i).$

5. The subterm (of any level): $subterm^* : T_v \times INT^* \rightarrow T_v$

- a) $\forall t \in T_v, subterm^*(t, \epsilon) = t;$
- b) $\forall t \in T_v, \forall p = (p_1, \dots, p_k) \in subterm_paths(t), p \neq \epsilon,$
 $subterm^*(t, p) = subterm^*(subterm(t, p_1),$
 $(p_2, \dots, p_k));$

- c) $\forall t \in T_v, \forall p \notin \text{subterm_paths}(t),$
 $\text{subterm}^*(t, p) = \text{NIL}.$

6. The substitution of the subterm:
 $\text{subterm_replace} : T_v \times T_v \times \text{INT}^* \rightarrow T_v$

- a) $\forall t, t_r \in T_v, \text{subterm_replace}(t, t_r, \varepsilon) = t_r ;$
 $\forall t, t_r \in T_v,$
b) $\forall p = (p_1, \dots, p_k) \in \text{subterm_paths}(t), p \neq \varepsilon,$
 $\text{subterm_replace}(t, t_r, p) =$
 $= \text{subterm_replace}(\text{subterm}(t, p_1), t_r,$
 $(p_2, \dots, p_k));$
 $\forall t, t_r \in T_v, \forall p \notin \text{subterm_paths}(t),$
c) $\text{subterm_replace}(t, t_r, p) = t.$

Further, we will introduce the operations of substitution and the bound unification.

7. The substitution of free variables is a set of pairs $[x_i, t_i]$ consisting of free variables x_i and terms t_i which are substituted instead of variables:

$$s = s[x_i, t_i] = \text{substitution}([x_1, t_1], \dots, [x_n, t_n]), x_i \in \Sigma_x, t_i \in T_v.$$

The set of substitutions will be denoted as S .

8. The substitution operation: $\text{subst} : T_v \times S \rightarrow T_v$

- a) $\forall t \in T_v, \forall s = s([x_1, t_1], [x_2, t_2], \dots, [x_n, t_n]) \in S,$
 $\text{subst}(t, s) = \text{subst}(\text{subst}(t, s([x_1, t_1])),$
 $s([x_2, t_2], \dots, [x_n, t_n]))$

The sequential application of this equality allows reducing the operation definition to substitutions with one variable;

- b) $\forall c \in \Sigma_c, \forall s = s[x_1, t_1] \in S, \text{subst}(c, s) = c ;$
c) $\forall x \in \Sigma_x, \forall s = s[x_1, t_1] \in S, \text{subst}(x, s) = \begin{cases} t_1, x = x_1; \\ x, x \neq x_1; \end{cases}$

- d) $\forall f \in \Sigma_t, \forall t_1, \dots, t_n \in T_v, \forall s \in S,$
 $\text{subst}(f(t_1, \dots, t_n), s) = f(\text{subst}(t_1, s), \dots, \text{subst}(t_n, s)).$

9. The bound unification: $\text{bound_unify} : T_v \times T_v \times S \rightarrow T_v \times S.$

Based on two terms and a current substitution, the given function computes a unifier and the specified substitution.

- a) $\forall t \in T_v, \forall s \in S, \text{bound_unify}(t, \text{NIL}, s) =$
 $= \text{bound_unify}(\text{NIL}, t, s) = (\text{NIL}, \emptyset);$
 $\forall c_1, c_2 \in \Sigma_c, \forall s \in S,$
- b) $\text{bound_unify}(c_1, c_2, s) = \begin{cases} (\text{NIL}, \emptyset), & c_1 \neq c_2; \\ (c_1, s), & c_1 = c_2; \end{cases}$
- c) $\forall x_1, x_2 \in \Sigma_x, \forall s \in S,$
 $\text{bound_unify}(x_1, x_2, s) = (x_1, s \cup [x_1, x_2]);$
 $\forall t \in T_v \setminus \Sigma_x, \forall x \in \Sigma_x, \forall s \in S,$
- d) $\text{bound_unify}(t, x, s) = \text{bound_unify}(x, t, s) =$
 $= (t, s \cup [x, t])$

(the requirement for t not to be a variable is due to both equalities, in this case, in this case would be applicable and they would give inconsistent results. The case, when both terms are variables, was considered in the previous subsection.)

- e) $\forall f, g \in \Sigma_t, \forall t_1, \dots, t_n, q_1, \dots, q_m \in T_v, \forall s \in S,$
 $\text{bound_unify}(f(t_1, \dots, t_n), g(q_1, \dots, q_m), s) =$
 $= \begin{cases} (\text{NIL}, \emptyset), & f \neq g \text{ or } m \neq n; \\ (f(\text{bound_unify1}(t_1, q_1, s), \dots, \\ \text{bound_unify1}(t_n, q_n, s)); \\ s \cup \bigcup_{i=1}^n \text{bound_unify2}(t_i, q_i, s)), \end{cases}$

That is, the unification occurs, only if names and arities of terms coincide; in this case, the unifier is obtained by a recursive application of operation, and all the substitutions found at subterm unification are

added to substitution. The symbols *bound_unify1* and *bound_unify2*, for brevity, designate the first and the second component of *bound_unify*, i.e. the unifier and the substitution.

Based on the definition of the bound unification, it is possible to introduce the concept of a free unification (for this purpose it is required to preliminary introduce some auxiliary concepts).

10. The set of free variables: $fv : T_v \rightarrow \Sigma_x^*$

a) $\forall c \in \Sigma_c, fv(c) = \emptyset;$

b) $\forall x \in \Sigma_x, fv(x) = \{x\};$

c) $\forall f \in \Sigma_f, \forall t_1, \dots, t_n \in T_v, fv(f(t_1, \dots, t_n)) = \bigcup_{i=1}^n fv(t_i);$

11. Renaming of variables for the purpose of conflict elimination: $free_fv : T_v \times \Sigma_x^* \times \Sigma_x^* \rightarrow T_v$

a) $\forall c \in \Sigma_c, \forall X, Y \subset \Sigma_x, free_fv(c, X, Y) = c;$

$\forall x \in \Sigma_x, \forall X, Y \subset \Sigma_x, free_fv(x, X, Y) =$

b)
$$= \begin{cases} x, x \notin X; \\ x_m : x_m \notin X \cup Y, x \in X; \end{cases}$$

c) $\forall f \in \Sigma_f, \forall t_1, \dots, t_n \in T_v, \forall X, Y \subset \Sigma_x,$
 $free_fv(f(t_1, \dots, t_n), X, Y) =$
 $= f(free_fv(t_1, X, Z), \dots, free_fv(t_n, X, Z)),$
 where $Z = Y \cup fv(f(t_1, \dots, t_n)).$

12. The free unification: $free_unify : T_v \times T_v \rightarrow S$. This function is similar to the bound unification but assumes that the variables included in input terms are not bound. Thus, for example, the variable x_1 which is included both in the first and the second term is considered different in the first and the second term. Therefore the renaming of variables is necessary, which is carried out with the help of *free_fv*.

a) $free_unify(t_1, t_2) =$
 $= bound_unify(t_1, free_fv(t_2, fv(t_1), \emptyset), \emptyset).$

3.2.4. Rewriting rules. Further, we define a concept of a rewriting rule which is used for transformation of terms.

The rewriting rule is a pair of terms $r = (t_{in}, t_{out})$ on which the additional condition $fv(t_{out}) \subseteq fv(t_{in})$ is imposed, i.e. all free variables of a target term t_{out} are present in the input term t_{in} . The rule can be written in the form of the term $rule(t_{in}, t_{out})$.

The application of the rule to the given term $t \in T_c$ is described by the following function:
 $apply(t, rule(t_{in}, t_{out})) = subst(t_{out}, free_unify(t, t_{in}))$.

It is necessary to notice that the additional condition $t_{in} \notin \Sigma_x$ is usually imposed on the input term, i.e. the input term cannot be a variable. This restriction is due to the fact that if the input term is a variable, the corresponding rule will be always applicable. We do not impose such a restriction, for the reasons which will be explained in Subsection 3.2.6.

Usually, the rule is applied not to a term, but to some its subterm. Thus, there is not the only result of application of a rule, but a set of results from rule application to various subterms:

$$Apply(t, r) = \bigcup_{p \in subterm_paths(t)} \{subterm_replace(t, apply(subterm^*(t, p), r), p)\}.$$

The rewriting rule system is the ordered sequence of rules: $R = (r_1, \dots, r_k)$. The application of a system of rules to a term is a union of results of application of separate rules to the given term (and its subterms): $apply(t, R) = \bigcup_{r \in R} Apply(t, r)$.

It is necessary to notice that the defined rewriting rules allow defining an abstract rewriting system (according to the definitions [13]). Indeed, let us fix the system of rewriting rules R . We will consider the set of specific terms T_c with the binary relation $\rightarrow$: $\forall x, y \in T_c, x \rightarrow y \Leftrightarrow y \in apply(x, R)$, defined on it. This struc-

ture can be considered as a rewriting system that will allow defining such properties as confluence, Noetherian, existence and uniqueness of a normal form. Such properties will depend on the chosen system of rules R .

3.2.5. Rewriting strategy. As can be seen from Subsection 3.2.4, the result of rewriting rule system applied to a term, in a general case is a set of terms, instead of a unique result term, i.e. the action of a system of rules is defined ambiguously. For practical application, it is necessary to define a method of selection of one result from the set $apply(t, R)$. The rewriting strategy is such a method.

The rewriting strategy is an arbitrary function selecting a unique result from the set of results: $str(t, R) = t_{res} \in apply(t, R)$. From the construction of the set $apply(t, R)$ the equivalent definition follows: $str(t, R) = (p_i, r_j), p_i \in subterm_paths(t), r_j \in R$, i.e. the strategy chooses a specific rule from the system and a subterm to which it is applied.

One of the ways to assign the strategy is ordering of sets $subterm_paths(t)$ and R , then the strategy chooses the first, in sense of the introduced order, elements for which the rule works. For the set of paths to subterms, it is possible to use a lexicographic order, i.e. $(i_1, \dots, i_n) < (j_1, \dots, j_m)$, if $i_1 < j_1$, or $i_1 = j_1, i_2 < j_2$, etc. At the same time, we will consider that the empty subsequence precedes any number. Such order on the set $subterm_paths(t)$ corresponds to a depth-first tree traversal. For a breadth-first traversal it is necessary to use the modified order, in which the sequence length is compared at first (the shorter one comes in the beginning), and only for the sequences of identical length the lexicographic order is used.

For the set of paths, it is possible to use a lexicographic order both in direct and opposite directions. The usage of a direct direction leads to the *TopDown* strategy, in which the subterms are checked beginning “from above”, i.e. the term itself, then its first subterm, etc. The opposite direction is implemented in the *BottomUp* strategy, where checking begins “from below”.

For the system of rules R , it is possible to use a total order, which is based on lexicographic comparison. But such ordering has

no special sense as it is based on superficial similarity. The partial ordering, based on the fact that more specific rules have higher priority, is more intelligent. Formally, it is defined as follows. Let us introduce the relation of a partial order for terms $t_1 \leq_c t_2 \Leftrightarrow \exists s \in S : \text{subst}(t_2, s) = t_1$ (i.e. there is a substitution transforming a less specific term to a more specific). The order relation for rules is defined on the basis of comparison of their input samples: $(t_{in}, t_{out}) \leq_c (q_{in}, q_{out}) \Leftrightarrow t_{in} \leq_c q_{in}$. For the rules incomparable over the introduced relation of a partial order, another criterion is applied, for example, the order of rules in a system.

3.2.6. Interaction with an environment. The model constructed above describes the term rewriting system which can be applied for transformation of an input term into an output on the basis of a system of rules. However, for the practical application, the additional possibilities can be demanded, such as execution of procedural actions or data access, which are not presented in an input term. In this connection, we will extend the system model by adding interaction with an external program environment.

As a model of an environment we will consider the triple $E = \langle X, \text{check}, \text{action} \rangle$, where E is a designation of the environment, X is an input state of the environment, $\text{check} : X \times T_c \rightarrow \text{BOOL}$ is the function allowing to request data from the environment, $\text{action} : X \times T_c \rightarrow X$ is the function modeling the influence on the environment. Thus, the state of the system at the moment of time i is described now by two parameters: t_i is a current term and x_i is a current state of the environment.

Let us also extend the definition of a rewriting rule: the quadruple $r = (t_{in}, t_{cond}, t_{out}, t_{act})$ will be called a rule, with additional conditions $\text{fv}(t_{cond}) \subseteq \text{fv}(t_{in})$, $\text{fv}(t_{out}) \subseteq \text{fv}(t_{in})$, $\text{fv}(t_{act}) \subseteq \text{fv}(t_{in})$. In this definition the following additional terms are added: t_{cond} is a condition of application of a rule and t_{act} is an additional action at rule activation. The rule application to a term (and all its subterms) is defined in the following modified manner:

$$\begin{aligned}
& \text{Apply}(t, r) = \\
& = \bigcup_{p \in \text{subterm_paths}(t) \wedge \text{rule_check}(\text{subterm}^*(t, p), r)} \{ \text{subterm_replace}(\\
& \quad t, \text{apply}(\text{subterm}^*(t, p), r), p) \},
\end{aligned}$$

where the function of checking the applicability of the rule is

$$\text{rule_check}(t, r) = \text{check}(x_i, \text{subst}(t_{\text{cond}}, \text{free_unify}(t, t_{\text{in}}))).$$

Thus, from the unification of a term and the input sample we obtain the substitution, which is applied to a condition term, then the result is checked with environment usage (the *check* function). Thus, the environment can influence the activation of rules.

The transformation of terms is defined the same as earlier, but taking into account the new definition of *Apply*(*t*, *r*). If the state of the system at the moment of time *i* is known (i.e. *t_i* and *x_i* are known), then the term during the following moment of time *i* + 1 is defined by the equation $t_{i+1} = \text{str}(t_i, R) \in \text{apply}(t_i, R)$. The state of the system at the moment of time *i* + 1 is defined from the equation $x_{i+1} = \text{action}(x_i, \text{subst}(t_{\text{act}}, \text{free_unify}(t_i, t_{\text{in}})))$.

Thus, the extended system is defined by the system of rules $R = \{r_1, \dots, r_n\}$ and the environment $E = \langle X, \text{check}, \text{action} \rangle$. If the input term *t₀* and the initial state *x₀* of the environment are set, it is possible to define the sequence $\{(t_i, x_i)\}$ of states of the system at the moment of time *i*. If for any *i* the $t_{i+1} = t_i$ is satisfied, the system (*R*, *E*) converges for the input data (*t₀*, *x₀*). In this case, the resulting term is *t_i*, which will be designated as follows: $t_i = \text{reduce}(t_0, R, x_0, E)$, or, at the fixed environment, $t_i = \text{reduce}(t_0, R)$.

As was mentioned in Subsection 3.2.4, we do not impose the restriction $t_{\text{in}} \notin \Sigma_x$ on rules, which is usually used to prevent an infinite loop in a system. It is explained by addition the condition *t_{cond}*

to a term rule. Thus, the rules of the form $v_i[cond(v_i)] \rightarrow t_{out}$ become possible. Such rules work for any term for which condition *cond* is satisfied. Actually, the declarative comparison with the sample is replaced with the procedural check implemented in the environment. We will notice that rules of such type can negatively influence the performance, as the computation of the condition *cond* will be carried out for every subterm.

3.2.7. TermWare language. For denoting terms and rules we use the TermWare language. Terms are written in a natural manner, i.e. $f(t_1, \dots, t_n)$. For separate symbols from Σ_t , which designate arithmetic operations, the reduced designations are used, i.e. the expression can be written in a natural form which then will be transformed into a term. For example, the expression $a + b * (c + d)$ will be transformed into the term *plus(a, multiply(b, plus(c, d)))*.

For writing the rules the following syntax is used: the rule $r = (t_{in}, t_{cond}, t_{out}, t_{act})$ is written as $t_{in}[t_{cond}] \rightarrow t_{out}[t_{act}]$. We will notice that this notation is also the reduction for the term *if_rule(t_{in}, t_{cond}, action(t_{out}, t_{act}))*. The components t_{cond} and t_{act} are unessential and can be omitted, i.e. the rule can look like $t_{in}[t_{cond}] \rightarrow t_{out}$ or $t_{in} \rightarrow t_{out}[t_{act}]$, or $t_{in} \rightarrow t_{out}$.

In TermWare, the propositional variables have identifiers that begin with the symbol \$, for example, \$x, \$y. As an illustration consider the rule $p(\$x, \$y) \rightarrow q(\$y, \$x)$. The result of application of this rule, for example, to the term $p(a, f(b))$ is the term $q(f(b), a)$.

The TermWare language contains special facilities for expression of lists. The list $(t_1, t_2, \dots, t_n)$ is represented in the form *cons(t₁, cons(t₂, ..., cons(t_n, NIL) ...))*. For such a construct, the reduction $[t_1, \dots, t_n]$ is used. Also, for writing the rules dealing with such lists, the reduction $[\$x : \$y] = \text{cons}(\$x, \$y)$ is useful. This reduction selects the first element of the list (\$x) and the rest of the list (\$y). We will notice that $[x : y] \neq [x, y]$: the first expression designates the list of any length, with the first element x and the rest of the list y,

whereas the second expression designates the list containing two elements x and y .

3.3. Existing software implementations

The system of rewriting rules is a software system that contains language for description of rules, the program for their application, and also, probably, the sets of ready-made rules or rewriting strategies. Often, such system is a part of more general metaprogramming platform [33].

We will consider the following examples of rewriting rule systems:

- Maude [105, 106, 169];
- Stratego [151, 152];
- ASF+SDF [28, 107];
- TXL [30, 160];
- Jess [59, 86];
- APS [8, 100];
- TermWare [38, 154].

3.3.1. Maude. The Maude system [105, 106, 169] is based on the theory of algebraic systems. The program is represented in the form of many-sorted algebra: the set of sorts (data types), and also operations on objects of these sorts are defined. Rules are defined in two ways. System definition includes the equations setting the correlations between operations. These correlations can be used for reduction of terms to the elementary form (which contains only basic operations). Besides, the rules, which are used for transition between terms and cannot be simplified with the usage of the equations, are defined separately. Formally, the unique distinction between the equations and the rules consists in that the equations are symmetric (i.e. the transition is possible from the left part to the right, and vice versa). Nevertheless, it is recommended to use these means for different purposes: the equations for defining the general structure of a problem, and a rule for defining specific transformations.

In Maude, considerable attention is given to metaprogramming, which is the modeling of the Maude system by means of this system. Standard libraries include the model of all programming facilities of Maude, and also the functions for transition between ab-

straction levels (allowing to construct a term analog in abstract model Maude based on a term and also to execute the reverse transformation). Besides, Maude includes the extensions implemented by means of the same system — the so-called full system (Full Maude) in comparison with the base system (Core Maude). In particular, these extensions include the means of object-oriented programming and parameterization of models.

Maude is considered as the high-grade programming system and does not assume embedding in external applications. In particular, the given system does not contain extension facilities with the usage of external languages (such as C or Java). Even the implementation of an additional rewriting strategy is made by Maude metaprogramming means.

3.3.2. Stratego. The Stratego system [151, 152] uses the standard model of rewriting systems: the working object is the term to which rules of the form $LHS \rightarrow RHS$ are applied. The feature of the Stratego is that rewriting strategy plays a key role in the system (in particular, it can be seen from its name too). The rewriting strategies are used in all systems of rewriting rules, as for practical application it is necessary to set a certain order of action of rules. However, the majority of systems contain one or several predetermined strategies, and, probably the means for implementation of new strategies. In Stratego, a different approach is provided: the set of elementary strategies and the means of their combination, which gives the possibility to declaratively define any strategy. The means of definition of strategies actually are base in the system: even rewriting rules are implemented as a strategy of a special kind.

The additional terms of a special kind, the so-called annotated terms (ATerm), are used in Stratego. Every subterm can contain an annotation of an arbitrary format. Annotations by default are not processed in any way by the system, in particular, they do not influence the possibility of application of rules. However, they can be used for the storage of data specific to an application. From the technical point of view, the implementation of terms is characterized by the usage of maximum combination of subterms: if the term contains two identical subterms (taking into account the annotations), they are

actually stored in memory only once. It allows to essentially increase the efficiency of storage and processing of terms.

One of the advantages of Stratego is the possibility to use the syntax of a target language at writing the rules, instead of the use of standard syntax of terms. It allows simplifying writing complex transformations, in particular, for applied developers who are not familiar very well with an abstract model of term rewriting.

Stratego is a part of the Stratego/XT platform [152], intended for solving various problems associated with transformations and processing of source code of programs. In addition to the actual Stratego rules system, the platform Stratego/XT contains code parsers based on SDF technology [155] and tools for generation of code for various languages. The platform has a modular architecture: it consists of a set of respectively independent applications, each implementing a certain functionality. The interaction of applications can be organized by means of an operational system (Unix pipeline). Also, the association of several components in one application with the usage of the XTC system [151] is supported.

Besides the independent usage, Stratego/XT components can also be embedded into C applications. Furthermore, Stratego can be extended by the implementation of new primitives in C.

3.3.3. ASF+SDF. As it can be seen from its name, ASF+SDF system [28, 107] consists of two parts (formalisms): the means for a description of SDF language syntax (Syntax Definition Formalism) [155] and facilities for description of ASF transformations (Algebraic Specification Formalism) [28]. ASF+SDF is used only for solving the problems of source code transformation, therefore the code analysis means are an integral part of the system. This is the difference of ASF+SDF from other rewriting rule systems, such as Stratego, where SDF means are also used, but they are independent of rewriting rules. Therefore, the Stratego system can be used for solving the problems which are not concerned with processing of source code of programs. For such problems, it is possible to use ASF+SDF too, but at that it will be necessary to implement a special language for a problem description that will allow using SDF means.

The rules in ASF+SDF consist of two parts corresponding to two used formalisms. The part corresponding to SDF, describes types of terms, possible operations over them, and also defines the syntax of operation calls. The part of a rule corresponding to ASF contains the actual transformation (written in the form of equality $LHS = RHS$, although a unidirectional application actually is meant). As well as Stratego, ASF+SDF allows to write down the rules with use of syntax of a source language (that actually is the natural form of notation for ASF+SDF, whereas the notation in the form of terms is a special case of the general syntax).

The ASF+SDF system contains a unique rewriting strategy and does not assume the possibility of addition of a new strategy. It is supposed that the system of rules should be confluent [13] (this should be provided by the developer of rules), therefore a rewriting strategy is of no significance. However, ASF+SDF contains some means for defining an order of application of rules. Separate rules can be defined as default rules, in this case, they will be executed only in that case when any other rule does not work. Besides, the functions of a tree traversal are used, which apply the given rule to all the subterms in a defined order; at that the transformation of subterms and/or accumulation of computation result can be made.

ASF+SDF is a part of the Meta-Environment platform [107] intended for processing and transformation of programs. One of the possibilities of this platform is the creation of the integrated development environments (IDEs) for languages with the syntax described by means of SDF. All the components of the platform provide both independent use (in a graphic or a console form) and embedding into applications in C and Java languages. Besides, the program interfaces for system extension are provided.

3.3.4. TXL. As well as ASF+SDF, the TXL system [30, 160] is intended for a description of transformations of source code of programs. Therefore, the TXL rules contain two parts too: the description of the language syntax and the description of transformations. Unlike ASF+SDF, where these descriptions are placed in different files and basically can be used independently, in TXL syntax and transformations are described in a shared file and use one language for their notation.

Rules in TXL are written in the syntax of the source language (similarly to ASF+SDF). There is a distinction between rules and functions: rules are applied to all subterms of an input term, whereas functions are applied to an input term entirely. Unlike many other rewriting systems, there are no built-in rewriting strategies: all applied rules should be specified in an explicit form, including an order of their application. Each rule can define a list of other rules which are called during the action of this rule. In every TXL program, there should be a basic function *main*, which at a high level defines all used rules. Thus, TXL is closer to the imperative style of languages such as C or Java, rather than to declarative style of other rewriting rule systems.

TXL has many built-in means which do not give essentially new functionality in comparison with basic means but facilitate implementation of typical problems. Hence, at defining the syntax of language there is special support for some frequently occurring constructs: comments, compound operators (such as `!=`, `<=`, `++`), keywords and separators. At defining a transformation it is possible to construct and deconstruct defined subterms. During the construction, the new variable is defined, which contains the given subterm and then can be used in several places in a rule. Similarly, at deconstruction one variable can be disassembled as subterm of a certain kind; it relieves of necessity to repeat identical subterms at defining a sample (these subterms are defined as one variable, and then their structure is restored with the help of deconstruction).

Unlike many other rewriting rule systems, source code of TXL is not distributed under the open-source license, although the finished application is accessible for free downloading from the site [160].

3.3.5. Jess. The Jess system [59, 86] was initially developed as a platform for the development of expert systems, therefore it has certain differences from other rewriting rule systems. The working object in Jess is the set of facts, each of which is represented in the form of a term. Rules, as well as in other systems, consist of the left part defining a condition of application of a rule, and the right part describing the action of a rule. However, unlike the other systems, in Jess the left part of a rule can contain several samples; thus the rule

works, if each sample is present among the current set of the facts. The right part is not necessarily rewriting (fact updating): other actions, such as addition or fact removal, an output of the debugging information or a call of any function, are possible.

The Jess system is implemented in Java and intended for embedding in Java applications. The facts in Jess can be both pure, i.e. defined at the level of the system itself, and also can refer to a Java object. Besides, there is a possibility of definition of user functions in Jess as Java classes implementing some interface. On the other hand, Jess contains a library for Java, allowing to create the facts and to execute rules. Thus, the interaction between the Java application and the rules is achieved.

The Jess system is the commercial application, though there is a special academic license [86].

3.3.6. APS. The APS system [8, 100] unites various programming paradigms: besides rewriting, there can be used imperative and functional paradigms for implementing strategies, as well as the logic paradigm in the form of unification during rewriting. In this regard, APS is close to Stratego or TXL. As well as Maude system, APS supports the implementation of additional possibilities with the use of the built-in language (in the case of APS it is the APLAN language).

A distinctive feature of APS is a large number of built-in rewriting strategies that allows choosing the most efficient way of solving a specific task by means of rewriting rules. On the other hand, such abundance of strategies and the necessity of their explicit defining can complicate the development for less experienced users. Besides, the APS system uses the system of designations which can mislead the users of other systems. For example, the rules are defined in the form $LHS = RHS$, and the symbol $\rightarrow$ is used for defining conditions, i.e. the rule with a condition is written in the form $cond \rightarrow (LHS = RHS)$.

The APS system is a part of the insertion modeling system IMS [82]. Both systems are commercial, but demonstration versions are accessible for downloading from the site [8].

3.3.7. TermWare. The TermWare system [38, 154] mainly uses the standard model of rewriting rules. Rules are defined in the

form $LHS \rightarrow RHS$. Basically, the notation in the form of terms is used, although the reduced form of notation is available for standard arithmetic and logic operations. Also, it is possible to attach additional parsers for supporting the syntax of various languages. Several built-in rewriting strategies are available, and additional strategies can be implemented with the use of Java program interfaces.

The feature of TermWare is as much as possible simplified syntax and absence of many auxiliary constructs. Thus, separate rules are not named, but only systems of rules (sets of rules for which a strategy is also defined). There is no necessity to define types or signatures of used terms. Accordingly, the most keywords necessary in other rewriting rule systems are absent. Such light-weight syntax promotes faster learning of TermWare language, though in some cases the absence of additional possibilities can lead to a complication of rules being developed.

Though TermWare can be used as an independent application (with the command line interface), the basic variant of the usage of this system consists in its embedding in Java applications. Every system of rules can specify the so-called facts database — the Java class implementing a certain interface. In this case, the system of rules can receive the information from the facts database and carry out the actions provided in the facts database. Thus, interaction of a declarative system of rules and the imperative Java program is achieved.

3.4. Applications of rewriting rules technique for working with program code

3.4.1. The main directions of use. Rewriting rules systems are a natural way of representation of transformations of source code of programs. Therefore there are many works describing the use of term rewriting technique for working with source code. The main directions of research in this area are the following:

- analysis of code (for the purpose of finding errors and inconsistencies with programming standards);
- transformation of code. Depending on the transformation purpose, the following directions are possible:

- refactoring, which includes the transformations preserving a behavior of a program, but improving code readability, conformity to coding standards and eliminating code duplication;
- optimization, which involves the transformations aimed at increase in performance of an application;
- security, which contains the transformations raising the security of a program by detection and elimination of potentially dangerous code fragments;
- legacy code, which is the transition from legacy code to a similar code on more modern platforms, support of legacy code and maintenance of its interaction with new applications;
- creation of new languages. Such languages can be implemented as independent domain-specific languages (DSLs) or as extensions of existing general-purpose languages;
- modeling. The creation and processing of high-level program models, the transition from high-level models to source code.

Further, we consider the use of term rewriting systems for each of the above-mentioned directions in more detail.

3.4.2. The analysis of source code. The static analysis of source code is an important factor in improvement of quality of applications. This method allows finding fragments of program code, which are syntactically correct (i.e. do not cause a compilation error), but can lead to errors during execution or complicate understanding and updating of source code. We would like to emphasize that it is a question of the static analysis, i.e. the analysis of source code without the use of the information at execution time.

The input data of the algorithms of the static analysis is a source code written in a high-level programming language. Some elementary errors can be already found in text representation (for example, use of unsafe functions). However for the full-scale analysis, it is necessary to translate source code to more structured representation, most often a parse tree. Since trees are naturally transformed into terms, it is possible to use rewriting rules after the syntactic analysis.

There are two approaches for implementation of static analysis algorithms: more formal and more practical. Under the formal approach, a formal model of program execution is constructed, for example, with the use of linear temporal logic (LTL) [142]. Then, some properties of the program, for example, the absence of blocking (deadlocks) [3] in the case of a parallel program, are formulated. After that, the tools of automatic proof are used, which can determine whether the input program has a given property.

Such an approach is used in work [58], where JavaFAN is developed, which is the tool based on Maude and intended for analysis of Java applications. Authors develop the Maude specification (a system of equations and rules), describing the semantics of execution for Java. To this specification, the description of the program obtained from source Java code or a bytecode with use of the corresponding parser is added. Rewriting with the use of such specifications corresponds to a transition between execution-time states, i.e. actually to program interpretation. In addition, the errors to be found in code are defined. The standard method for rewriting technique is used, i.e. defining a pattern, not in source code, but in execution-time model. The Maude system goes through all possible variants of rewriting (execution time states) and reports an error in the program if it detects the defined pattern.

The advantage of such an approach is that a certain class of errors is searched, instead of standard variants of a code which can lead to an error. Therefore all errors of the defined type are found. Besides, if the complete search of states has been finished, and the defined pattern has not been found, an absence of the given class of errors in a program thereby is proved. However, a formal approach has the essential lack, namely, the considerable complexity of calculations. Since the size of the space of program states is essentially larger than the program size and can be infinite, the formal approach is practically applicable to small programs only.

More applicable in practice is the approach based on a search of standard templates of erroneous code. Such approach cannot guarantee detection of all errors; besides, false positives are possible, when correct code corresponds to the defined pattern of an error. Nevertheless, such approach does not require a search of a large

number of states, therefore it is applicable to a code of any size. An example of such an approach is the JavaChecker system, intended for searching standard errors in Java code [85]. The system is based on TermWare and uses rewriting rules for searching the defined patterns of errors. The input code is not modified: when the error is detected, the system reports the message with the use of means of interaction with an environment.

A similar approach is typical for many other studies. For example, the RUST system [70] is intended for interactive refactoring of C++ code and uses TXL rules for finding incorrect code fragments. Unlike JavaChecker, RUST offers variants of correction of found errors. In work [156], the method for locating the errors connected with the use of types, for various languages, is proposed. The ASF+SDF system is used for defining the information about the type system of a language, and also rules for searching an exact place, where an error occurred.

One of the advantages of the rewriting rules technique is the declarative notation, which allows to easily add new rules for detection of new types of errors. Such situation is characteristic for the considered systems [70, 85, 156], where system extension is made by addition of new rules. For even further simplification of the process of addition of new patterns of errors, the Proteus system [166] was developed. This system implements the YATL language for simplified definition of templates of erroneous code in C language. The templates defined with use of YATL are automatically transformed into Stratego rules.

Since rewriting rules assume implementation of some transformation, and at the code analysis there is no need to transform it, sometimes rules are used for performing auxiliary actions at the analysis. The example is the NICAD system intended for searching similar methods (clones) in C and Java code [145]. Clones show that copying of code instead of correct reuse was applied; they complicate code support, as often changes are brought or errors are corrected only in one copy of a method. The NICAD system considers possibilities of various formatting of clones, using TXL for reduction of methods to an initial form (i.e. parsing with subsequent generation of standard representation is done; the identical transformation is actu-

ally applied). Methods, for which most of the lines in initial representation coincide, are considered as clones.

One more example of the auxiliary use of rewriting rules is considered in work [150]. This work uses the fact that at the present moment there is a large number of tools for checking the correctness of parallel programs for C/C++ language, and there are significantly fewer tools for high-level languages like Java. Authors propose TXL rules for transformation of Java code into a similar C++ code, which then is checked for presence of blocking with the use of one of the tools available for C. If blocking is found in C code, the same blocking should be also present in Java code.

3.4.3. Refactoring. Refactoring is the transformation of a source code preserving behavior of a program, but changing the structure of code [61]. Refactoring can be applied for simplification of code and improvement of its readability, elimination of code duplication and alignment with chosen programming standards. Classical examples of refactoring are renaming of a method (and all references to it), allocation of reusable code into a separate method.

The proof of transformations correctness, i.e. that transformations do not change a program behavior, is very important for refactoring. For this purpose, as well as in a case with the code analysis, formal methods are applied. In work [61], Maude is applied for formal representation of some transformations of Java programs and proof of their correctness. For this purpose, the specification of execution time semantics for Java, constructed in [58] is used. It is necessary to notice that the proof requires human intervention, although it intensively uses the possibilities of Maude. Similar results for a preprocessor of C language were obtained in [60].

Rewriting rules systems may also be used for specifying the transformations, which define the refactoring. In already mentioned work [70], transformations for C++ are defined in the form of TXL rules. The RUST system described in [70] is designed for interactive use: it detects incorrect fragments of code and offers the developer the variants of their correction. Thus, the developer has control over the transformations being applied.

Sometimes refactoring may be used for transition to a new, more high-level platform. For example, work [12] describes refactor-

ing for the transition from object-oriented (Java) to aspect-oriented (AspectJ) code. TXL rules are used for the definition of code fragments, which can be presented in the form of aspects. Rules also describe the transition from Java to AspectJ. As well as in [70], the developer's intervention is required for indicating which of the found transformations are necessary for applying.

In some cases, refactoring allows not only to improve code structure, but also to increase performance. For example, work [163] describes the refactoring of legacy COBOL code. The ASF+SDF rules reduce the operating structures of code to a normal form. It not only promotes simplification of code support, but also improves performance and allows to define interfaces with external systems.

3.4.4. Optimization. One more important class of transformations are *optimizing transformations*, i.e. transformations which preserve a program behavior (a result for given input data), but increase the performance of the program, for example, concerning the execution time. Optimizing transformations can be applied both at the level of source code and at the level of intermediate representation in the compiler. The latter variant is used most often, as in this case transformations are carried out automatically, and the source code does not contain the special optimized constructs which hide the purpose of code and complicate its understanding. For the implementation of optimizing transformations inside the compiler, rewriting rules systems can also be used, as shown in [165]. The mentioned work proposes to use Stratego for implementation of optimizers. It is supposed that optimizing transformations for the given language are defined in the form of rules and strategies of Stratego. The strategies are interpreted at debugging of transformations and compiled into C code for final use that increases the performance of the optimizer.

Despite all the advantages of optimizing transformations at the level of intermediate representation, such transformations cannot express many optimization methods. The reason is that at the level of intermediate representation, a lot of information accessible in source code is lost. Therefore many transformations need to be applied at the level of source code. Examples of such transformations are given in [14], where the CodeBoost system intended for optimization of

code of numerical methods in C++, is described. The system allows the developer to implement a program in a high-level form close to a mathematical representation. Further rewriting rules of Stratego transform code for increasing its efficiency. Another example of optimizing transformations at the level of source code is given in [132], where Stratego is used for automatic substitution of values of constants, which allows executing a part of computing during compilation, instead of execution.

For some optimizing transformations, even the level of a source code appears too low, therefore they work with models of a higher level. An example of such an approach is described in the work [87], where Jess is used for detection and elimination of problems with performance at a level of system architecture. As input data, UML interaction diagrams annotated with data about the performance of executing certain actions are used. The performance model thus obtained is processed by Jess rules for defining and elimination of bottlenecks in architecture.

3.4.5. Security. Increasing the security of applications is becoming more and more important in connection with the development of Internet technologies and widespread harmful programs. The use of rewriting rules systems may improve the security of applications, for example, due to detection of errors at the static analysis as it is described in Subsection 3.4.2. Besides, the rules can be used for implementation of transformations raising the security.

Examples of such transformations are described in [167], where TXL rules for automatic inclusion of additional assumptions (assertions) in C code are used. Assumptions check the correctness of execution of certain operations. For example, at accessing an array, it is checked that the index does not exceed the array dimensions. If the assumption is not satisfied, the execution of a program is interrupted with an error message that allows the developer to find an error. Besides, the execution of a program in an incorrect state that leads to vulnerability, such as buffer overflow, is prevented.

Rewriting rules may also be applied for fighting against harmful programs (viruses, worms). Thus, work [129] describes the application of TXL for detection of worms that change their code. Rewriting rules reduce code to a normal form that allows revealing

the modified versions of a worm. Work [147] proposes the use of TXL for automatic generation of patches that close the vulnerabilities found. Application of rewriting rules allows to considerably accelerate the process of reaction to the occurrence of a new virus.

Rewriting rules are also applied for increasing the security of Web applications. Work [71] proposes the language for description of access rights for a website. The language allows defining access rights to separate resources in a declarative form, not mixing this information with other data. The approach uses the platform for developing Web applications WebDSL [164] applying Stratego for the transition from specialized language to Java facilities for Web development.

3.4.6. Legacy code. When working with legacy code, there are a lot of problems concerned with the absence of documentation, loss of information on system architecture, usage of no longer supported technologies. Rewriting rules can extract additional information from source code and also carry out transformations to simplify interaction of old applications with new platforms. The transfer of a whole application or its part to new technology is also possible.

Extraction of information from legacy code is described, for example, in [34]. It is a matter of recovery of data structures from code for languages not supporting abstract data structures. TXL rules find sets of variables that are used together; such sets correspond to data structures in modern programming languages. A similar approach is used in [157], where TXL rules are used for extraction of information on system architecture.

The automated transition from one language to another is described in [53] on an example of the transition from Java to C#. TXL rules map the constructs of a subset of Java language to similar C# constructs. The advantage of the described approach is the extensibility of the system by addition of new rules.

Transformations improving the quality of legacy code without transition to new technology are also possible. Similar transformations are described in [132], where ASF+SDF rules reduce control structures in COBOL program to a normal form. This transformation increased program performance and allowed eliminating not sup-

ported constructs, which hindered the development of Web interface of the system.

Rewriting rules may be applied to legacy programs, which work with a source code themselves. Thus, work [20] describes the application of Stratego for extraction of information on a priority of operations from grammars. Such information can be coded in the form of grammar productions. Its explicit representation facilitates the comparison of various grammars of the same language (especially if grammars are implemented using different technologies). Besides, the transition to new technologies of parser development is facilitated.

3.4.7. Domain-specific languages. At present, there is considerable interest in domain-specific languages [164]. Such languages allow developing programs in a given subject domain easier and faster than general-purpose languages, as they contain language constructs implementing the concepts of a subject domain. But there is a need to create corresponding language (or a set of languages) for each subject domain, unlike general-purpose languages, where one language can be applied for solving various problems. Therefore the problem of automation of development of DSLs is actual.

Most often, the transformation to one of the general-purpose languages is used for DSLs instead of development of full-scale compiler. For the implementation of such transformation, it is possible to use rewriting rules. For example, work [73] describes the Apply language for solving computer vision tasks. This language is translated to C by means of Stratego rules. Work [55] considers the implementation of aspect-oriented domain-specific language KALA for description of transactions. This language is implemented on the basis of Java platform Reflex; Stratego is used for transition from constructs of KALA language to calls of methods of Reflex platform. In [164], the language for development of Web applications Web-DSL is described. Stratego rules translate the declarative description of Web sites in this language to Java code using JavaServer Faces and Seam platforms for Web development.

At the stage of development of a new language, it is helpful to use the interpreter allowing to check various constructs of the language and their interaction. The approach to the development of such

interpreters is proposed in [37]. Stratego rules are used for computation of results of application of separate constructs and also specific strategies are applied for efficient traversal of a program. Such an approach allows quite rapidly develop interpreters for new languages, and also easily make changes at language development.

3.4.8. Extensions of languages. Besides independent domain-specific languages, it is possible to implement languages that are built into existing general-purpose language, extending its facilities. Actually such languages extend libraries of some subject domain, adding convenient syntax for method calls. For example, an extension of Java language can allow using SQL queries directly in Java code, thus guaranteeing correct passing of parameters. The advantage of such languages consists in convenience of use of the constructs specific to a subject domain, directly from general-purpose language, without the need to split files written in different languages and to use additional interfaces between them. The disadvantage is a complication of development tools (compilers, integrated development environments): they must take into account the syntax of not one, but several languages simultaneously.

In [21], the StringBorg platform for the development of such built-in languages is proposed. Grammars of both languages (basic and built-in) are defined independently, after which the parser of conjoint language is automatically generated. Constructs of built-in language are converted to API calls with the use of Stratego rules. Automatic generation of API for the construction of safe string representation of built-in language is possible (for example, in the case of SQL, the replacement of special symbols is made, which prevents SQL code injections). Another advantage of StringBorg platform is the possibility to use any combination of basic and built-in languages.

The special case of language extension is presented in [92], where Java was chosen as the basic language, and a bytecode, i.e. low-level representation of the same language, was used as the extension. This allows Java programs to use constructs that have not been taken out to a high-level language, for example, unconditional jump goto. Besides, there is a possibility of more efficient implementation of critical fragments of code. For the conjoint language, the

new compiler Dryad was developed, which reduces code to a normal form (i.e. byte code) by application of Stratego rules.

3.4.9. Modelling. For many problems, it is convenient to use high-level program models (the “high level” refers to a level higher than source code). Such models can have various presentations: specialized languages, UML diagrams, formal specifications. But in any case, elements of a model can be represented in the form of terms. Therefore rewriting rules can be applied to models as well as to source code.

There are the following classes of problems concerned with program models:

- 1) transition from source code to a model;
- 2) transition from a model to source code;
- 3) model transformation.

In the first case, it is a matter of recovery of some information implicitly present in code. Rewriting rules are well suited for solving this task since the necessary information can be represented in the form of source code templates. This approach is applied in [4], where UML entity-relationship diagram is recovered from data description in SQL language. The recovery is implemented using TXL. A similar approach is used in [157] for recovery of the architecture of the legacy system.

The transition from a model to a source code actually means the generation of code from a model. This approach is provided in model-driven development (MDD) and model-driven architecture (MDA) [109]. Rewriting rules can be used directly for implementation of such transformation. Thus, work [76] describes generation of code for Java platforms Seam and JavaServer Faces from high-level descriptions of data. The project uses Stratego and is a part of the implementation of already mentioned WebDSL language [164]. The transformation of models with the use of TXL is described in [134].

Rewriting rules can also be used for development of additional tools that transform models. For example, work [9] proposes the construction of code generator on the basis of templates, which comprehends the syntax of a target language. The SDF grammar is used, which extends the syntax of a target language with a set of metaconstructs for designing templates (such as a reference to model

elements or traversal of a model). Generation is implemented with the use of ASF+SDF rules which interpret metaconstructs, extracting the data from a model.

Transformations of models, as well as source code, can be made for a variety of purposes. For example, in [87] the transformation of performance models for the purpose of detection and elimination of bottlenecks in architecture is described. For this purpose, UML diagrams and Jess rules are used. In [105], Maude is used for analyzing and checking correctness of network protocols. APS system is applied as a part of tools of insertion modeling for formal description of protocols in order to check their correctness [81, 153].

Control questions and exercises

1. Give the definition of an abstract rewriting system.
2. What is normalizing ARS?
3. Give the definition of terminating ARS.
4. How the termination of transitions of ARS is guaranteed?
5. Give the definition of confluent and locally confluent ARS.
6. Prove the equivalence of confluence and Church-Rosser property.
7. Give examples of ARSs.
8. Formulate the definition of a rewriting rule and give examples of rewriting rules and their application.
9. For each of the following systems of rewriting rules determine which ARS properties (normalizing, confluent, locally confluent, terminating) it has:

a) $R_1 = \{a \cdot b \rightarrow b \cdot a\};$

b) $R_2 = \begin{cases} a \rightarrow b, \\ a \rightarrow c; \end{cases}$

c) $R_3 = \begin{cases} a \cdot b \rightarrow b \cdot a, \\ a \rightarrow b, \\ a \rightarrow c; \end{cases}$

$$\text{d) } R_4 = \begin{cases} a \rightarrow f(a, b), \\ f(a, b) \rightarrow f(b, a); \end{cases}$$

$$\text{e) } R_5 = \begin{cases} f(x, x) \rightarrow a, \\ f(x, g(x)) \rightarrow b, \\ c \rightarrow g(c); \end{cases}$$

$$\text{f) } R_6 = \begin{cases} h(x) \rightarrow k(b), \\ k(a) \rightarrow h(a), \\ a \rightarrow b; \end{cases}$$

$$\text{g) } R_7 = \begin{cases} b \rightarrow a, \\ b \rightarrow c, \\ c \rightarrow b, \\ c \rightarrow d. \end{cases}$$

10. Formulate the definition of a term and describe operations over terms.
11. What is a rewriting strategy?
12. How the interaction with the environment is carried out in the formal model of a term rewriting system?
13. Give examples of software systems based on rewriting rules. What is specific about each system?
14. What are the main directions of using rewriting rules systems?
15. Consider the following TermWare rule system:

```
ruleset(
  p($x, q) → q,
  p(q) → y
)
```

What term will be obtained as a result of the application of this system to the term $f(p(p(q), q))$?

16. Represent the following Boolean algebra axioms as TermWare rewriting rules:

$$\begin{aligned}x \cdot (y \vee z) &= x \cdot y \vee x \cdot z; \\x \cdot \bar{x} &= 0; \\x \vee 0 &= x; \\0 \vee x &= x.\end{aligned}$$

Basic Boolean operations in TermWare can be written in functional or truncated form as follows:

- 1) disjunction: *logical_or*(x, y) or $x \parallel y$;
- 2) conjunction: *logical_and*(x, y) or $x \&\&y$;
- 3) negation: *logical_not*(x) or $!x$.

Apply the obtained rules to the following expression:

$$x \&\& (!x \parallel !y) \parallel y \&\& (!x \parallel !y)$$

Chapter 4

Generalization of Glushkov algorithmic algebra systems to the case of programs on hierarchical data

In this chapter, we consider algebras of sequential imperative programs which generalize Glushkov algorithmic algebra systems and allow one to define programs over complex (including hierarchical) data structures. As a universal mathematical model of data, we will use nominative data [84, 110, 117, 121, 126, 127, 149], which are based on the name-value relation, and which allow one to conveniently represent many data structures which frequently occur in programming. In the simplest case nominative data can be considered as associations of values with names, and in more complex cases they are hierarchical structures.

The set of functions and the set of predicates on nominative data which can be obtained from base functions and predicates using program compositions such as choice, loop, etc. form an algebraic structure, *an associative nominative algorithmic Glushkov algebra* [149] which generalizes Glushkov algorithmic algebra systems.

4.1. Different types of nominative data

Nominative data are built over classes of names V and atoms A with the help of naming relations. Thus in the first approximation, a nominative data d is either an atom A , or an expression of the form $[v_1 \mapsto d_1, \dots, v_n \mapsto d_n]$, where $v_1, \dots, v_n$ are different names from V , and $d_1, d_2, \dots, d_n$ — are atoms or other nominative data.

To define nominative data formally let us denote by $V \xrightarrow{n} B$ the class of all partial functions from V to a set of values B , which has a finite graph.

We classify nominative data in accordance with the following parameters:

- *values* can be abstract (unstructured) or complex (structured),
- *names* can be abstract (unstructured or complex (structured)).

The possible values of parameters give 4 types of nominative data. Let us clarify the notion of a complex name and a complex value.

Data with *complex values* are nominative data in which values associated with names can themselves be nominative data.

To clarify the notion of a *complex name* we will follow the principle of development (from abstract to concrete) and consider an abstract form of construction and processing of names. The main operation on complex names will be concatenation. This operation is associative.

Principle of associative construction and processing of complex names: *complex names are constructed from abstract names using concatenation, and data with complex names have to be processed using operations that take into account associativity of concatenation of names.*

Besides, we will require that data with complex names satisfy the following principle.

Principle of unambiguous associative naming: *one complex name should have no more than one corresponding value in a given data.*

Taking into account the mentioned principles let us give formal (mathematical) definitions of nominative data of different types.

1. The class of nominative data with abstract names and abstract values over a nonempty set of names V and a nonempty set of abstract values A is:

$$D_0 = V \xrightarrow{n} A.$$

This class contains data of the most abstract type. For example, if u, v are abstract names from V , and a, b are values from A , then

$$[u \mapsto a, v \mapsto b] \in D_0.$$

We will call the element of D_0 data of the type TND_{AA} (data with abstract names and values).

2. The class of nominative data with abstract names and complex values over a nonempty set of names V and a nonempty set of abstract values A is:

$$D_1 = ND(V, A),$$

where $ND(V, A)$ is

$$ND(V, A) = \bigcup_{k \geq 0} ND_k(V, A),$$

where

$$\begin{aligned} ND_0(V, A) &= A \cup \{\emptyset\}, \\ ND_{k+1}(V, A) &= A \cup \left(V \xrightarrow{n} ND_k(V, A) \right), \quad k \geq 0. \end{aligned}$$

Here we denote as $\emptyset$ the empty nominative data (the same notation is used for the empty set). For the empty nominative data, we will also use the notation $[]$.

Data of this class have hierarchical structure, e.g. if $u, v, w \in V$ and $a, b \in A$, then

$$[u \mapsto a, v \mapsto [w \mapsto b]] \in ND(V, A).$$

Such data can be represented as oriented trees with arcs labeled by names and leafs labeled by atoms or empty data. A *path* is a nonempty finite sequence $(v_1, v_2, \dots, v_k)$ of names $v_1, \dots, v_k \in V$. For any data d , the *value of the path* $(v_1, v_2, \dots, v_k)$ in d is the value of the expression

$$d(v_1, v_2, \dots, v_k) \cong (\dots((d(v_1))(v_2))\dots(v_k)).$$

We say that a path $(v_1, v_2, \dots, v_k)$ is a *path in data* $d \in ND(V, A)$, if the value $(v_1, v_2, \dots, v_k)$ in d is defined, i.e. $d(v_1, v_2, \dots, v_k) \downarrow$.

A *terminal path* in a data $d \in ND(V, A)$ is a path in d such that its value belongs to $A \cup \{\emptyset\}$.

The least k such that $d \in ND_k(V, A)$ is called the *rank* of d .

The elements of D_1 are called the data of type TND_{AC} (data with abstract names and complex values).

3. Class of nominative data with complex names and abstract values over a nonempty set of names V and a nonempty set of abstract values A is:

$$D_2 = NDVS(V, A),$$

where $NDVS(V, A)$ is a set of all elements $A \cup (V^+ \xrightarrow{n} A)$ such that either $d \in A$, or $d \in V^+ \xrightarrow{n} A$ and all words in $dom(d)$ are pairwise incomparable in the sense of the prefix relation (*the principle of unambiguous associative naming*).

For example, if $u, v, w \in V$ and $a, b, c \in A$, then

$$[uv \mapsto a, uw \mapsto b, w \mapsto c] \in NDVS(V, A).$$

The data of this class have complex names, i.e. the names are words in the alphabet V .

The elements of D_2 are called data of type TND_{CA} (data with complex names and abstract values).

4. The class of nominative data with complex names and complex values over a nonempty set of names V and a nonempty set of abstract values A is:

$$D_3 = NDVC(V, A),$$

where $NDVC(V, A)$ is the class of all data $d \in ND(V^+, A)$ such that for any two paths $(u_1, u_2, \dots, u_k)$ and $(v_1, v_2, \dots, v_l)$ in d , neither of which is a prefix of another one, the words $u_1 u_2 \dots u_k$ and $v_1 v_2 \dots v_l$ are incomparable in the sense of the prefix relation (the principle of unambiguous associative naming). Such data are called *complex-named data*. For example, if $a \in A$ and $u, v, w \in V$, then

$$[uv \mapsto a, w \mapsto [uw \mapsto \emptyset]] \in NDVC(V, A).$$

Such data are hierarchical and have complex names and unambiguous naming.

The elements of D_3 are called data of type TND_{CC} (data with complex names and complex values).

4.2. Representation of data structures by nominative data

Let us give some arguments in support of the following principle:

The principle of representative completeness of nominative data [149]: *different forms of data used in computer information processing systems can be adequately represented as nominative data*

The examples are given below. Representations are chosen in such a way that basic operations of the mentioned structures correspond to simple operations on nominative data.

1. Array $(a_1, a_2, \dots, a_n)$ of elements of type T_0 .

The type of nominative data is: TND_{AA} , $V = \{1, 2, \dots\}$, $A = T_0$.

Representation:

$$[1 \mapsto a_1, 2 \mapsto a_2, \dots, n \mapsto a_n].$$

2. Two-dimensional array $\begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ \dots & \dots & \dots & \dots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{pmatrix}$ of ele-

ments of type T_0 .

The type of nominative data is: TND_{CA} (complex names),

$V = \{1, 2, \dots\}$, $A = T_0$.

Representation:

$$[i.j \mapsto a_{i,j} \mid i = 1, \dots, n, j = 1, \dots, m]$$

(here “.” denotes the concatenation of characters which forms a complex name).

3. An array of arrays $((a_{i,j})_{j=1}^m)_{i=1}^n$ of elements of type T_0 .

The type of nominative data is: TND_{AC} , $V = \{1, 2, \dots\}$,

$A = T_0$.

Representation:

$$[i \mapsto [j \mapsto a_{i,j} \mid j = 1, \dots, m] \mid i = 1, \dots, n].$$

4. An associative array $(a_k)_{k \in K}$ of elements of type T_0 , where K is a set of keys.

The type of nominative data is: TND_{AA} , $V = K$, $A = T_0$.

Representation:

$$[k \mapsto a_k \mid k \in K].$$

5. A **table**, where **Key** is a key attribute:

Key	Attr1	Attr2	...	AttrN
key_1	val_{11}	val_{21}	$\dots$	val_{N1}
key_2	val_{12}	val_{22}	$\dots$	val_{N2}
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
key_M	val_{1M}	val_{2M}	$\dots$	val_{NM}

The type of nominative data is: TND_{AC} ,
 $V = \{Attr_1, Attr_2, \dots, Attr_M\}$.

Representation:

$$\begin{aligned}
 &[key_1 \mapsto [Attr_1 \mapsto val_{11}, Attr_2 \mapsto val_{12}, \dots, Attr_M \mapsto val_{1M}], \\
 &key_2 \mapsto [Attr_1 \mapsto val_{21}, Attr_2 \mapsto val_{22}, \dots, Attr_M \mapsto val_{2M}]] . \\
 &\dots \\
 &key_N \mapsto [Attr_1 \mapsto val_{N1}, Attr_2 \mapsto val_{N2}, \dots, Attr_M \mapsto val_{NM}]] .
 \end{aligned}$$

6. A **linked list** of elements $e_1, e_2, \dots, e_n$.

The type of nominative data is: TND_{AC} , $V = \{data, next\}$.

Representation:

$$\begin{aligned}
 &[data \mapsto e_1, next \mapsto [data \mapsto e_2, next \mapsto \\
 &\mapsto [\dots data \mapsto e_n, next \mapsto \emptyset \dots]]] .
 \end{aligned}$$

7. A **cyclic list** of elements $e_1, e_2, \dots, e_n$.

The type of nominative data is: TND_{AC} ,

$$V = \{head, list, data, next\} \cup \{x_1, x_2, x_3, \dots, x_n\} .$$

Here $x_1, x_2, \dots$ are auxiliary names which represent the location of the elements of the list.

Representation:

$$\begin{aligned}
& [head \mapsto loc_1, list \mapsto [\\
& x_1 \mapsto [data \mapsto e_1, next \mapsto x_2], \\
& x_2 \mapsto [data \mapsto e_2, next \mapsto x_3], \\
& \dots \\
& x_n \mapsto [data \mapsto e_n, next \mapsto x_1]]].
\end{aligned}$$

8. A binary tree with nodes labeled by the elements $e_1, e_2, e_3, \dots$

The type of nominative data is: TND_{AC} , $V = \{data, left, right\}$.

Representation:

$$\begin{aligned}
& [data \mapsto e_1, \\
& left \mapsto [data \mapsto e_2, left \mapsto [\dots], right \mapsto [\dots]], \\
& right \mapsto [data \mapsto e_3, left \mapsto [\dots], right \mapsto [\dots] \\
&]].
\end{aligned}$$

9. Inductive data types. Data of such types are widely used in functional programming languages. They can be interpreted as terms which satisfy special conditions.

Let S be a finite set of sorts. Let Σ be a finite signature $\{c_1, c_2, \dots, c_n\}$, where c_i are *constructor* names. Assume that each constructor c_i has a (unique) associated *type* of the form $s_1 \times \dots \times s_m \rightarrow s$, where $s_i, s \in S$, $m \geq 0$. Let D be a free algebra of the signature Σ freely generated by some set A . Then the elements of the carrier D can be considered as elements of an inductive data type. Such data can be constructed by applying constructors.

Let us define a mapping τ which gives a representation of a data from D in the form of a nominative data of type TND_{AC} over the class of names $V = \{constructor, 1, 2, 3, \dots\}$ and atoms A :

- $\tau(x) = x$, if $x \in A$;

- $\tau(x) = [constructor \mapsto c, 1 \mapsto \tau(x_1), 2 \mapsto \tau(x_2), \dots, n \mapsto \tau(x_n)]$, if x has the form $c(x_1, x_2, \dots, x_n)$, where $c \in \Sigma$.

The application of constructors to such data corresponds to the naming operation on nominative data.

4.3. Algebras of nominative data

The main operations on nominative data include:

- *denaming* (obtains a value which corresponds to a name),
- *naming* (gives a new value to a name),
- *overlapping* (joins two data).

Let us define these operations formally for data of types TND_{CA} , TND_{AC} , and TND_{CC} . Similar operations can be easily defined for data of type TND_{AA} .

We will use the same notation for denaming, naming, and overlapping for each type TND_{CA} , TND_{AC} , and TND_{CC} . The interpretation of symbols of operations should be clear from the context.

Let V and A be fixed sets of names and atoms.

Definition 4.1 (denaming [149]).

1. For nominative data of type TND_{AC} *denaming* is a unary operation $v \Rightarrow_a$ with parameter $v \in V^+$ defined by induction on the length of v (the length of a word is denoted as $|v|$) as follows:

- if $|v| = 1$, then $v \Rightarrow_a (d) \equiv d(v)$;
- if $|v| = n > 1$, then $v \Rightarrow_a (d) \equiv v_1 \Rightarrow_a (x \Rightarrow_a (d))$, where

$$v = xv_1, \quad x \in V, \quad v_1 \in V^{n-1}.$$

For nominative data of types TND_{CA} and TND_{CC} *associative denaming* is a unary operation $v \Rightarrow_a$ with parameter $v \in V^+$ defined by induction on the length of v as follows:

- if $v \in V$, then

$$v \Rightarrow_a (d) \cong \begin{cases} d(v), & d(v) \downarrow; \\ d / v, & d(v) \uparrow \text{ or } d / v \neq \emptyset; \\ \text{undefined}, & d(v) \uparrow \text{ or } d / v = \emptyset, \end{cases}$$

where $d / u = [v_1 \mapsto d(v) \mid d(v) \downarrow, v = uv_1, v_1 \in V^+]$ (a *division* of a data by name);

- if $v \in V^n$ and $n > 1$, then $v \Rightarrow_a (d) \cong v_1 \Rightarrow_a (x \Rightarrow_a (d))$,

where $v = xv_1$, $x \in V$, $v_1 \in V^{n-1}$.

The following example illustrates these operations:

$$\begin{aligned} u \Rightarrow_a ([u \mapsto 1, v \mapsto 2]) &= 1; \\ (uv) \Rightarrow_a ([u \mapsto [vw \mapsto 1, u \mapsto 2]]) &= [w \mapsto 1]. \end{aligned}$$

The name of the operation (associative denaming) is related to the following property, which is called *associativity of denaming*:

$$u \Rightarrow_a (d) \cong u_n \Rightarrow_a (u_{n-1} \Rightarrow_a (...u_1 \Rightarrow_a (d)...))$$

for all $u, u_1, u_2, \dots, u_n \in V^+$ such that $u = u_1 u_2 \dots u_n$.

Definition 4.2 (naming [149]).

1. For nominative data of type TND_{AC} *naming* is a unary operation $\Rightarrow v$ with a parameter $v \in V^+$, defined by induction on the length of v as follows:

- $\Rightarrow v(d) = [v \mapsto d]$, if $v \in V$;
- $\Rightarrow v(d) = [v_1 \mapsto (\Rightarrow v_2(d))]$, if $v = v_1 v_2$, $v_1 \in V$

and $v_2 \in V^+$.

2. For nominative data of type TND_{CA} *naming* is a unary operation $\Rightarrow v$ with a parameter $v \in V^+$, defined by induction as follows:

- $\Rightarrow v(d) = [v \mapsto d]$, if $d \in A \cup \{\emptyset\}$;
- $\Rightarrow v(d) = [vu \mapsto d(u) \mid u \in \text{dom}(d)]$, if

$d \notin A \cup \{\emptyset\}$.

3. For nominative data of type TND_{CC} , *naming* is a unary operation $\Rightarrow v$ with a parameter $v \in V^+$ such that $\Rightarrow v(d) = [v \mapsto d]$.

Overlapping is an operation which updates the value in its first argument with the values defined in the second argument which correspond to the respective names.

For nominative data with complex names and/or values, one can consider different kinds of overlapping. We will consider two kinds of overlapping — **global** and **local** overlapping.

The global (associative or structural) overlapping ∇_a updates several values in the first argument, while the local overlapping ∇_a^v (with name parameter v) updates only one value which corresponds to the name v . The global overlapping can be used to formalize the semantics of procedure call, and local overlapping can be used to formalize the semantics of the assignment operator in programming languages. Informally, this operation joins two data and resolves name conflict in favor of the *second* argument.

Definition 4.3 (global overlapping [149]).

1. For nominative data of type TND_{AC} *global overlapping* is a partial binary operation ∇_a such that

- $d_1 \nabla_a d_2 = d_2 \cup d_1|_{\text{dom}(d_1) \setminus \text{dom}(d_2)}$, if $d_1 \notin A$ and $d_2 \notin A$;
- $d_1 \nabla_a d_2 \uparrow$, if $d_1 \in A$ or $d_2 \in A$.

2. For nominative data of type TND_{CA} *global overlapping* is a binary operation ∇_a such that

- $d_1 \nabla_a d_2 = d_2 \cup d_1|_{\text{dom}(d_1) \setminus (\text{dom}(d_2)V^*)}$;
- $d_1 \nabla_a d_2 \uparrow$, if $d_1 \in A$ or $d_2 \in A$,

where $\text{dom}(d_2)V^*$ denotes the set of all words of the form uv , where $u \in \text{dom}(d_2)$ and $v \in V^*$.

3. For nominative data of type TND_{CC} , *global overlapping* is a partial binary operation ∇_a , defined inductively by the rank of the first argument in the following way. Let

$$NDVC_k(V, A) = NDVC(V, A) \cap ND_k(V^+, A)$$

be the set of data from $NDVC(V, A)$ the rank of which does not exceed k .

Induction base. If $d_1 \in NDVC_0(V, A)$, then

$$v \Rightarrow_a (d) \cong \begin{cases} d_2, & d_1 = \emptyset \wedge d_2 \in NDVC(V, A) \setminus A; \\ \text{undefined}, & d_1 \in A \vee d_2 \in A. \end{cases}$$

Induction step. Assume that the value $d_1 \nabla_a d_2$ is already defined for all d_1, d_2 such that $d_1 \in NDVC_k(V, A)$. Let

$$d_1 \in NDVC_{k+1}(V, A) \setminus NDVC_k(V, A).$$

Then $d_1 \nabla_a d_2 = d$, where the data d is defined on names $u \in V^+$ as follows:

1) $d(u) = d_2(u)$, if $u \in \text{dom}(d_2)$ and u does not have a proper prefix which belongs to $\text{dom}(d_1)$;

2) $d(u) = d_1(u) \nabla_a (d_2 / u)$, if $d_1(u)$ is defined and does not belong to A , and u is a proper prefix of some element of $\text{dom}(d_2)$, where $d_2 / u = [v_1 \mapsto d_2(v) \downarrow, v = uv_1, v_1 \in V^+]$ is division of data by name;

3) $d(u) = d_2 / u$, if $d_1(u)$ is defined and belongs A , and u is a proper prefix of some element of $\text{dom}(d_2)$;

4) $d(u) = d_1(u)$, if $d_1(u)$ is defined and the word u is incomparable (in the sense of prefix relation) with each element of $\text{dom}(d_2)$;

5) $d(u) \uparrow$ otherwise.

The global overlapping on data of type TND_{CC} has the following properties [121, 126]:

- $[u \mapsto d_1] \nabla_a [v \mapsto d_2] = [u \mapsto d_1, v \mapsto d_2], u, v \in V, u \neq v;$
- $[uv \mapsto d_1] \nabla_a [u \mapsto d_2] = [u \mapsto d_2], u, v \in V^+,$ i.e. the value which corresponds to the name u in the 2-nd argument overrides the values of names in the 1-st argument which are extensions of u ;
- $[u \mapsto d_1] \nabla_a [uv \mapsto d_2] = [u \mapsto (d_1 \nabla_a [v \mapsto d_2])],$ if $u, v \in V^+, d_1 \notin A,$ i.e. the value which corresponds to the name uv in the second argument modifies the value of the names-prefixes of uv in the first argument;
- $\emptyset \nabla_a d = d \nabla_a \emptyset = d,$ if $d \notin A;$
- $d_1 \nabla_a d_2 \uparrow,$ if $d_1 \in A$ or $d_2 \in A.$

Definition 4.4 (local overlapping [149]).

1. For nominative data of type TND_{CA} *local overlapping* is a binary operation ∇_a^v with parameter $v \in V^+$ defined as follows:

$$d_1 \nabla_a^v d_2 \cong d_1 \nabla_a (\Rightarrow v(d_2)).$$

2. For nominative data of type TND_{AC} *local overlapping* is a binary operation ∇_a^v parameter $v \in V^+$ defined by induction on the length of the word v as follows:

- if $v \in V,$ then $d_1 \nabla_a^v d_2 \cong d_1 \nabla_a [v \mapsto d_2];$
- if $v = v_1 v_2,$ where $v_1 \in V, v_2 \in V^+, d_1(v_1) \downarrow, d_1(v_1) \notin A,$ then

$$d_1 \nabla_a^v d_2 \cong d_1 \nabla_a [v_1 \mapsto d_1(v_1) \nabla_a^{v_2} d_2];$$

- if $v = v_1 v_2 v_3 \dots v_n,$ where $v_i \in V,$ and $d_1(v_1) \uparrow$ or $d_1(v_1) \in A,$ then

$$d_1 \nabla_a^v d_2 \cong d_1 \nabla_a [v_1 \mapsto [v_2 \mapsto \dots \mapsto [v_n \mapsto d_2] \dots]].$$

3. For nominative data of type TND_{CC} , *local overlapping* is a binary operation ∇_a^v with parameter $v \in V^+$ defined as follows:

$$d_1 \nabla_a^v d_2 \cong d_1 \nabla_a (\Rightarrow v(d_2)).$$

Definition 4.5 (Algebras of nominative data).

1. An algebra of nominative data of type TND_{AC} is an algebra

$$NDA_{AC}(V, A) = \langle ND(V, A); \{v \Rightarrow_a\}_{v \in V^+}, \{\Rightarrow v\}_{v \in V^+}, \{\nabla_a^v\}_{v \in V^+} \rangle,$$

where $ND(V, A)$ is the carrier set, $\{v \Rightarrow_a\}_{v \in V^+}, \{\Rightarrow v\}_{v \in V^+}, \{\nabla_a^v\}_{v \in V^+}$ are operations on $ND(V, A)$.

2. An algebra of nominative data of type TND_{CA} is an algebra

$$NDA_{CA}(V, A) = \langle NDVS(V, A); \{v \Rightarrow_a\}_{v \in V^+}, \{\Rightarrow v\}_{v \in V^+}, \{\nabla_a^v\}_{v \in V^+} \rangle,$$

where $NDVS(V, A)$ is the carrier set, $\{v \Rightarrow_a\}_{v \in V^+}, \{\Rightarrow v\}_{v \in V^+}, \{\nabla_a^v\}_{v \in V^+}$ are operations on $NDVS(V, A)$.

3. An algebra of nominative data of type TND_{CC} is an algebra

$$NDA_{CC}(V, A) = \langle NDVC(V, A); \{v \Rightarrow_a\}_{v \in V^+}, \{\Rightarrow v\}_{v \in V^+}, \{\nabla_a^v\}_{v \in V^+} \rangle,$$

where $NDVC(V, A)$ is the carrier set, $\{v \Rightarrow_a\}_{v \in V^+}, \{\Rightarrow v\}_{v \in V^+}, \{\nabla_a^v\}_{v \in V^+}$ are operations on $NDVC(V, A)$.

Complex-named data have a hierarchical structure of naming. However, the information content accessible using associative denaming is identical for some distinct pairs of such data, e.g. $[v_1 \mapsto [v_2 \mapsto [v_3 \mapsto 1]]]$ and $[v_1 v_2 v_3 \mapsto 1]$ have a different hierarchical structure, but can be considered equivalent since the associative denaming behaves similarly on them.

The following relation on complex-named data, which is called nominative equivalences, formalizes this observation.

Definition 4.6 (path and terminal path [121, 126]).

1. A *path* in a complex-named data $d \in NDVC(V, A)$ is a nonempty sequence $(v_1, v_2, \dots, v_n)$ of names from V^+ such that $((d(v_1))(v_2) \dots)(v_n)$ is defined. The value $((d(v_1))(v_2) \dots)(v_n)$ is called the value of the path $(v_1, v_2, \dots, v_n)$ in d .

2. A *terminal path* in a complex-named data $d \in NDVC(V, A)$ is a path in d such that its value in d belongs to $A \cup \{\emptyset\}$.

3. A complex-named data $d_1 \in NDVC(V, A)$ is *weakly nominatively included* in a complex-named data $d_2 \in NDVC(V, A)$ (denoted as $d_1 \ll_w d_2$), if either $d_1, d_2 \in A$ and $d_1 = d_2$, or $d_1, d_2 \notin A$ and for each terminal path $(v_1, v_2, \dots, v_n)$ in d_1 which ends with an atom there exists a terminal path $(v_{1'}, v_{2'}, \dots, v_{m'})$ in d_2 such that $v_1 v_2 \dots v_n = v_{1'} v_{2'} \dots v_{m'}$ and the values of $(v_1, v_2, \dots, v_n)$ in d_1 and of $(v_{1'}, v_{2'}, \dots, v_{m'})$ in d_2 coincide, and also for each complex name $u \in V^+$ such that $u \Rightarrow_a (d_1) \downarrow \notin W$ the following holds: $u \Rightarrow_a (d_2) \downarrow \notin W$.

4. A complex-named data $d_1 \in NDVC(V, A)$ is *nominatively included* in a complex-named data $d_2 \in NDVC(V, A)$ (denoted as $d_1 \ll d_2$), if either $d_1, d_2 \in A$ and $d_1 = d_2$, or $d_1, d_2 \notin A$ and for each terminal path $(v_1, v_2, \dots, v_n)$ in d_1 there exists a terminal path $(v_{1'}, v_{2'}, \dots, v_{m'})$ in d_2 such that $\text{c} v_1 v_2 \dots v_n = v_{1'} v_{2'} \dots v_{m'}$ and the value of $(v_1, v_2, \dots, v_n)$ in d_1 and of $(v_{1'}, v_{2'}, \dots, v_{m'})$ in d_2 coincide.

5. Complex-named data $d_1, d_2 \in NDVC(V, A)$ are called *nominatively equivalent* (which is denoted as $d_1 \approx d_2$), if d_1 is *nom-*

inatively included in d_2 , and d_2 is *nominatively* included in d_1 .

The relation of nominative inclusion is a preorder on $NDVC(V, A)$, and nominative equivalence is an equivalence relation on $NDVC(V, A)$.

Nominatively equivalent data can have different hierarchical structures of naming, but they are equivalent to the same “flat” data of type TND_{CA} , e.g.:

- $[v_1 \mapsto [v_2 v_3 \mapsto 1, v_2 v_4 \mapsto 2]] \approx [v_1 v_2 v_3 \mapsto 1, v_1 v_2 v_4 \mapsto 2];$
- $[v_1 v_2 \mapsto [v_3 \mapsto 1, v_4 \mapsto 2]] \approx [v_1 v_2 v_3 \mapsto 1, v_1 v_2 v_4 \mapsto 2].$

Theorem 4.1. Nominative equivalence is a congruence relation on $NDA_{CC}(V, A)$.

Proof. From [117] it follows that for each pair $d_1, d_2 \in NDVC(V, A)$ the naming and associative denaming operations have the following property (called *nominative stability*): if $d_1 \approx d_2$ and the operation is defined on d_1 , then it is defined on d_2 and its values on d_1 and d_2 are nominatively equivalent. Similarly, from [117] it follows that if $v \in V^+$, $d_1, d_2 \notin A$, $d_1 \approx d_1'$, $d_2 \approx d_2'$, then $d_1 \nabla_a^v d_2 \approx d_1' \nabla_a^v d_2'$. Then nominative equivalence is a congruence on $NDA_{CC}(V, A)$.

This result allows one to define a quotient algebra of $NDA_{CC}(V, A)$.

Denote as $NDA_{CC}^{\approx}(V, A)$ the quotient algebra of $NDA_{CC}(V, A)$ by the relation of nominative equivalence $\approx$.

Lemma 4.1. $NDA_{AC}(V, A)$ is isomorphic to $NDA_{CC}^{\approx}(V, A)$.

Proof. Let us define $\iota_{13}^{\approx}: ND(V, A) \rightarrow 2^{NDVC(V, A)}$ as follows $\iota_{13}^{\approx}(d) = \{d' \in NDVC(V, A) \mid d' \approx d\}$ (note that $ND(V, A) \subset NDVC(V, A)$). It is easy to check that each nomina-

tive equivalence class contains the unique element of $ND(V, A)$ and that the operations on equivalence classes correspond to the operations on elements of TND_{AC} . Then $\iota_{13}^{\approx}$ is an isomorphism from $NDA_{AC}(V, A)$ to $NDA_{CC}^{\approx}(V, A)$.

Lemma is proved.

Lemma 4.2. $NDA_{CA}(V, A)$ is isomorphic to $NDA_{CC}^{\approx}(V, A)$.

Proof. Let $\iota_{23}^{\approx} : NDVS(V, A) \rightarrow 2^{NDVC(V, A)}$ be the function:
 $\iota_{23}^{\approx}(d) = \{d' \in NDVC(V, A) \mid d' \approx d\}$ (note that $NDVS(V, A) \subset NDVC(V, A)$). It is easy to check that each class of nominative equivalence contains the unique element which belongs to $NDVS(V, A)$ and that the operations on equivalence classes correspond to the operations on elements of TND_{CA} . Then $\iota_{23}^{\approx}$ is an isomorphism from $NDA_{CA}(V, A)$ to $NDA_{CC}^{\approx}(V, A)$.

Lemma is proved.

Theorem 4.2. Algebras $NDA_{AC}(V, A)$, $NDA_{CA}(V, A)$, and $NDA_{CC}^{\approx}(V, A)$ are isomorphic.

Proof follows from Lemma 4.1 and Lemma 4.2.

Denote as $D_3 / \approx$ the carrier of the algebra $NDA_{CC}^{\approx}(V, A)$.

Let $\iota_{13}^{\approx}$ and $\iota_{23}^{\approx}$ be isomorphisms, defined in the proofs of Lemma 4.1 and Lemma 4.2. For each $d \in D_3$, let $\iota_3^{\approx}(d)$ be the $\approx$ -equivalence class of the element d . Let us introduce the following inclusion maps for nominative data classes:

$$\iota_{01} : D_0 \hookrightarrow D_1, \iota_{02} : D_0 \hookrightarrow D_2, \iota_{13} : D_1 \hookrightarrow D_3, \iota_{23} : D_2 \hookrightarrow D_3.$$

It is easy to check that the diagram in Fig. 4.1 is commutative.

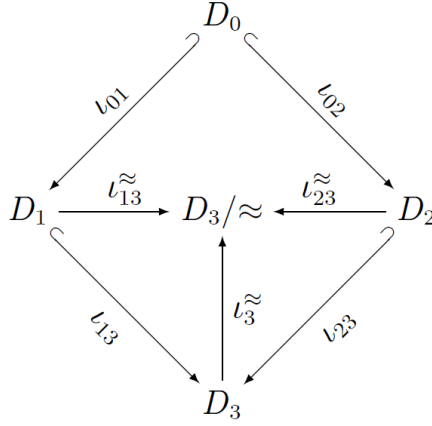


Fig. 4.1. Classes of nominative data and maps between them.

4.4. Operational semantics of operations on nominative data

In this section, we consider the operational semantics of programs over nominative data with complex names and complex values based on a special class of term rewriting systems. This semantics allows one to apply the existing methods applicable to term rewriting systems to the problems of transformation and verification of programs on complex data structures represented using nominative data.

Let V be a fixed set of basic names, W be a set of basic values (atoms). Then $NDVC(V, W)$ is the set of all complex-named data over V and W . We will denote as $pref(u) = \{v \in V^+ \mid \exists w \in V^* u = vw\}$ the set of nonempty prefixes of a word $u \in V^+$.

Consider the language of programs $SICON_{as}$ (a simple composition-nominative language with associative denaming). Each program in this language is considered as a partial function of the class $NDVC(V, W) \rightarrow NDVC(V, W)$, which can be obtained from the base functions using compositions.

Base functions are the family of naming operations $\Rightarrow v$ (for $v \in V^+$), the family of associative denaming operations $v \Rightarrow_a$ (for $v \in V^+$) and the constant function $\bar{\emptyset}$ with the value $\emptyset$.

The compositions are the simplest compositions of structured programming:

1. The sequence composition is the binary composition which maps a pair of functions f, g to a function h (denoted as $h = f \bullet g$) such that $h(d) \cong g(f(d))$.

2. *The loop* is a binary composition which maps a pair of functions p (condition) and g (loop body) to a function h (denoted as $h = p * g$) such that

- $h(d) = d$, if $p(d) \downarrow = \emptyset$,
- $h(d) = g^{(k)}(d)$, if $p(g^{(k)}(d)) \downarrow = \emptyset$ and $p(g^{(i)}(d)) \not\downarrow = \emptyset$ for all $i \in \{1, 2, \dots, k-1\}$ (where $g^{(k)}(d) = g(g(\dots g(d)\dots))$ is a k -time iteration of g applied to d),
- $h(d)$ is undefined, in other cases.

3. *Assignment* Asg^v with a parameter $v \in V^+$ is a unary composition defined by the expression $\text{Asg}^v(f)(d) \cong d \nabla_s [v \mapsto f(d)]$.

Functions expressible in this language form the set $\text{Sic}(V, W) = \bigcup_{k \geq 0} \text{Sic}_k(V, W)$, where

- $\text{Sic}_0(V, W) = \{\bar{\emptyset}\} \cup \{\Rightarrow u \mid u \in V^+\} \cup \{u \Rightarrow_a \mid u \in V^+\}$;
- $\text{Sic}_{k+1}(V, W) = \text{Sic}_k(V, W) \cup \{f \bullet g \mid f, g \in \text{Sic}_k(V, W)\} \cup \{p * g \mid p, g \in \text{Sic}_k(V, W)\} \cup \{\text{Asg}^v(f) \mid f \in \text{Sic}_k(V, W) \wedge \wedge v \in V^+\}$.

Despite its simplicity, the language SICON_{as} is quite expressive: it allows modeling of all partial recursive functions (i.e. this language is functionally complete) and it allows one to represent many common compositions as derived compositions (various conditional, switch, loop operators, etc.). It should be noted that all programs (functions) expressible in $\text{Sic}(V, W)$ are nominative stable.

4.4.1. Terms and term rewriting systems. Assume that each function symbol has a corresponding fixed arity.

Let $X = \{x_1, x_2, \dots\}$ be an infinite set of variable names.

For each signature Σ denote as $T(\Sigma, X)$ the set of terms in the signature Σ with variable names from V . Then $T(\Sigma, X)$ is the set of *closed* terms.

Let us introduce the notation $t|_p$ for each term $t \in T(\Sigma, X)$ and a finite sequence of positive integers $p = (i_1, \dots, i_n)$, $n \geq 0$ as follows:

- $t|_p \uparrow$, if $t \in X$ and $n \geq 1$ or $t = \bar{f}(t_1, \dots, t_m)$, where $\bar{f} \in \Sigma$, $m \geq 0$, $n \geq 1$ and $i_1 \notin \overline{1, m}$;
- $t|_p = t$, if $n = 0$;
- $\bar{f}(t_1, \dots, t_m)|_p \cong t_{i_1}|_{(i_2, \dots, i_n)}$, if $m \geq 1$, $n \geq 1$ and $j_1 \in \overline{1, n}$.

The sequences p such that $t|_p \downarrow$ are called *positions* in a term t . The empty sequence is called the *root position*.

The expression $t|_p$ is a term which corresponds to the position p .

The positions $p_1, \dots, p_k$ in t are called (pairwise) *incomparable*, if there is no pair of distinct indices $i, j \in \overline{1, k}$ such that one of the positions p_i and p_j is a prefix of another one.

Let us introduce the following notation (t, t_1, t_2 are terms):

- $\text{Var}(t)$ is the set of variable names which occur in t ;
- $t_1 \leq t_2$ — t_1 is a subterm of t_2 , i.e. $t_2 = t_1|_p$ for some position p in t_1 ;
- $t_1 < t_2$ — t_1 is a strict subterm of t_2 , i.e. $t_1 \leq t_2$ and $t_1 \neq t_2$.

A substitution is a total mapping $\sigma: X \rightarrow T(\Sigma, X)$ such that $\sigma(x) \neq x$ for a finite subset of elements $x \in X$.

We denote the application of substitution to a term t as $t\sigma$. Note that

- $t\sigma = \sigma(t)$, if $t \in X$,
- $t\sigma = \bar{f}(t_1\sigma, t_2\sigma, \dots, t_n\sigma)$, if $t = \bar{f}(t_1, \dots, t_n)$ for some $\bar{f} \in \Sigma$, $n \geq 0$ and terms $t_1, \dots, t_n$.

For partial function $\bar{\sigma}: X \rightarrow T(\Sigma, X)$, we denote as $\text{ext}(\bar{\sigma})$ the substitution which extends $\bar{\sigma}$ on X and satisfies $\text{ext}(\bar{\sigma})(x) = x$, if $x \in X$ and $\bar{\sigma}(x) \uparrow$.

Let $t, t_1, \dots, t_n \in T(\Sigma, X)$. In some cases, we will use the following notation: if $i_1, i_2, \dots, i_n$ are distinct natural numbers, then $t(t_1/x_{i_1}, \dots, t_n/x_{i_n})$ denotes the term obtained from t by placing t_j in place of x_{i_j} for $j = \overline{1, n}$.

If $p_1, \dots, p_n$ are incomparable positions in t , we will denote as $t(t_1/p_1, \dots, t_n/p_n)$ the term obtained from t by replacing a sub-term at the position p_i with t_i for each $i = \overline{1, n}$.

Definition 4.4.1. A constructor-orthogonal term rewriting system (CO TRS) is a triple $R = (\Sigma, C, P)$, where Σ is a signature, $C \subseteq \Sigma$ is a subset of *constructors*, and P is a set of rules of the form $l \rightarrow r$, $l, r \in T(\Sigma, X)$ such that:

1. Each rule $l \rightarrow r \in P$ satisfies the conditions:
 - the left-hand side l has a form $\bar{f}(p_1, p_2, \dots, p_n)$, where $\bar{f} \in \Sigma \setminus C$ is a function symbol, $p_1, \dots, p_n \in T(C, X)$, $n \geq 0$ are terms constructed from constructors and variable names (*patterns*);
 - $\text{Var}(r) \subseteq \text{Var}(l)$, i.e. variable names on the right-hand side must occur on the left-hand side;
 - no variable name occurs in l more than once (*left-linearity*).
2. The following *symbol non-ambiguity* condition is satisfied: in P there are no two distinct rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ such that $l_1\sigma = l_2\sigma'$ for some substitutions $\sigma, \sigma': X \rightarrow T(\Sigma, X)$.

Denote as $\rightarrow_R$ the rewrite relation for R , i.e. $t_1 \rightarrow_R t_2$, where $t_1, t_2 \in T(\Sigma, X)$, if $t_1|_p = l\sigma$ and $t_2 = t_1(r\sigma/p)$ for some rule $l \rightarrow r \in P$, position p in t_1 and a substitution σ . Denote as $\rightarrow_R^*$ the reflexive-transitive closure of $\rightarrow_R$.

Each CO TRS R is confluent, so each term $t \in T(\Sigma, X)$ has no more than one normal form, (i.e. a term t' such that $t \rightarrow_R^* t'$ and there is no t'' such that $t' \rightarrow_R t''$).

Let us introduce a partial function $nf_R : T(\Sigma, X) \rightarrow T(C, X)$:

- $nf_R(t)$ is a normal form of a term t , if it exists and belongs to $T(C, X)$;
- $nf_R(t)$ is undefined, otherwise.

CO TRS can be used to give recursive definitions to functions on the initial algebra of the signature C . However, here we use TRS to define function over an algebra which models complex-named data.

To define a function on such an algebra using CO TRS, we will introduce an additional unary function symbol ω which wraps variables which represent atomic values. It can be used in rules in order to distinguish atomic values and non-atomic values. We will also put some restrictions on rules in order to guarantee that the symbol ω is used correctly. Alternatively, order-sorted TRS could be used to define such functions, however, here we do not consider such TRS.

Let us fix a signature of constructors C .

Let F be a signature, $F \cap C = \emptyset$, $\omega \notin F \cup C$ is a unary function symbol.

Denote by $C_w = C \cup \{\omega\}$ the extended constructor signature.

Let $\mathcal{G} : T(C \cup F, X) \rightarrow T(C_w \cup F, X)$ be a mapping such that $\mathcal{G}(t) = t\sigma$, where σ is the substitution $ext(\{x \mapsto \omega(x) \mid x \in Var(t)\})$. This mapping is injective.

Denote by T_w the image of the set $T(C \cup F, X)$ under $\mathcal{G}$. Then the following equality holds: $T_w = \{t\sigma \mid t \in T(C, X)\}$,

$\sigma: X \rightarrow \{\omega(x) \mid x \in X\}$ is a substitution $\}$. For example, $c(\omega(x_1), \omega(x_2)) \in T_w^F$, if $c \in C$ is a binary symbol.

Also, denote by $T_w^F = \{t\sigma \mid t \in T(C \cup F, X)\}$, $\sigma: X \rightarrow X \cup \{\omega(x) \mid x \in X\}$ the substitution and there is no $x \in X$ such that $x = \sigma(y)$ and $\omega(x) = \sigma(y')$ for some $y, y' \in X$ $\}$. For example, $f(c(\omega(x_1)), x_2) \in T_w^F$, if $f \in F$ is a binary symbol, and $c \in C$ is a unary symbol.

Note that if $t \in T_w^F$ and $\sigma: X \rightarrow T_w$ is a substitution, then $t\sigma \in T_w$.

Denote by **R** the class of CO TRS $R = (C_w \cup F, C_w, P)$ such that for each rule $l \rightarrow r \in P$:

- if $\omega(t) \trianglelefteq l$ or $\omega(t) \trianglelefteq r$, then $t \in X$;
- if $\omega(x) \trianglelefteq l$, where $x \in X$, then $\omega(x) \trianglelefteq t$ for each t such that $x \triangleleft t$ and $t \trianglelefteq r$;
- if $\omega(x) \trianglelefteq r$, where $x \in X$, then $\omega(x) \trianglelefteq l$.

These conditions imply that if $t_1 \in T_w^F$ and $t_1 \rightarrow_R t_2$, then $t_2 \in T_w^F$.

Let us fix a CO TRS R of the class **R**.

Lemma 4.4.1. Let $t \in T(C_w \cup F, X)$. Then $t \in T_w$ if and only if the following conditions hold:

1. if $\omega(t') \trianglelefteq t$, where t' is a term, then $t' \in X$;
2. if $x \in X$, then $\omega(x) \trianglelefteq t'$ for each t' such that $x \triangleleft t'$ and $t' \trianglelefteq t$;
3. $t \notin X$.

Proof.

“Only if”: assume $t \in T_w$. Then $t = t_0\sigma$ for some $t_0 \in T(C \cup F, X)$ and a substitution $\sigma = \text{ext}(\{x \mapsto \omega(x) \mid x \in \text{Var}(t_0)\})$. Then from $\omega(t') \trianglelefteq t$ it follows that $\omega(t') \trianglelefteq \sigma(x)$ for some $x \in X$, whence $t' \in X$. Thus the condition 1 holds.

Let $x \triangleleft t'$ and $t' \trianglelefteq t$ for some $x \in X$ and a term t' . Then there exists $y \in \text{Var}(t_0)$ such that either $x \trianglelefteq \sigma(y) \trianglelefteq t'$, or $x \triangleleft t' \trianglelefteq \sigma(y)$. Then $\sigma(y) = \omega(x)$ and $\omega(x) \trianglelefteq t'$. I.e. the condition 2 holds.

Obviously, the condition 3 also holds.

“If”: assume the conditions 1-3. Let A be the set of positions p in t such that $t|_p$ has the form $\omega(t')$ for some t' . By the condition 1, for each $p \in A$, there exists (a unique) $x \in X$ which will be denoted by $\phi(p)$ such that $t|_p = \omega(x)$. It is easy to see that the positions in A are pairwise incomparable. Let $p_1, \dots, p_n$ be all distinct elements of A . Let $t_0 = t(\phi(p_1)/p_1, \dots, \phi(p_n)/p_n)$ and $\sigma = \text{ext}(\{x \mapsto \omega(x) \mid x \in \text{Var}(t_0)\})$. Then $t_0 \in T(C \cup F, X)$. From the conditions 2 and 3 it follows that for each position $p = (i_1, \dots, i_k)$ in t such that $t|_p \in X$ it holds $k \geq 1$ and $t|_{(i_1, \dots, i_{k-1})} = \omega(t|_p)$, so $(i_1, \dots, i_{k-1}) \in A$, $t_0|_{(i_1, \dots, i_{k-1})} = t|_p$ and $\sigma(t_0|_{(i_1, \dots, i_{k-1})}) = \omega(t|_p) = t|_{(i_1, \dots, i_{k-1})}$. Then $t_0\sigma = t$, thus $t \in T_w$.

Lemma is proved.

Corollary 1. If $t \in T_w$, $t' \trianglelefteq t$ and $t' \notin X$, then $t' \in T_w$.

Corollary 2. If $t, t' \in T_w$ and p is a position in t such that $t|_p \notin X$, then $t(t'/p) \in T_w$.

Lemma 4.4.2. If $l \rightarrow r \in P$, $\sigma: X \rightarrow T(C_w \cup F, X)$ is a substitution and $l\sigma \in T_w$, then $r\sigma \in T_w$.

Proof.

For each $x \in \text{Var}(l)$ consider the following cases:

- $\omega(x) \trianglelefteq l$. Then $\omega(\sigma(x)) \trianglelefteq l\sigma$ and by Lemma 4.4.1(1), $\sigma(x) \in X$.

- $\omega(x) \not\trianglelefteq l$. Then $\sigma(x) \trianglelefteq l\sigma$. If $\sigma(x) \in X$, then there exists a term $t' \trianglelefteq l\sigma$ such that $\sigma(x) \triangleleft t'$ and $\omega(\sigma(x)) \not\trianglelefteq t'$, which by Lemma 4.4.1(2) contradicts the assumption $l\sigma \in T_w$. Then $\sigma(x) \notin X$ and by Corollary 1 from Lemma 4.4.1, we have $\sigma(x) \in T_w$.

Thus for each $x \in \text{Var}(l)$, either $\sigma(x) \in X$, or $\sigma(x) \in T_w$.

Let us check the conditions of Lemma 4.4.1 for $r\sigma$.

1) Let $\omega(t') \leq r\sigma$, where t' is a term. Consider the cases:

- There exists a term t'' such that $\omega(t'') \leq r$ and $t''\sigma = t'$.

Then $t'' \in X$, $\omega(t'') \leq l$ and $\sigma(t'') \in X$. Then $t' \in X$.

- There exists $x \in \text{Var}(r)$ such that $\sigma(x) = \omega(t')$. Then $x \in \text{Var}(l)$ and $\sigma(x) \in T_w$. Then $t' \in X$.

Thus the condition 1 of Lemma 4.4.1 holds.

2) Let $x \in X$ and t' is a term such that $x \triangleleft t'$ i $t' \leq r\sigma$.

Consider the cases:

- There exists a term t'' such that $t'' \leq r$ and $t''\sigma = t'$.

Then $x \in \text{Var}(\sigma(y))$ and for some $y \in \text{Var}(t'')$. Then $y \in \text{Var}(r) \subseteq \text{Var}(l)$. If $\sigma(y) \in X$, then $\omega(y) \leq l$ and $x = \sigma(y)$. Then $\omega(x) \leq r$ and $\omega(x) \leq t'$. If $\sigma(y) \notin X$, then $\sigma(y) \in T_w$, so $\omega(x) \leq \sigma(y)$ and $\omega(x) \leq t'$.

- There exists $y \in \text{Var}(r)$ such that $\sigma(y) = t'$. Then $t' \notin X$, $y \in \text{Var}(l)$ and as above, $\sigma(y) \in T_w$. Then $t' \in T_w$, whence $\omega(x) \leq t'$.

Thus the condition 2 of Lemma 4.4.1 holds.

3) Let us check that $r\sigma \notin X$. If $r\sigma \in X$, then $r \in X$ and $\sigma(r) \in X$. Then $r \in \text{Var}(l)$ and $\omega(r) \not\leq l$, which contradicts the membership $l\sigma \in T_w$. Thus the condition 3 of Lemma 4.4.1 holds.

Thus $r\sigma \in T_w$.

Lemma is proved.

Lemma 4.4.3. If $t_1 \in T_w$ and $t_1 \rightarrow_R t_2$, then $t_2 \in T_w$.

Proof.

Let $t_1 \in T_w$, $t_1 \rightarrow_R t_2$. Then there exists a rule $l \rightarrow r \in P$, a position p in t_1 and a substitution σ such that $t_1|_p = l\sigma$ and $t_2 = t_1(r\sigma/p)$. Since $l\sigma \leq t_1$, $l\sigma \notin X$ and $l\sigma \in T_w$, by the Corollary 1 from Lemma 4.4.1, $l\sigma \in T_w$. Then by Lemma 4.4.2, $r\sigma \in T_w$ and by Corollary 2 from Lemma 4.4.1, $t_2 \in T_w$.

Lemma is proved.

4.4.2. Interpretation of terms and function symbols. Let $\mathbf{D}$ be a free algebra of signature C with the set of generators W . The notation $a \in \mathbf{D}$ will denote that a is an element of the carrier of $\mathbf{D}$.

Denote as $\mathbf{D}_*$ the algebra with the carrier $\mathbf{D}$ and the signature $C \cup W$, where the elements of W are considered as constants (0-ary operations).

Denote $T_*^0 = T(C \cup W, \emptyset)$. For each term $t \in T_*^0$ denote as $[t]_0$ (interpretation of t) the element of the carrier of $\mathbf{D}$ which is the value of t under interpretation in $\mathbf{D}_*$. Note that $[t]_0$ is a bijection between T_*^0 and the carrier of $\mathbf{D}$.

Let $t \in T_*^0$ and $p_1 < p_2 < \dots < p_n$ ($n \geq 0$) be all positions in the term t , at which there are constants from W . Denote:

- $ord(t) = n$,
- $\nu_s(t) = t(x_{s+1} / p_1, \dots, x_{s+n} / p_n) \in T(C, X)$,
- $\beta_s(t)$ is the substitution

$$ext(\{x_{s+j} \mapsto t(p_j) \mid j = \overline{s+1, s+n}\}).$$

Informally, $\nu_s(t)$ is a term obtained from t by replacing constants from W with different variables with indices which start with s , and the substitution $\beta_s(t)$ allows one to restore the term t from $\nu_s(t)$.

Let $R = (C_w \cup F, C_w, P)$ be a CO TRS of class $\mathbf{R}$.

Let us define the interpretation of terms from $T(C_w \cup F, X)$ using R .

Let $t \in T(C_w \cup F, X)$, $Var(t) = \{x_{i_1}, \dots, x_{i_k}\}$ and $m = i_k$, where $k \geq 0$ and $i_1 < i_2 < \dots < i_k$. Denote $[t]_{R, \mathbf{D}}$ (interpretation of t) be a partial function $\mathbf{D}^m \rightarrow \mathbf{D}$ defined as follows.

Let $D_1, \dots, D_m \in \mathbf{D}$. Denote as $t_1, \dots, t_m \in T_*^0$ terms such that $D_j = [t_j]_0$, $j = \overline{1, m}$. Let the numbers $s_1, \dots, s_m$ be defined as follows:

$s_1 = 0$, $s_j = s_{j-1} + \text{ord}_B(t_j)$, $j = \overline{1, m}$. Denote $t' = t(\nu_{s_1}(t_1)/x_1, \dots, \nu_{s_m}(t_m)/x_m)$. Note that thanks to the choice of s_j , the sets of names of variables occurring in terms $\nu_{s_1}(t_1), \dots, \nu_{s_m}(t_m)$ are pairwise disjoint. Then

- $[t]_{R, \mathbf{D}}(D_1, D_2, \dots, D_m) \Downarrow = [t''\beta_{s_1}(t_1)\beta_{s_2}(t_2)\dots\beta_{s_m}(t_m)]_0$, if $\mathcal{G}^{-1}(nf_R(\mathcal{G}(t')))\Downarrow = t''$;
- $[t]_{R, \mathbf{D}}(D_1, D_2, \dots, D_m) \Uparrow$, if $\mathcal{G}^{-1}(nf_R(\mathcal{G}(t')))\Uparrow$.

Note that from Lemma 4.4.3 it follows that if $nf_R(\mathcal{G}(t'))\Downarrow$, then $\mathcal{G}^{-1}(nf_R(\mathcal{G}(t')))\Downarrow$.

We will mainly consider interpretation not of arbitrary terms from $T(C_w \cup F, X)$, but of terms from the set T_w^F .

Let us define the interpretation of function symbols from F . For each symbol $\bar{f} \in F$ of arity m let $I_{R, \mathbf{D}}(\bar{f}) = [\bar{f}(x_1, x_2, \dots, x_m)]_{R, \mathbf{D}}$ (interpretation of $\bar{f}$).

Let us describe the class of functions which can be defined using CO TRS as follows: let $CTR_{\mathbf{D}}^m$, $m \geq 1$ be the class of partial functions $f : \mathbf{D}^m \rightarrow \mathbf{D}$ such that there exists a signature F , an m -ary symbol $\bar{f} \in F$ and CO TRS $R = (C_w \cup F, C_w, P)$ of class $\mathbf{R}$ such that $f = I_{R, \mathbf{D}}(\bar{f})$.

We will denote $CTR_{\mathbf{D}} = CTR_{\mathbf{D}}^1$ (unary functions).

Lemma 4.4.4. Let $t_1 \rightarrow_R^* t_2$ and $\text{Var}(t_1) = \text{Var}(t_2)$ for $t_1, t_2 \in T(C_w \cup F, X)$. Then $[t_1]_{R, \mathbf{D}} = [t_2]_{R, \mathbf{D}}$.

Proof.

Let $\text{Var}(t_1) = \{x_{i_1}, \dots, x_{i_k}\}$ and $m = i_k$, where $k \geq 0$ and $i_1 < i_2 < \dots < i_k$.

Let $D_1, \dots, D_m \in \mathbf{D}$, $t_1, \dots, t_m \in T_w^0$ be terms such that $D_j = [t_j]_0$, $j = \overline{1, m}$ and the numbers $s_1, \dots, s_m$ be defined as follows:

$s_1 = 0$, $s_j = s_{j-1} + \text{ord}_B(t_j)$. Denote $t'_l = t_l(v_{s_1}(t_1)/x_1, \dots, v_{s_m}(t_m)/x_m)$, $l = \overline{1, 2}$. Then

$$\mathcal{G}(t'_l) = t_l(v_{s_1}(t_1)/x_1, \dots, v_{s_m}(t_m)/x_m)\sigma = t_l(v_{s_1}(t_1)\sigma/x_1, \dots, v_{s_m}(t_m)\sigma/x_m)$$

for $l = \overline{1, 2}$, where $\sigma = \text{ext}(\{x \mapsto \omega(x) \mid x \in \bigcup_{1 \leq j \leq m} \text{Var}(v_{s_j}(t_j))\})$, $j = \overline{1, m}$, since $\text{Var}(t_l) \subseteq \{x_1, \dots, x_m\}$. Then there exists a substitution σ' such that $\mathcal{G}(t'_l) = t_l\sigma'$, $l = \overline{1, 2}$. Since $t_1 \rightarrow_R^* t_2$, we have $t_1\sigma' \rightarrow_R^* t_2\sigma'$.

If $nf_R(\mathcal{G}(t'_1)) \downarrow$, then $nf_R(\mathcal{G}(t'_2)) \downarrow = nf_R(\mathcal{G}(t'_1))$ (since R is confluent), so the values $\mathcal{G}^{-1}(nf_R(\mathcal{G}(t'_l)))$, $l = \overline{1, 2}$ are defined and equal, and $[t_1]_{R, \mathbf{D}}(D_1, D_2, \dots, D_m) = [t_2]_{R, \mathbf{D}}(D_1, D_2, \dots, D_m)$.

If $nf_R(\mathcal{G}(t'_1)) \uparrow$, then $nf_R(\mathcal{G}(t'_2)) \uparrow$, so $[t_1]_{R, \mathbf{D}}(D_1, D_2, \dots, D_m) \cong [t_2]_{R, \mathbf{D}}(D_1, D_2, \dots, D_m)$.

Then, since $D_1, \dots, D_m$ are arbitrary, we have $[t_1]_{R, \mathbf{D}} = [t_2]_{R, \mathbf{D}}$.

Lemma is proved.

Lemma 4.4.5. If $\bar{c} \in C$ is a function symbol, c is the corresponding operation on $\mathbf{D}$, then $I_{R, \mathbf{D}}(\bar{c}) = c$.

Proof.

Let m be the arity of c . It is sufficient to show that $[t]_{R, \mathbf{D}} = c$, where $t = \bar{c}(x_1, x_2, \dots, x_m)$.

Let $D_1, \dots, D_m \in \mathbf{D}$, $t_1, \dots, t_m \in T^0$ be terms such that $D_j = [t_j]_0$, $j = \overline{1, m}$ and the numbers $s_1, \dots, s_m$ be defined as $s_1 = 0$, $s_j = s_{j-1} + \text{ord}_B(t_j)$, $j = \overline{1, m}$.

Let $t' = t(v_{s_1}(t_1)/x_1, \dots, v_{s_m}(t_m)/x_m) = \bar{c}(v_{s_1}(t_1), \dots, v_{s_m}(t_m))$. Then $t' \in T(C, X)$, so $\mathcal{G}(t') \in T(C_w, X)$, so $nf_R(\mathcal{G}(t')) = \mathcal{G}(t')$. Then

$$\begin{aligned}
[t]_{R,D}(D_1, D_2, \dots, D_m) \downarrow &= [t' \beta_{s_1}(t_1) \beta_{s_2}(t_2) \dots \beta_{s_m}(t_m)]_0 = \\
&= [\bar{c}(t_1, \dots, t_m)]_0 = c([t_1]_0, \dots, [t_m]_0) = c(D_1, \dots, D_m).
\end{aligned}$$

Since $D_1, \dots, D_m$ are arbitrary, we have $I_{R,D}(\bar{c}) = [t]_{R,D} = c$.

Lemma is proved.

4.4.3. Term rewriting system representation of programs over complex-named data. Let V be a finite set of names. We will assume that V consists of two names v_1, v_2 , since in the general case the constructions described below can be done analogously.

Let us construct a free algebra $\mathbf{D}$ with a finite signature C such that the functions from $CTR_{\mathbf{D}}$ in a certain sense can be used to perform complex-named data processing. More specifically, each function f expressible in $SICON_{as}$ can be represented as $f = \gamma \bullet g \bullet \gamma^{-1}$, where $g \in CTR_{\mathbf{D}}$, and γ is a fixed bijection between the carrier of $\mathbf{D}$ and $NDVC(V, W)$. Then a CO TRS for g can be effectively constructed from the term of f in $SICON_{as}$ language.

Let $C = \{e, \phi_1, \phi_2, \phi'_1, \phi'_2, \psi_{11}, \psi_{12}, \psi_{21}, \psi_{22}\}$, where e is a constant symbol, $\phi_1, \phi_2, \phi'_1, \phi'_2$ are unary function symbols, $\psi_{11}, \psi_{12}, \psi_{21}, \psi_{22}$ are binary function symbols.

Let $\mathbf{D}$ be a free algebra with signature C and base W . For each function symbol $c \in C$ let us denote as $\hat{c}$ the corresponding operation (function) on $\mathbf{D}$.

Let us define the following auxiliary mapping $\chi: NDVC(V, W) \rightarrow NDVC(V, W) \setminus (W \cup \{\emptyset\})$:

- $\chi(d) = [v_1 \mapsto d]$, if $d \in ND(\{v_1\}, W)$;
- $\chi(d) = d$ otherwise.

Note that

$$\begin{aligned}
ND(\{v_1\}, W) &= W \cup \{\emptyset\} \cup \\
&\cup \{[v_1 \mapsto [v_1 \mapsto \dots \mapsto [v_1 \mapsto w] \dots]] \mid w \in W \cup \{\emptyset\}\}.
\end{aligned}$$

The definition implies that χ indeed takes values in $NDVC(V, W) \setminus (W \cup \{\emptyset\})$.

It is easy to see that the mapping $\chi^{-1} : NDVC(V, W) \setminus (W \cup \{\emptyset\}) \rightarrow NDVC(V, W)$, is defined below is an inverse to χ , so χ is a bijection:

- $\chi^{-1}(d) = d(v_1)$, if $d \in ND(\{v_1\}, W) \setminus (W \cup \{\emptyset\})$;
- $\chi^{-1}(d) = d$ otherwise.

Let us introduce the following operations on $NDVC(V, W)$: $v_i \cdot$, where $i \in \{1, 2\}$, is a unary operation (product of a simple name and a data), defined as follows: $v_i \cdot d = [v_i u \mapsto d(u) \mid u \in rn(d)]$, if $d \notin W$ and $v_i \cdot d \uparrow$, if $d \in W$.

Let us defined a mapping $\gamma : \mathbf{D} \rightarrow NDVC(V, W)$ as follows (where $D, D_1, D_2 \in \mathbf{D}$):

- $\gamma(e) = \emptyset$;
- $\gamma(a) = a$, $a \in W$;
- $\gamma(\varphi_1(D)) = [v_1 \mapsto \gamma(D)]$;
- $\gamma(\varphi_2(D)) = [v_2 \mapsto \gamma(D)]$;
- $\gamma(\varphi'_i(D)) = v_i \cdot \chi(\gamma(D))$, $i = 1, 2$;
- $\gamma(\psi_{11}(D_1, D_2)) = [v_1 \mapsto \gamma(D_1), v_2 \mapsto \gamma(D_2)]$;
- $\gamma(\psi_{12}(D_1, D_2)) = [v_1 \mapsto \gamma(D_1)] \cup v_2 \cdot \chi(\gamma(D_2))$;
- $\gamma(\psi_{21}(D_1, D_2)) = v_1 \cdot \chi(\gamma(D_1)) \cup [v_2 \mapsto \gamma(D_2)]$;
- $\gamma(\psi_{22}(D_1, D_2)) = v_1 \cdot \chi(\gamma(D_1)) \cup v_2 \cdot \chi(\gamma(D_2))$.

The definition implies that γ indeed takes values in $NDVC(V, W)$.

It is easy to see that the mapping $\gamma^{-1} : NDVC(V, W) \rightarrow \mathbf{D}$ defined below is inverse for χ , so χ is a bijection:

- $\gamma^{-1}(\emptyset) = e$;
- $\gamma^{-1}(a) = a$, $a \in W$;
- $\gamma^{-1}([v_1 \mapsto d]) = \varphi_1(\gamma^{-1}(d))$;

- $\gamma^{-1}([v_2 \mapsto d]) = \varphi_2(\gamma^{-1}(d))$;
- $\gamma^{-1}(v_i \cdot d) = \varphi'_i(\gamma^{-1}(\chi^{-1}(d)))$, $d \notin W \cup \{\emptyset\}$, $i = 1, 2$;
- $\gamma^{-1}([v_1 \mapsto d_1, v_2 \mapsto d_2]) = \psi_{11}(\gamma^{-1}(d_1), \gamma^{-1}(d_2))$;
- $\gamma^{-1}([v_1 \mapsto d_1] \cup v_2 \cdot d_2) = \psi_{12}(\gamma^{-1}(d_1), \gamma^{-1}(\chi^{-1}(d_2)))$,
 $d_2 \notin W \cup \{\emptyset\}$;
- $\eta(v_1 \cdot d_1 \cup [v_2 \mapsto d_2]) = \psi_{21}(\gamma^{-1}(\chi^{-1}(d_1)), \gamma^{-1}(d_2))$,
 $d_1 \notin W \cup \{\emptyset\}$;
- $\gamma^{-1}(v_1 \cdot d_1 \cup v_2 \cdot d_2) = \psi_{22}(\gamma^{-1}(\chi^{-1}(d_1)), \gamma^{-1}(\chi^{-1}(d_2)))$,
 $d_1, d_2 \notin W \cup \{\emptyset\}$.

Let us denote: if $\bar{f}$ is a unary function symbol and t is a term, then $f^{(n)}(t)$ denotes the term $\underbrace{f(f(\dots f(t)\dots))}_n$, $n \geq 0$.

Lemma 4.4.6. The functions $\gamma \bullet \chi \bullet \gamma^{-1}$ and $\gamma \bullet \chi^{-1} \bullet \gamma^{-1}$ (the latter is partial) belong to CTR_D .

Proof.

1. Let us prove that $\gamma \bullet \chi \bullet \gamma^{-1} \in CTR_D$.

Let $F = \{\bar{f}\}$ and let us define a CO TRS R with the rules:

- $\bar{f}(e) \mapsto \varphi_1(e)$;
- $\bar{f}(\omega(x_1)) \mapsto \varphi_1(\omega(x_1))$;
- $\bar{f}(\varphi_1(x_1)) \mapsto \varphi_1(\bar{f}(x_1))$;
- $\bar{f}(c(x_1)) \mapsto c(x_1)$ for $c \in \{\varphi_2, \varphi'_1, \varphi'_2\}$;
- $\bar{f}(c(x_1, x_2)) \mapsto c(x_1, x_2)$ for $c \in \{\psi_{i,j} \mid i, j = \overline{1, 2}\}$.

It is easy to see that if $t \in T_w$, then

- $nf_R(\bar{f}(t)) \downarrow = \varphi_1^{(n+1)}(e)$, if $t = \varphi_1^{(n)}(e)$ for some $n \geq 0$;
- $nf_R(\bar{f}(t)) \downarrow = \varphi_1^{(n+1)}(\omega(x))$, if $t = \varphi_1^{(n)}(\omega(x))$ for some $n \geq 0$, $x \in X$;
- $nf_R(\bar{f}(t)) \downarrow = t$ otherwise.

Denote $f = I_{R,D}(\bar{f})$. Then the following holds:

- $f(\hat{\phi}_1^{(n)}(D)) \downarrow = \hat{\phi}_1^{(n+1)}(D)$, if $n \geq 0$ and $D \in W \cup \{e\}$;
- $f(D) \downarrow = D$, if D cannot be represented as $\hat{\phi}_1^{(n)}(D')$,

where $D' \in W \cup \{e\}$, $n \geq 0$.

Then for each $d \in NDVC(V, W)$,

- $f(\gamma^{-1}(d)) = \gamma^{-1}([v_1 \mapsto d])$, if $d \in ND(\{v_1\}, W)$.
- $f(\gamma^{-1}(d)) = \gamma^{-1}(d)$, if $d \notin ND(\{v_1\}, W)$;

Thus $f(\gamma^{-1}(d)) = \gamma^{-1}(\chi(d))$ for all $d \in NDVC(V, W)$. Then $f(D) = \gamma^{-1}(\chi(\gamma(D)))$ for all $D \in \mathbf{D}$, whence $f = \gamma \bullet \chi \bullet \gamma^{-1} \in CTR_{\mathbf{D}}$.

2. Let us prove that $\gamma \bullet \chi^{-1} \bullet \gamma^{-1} \in CTR_{\mathbf{D}}$.

Let $F = \{\bar{f}\}$ and let us define a CO TRS R with the rules:

- $\bar{f}(\phi_1(e)) \mapsto e$;
- $\bar{f}(\phi_1(\omega(x_1))) \mapsto \omega(x_1)$;
- $\bar{f}(\phi_1(\phi_1(x_1))) \mapsto \phi_1(\bar{f}(\phi_1(x_1)))$;
- $\bar{f}(c(x_1)) \mapsto c(x_1)$ for $c \in \{\phi_2, \phi'_1, \phi'_2\}$;
- $\bar{f}(c(x_1, x_2)) \mapsto c(x_1, x_2)$ for $c \in \{\psi_{i,j} \mid i, j = \overline{1, 2}\}$.

Then if $t \in T_w$, then

- $nf_R(\bar{f}(t)) \downarrow = \phi_1^{(n-1)}(e)$, if $t = \phi_1^{(n)}(e)$ for some $n \geq 1$;
- $nf_R(\bar{f}(t)) \downarrow = \phi_1^{(n-1)}(\omega(x))$, if $t = \phi_1^{(n)}(\omega(x))$ for some $n \geq 1$, $x \in X$;
- $nf_R(\bar{f}(t)) \uparrow$, if $t = e$ or $t = \omega(x)$ for some $x \in X$;
- $nf_R(\bar{f}(t)) \downarrow = t$ otherwise.

Denote $f = I_{R, \mathbf{D}}(\bar{f})$. Then the following holds:

- $f(\hat{\phi}_1^{(n)}(D)) \downarrow = \hat{\phi}_1^{(n-1)}(D)$, if $n \geq 1$ and $D \in W \cup \{e\}$;
 - $f(D) \uparrow$, if $D \in W \cup \{e\}$;
 - $f(D) \downarrow = D$, if D cannot be represented as $\hat{\phi}_1^{(n)}(D')$,
- where $D' \in W \cup \{e\}$, $n \geq 0$.

Then for each $d \in NDVC(V, W)$,

- $f(\gamma^{-1}(d)) = \gamma^{-1}(d(v_1))$, if $d \in ND(\{v_1\}, W) \setminus (W \cup \{\emptyset\})$

- $f(\gamma^{-1}(d)) \uparrow$, if $d \in W \cup \{\emptyset\}$;
- $f(\gamma^{-1}(d)) = \gamma^{-1}(d)$, if $d \notin ND(\{v_1\}, W)$.

Thus $f(\gamma^{-1}(d)) \cong \gamma^{-1}(\chi^{-1}(d))$ for all $d \in NDVC(V, W)$.

Then $f(D) \cong \gamma^{-1}(\chi^{-1}(\gamma(D)))$ for all $D \in \mathbf{D}$, whence $f = \gamma \bullet \chi^{-1} \bullet \gamma^{-1} \in CTR_{\mathbf{D}}$.

Lemma is proved.

Lemma 4.4.7. (About composition). Let $f_0 \in CTR_{\mathbf{D}}^n$ and $f_1, \dots, f_n \in CTR_{\mathbf{D}}^m$, where $n, m \geq 1$. Then $h \in CTR_{\mathbf{D}}$, where h is a function such that $h(D_1, \dots, D_m) \cong f(f_1(D_1, \dots, D_m), \dots, f_n(D_1, \dots, D_m))$ for all $D_1, \dots, D_m \in \mathbf{D}$. Moreover, CO TRS for h can be effectively constructed from CO TRS for $f_0, f_1, \dots, f_n$.

Proof.

Let for each $i = \overline{0, n}$, $R_i = (F_i \cup C_w, C_w, P_i)$ be CO TRS of class **R** with distinguished symbol, $\bar{f}_i \in F_i$ such that $I_{R, \mathbf{D}}(\bar{f}_i) = f_i$. Without loss of generality, we can assume that the sets F_i , $i = \overline{0, n}$ are pairwise disjoint. Let $\bar{g}, \bar{h}_1, \dots, \bar{h}_n \notin \bigcup_{i=0}^n F_i$ be new function symbols, $\bar{g}$ is a binary symbol, $\bar{h}_1, \dots, \bar{h}_n$ are m -ary symbols. Then $R = (\{\bar{g}, \bar{h}_1, \dots, \bar{h}_n\} \cup \bigcup_{i=0}^n F_i \cup C_w, C_w, P \cup \bigcup_{i=0}^n P_i)$ is a CO TRS of class **R**, where P is the set of rules

$$\begin{aligned}
& \bar{h}_1(x_1, \dots, x_m) \mapsto \bar{g}(\bar{f}_0(\bar{f}_1(x_1, \dots, x_m), \dots, \bar{f}_n(x_1, \dots, x_m), \bar{f}_1(x_1, \dots, x_m))) \\
& \bar{h}_i(x_1, \dots, x_m) \mapsto \bar{g}(\bar{h}_{i-1}(x_1, \dots, x_m), \bar{f}_i(x_1, \dots, x_m)), \quad i = \overline{2, n} \\
& \bar{g}(x_1, e) \mapsto x_1, \quad \bar{g}(x_1, \omega(x_2)) \mapsto x_1 \\
& \bar{g}(x_1, c(x_2)) \mapsto \bar{g}(x_1, x_2) \text{ for } c \in \{\varphi_1, \varphi_2, \varphi'_1, \varphi'_2\} \\
& \bar{g}(x_1, c(x_2, x_3)) \mapsto \bar{g}(\bar{g}(x_1, x_2), x_3) \text{ for } c \in \{\psi_{i,j} \mid i, j \in \overline{1, 2}\}
\end{aligned}$$

These rules imply that if $t_1, t_2 \in T_w^F$, then $nf_R(\bar{g}(t_1, t_2)) \downarrow$ if and only if $nf_R(t_1) \downarrow$ and $nf_R(t_2) \downarrow$. Moreover,

$nf_R(\bar{g}(t_1, t_2)) = nf_R(t_1)$, if $nf_R(\bar{g}(t_1, t_2)) \downarrow$. Then we have
 $I_{R, \mathbf{D}}(\bar{h}_n) = h \in CTR_{\mathbf{D}}$, where

$h(D_1, \dots, D_m) \cong f(f_1(D_1, \dots, D_m), \dots, f_n(D_1, \dots, D_m))$ for $D_1, \dots, D_m \in \mathbf{D}$.

Lemma is proved.

Lemma 4.4.8. (About condition). Let $f_0, f_1, f_2 \in CTR_{\mathbf{D}}$ and the function $h: \mathbf{D} \rightarrow \mathbf{D}$ is defined as $h(D) \cong f_1(D)$, if $f_0(D) \downarrow = e$, $h(D) \cong f_2(D)$, if $f_0(D) \downarrow \neq e$, and $h(D) \uparrow$, if $f_0(D) \uparrow$. Then $h \in CTR_{\mathbf{D}}$, and a CO TRS for h can be effectively constructed from CO TRS for f_0, f_1, f_2 .

Proof.

Let for each $i = \overline{0, 2}$, $R_i = (F_i \cup C_w, C_w, P_i)$ be CO TRS of class **R** with distinguished symbol, $\bar{f}_i \in F_i$ such that $I_{R, \mathbf{D}}(\bar{f}_i) = f_i$. Without loss of generality, we can assume that the sets F_i , $i = \overline{0, 2}$ are pairwise disjoint. Let $\bar{g}, \bar{h}, \bar{h}' \notin \bigcup_{i=0}^2 F_i$ be new function symbols, $\bar{g}$ is a binary symbol, $\bar{h}$ is a unary symbol, $\bar{h}'$ is a 3-ary symbol. Then $R = (\{\bar{g}, \bar{h}, \bar{h}'\} \cup \bigcup_{i=0}^2 F_i \cup C_w, C_w, P \cup \bigcup_{i=0}^2 P_i)$ is a CO TRS of class **R**, where P is the set of rules:

$$\begin{aligned} \bar{h}(x_1) &\mapsto \bar{g}(\bar{h}'(\bar{f}_0(x_1), \bar{f}_1(x_1), \bar{f}_2(x_1)), \bar{f}_0(x_1)), \\ \bar{h}'(e, x_2, x_3) &\mapsto x_2, \\ \bar{h}'(c(x_1), x_2, x_3) &\mapsto x_3 \text{ for } c \in \{\omega, \varphi_1, \varphi_2, \varphi'_1, \varphi'_2\}, \\ \bar{h}'(c(x_1, x_2), x_2, x_3) &\mapsto x_3 \text{ for } c \in \{\psi_{i,j} \mid i, j \in \overline{1, 2}\}, \\ \bar{g}(x_1, e) &\mapsto x_1, \bar{g}(x_1, \omega(x_2)) \mapsto x_1, \\ \bar{g}(x_1, c(x_2)) &\mapsto \bar{g}(x_1, x_2) \text{ for } c \in \{\varphi_1, \varphi_2, \varphi'_1, \varphi'_2\}, \\ \bar{g}(x_1, c(x_2, x_3)) &\mapsto \bar{g}(\bar{g}(x_1, x_2), x_3) \text{ for } c \in \{\psi_{i,j} \mid i, j \in \overline{1, 2}\}. \end{aligned}$$

These rules imply that if $t_1, t_2 \in T_w^F$, then $nf_R(\bar{g}(t_1, t_2)) \downarrow$ if and only if $nf_R(t_1) \downarrow$ and $nf_R(t_2) \downarrow$. Moreover,

$nf_R(\bar{g}(t_1, t_2)) = nf_R(t_1)$, if $nf_R(\bar{g}(t_1, t_2)) \downarrow$. Then we have $I_{R, \mathbf{D}}(\bar{h}) = h \in CTR_{\mathbf{D}}$, where $h(D) \cong f_1(D)$, if $f_0(D) \downarrow = e$, $h(D) \cong f_2(D)$, if $f_0(D) \downarrow \neq e$ and $h(D) \uparrow$, if $f_0(D) \uparrow$.

Lemma is proved.

Lemma 4.4.9. (About loop). Let $f_0, f_1 \in CTR_{\mathbf{D}}$ and the function $h: \mathbf{D} \rightarrow \mathbf{D}$ is defined as follows:

- 1) $h(D) = D$, if $f_0(D) \downarrow = e$,
- 2) $h(D) = f_1^{(n)}(D)$, if $n \geq 1$, $f_0(f_1^{(n)}(D)) \downarrow = e$ and $f_0(f_1^{(i)}(D)) \downarrow \neq e$ for $i = \overline{0, n-1}$,
- 3) $h(D) \uparrow$ otherwise.

Then $h \in CTR_{\mathbf{D}}$ and CO TRS for h can be effectively constructed from CO TRS for f_0 and f_1 .

Proof.

For each $i = \overline{0, 1}$ let $R_i = (F_i \cup C_w, C_w, P_i)$ be CO TRS of class $\mathbf{R}$ with distinguished symbol, $\bar{f}_i \in F_i$ such that $I_{R, \mathbf{D}}(\bar{f}_i) = f_i$. Without loss of generality, we can assume that the sets F_i , $i = \overline{0, 1}$ are pairwise disjoint. Let $\bar{g}, \bar{h}, \bar{h}' \notin F_0 \cup F_1$ be new function symbols, $\bar{g}$ is a binary symbol, $\bar{h}$ is unary symbol, $\bar{h}'$ is 3-ary symbol. Then $R = (\{\bar{g}, \bar{h}, \bar{h}'\} \cup F_0 \cup F_1 \cup C_w, C_w, P \cup P_0 \cup P_1)$ is a CO TRS of class $\mathbf{R}$, where P is the set of rules:

$$\begin{aligned}
& \bar{h}(x_1) \mapsto \bar{g}(\bar{h}'(\bar{f}_0(x_1), x_1, \bar{h}(\bar{f}_1(x_1))), \bar{f}_0(x_1)), \\
& \bar{h}'(e, x_2, x_3) \mapsto x_2, \\
& \bar{h}'(c(x_1), x_2, x_3) \mapsto x_3 \text{ for } c \in \{\omega, \varphi_1, \varphi_2, \varphi'_1, \varphi'_2\}, \\
& \bar{h}'(c(x_1, x_2), x_2, x_3) \mapsto x_3 \text{ for } c \in \{\psi_{i,j} \mid i, j \in \overline{1, 2}\}, \\
& \bar{g}(x_1, e) \mapsto x_1, \bar{g}(x_1, \omega(x_2)) \mapsto x_1, \\
& \bar{g}(x_1, c(x_2)) \mapsto \bar{g}(x_1, x_2) \text{ for } c \in \{\varphi_1, \varphi_2, \varphi'_1, \varphi'_2\}, \\
& \bar{g}(x_1, c(x_2, x_3)) \mapsto \bar{g}(\bar{g}(x_1, x_2), x_3) \text{ for } c \in \{\psi_{i,j} \mid i, j \in \overline{1, 2}\}.
\end{aligned}$$

From these rules, it follows that if $t_1, t_2 \in T_w^F$, then $nf_R(\bar{g}(t_1, t_2)) \downarrow$ if and only if $nf_R(t_1) \downarrow$ and $nf_R(t_2) \downarrow$. Moreover, $nf_R(\bar{g}(t_1, t_2)) = nf_R(t_1)$, if $nf_R(\bar{g}(t_1, t_2)) \downarrow$. Then we have: if $t \in T_w$, $n \geq 0$, $nf_R(\bar{f}_0(\bar{f}_1^{(i)}(t))) \downarrow \neq e$ for $i = \overline{0, n-1}$ and $nf_R(\bar{f}_0(\bar{f}_1^{(n)}(t))) \downarrow = e$, then

$$\begin{aligned} \bar{h}(t) &\rightarrow_R \bar{g}(\bar{h}(\bar{f}_1(t)), \bar{f}_0(t)) \rightarrow_R \bar{g}(\bar{g}(\bar{h}(\bar{f}_1(\bar{f}_1(t))), \bar{f}_1(\bar{f}_0(t_1))), \\ \bar{f}_0(t_1)) &\rightarrow_R \bar{g}(\dots \bar{g}(\bar{g}(\bar{h}(\bar{f}_1^{(n)}(t)), \bar{f}_{n-1}(\dots(\bar{f}_1(\bar{f}_0(t)))\dots), \dots, \bar{f}_0(t)) \rightarrow_R \\ &\rightarrow_R \bar{h}(\bar{f}_1^{(n)}(t)) \rightarrow_R \rightarrow_R \bar{g}(\bar{h}'(\bar{f}_0(\bar{f}_1^{(n)}(x_1)), \bar{f}_1^{(n)}(t), \bar{h}(\bar{f}_1(\bar{f}_1^{(n)}(t)))), \dots \\ \bar{f}_0(\bar{f}_1^{(n)}(t))) &\rightarrow_R \bar{g}(\bar{h}'(e, \bar{f}_1^{(n)}(t), \bar{h}(\bar{f}_1(\bar{f}_1^{(n)}(t)))), \bar{f}_0(\bar{f}_1^{(n)}(t))) \rightarrow_R \\ &\bar{g}(\bar{f}_1^{(n)}(t), \bar{f}_0(\bar{f}_1^{(n)}(t))) \rightarrow_R \bar{f}_1^{(n)}(t). \end{aligned}$$

On the other hand, if either $nf_R(\bar{f}_0(\bar{f}_1^{(n)}(t))) \uparrow$ for some $n \geq 0$ such that $nf_R(\bar{f}_0(\bar{f}_1^{(i)}(t))) \downarrow \neq e$ for all $i = \overline{0, n-1}$, or $nf_R(\bar{f}_0(\bar{f}_1^{(n)}(t))) \downarrow \neq e$ for all $n \geq 0$, then $nf_R(\bar{h}(t)) \uparrow$. Thus $I_{R,D}(\bar{h}) = h \in CTR_D$, where $h(D) = D$, if $f_0(D) \downarrow = e$, and $h(D) = f_1^{(n)}(D)$, if $n \geq 1$, and $f_0(f_1^{(n)}(D)) \downarrow = e$ and $f_0(f_1^{(i)}(D)) \downarrow \neq e$ for $i = \overline{0, n-1}$, and $h(D) \uparrow$ otherwise.

Lemma is proved.

Lemma 4.4.10. (About basic functions).

(1) $\gamma \bullet \bar{\emptyset} \bullet \gamma^{-1} \in CTR_D$, where $\bar{\emptyset}$ is the empty constant function.

(2) $\gamma \bullet \Rightarrow u \bullet \gamma^{-1} \in CTR_D$ for each $u \in V^+$;

(3) $\gamma \bullet u \Rightarrow_a \bullet \gamma^{-1} \in CTR_D$ for each $u \in V^+$;

Moreover, CO TRS for $\gamma \bullet \Rightarrow u \bullet \gamma^{-1}$ and $\gamma \bullet u \Rightarrow_a \bullet \gamma^{-1}$ can be effectively constructed from u .

Proof.

1. Since $\gamma \bullet \bar{\emptyset} \bullet \gamma^{-1} = \hat{e}$, we have $\gamma \bullet \bar{\emptyset} \bullet \gamma^{-1} \in CTR_D$ by Lemma 4.4.5.

2. If $u = v_i$ for $i \in \{1, 2\}$, then $\gamma \bullet \Rightarrow u \bullet \gamma^{-1} = \hat{\phi}_i \in CTR_{\mathbf{D}}$, so consider the case when $u \notin V$. Let $u = v_{i_1} v_{i_2} \dots v_{i_k}$, where $k \geq 2$, $i_1, \dots, i_k \in \{1, 2\}$ and $v_{i_1}, \dots, v_{i_k} \in V$. Denote $h = \hat{\phi}_{i_k} \bullet f_{k-1} \bullet \dots \bullet f_1$, where $f_j = (\gamma \bullet \chi^{-1} \bullet \gamma^{-1}) \bullet \hat{\phi}'_{i_j}$, $j = \overline{1, k-1}$. From Lemma 4.4.5 and Lemma 4.4.7 it follows that $h \in CTR_{\mathbf{D}}$, and the TRS for h can be effectively constructed from u . Taking into account the definition of γ and the fact that $\chi^{-1} \bullet \chi$ is a (total) identity function, we have:

$$\begin{aligned}
& \gamma^{-1} \bullet h \bullet \gamma = \gamma^{-1} \bullet \hat{\phi}_{i_k} \bullet f_{k-1} \bullet \dots \bullet (f_1 \bullet \gamma) = \\
& = \gamma^{-1} \bullet \hat{\phi}_{i_k} \bullet f_{k-1} \bullet \dots \bullet f_2 \bullet (\gamma \bullet \chi^{-1} \bullet \gamma^{-1} \bullet \hat{\phi}'_{i_1} \bullet \gamma) = \\
& = \gamma^{-1} \bullet \hat{\phi}_{i_k} \bullet f_{k-1} \bullet \dots \bullet f_2 \bullet \gamma \bullet (\chi^{-1} \bullet \gamma^{-1} \bullet \gamma \bullet \chi \bullet (v_{i_1} \cdot)) = \\
& = \gamma^{-1} \bullet \hat{\phi}_{i_k} \bullet (f_{k-1} \bullet \dots \bullet f_2 \bullet \gamma) \bullet (v_{i_1} \cdot) = \\
& = \gamma^{-1} \bullet \hat{\phi}_{i_k} \bullet \gamma \bullet (v_{i_{k-1}} \cdot) \bullet \dots \bullet (v_{i_2} \cdot) \bullet (v_{i_1} \cdot) = \\
& = (\Rightarrow v_{i_k}) \bullet (v_{i_{k-1}} \cdot) \bullet \dots \bullet (v_{i_2} \cdot) \bullet (v_{i_1} \cdot) \Rightarrow (v_{i_1} \dots v_{i_k}) \Rightarrow u.
\end{aligned}$$

Thus $h = \gamma \bullet \Rightarrow u \bullet \gamma^{-1} \in CTR_{\mathbf{D}}$.

3. Taking into account the associativity property of denaming and Lemma 4.4.7, it is sufficient to show that $\gamma \bullet v_i \Rightarrow_a \bullet \gamma^{-1} \in CTR_{\mathbf{D}}$ for $i = \overline{1, 2}$. Consider the case $i = 1$, since the case $i = 2$ is similar.

By Lemma 4.4.6, let us construct a CO TRS $R = (F \cup C_w, C_w, P)$ of class **R** with distinguished symbol $\bar{f}$ such that $I_{R, \mathbf{D}}(\bar{f}) = \gamma \bullet \chi \bullet \gamma^{-1}$. Let $\bar{g} \notin F$ be a new function symbol, and $P_{\bar{g}}$ be the set of rules: $\bar{g}(\varphi_1(x_1)) \mapsto x_1$, $\bar{g}(\varphi'_1(x_1)) \mapsto \bar{f}(x_1)$, $\bar{g}(\psi_{1,j}(x_1, x_2)) \mapsto x_1$, $\bar{g}(\psi_{2,j}(x_1, x_2)) \mapsto f(x_1)$, $j = \overline{1, 2}$. Then

$R' = (F \cup \{\bar{g}\} \cup C_w, C_w, P \cup P_{\bar{g}})$ is a CO TRS of class **R**. Denote $g = I_{R',D}(\bar{g})$ and $h = \gamma^{-1} \bullet g \bullet \gamma$. Then

- $g(\varphi_1(D)) = D$ for all $D \in \mathbf{D}$, so $h([v_1 \mapsto d]) = d$ for all $d \in NDVC(V, W)$,

- for each $D \in \mathbf{D}$ it holds $g(\varphi'_1(D)) = \gamma(\chi(\gamma^{-1}(D)))$, so $(v_1 \cdot) \bullet h = (v_1 \cdot) \bullet \gamma^{-1} \bullet g \bullet \gamma = \chi^{-1} \bullet \gamma^{-1} \bullet (\hat{\varphi}'_1 \bullet g) \bullet \gamma = \chi^{-1} \bullet \gamma^{-1} \bullet \gamma \bullet \chi \bullet \gamma^{-1} \bullet \gamma = id$ (identity function), whence $h(v_1 \cdot d) = d$ for all $d \in NDVC(V, W) \setminus (W \cup \{\emptyset\})$

Similarly, we have $h([v_1 \mapsto d_1, v_2 \mapsto d_2]) = d_1$, $h([v_1 \mapsto d_1] \cup v_2 \cdot d_2) = d_1$ (when $d_2 \notin W$), $h(v_1 \cdot d_1 \cup [v_2 \mapsto d_2]) = d_1$ (when $d_1 \notin W$), $h(v_1 \cdot d_1 \cup v_2 \cdot d_2) = d_1$ (when $d_1, d_2 \notin W$) and $h(d) \uparrow$, if $v_1 \notin rp_0(d)$. Thus $h = v_1 \Rightarrow_a$, so $\gamma \bullet v_1 \Rightarrow_a \bullet \gamma^{-1} \in CTR_{\mathbf{D}}$.

Lemma is proved.

Lemma 4.4.11. (About assignment). There exists $h \in CTR_{\mathbf{D}}^2$ such that for all elements $D_1, D_2 \in \mathbf{D}$ such that the set $rn(\gamma(D_2))$ is a singleton the following holds: $\gamma(h(D_1, D_2)) \cong \gamma(D_1) \nabla_s \gamma(D_2)$.

Proof.

Using Lemma 4.4.6 let us construct a CO TRS $R = (F \cup C_w, C_w, P)$ of class **R** with distinguished symbols $\bar{f}$ and $\bar{g}$ such that $I_{R,D}(\bar{f}) = \gamma \bullet \chi \bullet \gamma^{-1}$ and $I_{R,D}(\bar{g}) = \gamma \bullet \chi^{-1} \bullet \gamma^{-1}$. Let $\bar{h} \notin F$ be a new function symbol, and $P_{\bar{h}}$ be the set of rules given below, where i ranges over $\{1, 2\}$:

$$\begin{aligned} \bar{h}(e, \varphi_i(x_2)) &\mapsto \varphi_i(x_1), \quad \bar{h}(\varphi_i(x_1), \varphi_i(x_2)) \mapsto \varphi_i(x_2), \\ \bar{h}(\varphi_{3-i}(x_1), \varphi_i(x_2)) &\mapsto \psi_{11}(x_{3-i}, x_i), \\ \bar{h}(\varphi'_i(x_1), \varphi_i(x_2)) &\mapsto \varphi_i(x_2), \quad \bar{h}(\varphi'_2(x_1), \varphi_1(x_2)) \mapsto \psi_{1,2}(x_2, \bar{f}(x_1)), \\ \bar{h}(\varphi'_1(x_1), \varphi_2(x_2)) &\mapsto \psi_{2,1}(\bar{f}(x_1), x_2), \\ \bar{h}(\psi_{1,i}(x_1, x_2), \varphi_1(x_3)) &\mapsto \psi_{1,i}(x_3, x_2), \\ \bar{h}(\psi_{2,i}(x_1, x_2), \varphi_1(x_3)) &\mapsto \psi_{1,i}(\bar{f}(x_3), x_2), \end{aligned}$$

$$\begin{aligned}
& \bar{h}(\psi_{i,1}(x_1, x_2), \varphi_2(x_3)) \mapsto \psi_{i,1}(x_1, x_3), \\
& \bar{h}(\psi_{i,2}(x_1, x_2), \varphi_2(x_3)) \mapsto \psi_{i,1}(x_1, \bar{f}(x_3)), \\
& \bar{h}(e, \varphi'_i(x_1)) \mapsto \varphi'_i(x_1), \quad \bar{h}(\varphi_i(x_1), \varphi'_i(x_2)) \mapsto \varphi_i(h(x_1, \bar{f}(x_2))), \\
& \bar{h}(\varphi_1(x_1), \varphi'_2(x_2)) \mapsto \psi_{12}(x_1, \bar{f}(x_2)), \\
& \bar{h}(\varphi_2(x_1), \varphi'_1(x_2)) \mapsto \psi_{21}(\bar{f}(x_2), x_1), \\
& \bar{h}(\varphi'_i(x_1), \varphi'_i(x_2)) \mapsto \varphi'_i(\bar{g}(\bar{h}(\bar{f}(x_1), \bar{f}(x_2)))), \\
& \bar{h}(\varphi'_{3-i}(x_1), \varphi'_i(x_2)) \mapsto \psi_{22}(x_{3-i}, x_i), \\
& \bar{h}(\psi_{1,i}(x_1, x_2), \varphi'_1(x_3)) \mapsto \psi_{1,i}(h(x_1, \bar{f}(x_3)), x_2), \\
& \bar{h}(\psi_{2,i}(x_1, x_2), \varphi'_1(x_3)) \mapsto \psi_{2,i}(g(\bar{h}(\bar{f}(x_1), \bar{f}(x_3))), x_2), \\
& \bar{h}(\psi_{i,1}(x_1, x_2), \varphi'_2(x_3)) \mapsto \psi_{i,1}(x_1, h(x_2, \bar{f}(x_3))), \\
& \bar{h}(\psi_{i,2}(x_1, x_2), \varphi'_2(x_3)) \mapsto \psi_{i,2}(x_1, g(\bar{h}(\bar{f}(x_2), \bar{f}(x_3)))).
\end{aligned}$$

Then $R' = (F \cup \{\bar{h}\} \cup C_w, C_w, P \cup P_{\bar{h}})$ is a CO TRS of class **R**. Let $h = I_{R,D}(\bar{h})$. Using the definition of γ it is easy to check that $\gamma(h(D_1, D_2)) \equiv \gamma(D_1) \nabla_s \gamma(D_2)$, whenever $rn(\gamma(D_2))$ is a singleton set.

Lemma is proved.

Theorem 4.4.1. For each function $h : NDVC(V, W) \rightarrow NDVC(V, W)$ which is expressible in $SICON_{as}$, there exists a function $f \in CTR_D$ such that $h = \gamma \circ f \circ \gamma^{-1}$. Moreover, a CO TRS R with a distinguished symbol $\bar{f}$ such that $f = I_{R,D}(\bar{f})$ can be effectively constructed from a term of h in $SICON_{as}$ language.

Proof. The statement can be proven by induction on the structure of the term for h using Lemmas 4.4.7, 4.4.9, 4.4.10, 4.4.11.

This theorem gives a method of modeling programs over complex-named data using term rewriting systems.

4.5. Compositions of functions and predicates over nominative data

Let V and A be fixed sets of basic names and atoms.

Let

$$\begin{aligned} Pr_{CC}(V, A) &= NDVC(V, A) \xrightarrow{\sim} \{T, F\}, \\ Fn_{CC}(V, A) &= NDVC(V, A) \xrightarrow{\sim} NDVC(V, A), \end{aligned}$$

where T and F denote truth values (*true* and *false*). We will assume that they do not belong to $NDVC(V, A)$.

We will call the elements of $Pr_{CC}(V, A)$ (*partial nominative*) *predicates*, and the elements of $Fn_{CC}(V, A)$ (*partial binominative*) *functions*.

Similarly, let us introduce the following sets of functions and predicates on data of types TND_{AC} and TND_{CA} :

$$\begin{aligned} Pr_{AC}(V, A) &= ND(V, A) \xrightarrow{\sim} \{T, F\}, \\ Fn_{AC}(V, A) &= ND(V, A) \xrightarrow{\sim} ND(V, A), \\ Pr_{CA}(V, A) &= NDVS(V, A) \xrightarrow{\sim} \{T, F\}, \\ Fn_{CA}(V, A) &= NDVS(V, A) \xrightarrow{\sim} NDVS(V, A). \end{aligned}$$

Let us consider logical and program compositions of predicates and functions. These compositions are defined in a similar way for functions and predicates for each type of nominative data, so below we give general definitions in which the symbol Fn has to be interpreted as Fn_{CC} , Fn_{CA} , or Fn_{AC} , and the symbol Pr has to be understood as Pr_{CC} , Pr_{CA} , Pr_{AC} (the meanings of all instances of the symbol Fn is the same in one definition and the meanings of all instances of the symbol Pr are the same in one definition).

Let us denote as $\bar{U}$ the set of all tuples $(u_1, u_2, \dots, u_n)$, $n \geq 1$ of complex names from V^+ such that if $i \neq j$, then u_i and u_j are incomparable in the sense of the prefix order.

1. Sequential composition of functions

$$\bullet : Fn(V, A) \times Fn(V, A) \rightarrow Fn(V, A)$$

is defined as follows: for any $f, g \in Fn(V, A)$ and data d

$$(f \bullet g)(d) \cong g(f(d)).$$

2. Predication composition

$$\cdot : Fn(V, A) \times Pr(V, A) \rightarrow Pr(V, A)$$

is defined as follows: for any $f \in Fn(V, A)$, $p \in Pr(V, A)$, and data d

$$(f \cdot p)(d) \cong p(f(d)).$$

3. Assignment composition $Asg^u : Fn(V, A) \rightarrow Fn(V, A)$,

where $u \in V^+$ is a parameter, is defined as follows: for each $f \in Fn(V, A)$ and data d ,

$$(Asg^u(f))(d) \cong d \nabla_a^u f(d),$$

where ∇_a^u denotes the local overlapping of nominative data of the respective type.

5. Composition of superposition into a function

$$S_F^{u_1, u_2, \dots, u_n} : Fn(V, A) \times (Fn(V, A))^n \rightarrow Fn(V, A)$$

with parameters $n \geq 1$ and $u_1, \dots, u_n \in V^+$ such that $(u_1, \dots, u_n) \in \bar{U}$:

$$\begin{aligned} & S_F^{u_1, u_2, \dots, u_n}(f, f_1, \dots, f_n)(d) \cong \\ & \cong f(\dots((d \nabla_a^{u_1} f_1(d)) \nabla_a^{u_2} f_2(d)) \dots \nabla_a^{u_n} f_n(d)) \dots). \end{aligned}$$

We will also use the following notation for this composition:
for a tuple $\bar{u} = (u_1, u_2, \dots, u_n) \in \bar{U}$, $S_F^{\bar{u}}$ denotes $S_F^{u_1, u_2, \dots, u_n}$.

6. Composition of superposition into a predicate

$$S_p^{u_1, u_2, \dots, u_n} : Pr(V, A) \times (Fn(V, A))^n \rightarrow Pr(V, A)$$

with parameters $n \geq 1$ and $u_1, \dots, u_n \in V^+$ such that $(u_1, \dots, u_n) \in \bar{U}$:

$$\begin{aligned} & S_p^{u_1, u_2, \dots, u_n}(p, f_1, \dots, f_n)(d) \cong \\ & \cong p(\dots((d \nabla_a^{u_1} f_1(d)) \nabla_a^{u_2} f_2(d)) \dots \nabla_a^{u_n} f_n(d)) \dots). \end{aligned}$$

We will also use the following notation for this composition:
for a tuple $\bar{u} = (u_1, u_2, \dots, u_n) \in \bar{U}$, $S_p^{\bar{u}}$ denotes $S_p^{u_1, u_2, \dots, u_n}$.

7. Branching composition

$IF : Pr(V, A) \times Fn(V, A) \times Fn(V, A) \rightarrow Fn(V, A)$ is defined as follows: for any $p \in Pr(V, A)$ and $f, g \in Fn(V, A)$:

$$IF(p, f, g)(d) \cong \begin{cases} f(d), & p(d) \downarrow = T; \\ g(d), & p(d) \downarrow = F; \\ \text{undefined}, & p(d) \uparrow. \end{cases}$$

8. Loop composition

$WH : Pr(V, A) \times Fn(V, A) \rightarrow Fn(V, A)$ is defined as follows: for any $p \in Pr(V, A)$, $f \in Fn(V, A)$, and data d :

- $WH(p, f)(d) \downarrow = f^{(n)}(d)$, if there exists $n \geq 0$ such that $(f^{(i)} \cdot p)(d) \downarrow = T$ for all $i \in \{0, 1, \dots, n-1\}$ and $(f^{(n)} \cdot p)(d) \downarrow = F$, where $f^{(n)}$ denotes n -times sequential composition of f (we assume that $f^{(0)}$ is the identity function),

- $WH(p, f)(d)$ is undefined, otherwise.

9. Negation composition on predicates

$\neg: Pr(V, A) \rightarrow Pr(V, A)$ be a composition such that for each $p \in Pr(V, A)$ and data d :

$$(\neg p)(d) \cong \begin{cases} T, & p(d) \downarrow = F; \\ F, & p(d) \downarrow = T; \\ \text{undefined}, & p(d) \uparrow. \end{cases}$$

10. Disjunction composition on predicates

$\vee: Pr(V, A) \times Pr(V, A) \rightarrow Pr(V, A)$ is a composition such that for any $p_1, p_2 \in Pr(V, A)$ and data d :

$$(p_1 \vee p_2)(d) = \begin{cases} T, & p_1(d) \downarrow = T \text{ or } p_2(d) \downarrow = T; \\ F, & p_1(d) \downarrow = F \text{ i } p_2(d) \downarrow = F; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

11. Identity composition $Id: Fn(V, A) \rightarrow Fn(V, A)$ is

defined as follows:

$$Id(f) = f \text{ for all } f \in Fn(V, A).$$

12. True predicate (0-ary composition) $True \in Pr(V, A)$ is

defined as follows:

$$True(d) \downarrow = T \text{ for all data } d.$$

13. Nowhere defined function (0-ary composition)
 $\perp_F \in Fn(V, A)$ is defined as follows:

$$\perp_F(d) \uparrow \text{ for all data } d.$$

14. Nowhere defined predicate (0-ary composition)
 $\perp_P \in Pr(V, A)$ is defined as follows:

$$\perp_P(d) \uparrow \text{ for all data } d.$$

15. Name checking predicate (0-ary composition)
 $u! \in Pr(V, A)$ with parameter $u \in V^+$ is defined as follows:

$$u!(d) = \begin{cases} T, & p \Rightarrow_a(d) \downarrow; \\ F, & p \Rightarrow_a(d) \uparrow. \end{cases}$$

16. Empty data function (0-ary composition)
 $Empty \in Fn(V, A)$ is defined as follows:

$$Empty(d) = \emptyset \text{ for all data } d.$$

17. Emptiness checking predicate (0-ary composition)
 $IsEmpty \in Fn(V, A)$ is defined as follows:

$$IsEmpty(d) = \begin{cases} T, & d = \emptyset, \\ F, & d \neq \emptyset. \end{cases}$$

4.6. Generalized systems of Glushkov algorithmic algebras

Compositions defined above allow one to introduce a sufficiently expressive program language. This language generalizes Glushkov algorithmic algebra systems, and also some program alge-

bras related to Floyd-Hoare logic, algorithmic logics and dynamic logics.

Definition 4.7. *An associative nominative algorithmic Glushkov algebra of predicates and functions over nominative data of type TND_{CC} is an algebraic structure*

$$NGA_{CC}^a(V, A) = \langle Pr_{CC}(V, A), Fn_{CC}(V, A); \vee, \neg, \bullet, IF, WH, :, (Asg^u)_{u \in V^+}, (S_F^{\bar{u}})_{\bar{u} \in \bar{U}}, (S_P^{\bar{u}})_{\bar{u} \in \bar{U}}, Id, True, \perp_F, \perp_P, (u!)_{u \in V^+}, Empty, IsEmpty \rangle.$$

with the carriers $Pr_{CC}(V, A)$ (predicates) and $Fn_{CC}(V, A)$ (functions) for some V and A , with the following operations: disjunction composition $\vee$ and negation composition $\neg$ on predicates, sequence composition of functions $\bullet$, branching composition IF , loop composition WH , predication composition $\cdot$, a family of assignment compositions Asg^u , $u \in V^+$, the family of superposition compositions $S_P^{\bar{u}}$, $S_F^{\bar{u}}$ for $\bar{u} \in \bar{U}$, the identity composition on functions Id , and the following constant elements of the carrier sets (0-ary compositions): true predicate $True$, nowhere defined predicate $\perp_P$, nowhere defined function $\perp_F$, a family of name checking predicates $u!$, $u \in V^+$, the empty data function $Empty$, and the emptiness checking predicate $IsEmpty$.

Similarly, algebras of functions and predicates over nominative data of types TND_{CA} and TND_{AC} can be defined:

Definition 4.8.

1. *An associative nominative algorithmic Glushkov algebra of predicates and functions over nominative data of type TND_{AC} is an algebraic structure*

$$NGA_{AC}^a(V, A) = \langle Pr_{AC}(V, A), Fn_{AC}(V, A); \vee, \neg, \bullet, IF, WH, :, (Asg^u)_{u \in V^+}, (S_F^{\bar{u}})_{\bar{u} \in \bar{U}}, (S_P^{\bar{u}})_{\bar{u} \in \bar{U}}, Id, True, \perp_F, \perp_P, (u!)_{u \in V^+}, Empty, IsEmpty \rangle.$$

2. An associative nominative algorithmic Glushkov algebra of predicates and functions over nominative data of type TND_{CA} is an algebraic structure

$$NGA_{CA}^a(V, A) = \langle Pr_{CA}(V, A), Fn_{CA}(V, A); \vee, \neg, \bullet, IF, WH, :, (Asg^u)_{u \in V^+}, (S_F^{\bar{u}})_{\bar{u} \in \bar{U}}, (S_P^{\bar{u}})_{\bar{u} \in \bar{U}}, Id, True, \perp_F, \perp_P, (u!)_{u \in V^+}, Empty, IsEmpty \rangle.$$

4.7. Stability and monotonicity of programs over nominative data

The binary relation of *nominative stability* formalizes the idea that the behavior of a program does not change, when the hierarchical structure of naming of data changes. As an example consider the following feature of Pascal programming language: the definitions **var** A: **array** [1..n, 1..m] **of real** and **var** A: **array** [1..n] **of array** [1..m] **of real** are equivalent and both syntaxes $A[i, j]$ and $A[i][j]$ can be used to access array elements regardless of the form of the definition. It should be noted that many languages such as C++ and Java do not have this feature.

Let us formalize the fact of independence of the behavior of a program of the hierarchical structure of data using the following notions of nominative stability of functions and predicates.

Definition 4.9. A predicate $p \in Pr_{CC}(V, A)$ is *nominative stable*, if for any $d_1, d_2 \in NDVC(V, A)$, if $p(d_1) \downarrow$ and $d_1 \approx d_2$, then $p(d_2) \downarrow$ and $p(d_1) = p(d_2)$.

Definition 4.10. A function $f \in Fn_{CC}(V, A)$ is *nominative stable*, if for any $d_1, d_2 \in NDVC(V, A)$, if $f(d_1) \downarrow$ and $d_1 \approx d_2$, then $f(d_2) \downarrow$ and $f(d_1) \approx f(d_2)$.

Let us denote by $PrNS(V, A)$ the set of all nominative stable predicates $p \in Pr_{CC}(V, A)$ and let us denote as $FnNS(V, A)$ the set of all nominative stable functions $f \in Fn_{CC}(V, A)$.

Let us introduce the following relation on $FnNS(V, A)$:

$$f \equiv_F g,$$

(where $f, g \in FnNS(V, A)$), if from $d_1 \approx d_2$ and $f(d_1) \downarrow$ or $g(d_2) \downarrow$ it follows that $f(d_1) \downarrow$, $g(d_2) \downarrow$ and $f(d_1) \approx g(d_2)$.

Let us introduce the following relation on $PrNS(V, A)$:

$$p_1 \equiv_p p_2$$

(where $p_1, p_2 \in PrNS(V, A)$), if from $d_1 \approx d_2$ and $p_1(d_1) \downarrow$ or $p_2(d_2) \downarrow$ it follows that $p_1(d_1) \downarrow$, $p_2(d_2) \downarrow$ and $p_1(d_1) = p_2(d_2)$.

Theorem 4.3. $\equiv_p$ is an equivalence relation on $PrNS(V, A)$.

Proof follows from the fact that $\approx$ is an equivalence relation on $NDVC(V, A)$.

Theorem 4.4. $\equiv_F$ is an equivalence relation on $FnNS(V, A)$.

Proof follows from the fact that $\approx$ is an equivalence relation on $NDVC(V, A)$.

Let D be a set. Let us introduce the following notation.

- $Pr(D)$ is the set of all partial predicates $p : D \rightarrow \{T, F\}$.

- $Fun(D)$ is the set of all partial functions $f : D \rightarrow D$.

- $Pr^n(D) = Pr(D^n)$ for each $n \in \mathbb{N}$.

- $Fun^n(D)$ is the set of all partial functions $f : D^n \rightarrow D$ for $n \in \mathbb{N}$.

For each preorder $\leq$ on D let us denote:

- $PrM(D, \leq)$ is the set of all $p \in Pr(D)$ such that for each $d_1, d_2 \in D$, if $p(d_1) \downarrow$ and $d_1 \leq d_2$, then $p(d_2) \downarrow$ and $p(d_1) = p(d_2)$.

- $FunM(D, \leq)$ is the set of all $f \in Fun(D)$ such that for each $d_1 \in D$, if $f(d_1) \downarrow$ and $d_1 \leq d_2$, then $f(d_2) \downarrow$ and $f(d_1) \leq f(d_2)$.

- $PrM^n(D, \leq) = PrM(D^n, \leq^n)$ for each $n \in \mathbb{N}$, where $\leq^n$ is a preorder on D^n induced by $\leq$, i.e. $(d_1, d_2, \dots, d_n) \leq^n (d'_1, d'_2, \dots, d'_n)$, if $d_1 \leq d'_1, d_2 \leq d'_2, \dots, d_n \leq d'_n$.

- $FunM^n(D, \leq)$ is the set of all $f \in Fun^n(D)$ such that for any $d_1, d_2, \dots, d_n, d'_1, d'_2, \dots, d'_n \in D$, if $f(d_1, d_2, \dots, d_n) \downarrow$ and $d_1 \leq d'_1, d_2 \leq d'_2, \dots, d_n \leq d'_n$, then $f(d'_1, d'_2, \dots, d'_n) \downarrow$ and $f(d_1, d_2, \dots, d_n) \leq f(d'_1, d'_2, \dots, d'_n)$.

Lemma 4.3.

1. if $\leq_1$ and $\leq_2$ are preorders on D and $\leq_1 \subseteq \leq_2$, then

$$PrM(D, \leq_2) \subseteq PrM(D, \leq_1).$$

2. if $\leq_1$ and $\leq_2$ are preorders on D , then $\leq_1 \cap \leq_2$ are preorders on D and

$$FunM(D, \leq_1) \cap FunM(D, \leq_2) \subseteq FunM(D, \leq_1 \cap \leq_2).$$

3. if $\leq$ is a preorder on D , then the inverse relation $\leq^{-1}$ is a preorder on D and

$$\begin{aligned} PrM(D, \leq) &\subseteq PrM(D, \leq \cap \leq^{-1}), \\ FunM(D, \leq) &\subseteq FunM(D, \leq \cap \leq^{-1}). \end{aligned}$$

Proof.

1. Let $\leq_1, \leq_2$ be preorders on D . Let $p \in \text{PrM}(D, \leq_2)$. If $d_1, d_2 \in D$, $d_1 \sqsubseteq_2 d_2$, and $p(d_1) \downarrow$, then $d_1 \leq_2 d_2$, so $p(d_2) \downarrow$ and $p(d_2) = p(d_1)$. Thus, $p \in \text{PrM}(D, \leq_1)$. So

$$\text{PrM}(D, \leq_2) \subseteq \text{PrM}(D, \leq_1).$$

2. Let $\leq_1, \leq_2$ be preorders on D . Obviously, $\leq_1 \cap \leq_2$ is a preorder on D . Let $f \in \text{FunM}(D, \leq_1) \cap \text{FunM}(D, \leq_2)$. Assume that $d_1, d_2 \in D$, $d_1 \leq_1 d_2$, $d_1 \leq_2 d_2$, and $f(d_1) \downarrow$. Then $f(d_2) \downarrow$, $f(d_1) \leq_1 f(d_2)$, and $f(d_1) \leq_2 f(d_2)$. So $f \in \text{FunM}(D, \leq_1 \cap \leq_2)$. Then

$$\text{FunM}(D, \leq_1) \cap \text{FunM}(D, \leq_2) \subseteq \text{FunM}(D, \leq_1 \cap \leq_2).$$

3. Let $\leq$ be a preorder on D . Obviously, $\leq^{-1}$ is a preorder on D and $\leq \cap \leq^{-1}$ is an equivalence relation on D . Then

$$\text{PrM}(D, \leq) \subseteq \text{PrM}(D, \leq \cap \leq^{-1})$$

follows from (1).

Let $f \in \text{FunM}(D, \leq)$. If $d_1 \leq d_2$, $d_1 \leq^{-1} d_2$, and $f(d_1) \downarrow$, then $d_2 \leq d_1$, $f(d_2) \downarrow$, and $f(d_1) \leq f(d_2)$ and $f(d_2) \leq f(d_1)$, whence $f(d_1) \leq f(d_2)$ and $f(d_1) \leq^{-1} f(d_2)$. Thus $f \in \text{FunM}(D, \leq \cap \leq^{-1})$.

So the inclusion

$$\text{FunM}(D, \leq) \subseteq \text{FunM}(D, \leq \cap \leq^{-1})$$

holds.

Lemma is proved.

Lemma 4.4 (Main properties of classes of n-ary monotone functions and predicates)

Let $n \in \mathbb{N}$.

1. if $\leq_1$ and $\leq_2$ are preorders on D and $\leq_1 \subseteq \leq_2$, then

$$PrM^n(D, \leq_2) \subseteq PrM^n(D, \leq_1).$$

2. if $\leq_1$ and $\leq_2$ are preorders on D , then $\leq_1 \cap \leq_2$ is a preorder on D and

$$FunM^n(D, \leq_1) \cap FunM^n(D, \leq_2) \subseteq FunM^n(D, \leq_1 \cap \leq_2).$$

3. if $\leq$ is a preorder on D , then the inverse relation $\leq^{-1}$ is a preorder on D and

$$\begin{aligned} PrM^n(D, \leq) &\subseteq PrM^n(D, \leq \cap \leq^{-1}), \\ FunM^n(D, \leq) &\subseteq FunM^n(D, \leq \cap \leq^{-1}). \end{aligned}$$

Proof is similar to the proof of Lemma 4.3.

For each preorder $\leq$ on $NDVC(V, A)$ and $n \in \mathbb{N}$ let us denote:

1. $PrM_{cc}(V, A, \leq) = PrM(NDVC(V, A), \leq)$;
2. $PrM_{cc}^n(V, A, \leq) = PrM^n(NDVC(V, A), \leq)$;
3. $FunM_{cc}(V, A, \leq) = FunM(NDVC(V, A), \leq)$;
4. $FunM_{cc}^n(V, A, \leq) = FunM^n(NDVC(V, A), \leq)$.

Lemma 4.5 (Preservation of monotonicity by the compositions). Let $\leq$ be a preorder on $NDVC(V, A)$, $u \in V^+$, $n \geq 1$, $u_1, u_2, \dots, u_n$ be pairwise incomparable names from V^+ (in the sense

of the relation $\leq$), $f, g, f_1, f_2, \dots, f_n \in \text{FunM}_{CC}(V, A, \leq)$,
 $p, q \in \text{PrM}_{CC}(V, A, \leq)$.

1. $p \vee q \in \text{PrM}_{CC}(V, A, \leq)$.
2. $\neg p \in \text{PrM}_{CC}(V, A, \leq)$.
3. $f \bullet g \in \text{FunM}_{CC}(V, A, \leq)$.
4. $IF(p, f, g) \in \text{FunM}_{CC}(V, A, \leq)$.
5. $WH(p, f) \in \text{FunM}_{CC}(V, A, \leq)$.
6. $f \cdot p \in \text{PrM}_{CC}(V, A, \leq)$.
7. If $\nabla_a^u \in \text{FunM}_{CC}^2(V, A, \leq)$,
then $\text{Asg}^u(f) \in \text{FunM}_{CC}(V, A, \leq)$.
8. If $\nabla_a^{u_j} \in \text{FunM}_{CC}^2(V, A, \leq)$ for all $j \in \{1, 2, \dots, n\}$, then
 $S_F^{u_1, u_2, \dots, u_n}(f, f_1, f_2, \dots, f_n) \in \text{FunM}_{CC}(V, A, \leq)$.
9. If $\nabla_a^{u_j} \in \text{FunM}_{CC}^2(V, A, \leq)$ for all $j \in \{1, 2, \dots, n\}$, then
 $S_P^{u_1, u_2, \dots, u_n}(p, f_1, f_2, \dots, f_n) \in \text{FunM}_{CC}(V, A, \leq)$.
10. $\text{Id}(f) \in \text{FunM}_{CC}(V, A, \leq)$.
11. $\text{True} \in \text{PrM}_{CC}(V, A, \leq)$.
12. $\perp_F \in \text{FunM}_{CC}(V, A, \leq)$.
13. $\perp_P \in \text{PrM}_{CC}(V, A, \leq)$.
14. $\text{Empty} \in \text{FunM}_{CC}(V, A, \leq)$.

Proof. The items 1–3, 6, and 10–14 can be proven directly by the definitions. Let us prove the items 4–5 and 7–9.

5. Assume that $d_1, d_2 \in \text{NDVC}(V, A)$, $d_1 \leq d_2$, and $IF(p, f, g)(d_1) \downarrow$. Then $p(d_1) \downarrow$ and $p(d_2) \downarrow = p(d_1)$. If $p(d_1) = T$, then $f(d_1) \downarrow = IF(p, f, g)(d_1)$, $f(d_1) \leq f(d_2)$, and $IF(p, f, g)(d_2) \downarrow = f(d_2)$,
whence

$IF(p, f, g)(d_1) \leq IF(p, f, g)(d_2)$. Similarly, if $p(d_1) = F$, we have $IF(p, f, g)(d_1) \leq IF(p, f, g)(d_2)$. So

$$IF(p, f, g) \in FunM_{CC}(V, A, \leq).$$

5. Assume that $d_1, d_2 \in NDVC(V, A)$, $d_1 \leq d_2$, and $WH(p, f)(d_1) \downarrow$. Then $p(d_1) \downarrow$ and $p(d_2) \downarrow = p(d_1)$. if $p(d_1) = F$, then $WH(p, f)(d_1) = d_1 \leq d_2 = WH(p, f)(d_2)$. if $p(d_2) = T$, then for some $n \geq 1$,

$$WH(p, f)(d_1) = (\underbrace{f \bullet \dots \bullet f}_n)(d_1).$$

Then $(\underbrace{f \bullet \dots \bullet f}_n)(d_2) \downarrow$ and

$$(\underbrace{f \bullet \dots \bullet f}_n)(d_1) \sqsubseteq (\underbrace{f \bullet \dots \bullet f}_n)(d_2),$$

whence $WH(p, f)(d_2) \downarrow$ and $WH(p, f)(d_1) \leq WH(p, f)(d_2)$. So

$$WH(p, f) \in FunM_{CC}(V, A, \leq).$$

7. Assume that $\nabla_a^u \in FunM_{CC}^2(V, A, \leq)$. Let $d_1, d_2 \in NDVC(V, A)$ and $d_1 \leq d_2$. Assume that $Asg^u(f)(d_1) \downarrow$. Then $Asg^u(f)(d_1) = d_1 \nabla_a^u f(d_1)$. Moreover, $f(d_2) \downarrow$ and $f(d_1) \leq f(d_2)$. Then $d_2 \nabla_a^u f(d_2) \downarrow$ and $Asg^u(f)(d_1) = d_1 \nabla_a^u f(d_1) \leq d_2 \nabla_a^u f(d_2) = Asg^u(f)(d_2)$. Then $Asg^u(f) \in FunM_{CC}(V, A, \sqsubseteq)$.

8. Assume that $\nabla_a^{u_j} \in FunM_{CC}^2(V, A, \leq)$ for all $j \in \{1, 2, \dots, n\}$. Let $d_1, d_2 \in NDVC(V, A)$ and $d_1 \leq d_2$. Assume that

$$S_F^{u_1, u_2, \dots, u_n}(f, f_1, f_2, \dots, f_n)(d_1) \downarrow = d.$$

Then

$$d = f(\dots((d_1 \nabla_s^{u_1} f_1(d_1)) \nabla_s^{u_2} f_2(d_1)) \dots \nabla_s^{u_n} f_n(d_1)) \dots).$$

Moreover, for all $j \in \{1, 2, \dots, n\}$, $f_j(d_2) \downarrow$ and $f_j(d_1) \leq f_j(d_2)$, since $f_j \in FunM_{CC}(V, A, \leq)$. Then

$$(\dots((d_1 \nabla_s^{u_1} f_1(d_2)) \nabla_s^{u_2} f_2(d_2)) \dots \nabla_s^{u_n} f_n(d_2)) \dots) \downarrow$$

and

$$\begin{aligned} & (\dots((d_1 \nabla_s^{u_1} f_1(d_1)) \nabla_s^{u_2} f_2(d_1)) \dots \nabla_s^{u_n} f_n(d_1)) \dots) \leq \\ & \boxed{\square} (\dots((d_2 \nabla_s^{u_1} f_1(d_2)) \nabla_s^{u_2} f_2(d_2)) \dots \nabla_s^{u_n} f_n(d_2)) \dots). \end{aligned}$$

Then $S_F^{u_1, u_2, \dots, u_n}(f, f_1, f_2, \dots, f_n)(d_2) \downarrow$ and

$$S_F^{u_1, u_2, \dots, u_n}(f, f_1, f_2, \dots, f_n)(d_1) \leq S_F^{u_1, u_2, \dots, u_n}(f, f_1, f_2, \dots, f_n)(d_2),$$

since $f \in FunM_{CC}(V, A, \leq)$. Then

$$S_F^{u_1, u_2, \dots, u_n}(f, f_1, f_2, \dots, f_n) \in FunM_{CC}(V, A, \leq).$$

9. Assume that $\nabla_a^{u_j} \in FunM_{CC}^2(V, A, \leq)$ for all $j \in \{1, 2, \dots, n\}$.

Let $d_1, d_2 \in NDVC(V, A)$ and $d_1 \leq d_2$. Assume that

$$S_p^{u_1, u_2, \dots, u_n}(p, f_1, f_2, \dots, f_n)(d_1) \downarrow = b.$$

Then

$$b = p(\dots((d_1 \nabla_s^{u_1} f_1(d_1)) \nabla_s^{u_2} f_2(d_1)) \dots \nabla_s^{u_n} f_n(d_1)) \dots).$$

Moreover, for all $j \in \{1, 2, \dots, n\}$, $f_j(d_2) \downarrow$ and $f_j(d_1) \leq f_j(d_2)$, then $f_j \in FunM_{CC}(V, A, \leq)$.

Then

$$(\dots((d_1 \nabla_s^{u_1} f_1(d_2)) \nabla_s^{u_2} f_2(d_2)) \dots \nabla_s^{u_n} f_n(d_2)) \dots) \downarrow$$

and

$$\begin{aligned} & (\dots((d_1 \nabla_s^{u_1} f_1(d_1)) \nabla_s^{u_2} f_2(d_1)) \dots \nabla_s^{u_n} f_n(d_1)) \dots) \leq \\ & \boxed{\dots((d_2 \nabla_s^{u_1} f_1(d_2)) \nabla_s^{u_2} f_2(d_2)) \dots \nabla_s^{u_n} f_n(d_2)) \dots). \end{aligned}$$

Then $S_p^{u_1, u_2, \dots, u_n}(f, f_1, f_2, \dots, f_n)(d_2) \downarrow$ and

$$S_p^{u_1, u_2, \dots, u_n}(f, f_1, f_2, \dots, f_n)(d_1) = S_p^{u_1, u_2, \dots, u_n}(f, f_1, f_2, \dots, f_n)(d_2),$$

since $p \in PrM_{CC}(V, A, \sqsubseteq)$, so

$$S_p^{u_1, u_2, \dots, u_n}(f, f_1, f_2, \dots, f_n) \in PrM_{CC}(V, A, \sqsubseteq)$$

Lemma is proved.

Theorem 4.5. Let $\leq$ be a preorder on $NDVC(V, A)$.

Assume that the following conditions hold:

- $\Rightarrow u \in FunM_{CC}(V, A, \leq)$ for all $u \in V^+$;
- $u \Rightarrow_a \in FunM_{CC}(V, A, \leq)$ for all $u \in V^+$;
- $\nabla_a^u \in FunM_{CC}^2(V, A, \leq)$ for all $u \in V^+$;
- $u! \in PrM_{CC}(V, A, \leq)$ for all $u \in V^+$;
- $IsEmpty \in PrM_{CC}(V, A, \leq)$.

Then $PrM_{CC}(V, A, \leq)$ and $FunM_{CC}(V, A, \leq)$ form a sub-algebra of $NGA_{CC}^a(V, A)$.

Proof follows from Lemma 4.5 and the definition of $NGA_{CC}^a(V, A)$.

Lemma 4.6.

1. $\Rightarrow u \in FunM_{CC}(V, A, \ll_w)$ for all $u \in V^+$.
2. $u \Rightarrow_a \in FunM_{CC}(V, A, \ll_w)$ for all $u \in V^+$.
3. If $d_1, d_1', d_2, d_2' \in NDVC(V, A)$, $d_1 \ll_w d_1'$, $d_2 \ll_w d_2'$, $d_1 \nabla_a d_2 \downarrow$, and $dom(d_2) = dom(d_2')$, then $d_1' \nabla_a d_2' \downarrow$ and $d_1 \nabla_a d_2 \ll_w d_1' \nabla_a d_2'$.
4. $\nabla_a^u \in FunM_{CC}^2(V, A, \ll_w)$ for all $u \in V^+$.

Proof.

1. Follows from [126], Lemma 7(1).
2. Follows from [126], Lemma 7(2).
3. Follows from [126], Lemma 10 and Lemma 13, and transitivity of $\ll_w$.

4. Let $u \in V^+$. Assume that $d_1, d_2, d_1', d_2' \in NDVC(V, A)$, $d_1 \ll_w d_1'$, $d_2 \ll_w d_2'$, and $d_1 \nabla_a^u d_2 \downarrow$. Then $d_1 \notin A$, so $d_1' \notin A$ and $d_1' \nabla_a^u d_2' \downarrow$. From (1) it follows that $[u \mapsto d_2] \ll_w [u \mapsto d_2']$. Then from (3) it follows that $d_1 \nabla_a^u d_2 = d_1 \nabla_a [u \mapsto d_2] \ll_w d_1' \nabla_a [u \mapsto d_2'] = d_1' \nabla_a^u d_2'$. since

d_1, d_2, d_1', d_2' are arbitrary, we conclude that $\nabla_a^u \in FunM_{CC}^2(V, A, \ll_w)$.

Lemma is proved.

Lemma 4.7.

1. $u \Rightarrow_a \in FunM_{CC}(V, A, \ll)$ for all $u \in V^+$;
2. If $d_1, d_1', d_2, d_2' \in NDVC(V, A)$, $d_1 \ll d_1'$, $d_2 \ll d_2'$, $d_1 \nabla_a d_2 \downarrow$, and $dom(d_2) = dom(d_2')$, then $d_1' \nabla_a d_2' \downarrow$ and $d_1 \nabla_a d_2 \ll d_1' \nabla_a d_2'$.

Proof.

1. Follows from [121], Lemma 7.2(2).
2. Follows from [121], Lemma 7.3 and Lemma 7.4, and transitivity of $\ll$.

Lemma is proved.

Lemma 4.8.

1. $\Rightarrow u \in FunM_{CC}(V, A, \approx)$ for all $u \in V^+$.
2. $u \Rightarrow_a \in FunM_{CC}(V, A, \approx)$ for all $u \in V^+$.
3. $\nabla_a^u \in FunM_{CC}^2(V, A, \approx)$ for all $u \in V^+$.
4. $u! \in PrM_{CC}(V, A, \approx)$ for all $u \in V^+$.
5. $IsEmpty \in PrM_{CC}(V, A, \approx)$.

Proof.

1. Follows from [121], Lemma 7.2(1).
2. Follows from Lemma 4.7 (п. 1) and Lemma 4.3 (п. 3), since $\approx = \ll \cap \ll^{-1}$.
3. Follows from Lemma 4.6 (п. 4) and Lemma 4.3 (п. 3), since $\approx = \ll_w \cap \ll_w^{-1}$.

4. Let $u \in V^+$. From (2) we have $u \Rightarrow_a \in FunM_{CC}(V, A, \approx)$. Let $d_1, d_2 \in NDVC(V, A)$ and $d_1 \approx d_2$. If $u!(d_1) = T$, then $u \Rightarrow_a (d_1) \downarrow$, so $u \Rightarrow_a (d_2) \downarrow$, whence $u!(d_2) = T$. If $u!(d_1) = F$, then $u \Rightarrow_a (d_1) \uparrow$, whence

$u \Rightarrow_a (d_2) \uparrow$ since $d_2 \approx d_1$. Then $u!(d_2) = F$. In both cases $u!(d_1) = u!(d_2)$. since d_1, d_2 are arbitrary. We conclude that $u! \in PrM_{CC}(V, A, \approx)$.

5. Let $d_1, d_2 \in NDVC(V, A)$ and $d_1 \approx d_2$. If $d_1 = \emptyset$, then $d_2 \notin A$ and $TPath(d_2) = \emptyset$, so $Path(d_2) = \emptyset$ and $d_2 = \emptyset$. Similarly, if $d_2 = \emptyset$, then $TPath(d_1) = \emptyset$, so $Path(d_1) = \emptyset$ and $d_1 = \emptyset$. Then $d_1 = \emptyset$ if and only if $d_2 = \emptyset$, whence $IsEmpty(d_1) = IsEmpty(d_2)$. Since d_1, d_2 are arbitrary, we conclude that $IsEmpty \in PrM_{CC}(V, A, \approx)$.

Lemma is proved.

Theorem 4.6. $PrM_{CC}(V, A, \approx)$ and $FunM_{CC}(V, A, \approx)$ form a subalgebra of the associative algorithmic Glushkov algebra $NGA_{CC}^a(V, A)$.

Proof follows from Lemma 4.8 and Theorem 4.5.

Theorem 4.7. Each function expressible in $NGA_{CC}^a(V, A)$ belongs to $PrM_{CC}(V, A, \approx)$.

Proof follows from Theorem 4.6.

Note that $PrNS(V, A) = PrM_{CC}(V, A, \approx)$ and $FunNS(V, A) = FunM_{CC}(V, A, \approx)$. Then from Theorem 4.6, it follows that the sets of nominative stable predicates and functions form a subalgebra of $NGA_{CC}^a(V, A)$.

Theorem 4.7 has an interesting interpretation. A programmer can construct nominative stable programs by assuming a specific hierarchical structure of naming, but the program will give an equivalent result, if the input data are replaced with nominative equivalent data. Such stability simplifies programming with complex data and makes it “softer”, since a programmer does not need to take into account the current structure of naming. Moreover, the facts described above allow one to reduce logics over nominative data to logics over flat data which are closer to the classical logic.

Questions and exercises

1. Give the definitions of the main types of nominative data.
2. Give the definitions of the operations of naming, associative denaming and overlapping.
3. Give the definitions of the main compositions of functions over nominative data.
4. What is an associative nominative algorithmic Glushkov algebra?
5. What is nominative stability?
6. Give an example of a nominative stable predicate.
7. Give an example of a nominative stable program.

PART II. APPLICATION OF FORMAL PROGRAM SPECIFICATION METHODS FOR SOFTWARE DEVELOPMENT AND VERIFICATION

This part is devoted to the developed methods and software tools based on the considered theoretical fundamentals of program specification.

In Chapter 5, the process of formal algorithm design based on algebras is described. We examine the metarules of scheme design, algorithmic language SAA/1, algebra-grammatical and algebra-dynamic models of sequential and parallel programs.

The programming automation tools — the algebra-algorithmic toolkit for design and synthesis of programs and the re-writing rules system TermWare are considered in Chapter 6.

Chapter 7 gives examples of usage of the algebra-algorithmic toolkit and TermWare for development and transformation of sequential and parallel programs. Formalization and verification of parallel programs using a proof assistant software Mizar is considered.

Chapter 5

Formal design of algorithms and programs

This chapter is devoted to facilities of formalized specification of structured schemes of sequential and parallel algorithms represented in analytical and natural-linguistic forms based on Glushkov system of algorithmic algebras.

5.1. Metarules of algorithm specification design

The algorithm design process is associated with applying the following metarules [159]:

- *involution* — descending design of algorithm schemes;
- *convolution* — transition to algorithm schemes belonging to the higher level of algorithm description;
- *reinterpretation* — combined design of algorithms, consisting in the consecutive application of convolution and involution;
- *transformation* — conversion of algorithms by means of algebraic equalities.

The metarules are applied for a transition between algorithms and generation of new algorithmic knowledge.

The involution, convolution and reinterpretation metarules are based on using the system of equalities of the form

$$J = \{v_1 = T_1; v_2 = T_2; \dots; v_r = T_r\},$$

where v_i are predicate and operator variables; T_i are algebraic terms, represented in SAA-M, $i=1, 2, \dots, r$.

The regular scheme of an algorithm which does not contain occurrences of variables v_i is called an *interpreted* scheme.

The scheme is called *partially interpreted*, if it contains occurrences of variables v_i and basic operators and/or predicates.

The scheme not containing basic elements is called *non-interpreted*.

Non-interpreted and partially interpreted schemes are called *processing strategies*.

The metarule of *convolution* of the regular scheme F by the system of equalities J , consists in finding all the entries of right parts T_i of equalities in the scheme F and replacing them with corresponding left parts (variables) v_i . The result of the convolution of the scheme F by the system J is the processing strategy $S_1 = F \uparrow J$. The convolution is aimed at abstracting the regular scheme.

The metarule of *involution* of the regular scheme F based on the system J , consists in finding all the entries of variables v_i in the scheme F and replacing them with corresponding right parts T_i of equalities. The result of the involution of the scheme F by the system J is the regular scheme $S_2 = F \downarrow J$. The involution is applied for a more detailed elaboration of the regular scheme.

The *reorientation* of the scheme F_1 into the scheme F_2 is a derivative metarule that consists in the convolution of F_1 by the system J_1 and involution of the obtained strategy by the system J_2 : $F_2 = (F_1 \uparrow J_1) \downarrow J_2$.

The special case of reorientation is the *reinterpretation* of a scheme, which is the replacement of its basic elements (operators and predicates). The reinterpretation defines interrelation between regular schemes which have identical algorithmic structure, but the different interpretation in terms of basic concepts of a subject domain.

The metarule of *transformation* is a conversion of schemes as a result of the application of equalities of the type:

$$t_1(x_1, x_2, \dots, x_s) = t_2(x_1, x_2, \dots, x_s),$$

where t_1 and t_2 are terms represented in SAA-M, which depend on variables $x_1, x_2, \dots, x_s$.

Example 5.1. Consider the use of metarules for a transition from the sort algorithm *Bubble* considered in Subsection 2.1 to a

search algorithm. The regular scheme of the sort algorithm is the following:

$$\text{Bubble} = \{[\text{Sorted}] \{[d(Y_1, K)]([l > r | Y_1] \text{Transp}(l, r | Y_1), E) * \\ * R(Y_1)\} * \text{SET}(Y_1, H)\}.$$

For the transition to a search algorithm, we will apply the convolution and involution metarules. The convolution is used for abstracting the regular scheme and is based on using the following system of equalities:

$$J' = \{N_1 = \text{Sorted}; N_2 = d(Y_1, K); N_3 = l > r | Y_1; \\ O_1 = \text{Transp}(l, r | Y_1); O_2 = \text{SET}(Y_1, H)\},$$

where N_i are predicate variables ($i=1,2,3$); O_j are operator variables ($j=1,2$).

The convolution of the scheme consists in replacement of predicates and operators by corresponding variables N_i and O_j . The result of the convolution is the following partially interpreted regular scheme:

$$S_1 = \{[N_1] \{[N_2] ([N_3] O_1, E) * R(P_1)\} * O_2\}.$$

Further, we will convert the above scheme S_1 to an algorithm of searching records in a file. The search algorithm will be applied to a file F_0 (a sequence of formatted records)

$$F_0 : H Y_1 a_1 a_2 \dots a_n K,$$

where H, K are markers fixing the beginning and the end of the file F_0 ; Y_1 is a pointer moving over the file F_0 ; a_i is a record in the file.

Records in the file F_0 are searched according to the array of queries

$$AQ: H \ Y_2 \ q_1 \ q_2 \ \dots \ q_s \ K,$$

where each query $q_i: D \rightarrow E_2$ is a predicate, which is defined on the set $D = \{a_1, a_2, \dots, a_n\}$ of records of the file F_0 and takes the values in the set $E_2 = \{0, 1\}$ for any $i = 1, \dots, s$. If $q_i(a) = 1$ for some element $a \in D$, then the element a satisfies the query q_i .

For converting the scheme S_1 to a search algorithm, we apply the involution metarule, based on the usage of the following system of equalities:

$$\begin{aligned} J'' &= \{N_1 = d(Y_2, K); N_2 = d(Y_1, K); N_3 = SQUERY(r | Y_1); \\ O_1 &= OUTPUT(r | Y_1); O_2 = R(Y_2)\}, \end{aligned}$$

where N_i are predicate variables ($i = 1, 2, 3$); O_j are operator variables ($j = 1, 2$); $d(Y_2, F)$ is a predicate, which takes the true value, if pointer Y_2 reached the end of the array of queries AQ ; $d(Y_1, F)$ is a predicate, which takes the true value, if the pointer Y_1 reached the end of the file of records F_0 ; $SQUERY(r | Y_1)$ is a predicate, which takes the true value, if the record is found; $OUTPUT(r | Y_1)$ is the operator outputting the search result; $R(Y_2)$ is the operator, shifting the pointer Y_2 over the array of queries AQ .

The involution consists in replacement of variables N_i and O_j by corresponding predicates and operators. The result of the involution of S_1 is the following scheme:

$$\begin{aligned} Search &= \{[d(Y_2, K)] \ [d(Y_1, K)] \\ &([SQUERY(r | Y_1)] \ OUTPUT(r | Y_1, E) * R(Y_1)) * \\ &* R(Y_2)\}. \end{aligned}$$

According to the above scheme, the pointer Y_1 scans in the direction from left to right over the file F_0 . The file F_0 may contain more than one record, satisfying the current query, which is observed by the pointer Y_2 . When such a record is found, the operator $OUTPUT(r|Y_1)$ is executed and the pointer Y_1 moves by one position to the right. Further, the search is continued until the pointer Y_1 reaches the end of the file. After the pointer Y_1 returns to the beginning of the file, the algorithm proceeds to process the next query.

Example 5.2. Consider the use of the transformation metarule for the conversion of the following fragment of *Bubble* scheme:

$$([l > r | Y_1] \text{ Transp}(l, r | Y_1), E) * R(Y_1).$$

The transformation of the above scheme is based on using the following equality, which is the distributive property of composition and branching operations:

$$R_1: ([u] A, B) * C = ([u] A * C, B * C),$$

where u is a predicate variable; A, B, C are operator variables.

The equality R_1 is applicable under any interpretations of included variables and is an example of identity in SAA-M. Let us introduce the following interpretations of variables included in R_1 :

$$u = l > r | Y_1; A = \text{Transp}(l, r | Y_1); B = E; C = R(Y_1).$$

We substitute the introduced interpretations as values of variables into equality R_1 and apply the equality in the direction from right to left. As a result, we obtain the following regular scheme:

$$([l > r | Y_1] \text{ Transp}(l, r | Y_1) * R(Y_1), E * R(Y_1)).$$

The metarules are useful for improving programs by a selected criterion, for example, used memory, execution time and other.

5.2. Algorithmic language SAA/1

The algorithmic language SAA/1 [5, 112] is intended for multilevel structured designing and documenting sequential and parallel algorithms and programs. The mathematical basis of this language is SAA and its modifications. The descriptions of algorithms in SAA/1 language are called SAA schemes and the purpose of this subsection consists in the explanation of possibilities of this language.

The basic objects of the SAA/1 language are abstractions of operators (converters of data) and conditions (predicates). The identifiers of operators and predicates are subdivided into standard and semantic and can be either basic (elementary) or compound. *Basic* elements are considered in an SAA scheme as a primary, atomic, indivisible abstractions. *Compound* elements are designed from elementary ones by means of the following SAA-M operations represented in a natural language form (see also Table 2.1 given in Chapter 2):

- disjunction: ‘condition 1’ OR ‘condition 2’;
- conjunction: ‘condition 1’ AND ‘condition 2’;
- negation: NOT ‘condition’;
- sequential execution of operators: “operator 1” THEN “operator 2”
(or “operator 1”; “operator 2”);
- branching: IF ‘condition’ THEN “operator1” ELSE “operator2” END IF;
- cyclic operator execution: WHILE NOT ‘condition-of-loop-termination’ LOOP “operator” END OF LOOP;
- asynchronous execution of operators: “operator1” PARALLEL “operator2”;
- control point: CP ‘condition’;
- synchronizer: WAIT ‘condition’.

Standard identifiers are defined in the alphabet consisting of English letters, numbers and a symbol of an underscore.

The semantic identifier of the operator (accordingly, condition) is the sequence of any length of any symbols set into double quotes (accordingly, single quotes). Usage of semantic identifiers essentially facilitates understanding of algorithms presented in SAA/1 language. For example, the following texts can serve as semantic identifiers of operators and conditions:

“Place the pointer Y(1) at the beginning of the array (M)”,
‘The array (M) is sorted’,

Example 5.3. As an illustration, consider the following SAA scheme of a bubble sort algorithm.

```
SCHEME BUBBLE SORT =====  
    “Sequential bubble sort”  
    END OF COMMENTS  
  
    “BUBBLE”  
    ===== “START”  
        THEN  
            “Place the pointer Y(1) at the beginning of  
            the array (M)”  
        THEN  
            WHILE NOT ‘The array (M) is sorted’  
            LOOP  
                WHILE NOT ‘The pointer Y(1) is at the end of  
                the array (M)’  
                LOOP  
                    IF ‘The elements l > r at pointer Y(1) in  
                    the array (M)’  
                    THEN “Transpose the elements l, r at pointer Y(1)  
                    in the array (M)”  
                    ELSE “Empty operator”  
                    END IF;  
                THEN
```

```

        "Shift the pointer Y(1) by (1) element in the
        array (M) to the right"
    END OF LOOP
    THEN
        "Place the pointer Y(1) at the beginning of the
        array (M)"
    END OF LOOP
    THEN
        "FIN";

"START"
==== "Open the file (F1) for reading"
    THEN
        "Read the data from the file (F1) to the array (M)"
    THEN
        "Close the file (F1)";

"FIN"
==== "Open the file (F2) for writing"
    THEN
        "Write the array (M) into the file (F2)"
    THEN
        "Close the file (F2)";

END OF SCHEME BUBBLE SORT

```

The SAA scheme begins with the SCHEME keyword followed by a name of the scheme and comments to it. The end of the title of the SAA scheme is designated by a keyword sentence END OF COMMENTS. The name of SAA scheme can be a character sequence of any length consisting of English alphabet letters, numbers, a point, a comma, minus and slash symbols, and also round, square and angle brackets. The presence of the comment (together with a chain of symbols "=") is optional. Any SAA scheme ends with the END keyword followed by an optional SCHEME keyword and a name of SAA scheme.

It is easy to see the functionality and narrative style of notation of the given algorithm, and also its multilevel structure. The levels of the scheme are marked by the left parts of equalities, and the structure of each level is concretized (in terms of SAA-M) by the right part of the corresponding equality. The left part of equality is separated from the right one by the chain of symbols “=”. Equalities are separated from each other by a semicolon.

In the given example, only semantic identifiers appear, which are abstractions of operators and conditions. Those abstractions, which identifiers are included in the left parts of equalities of SAA schemes, are abstractions of compound operators and conditions, and the rest are elements primary at a given level of algorithm description. Thus, in the above illustrative example, abstractions of compound operators are designated by semantic identifiers “BUBBLE”, “START” and “FIN”.

The prominent features of SAA/1 language are the following:

- the multilevel structure of SAA schemes represented in a form of a list of equalities; each next equality concretizes (by means of SAA-M) an abstraction of some operator or a condition, which identifier is included in one of previous equalities;
- the use of semantic and standard identifiers of operators and conditions, which allows designing algorithms both in the style close to a natural language and in a form of algebraic formulas;
- the redundancy of syntactic representation of the same objects and operations of the language;
- an arbitrary length of identifiers of conditions and operators.

Thus, the basis of the SAA/1 language is the facilities of Glushkov algorithmic algebras and their modifications, focused on formalization of sequential and parallel algorithms by means of structured schemes. The main feature of SAA/1 language is its focus on multilevel descending, ascending and combined designing of classes of algorithms and programs, associated with a chosen subject domain. Synthesis of programs on the basis of SAA schemes can be carried out in nonautomated and automated modes. In the first case, the algorithm represented as an SAA scheme is translated manually

to a program text in a chosen programming language (target language). The SAA/1 language is the basis of the method of multilevel structured program design and its tools [5, 11, 43, 112, 174] providing automated and automatic (under certain conditions) synthesis of programs in a target language. During the synthesis, the semantic libraries containing interpretations of basic operations of SAA/1 language are used. These interpretations are represented in terms of a target language. Such interpretation is used not only in connection with implementation of well-known algorithmic structures, but also data structures. Thereby, it is a question of an interrelated designing of algorithms and data structures. Along with semantic libraries, the libraries of basic concepts adequate to actual subject domains are formed. The formation of semantic libraries and libraries of subject domains is associated with a construction of a family of languages of SAA schemes based on SAA/1 language. So, at the formation of the corresponding library of basic interpretations, the construction of specialized languages of SAA schemes for a description of sequential and parallel algorithms of sorting, search in multilevel files, designing of language processors, etc. [5, 11, 43] is carried out.

5.3. Algebra-grammatical models for generation of algorithm specifications

One of the essential problems of the algebra of algorithmics is to increase the adaptability of programs to specific conditions of their use. In particular, the problem can be solved by using a parameter-driven generation of algorithm specifications by means of higher-level algorithms, which are called hyperschemes [174, 177]. In this subsection, the approach to generation of sequential and parallel algorithms on the basis of the algebra of hyperschemes is proposed. Hyperscheme is a parameterized algorithm for solving a certain class of problems; setting specific values of parameters and subsequent interpretation of a hyperscheme allows obtaining algorithms adapted to specific conditions of their use. Hyperschemes are adjacent to well-known methods of transformational synthesis: term rewriting systems (see Chapter 3 and also [13, 35, 38]), mixed computations [22] and macrogeneration [52].

The basis of an algebraic apparatus of generation of schemes is an abstract automaton model of the parameter-driven generator of texts [177] and also the means of SAA-M. Similarly to the abstract automaton model of a computer, considered in Subsection 1.2, the model of the parameter-driven generator of texts works according to a feedback principle. A stack automaton $\bar{\Psi}$ is used as a control automaton and an automaton $\bar{\Phi}$ with tape $\tilde{L}$ is used as an operational one (Fig. 5.1).

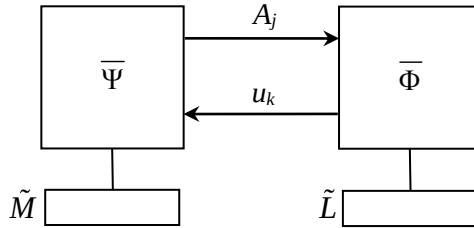


Fig. 5.1. An abstract automaton model of the parameter-driven generator of texts

The tape $\tilde{L}$ is used for writing the text of RS being generated. The set $\bar{M}$ of states of the automaton $\bar{\Phi}$ is associated with parameters, which control the generation of schemes. The elements of the information set $\bar{P} = \bar{M} \times \bar{L}$ (where $\bar{L}$ is a set of states of the tape $\tilde{L}$) are called the states of the operational structure. On each step of work of the automaton, the set of logical conditions $\{u_k | k \in I\}$ defined on the set $\bar{P}$ comes from the operational to the control automaton. According to these conditions and a content of the stack $\tilde{M}$, the control automaton initiates the execution of some operator. The set of operators $Op = \{A_j | j \in I\}$ is divided into two disjoint sets: R_T (terminal operators) and R_N (nonterminal operators). The execution of a terminal operator from the set R_T consists in a change of a current state of the operational structure, which, in particular, can be writing some text on the tape $\tilde{L}$. The execution of an operator

$A \in R_N$ at the current state $p \in \bar{P}$ consists in writing some term $F_N(A, p)$ to the stack $\tilde{M}$ and its further interpretation by the control automaton. The term $F_N(A, p)$ is an analog of notions of macro definition, procedure, subroutine, etc. The stack $\tilde{M}$ is used for processing nested and recursive terms. The generated text is a content of the tape $\tilde{L}$ at a final state of the operational structure.

5.3.1. Algebra of hyperschemes. The algebra of hyperschemes is the formalism, which is based on the above abstract automaton model and is applied for the parameter-driven generation of regular schemes of algorithms represented in SAA-M. The algebra of hyperschemes is a two-sorted algebra

$$AHS = \langle \{Pr, Op\}; \Omega_{AHS} \rangle,$$

where Pr is a set of predicates defined on the information set $\bar{P}$ and taking the values of four-valued logic $E_4 = \{0, 1, \mu, \eta\}$, where 0 is for false, 1 is for true, μ is for unknown, η is for “not computed”; Op is a set of operators, defined on an information set $\bar{P}$ and taking values from the set $\bar{P}$; Ω_{AHS} is a signature of operations.

The set of predicates is associated with parameters, which control the process of generation of a regular scheme of an algorithm. The operations of the signature Ω_{AHS} are similar to the operations of SAA-M signature (see Subsection 2.3). In particular, the signature Ω_{AHS} includes Boolean operations, i.e. disjunction $u \vee u'$, conjunction $u \wedge u'$, negation $\bar{u}$, prediction $A \cdot u$, and operator operations: composition $A * B$, branching $([u] A, B)$, loop $\{[u] A\}$, selection operation $SELECT(u_1 \rightarrow A_1, \dots, u_n \rightarrow A_n)$ and other [177]. The difference from SAA-M is that predicates from the set Pr map elements of the information set $\bar{P}$ to the elements of the set $E_4 = \{0, 1, \mu, \eta\}$, where the additional value η stands for “not computed”. The element η is used to indicate that the value of a condition cannot be calculated due to the lack of information about

parameter values. The truth tables for this four-valued logic are given in [177].

The application of the operator $A \in Op$ at the state $p \in \bar{P}$ leads to a transition to the new state $A(p) \in \bar{P}$ and writing the fragment $F(A, p)$ of a regular scheme being generated on the tape $\tilde{L}$, where $F(A, p)$ is the function that specifies the generation technique for all operations of signature of *AHS* and will be considered below.

By analogy with SAA-M, an operator representation of an algorithm in *AHS* is called a *regular hyperscheme (RHS)* [177]. Each RHS A , being applied to a state $p \in \bar{P}$, generates RS $F(A, p)$. The hyperscheme A defines the class $L(A)$ of regular schemes in SAA: $L(A) = \{F(A, p) \mid p \in \bar{P}\}$.

The function $F(A, p)$ for the main operations of *AHS* was defined in work [177]. Particularly, the composition operation $A * B$ generates the operator $C = A * B$ without changes, according to the function $F(C, p) = F(A, p) * F(B, p)$, where $p \in \bar{P}$.

The operation of alternative $([u] A, B)$, where $u \in Pr$, $A \in Op$, $B \in Op$ generates the operator $C = ([u] A, B)$, such that for each $p \in \bar{P}$

$$F(C, p) = \begin{cases} F(A, p), & \text{if } u(p) = 1; \\ F(B, p), & \text{if } u(p) = 0; \\ ([u(p)] F(A, p), F(B, A(p))), & \text{if } u(p) = \eta; \\ e, & \text{if } u(p) = \mu, \end{cases}$$

where e is an empty text.

According to the given definition, the result of the interpretation of the operation of alternative will be the text of the operator A , if the value of the condition $u(p)$ is true. The text of the operator B is generated, if the condition $u(p)$ is false. The text of the operation of an alternative without changes is generated, if the value of $u(p)$

was not computed, and the empty text will be the result, if an error occurred during the interpretation.

Example 5.4. Consider the application of *AHS* to transformation of a hybrid sort algorithm *HSort*. The algorithm reads a set of input numerical arrays and calls one of the sorting algorithms (*insertionSort*, *quickSort* or *mergeSort*) depending on the size n of an input array [174]. The arrays of size $n < MIN$ are sorted by *insertionSort*, the arrays of size $n > MAX$ are processed by *quickSort* and arrays getting to an interval $[MIN, MAX]$ are sorted by *mergeSort* algorithm. The regular scheme of the algorithm is

$$\begin{aligned}
 HSort = & INIT * \{ [END_OF_SET] READ_ARRAY(A) * \\
 & * ([n < MIN] insertionSort(A, n), \\
 & ([n > MAX] quickSort(A, n), mergeSort(A, n))) \},
 \end{aligned}$$

where *INIT* is an initialization operator; *END_OF_SET* is the condition, which takes the true value, if all the arrays from the input set have been processed, and takes false value otherwise; *READ_ARRAY(A)* is an operator that reads the array; *A* is an input array of the length n .

Let it is in advance known that the algorithm *HSort* will be applied in conditions when the size of all input arrays is in a certain range, say, not less than MIN . Then the given RS becomes superfluous and for its reduction, we will regard it as the hyperscheme with the parameter n . At the stage of interpretation of the hyperscheme *HSort*, the predicate $n < MIN$ takes the value 0, whereas $n > MAX$ takes the value η . The condition $n > MAX$ cannot be computed, because we do not have enough information about the value of the parameter n (it is only known that $n \geq MIN$). Assuming that the function F is identical on a set of all basic operators and conditions of the hyperscheme, we will obtain the reduced RS

$$\begin{aligned}
 F(HSort, p_0) = & INIT * \{ [END_OF_SET] READ_ARRAY(A) * \\
 & * ([n > MAX] quickSort(A, n), mergeSort(A, n))) \},
 \end{aligned}$$

where $p_0 \in \overline{P}$ is the initial state of the information set IS .

Thus, by setting various values of the parameter n of the hyperscheme $HSort$ (in the considered case the value $n \geq MIN$ was set), it is possible to obtain RS, optimal to given conditions.

Example 5.5. The paper [173] describes a more complicated hybrid sorting scheme, designed with the help of machine learning technique. This algorithm also can be regarded as a hyperscheme, below we give its fragment in a natural-linguistic form.

```

HYPERSCHEME hybridSort2(A, n)
===== IF 'size <= 30' THEN
    IF 'runs <= 0.7' THEN "insertionSort(A, n)"
    ELSE IF 'runs > 0.7' THEN
        IF 'size <= 10' THEN "insertionSort(A, n)"
        ELSE IF 'size > 10' THEN "quickSort(A, n)"
        END IF
    END IF
END IF
ELSE IF 'size > 30' THEN
    IF 'size <= 40' THEN "insertionSort(A, n)"
    ELSE IF 'size > 40' THEN "quickSort(A, n)"
    END IF
END IF
END IF
END IF

```

In the given scheme, the selection of one of the algorithms is implemented based on the size n of the input array A and its presortedness degree ($runs$). The presortedness degree of the array A is computed according to the formula [173]: $runs(A) = runs_up(A) / n$, where $runs_up(A)$ is the number of ascending substrings or the “runs up” of the array A ; $runs(A)$ takes values in the range $(0 \dots 1]$. Let it is in advance known that the algorithm described by the given scheme will be applied in conditions when the size of all input arrays is $n < 20$, and the degree of

presortedness is $runs = 0.9$. Then as a result of interpretation of the hyperscheme, we obtain the reduced SAA scheme:

```
SCHEME hybridSort2'(A, n)
==== IF 'size <= 10' THEN
      "insertionSort(A, n)"
ELSE
      IF 'size > 10' THEN
        "quickSort(A, n)"
      END IF
    END IF
```

By analogy with the modified SAA, in [177] the modified *AHS* was proposed, oriented on the sequential and parallel generation of sequential and parallel RS represented in SAA-M. The signature of the modified *AHS*, in particular, contains the operation of asynchronous disjunction $A // B$, consisting in parallel execution of operators A and B . For generation of asynchronous disjunction, the operation of its sequential generation $(//)(A, B)$ or asynchronous generation $(///)(A, B)$ is used. For synchronization of processes, initiated by asynchronous disjunctions, control points and synchronizers are used. Each control point $CP(u)$ is associated with the condition u , which is false while a computation process has not yet reached the point $CP(u)$, and true from the moment of achievement of the given point. The synchronizer $S(u)$ implements the delay of the computation process at a false value of u until u gets a true value. The application of these constructs is considered further.

5.3.2. Application of hyperschemes to a representation of derivation algorithms in generative grammars. In this subsection, the algebra of hyperschemes is applied for representation algorithms of inference control in structured design grammars. The approach is illustrated on an example of an algorithm from the subject domain of linear algebra.

Structured design grammar (SDG) [5] is the 5-tuple

$$G^D = (T^D, N^D, \sigma^D, P^D, U^D)$$

where $T^D = \bar{Z} \cup \bar{R}$ is a set of terminal symbols, $\bar{Z}$ is a set of basic conditions, operators and data objects, $\bar{R}$ is a set of separators: symbols of SAA-M operations, brackets etc.; N^D is a set of non-terminal symbols (logical, operator and object metavariables); $\sigma^D \in N^D$ is a start symbol; $P^D = \{\beta_i; v_i \rightarrow w_i \mid i \in I\}$ is a set of rules; U^D is a derivation control algorithm. In this work derivation control algorithm is represented in the algebra of hyperschemes.

The set $L(G^D) = \{X \mid \sigma^D \xRightarrow{*} X\}$ of PRSs over the basis Σ , which are generated from a start symbol σ^D in SDG G^D , forms the language $L(T^D)$, associated with a class of programs being designed.

A variant of a structured design grammar is a matrix SDG. The *matrix* SDG is the SDG $G^D = (T^D, N^D, \sigma^D, P^D, U^D)$ the set of rules of which $P^D = \{m_j \mid j \in I\}$ consists of generalized (matrix) productions of the form:

$$m_j: \left\| \begin{array}{l} \gamma_1; \gamma_2; \dots \\ \dots \quad \dots \quad \dots \\ \gamma_l; \gamma_{l+1}; \dots \end{array} \right\|,$$

where γ_l are productions of the form $v_l \rightarrow w_l$, $l \in I$. In the process of derivation, generalized productions can be applied both sequentially and in parallel. In the case of sequential application, the rules are written in one line and are separated by a semicolon, and at parallel application they are written in a column.

Example 5.6. Let us illustrate the process of generating a scheme of parallel matrix multiplication algorithm with the usage of SDG and *AHS*. The algorithm multiplies two rectangular matrices:

$A = (a_{lj})_{M \times N}$ and $B = (b_{lj})_{N \times Q}$. The elements of a resultant matrix $C = (c_{lj})_{M \times Q} = A \times B$ are defined according to the formula

$$c_{lj} = \sum_{k=0}^{N-1} a_{lk} \cdot b_{kj}, \quad l = 0, \dots, M-1, j = 0, \dots, Q-1,$$

where the elements of matrices are indexed beginning with zero.

In the parallel algorithm computations are performed by K processors in such a way that the first processor computes the first $(M \cdot Q) / K$ elements of a resultant matrix, the second one processes the following $(M \cdot Q) / K$ elements, and so on. Thus, the initial matrix A and the final matrix C are divided into horizontal blocks shown in Fig. 5.2. The processor with the number i multiplies block A_i by matrix B and as result, the block C_i is obtained. All blocks of matrices are also indexed beginning with zero.

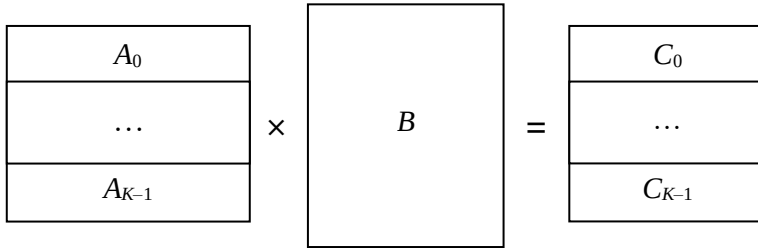


Fig 5.2. Splitting matrices into blocks in the parallel matrix multiplication algorithm

The parallel regular scheme of the algorithm is

$$\begin{aligned}
 & \text{MatrixMultiplication}(K) = \text{START}(K) * \\
 & * (\text{Thread}(A_0, B) // \text{Thread}(A_1, B) // \dots // \text{Thread}(A_{K-1}, B)) * \\
 & * S(\text{PROC_FIN}) * \text{FIN},
 \end{aligned}$$

where $START(K)$ is the operator of initialization of matrices and preparation for launching K parallel threads; $Thread(A_i, B)$ is the operator executing the multiplication of the i -th block of the matrix A by the matrix B ; $S(PROC_FIN)$ is the synchronizer operation, which delays the computation until all threads complete their work; FIN is the final operator which outputs a resultant matrix C .

The regular scheme of the compound operator $Thread(A_i, B)$ is presented below.

$$\begin{aligned}
 &Thread(A_i, B) = \\
 &\quad (start := M / K * i) * \\
 &\quad * (end := M / K * (i + 1)) * \\
 &\quad * For(l, start, end - 1) \\
 &\quad \{ \\
 &\quad \quad For(j, 0, Q - 1) \\
 &\quad \quad \{ \\
 &\quad \quad \quad (value := A[l][0] * B[0][j]) * \\
 &\quad \quad \quad * For(k, 1, N - 1) \\
 &\quad \quad \quad \quad \{ value := value + A[l][k] + B[k][j] \} * \\
 &\quad \quad \quad * (C[l][j] := value) \\
 &\quad \quad \quad \} \\
 &\quad \quad \} * CP(PROC_FIN(i)),
 \end{aligned}$$

where $CP(PROC_FIN(i))$ is a control point fixing the moment of completion of computations in the thread with the index i .

Consider the matrix SDG $G_1^D = (T_1^D, N_1^D, \sigma_1^D, P_1^D, U_1^D)$ intended for generating the class of parallel regular schemes $MatrixMultiplication(K)$. The SDG generates algorithms from the mentioned class with a various number of parallel threads K . The rules of the grammar G_1^D provide the generation of parallel threads

$Thread(A_i, B)$ by changing the values of the parameter i from 0 to $K-1$. The set of rules P_1^D of the SDG G_1^D is the following:

$$m_0: \left\| \sigma_1^D \rightarrow START(0) * PRS1 * S(PROC_FIN) * FIN \right\|,$$

$$m_1: \left\| PRS1 \rightarrow Thread(A_0, B) // PRS1 \right\|,$$

$$m_2:$$

$$\left\| \begin{array}{l} Thread(A_i, B) // PRS1 \rightarrow Thread(A_i, B) // Thread(A_{i+1}, B) // PRS1 \\ START(i) \rightarrow START(i+1) \end{array} \right\|,$$

$$\text{at } i < K-2,$$

$$m_3: \left\| \begin{array}{l} Thread(A_i, B) // PRS1 \rightarrow Thread(A_i, B) // Thread(A_{i+1}, B) \\ START(i) \rightarrow START(i+2) \end{array} \right\|,$$

$$\text{at } i = K-2,$$

where $START(i) \in N_1^D$, $PRS1 \in N_1^D$ are operator nonterminals; A_i is an object nonterminal.

The process of generation of the algorithm scheme according to given rules is the following. The rule m_0 forms the general structure of the scheme. With the help of the rule m_1 , the thread with the index 0 is formed. Then the rule set m_2 recursively generates the next threads of the PRS at $i < K-2$. The process completes with the application of the rule set m_3 at $i = K-2$.

Further, the process of designing hyperschemes representing algorithms in context-free SDGs according to the method given in [177] is considered.

Let $AHS = \langle \{Pr, Op\}; \Omega_{AHS} \rangle$ be defined on the information set $\bar{P}$ associated with parameters of derivation control in SDG

$G^D = (T^D, N^D, \sigma^D, P^D, U^D)$. The rules $\beta_i: v_i \rightarrow w_i$ of the grammar G^D with identical left parts are written as the set of equalities f_i :

$$\{f_i: v_i = w_1^i \mid w_2^i \mid \dots \mid w_{t_i}^i \mid i = 1, 2, \dots, n\},$$

where t_i is the number of alternative rules in the i -th equality; v_i are the operator and logical metavariables (nonterminals); $w_1^i, w_2^i, \dots, w_{t_i}^i$ are the right parts of the rules, which are operators and conditions including both terminal and nonterminal symbols. The mentioned metavariables are associated with corresponding compound operators and conditions of *AHS*. Basic operators and conditions of SAA-M, included in the right parts of rules of G^D , are associated with analogous basic elements of *AHS*. The application of a basic operator or a condition of *AHS*, which has the value η , at the state $p \in \bar{P}$ leads to writing its text without changes on the tape $\tilde{L}$. The symbols of synchronous and asynchronous disjunctions (if they are present) included into expressions w_j^i ($i = 1, 2, \dots, n; j = 1, 2, \dots, t_i$), are substituted with the symbols of operations of asynchronous and sequential generation of synchronous and asynchronous disjunctions. The nonterminals v_i included into the equalities f_i , which have several alternative rules, are associated with the compound operator which is the composition of some operator $O_i \in Op$ and a selection operation:

$$v_i = O_i * \text{SELECT}([u_1^i] \rightarrow w_1^i, [u_2^i] \rightarrow w_2^i, \dots, [u_{t_i}^i] \rightarrow w_{t_i}^i),$$

where u_j^i are conditions, such that $\forall p \in P: u_j^i(p) \neq \eta$ ($j = 1, 2, \dots, t_i$). The operator O_i sets the true value of one of the conditions and the false value of other conditions u_j^i , depending on the current state $p \in \bar{P}$. At the execution of operator O_i , writing a text on the

tape $\tilde{L}$ is not carried out. (The operator O_i can be the operator $INC(i)$ increasing the parameter i by value 1.) The operators w_j^i in the selection operation are expressions obtained on the basis of the right parts w_j^i of the equalities f_i .

The grammar G^D is associated with the regular hyperscheme v_1 :

$$v_1 = O_1 * \text{SELECT}([u_1^1] \rightarrow w_1^1, [u_2^1] \rightarrow w_2^1, \dots, [u_{t_1}^1] \rightarrow w_{t_1}^1);$$

$$v_2 = O_2 * \text{SELECT}([u_1^2] \rightarrow w_1^2, [u_2^2] \rightarrow w_2^2, \dots, [u_{t_2}^2] \rightarrow w_{t_2}^2);$$

...

$$v_n = O_n * \text{SELECT}([u_1^n] \rightarrow w_1^n, [u_2^n] \rightarrow w_2^n, \dots, [u_{t_n}^n] \rightarrow w_{t_n}^n).$$

The result of application of the regular hyperscheme v_1 at the state $p \in \bar{P}$ is the PRS $F(v_1, p)$. Here $F(v_1, p) \in L(G^D)$, where $L(G^D)$ is the language generated by grammar G^D .

Example 5.7. Consider the regular hyperscheme *MatrixMultHS*, which is the derivation control algorithm for SDG G_1^D built in Example 2.6 and intended for generation of the class of asynchronous PRSs of matrix multiplication. The hyperscheme was built according to the above method for designing derivation control algorithms for formal grammars and is the following:

$$\begin{aligned} \text{MatrixMultHS} = & (i := -1) * (K := 4) * \text{START}(K) * \text{PRS1} * \\ & * S(\text{PROC_FIN}) * \text{FIN}; \end{aligned}$$

$$\begin{aligned} \text{PRS1} = & \text{INC}(i) * \text{SELECT}([i < K - 1] \rightarrow (//)(\text{Thread}(A, B) // \text{PRS1}), \\ & [i = K - 1] \rightarrow \text{Thread}(A, B)). \end{aligned}$$

Here K is a parameter of the hyperscheme (the number of parallel threads); $PRS1$ is the compound operator that recursively generates a sequence of threads; $INC(i)$ is an increment operator that adds 1 to the value of the index variable i .

In the given hyperscheme, the operation of sequential generation of an asynchronous disjunction is used. The start symbol σ_1^D of grammar G_1^D corresponds to the compound operator *MatrixMultHS* of the hyperscheme, and the nonterminal $PRS1$ corresponds to the compound operator with the same name. The operators $START(K)$, $Thread(A_i, B)$ and FIN are identical on information set $\bar{P}$ and their execution consists in writing on the tape $\tilde{L}$ the text of the corresponding operator with current values of variables i and K .

The execution of the hyperscheme *MatrixMultHS* begins with the initialization of the variables i and K . The parameter K , which corresponds to a number of parallel threads, is assigned the value 4. Then the text of the operator $START(4)$ is generated. After this the compound operator $PRS1$ is applied, in which with the help of changing the value of i from 0 to $K-1$ and a selection operation, the branches $Thread(A_i, B)$ are recursively formed. The generation of PRS is finished with writing the text of basic operator FIN on the tape $\tilde{L}$.

The result of the application of the hyperscheme *MatrixMultHS* at the state $p \in \bar{P}$ at the value of the parameter $K=4$ is the PRS *MatrixMultiplication(4)*, written on the tape $\tilde{L}$:

$$\begin{aligned} MatrixMultiplication(4) = & START(4) * \\ & * (Thread(A_0, B) // Thread(A_1, B) // \\ & // Thread(A_2, B) // Thread(A_3, B)) * \\ & * S(PROC_FIN) * FIN. \end{aligned}$$

Thus, setting specific values of the parameter K and subsequent interpretation of the hyperscheme allows obtaining regular

schemes of matrix multiplication algorithms with a corresponding number of threads.

For automating the design of algorithm schemes, hyper-schemes and generation of algorithms and programs, the software toolkit was developed [174], which is considered in Chapter 6.

5.4. Algebra-dynamic models of parallel programs

The development of multicore processors leads to increasing importance of parallel programming aimed at standard, widely accessible computers, and not just for specialized high-performance systems. However there is one more direction of parallel programming which has received especial development recently, namely, the programming of general-purpose tasks for graphics processing units (GPUs) [74]. Market requirements have led to rapid development of GPUs and, at present, their computing capacity considerably exceeds the capabilities of usual processors. Therefore, GPUs were applied for solving the problems not connected directly with graphics processing. Research in these directions is supported by GPU developers: in particular, NVIDIA company provides CUDA platform [130] for general-purpose computations on GPUs.

Despite the presence of specialized facilities for CUDA, development of GPU programs remains a labor-consuming work, which demands from a developer the knowledge about low-level details of hardware and software platform. Therefore, there is a need for research in the area of automation of software development process for GPU. In this chapter, we describe the development of formal design methods, based on concepts of algebraic programming (see Chapter 3) and algebra-dynamic models of programs [5] with the usage of rewriting rules technique [38, 154] for automated development of efficient programs for GPUs. High-level models of programs and models of program execution are developed for central processing unit (CPU) and GPU. Application of rewriting rules and high-level models for automated parallelization and optimization of programs for GPUs is described. The method of automated transition between a high-

level model of a program and a source code, which is based on usage of special rewriting rules is proposed.

5.4.1. The concept of a transition system. For describing program execution, general concepts of the theory of transition systems (also called discrete dynamic systems) [5, 93] are used.

The *transition system* is defined as a triple

$$(S_0, S, d),$$

where S is the state space; $S_0 \subseteq S$ is the set of initial states; $d \subset S \times S$ is the binary transition relation over the state space. The system can move from the state s_i into the state s_j , if $(s_i, s_j) \in d$.

Let $P(S_0)$ designate the set of all finite processes, i.e. sequences of states $p = s_0 s_1 \dots s_n \dots$, beginning at S_0 , $s_0 \in S_0$, $n \geq 1$, then d can be represented as the relation

$$d \subseteq P(S_0) \times S. \quad (5.1)$$

The transition relation d and the set of initial states S_0 unambiguously define the set of admissible processes F of the system as the union $F = \bigcup_{t=0}^{\infty} F_t$, where F_t is a set of admissible processes of length t .

The sets F_t are defined recursively as follows:

1. $F_0 = S_0$;
2. $\forall t > 0, F_t = \{ps \mid p \in F_{t-1}, (p, s) \in d\}$ (i.e. the processes of the length t are obtained from the processes of the length $t-1$ by adding the states defined by the transition relation).

Therefore, the system (S_0, S, d) can be defined in an alternative way as a pair (S, F) .

Let us describe some general properties of transition systems [48]. The system (S_0, S, d) is called *finite-state automaton*, if

$(ps_1, s_2) \in d \Leftrightarrow (s_1, s_2) \in d$. For the finite-state automaton system, admissible transitions depend only on a current state and do not depend on prehistory. All the systems considered in this subsection are finite-state automaton. Therefore, instead of relation (5.1), further we will use $d \subseteq S \times S$. The system is called *deterministic*, if $\forall p \in P(S_0), \exists! s \in S : (p, s) \in d$, i.e. the transition is defined unambiguously. For the deterministic finite-state automaton systems the relation d is a function $d : S \rightarrow S$. The transition system is called *multicomponent*, if the set of its states S is contained in the Cartesian product $S \subseteq S_1 \times \dots \times S_m$. Sets $S_1, \dots, S_m$, which make the least Cartesian product containing S , are called *system components*. If system components are multicomponent transition systems, then the system is called *multilevel*. Multicomponent multilevel systems are used as a model of execution of parallel programs.

5.4.2. Sequential program execution model. Algebra-dynamic models of programs consist of two parts: a model of program structure and a model of program execution. For modeling a code we use Glushkov SAA (see Subsection 2.3). It should be reminded that SAA is the two-sorted algebra $GA = \langle \{Pr, Op\}; \Omega_{GA} \rangle$ containing the set of conditions (predicates) Pr and the set of operators Op .

The operations defined in GA , are the following.

1. Logic operations of conjunction, disjunction and negation:

$$AND : Pr \times Pr \rightarrow Pr;$$

$$OR : Pr \times Pr \rightarrow Pr;$$

$$NOT : Pr \times Pr \rightarrow Pr.$$

2. Checking a condition after operator execution:

$$AFTER : Op \times Pr \rightarrow Pr.$$

3. Sequential composition of operators:

$$THEN : Op \times Op \rightarrow Op.$$

4. Branching:

$$IF : Pr \times Op \times Op \rightarrow Op.$$

5. Iteration (a loop construct):

$$WHILE : Pr \times Op \rightarrow Op.$$

We consider the following program model. The program consists of a set of components corresponding to separate programming language functions (methods, procedures) $P = \{P_1, \dots, P_k\}$. We regard that the component is described by an identifier (a name unique within the program) and also by the model of a code:

$$P_i = (name_i, code_i), \quad name_i \in STRING, \quad code_i \in Op.$$

Procedural code is represented as an expression of Glushkov algebra.

On the sets of operators and conditions, basic elements are defined, and then it is possible to build various algebra expressions which will be described by compound operators and conditions. Basic operators and conditions usually depend on the subject domain. We will select one basic operator common for all subject domains: a call of a function $call(P_i)$.

A particular feature of programs for GPUs is their division into two parts: a code executed on CPU and specialized code for GPU. These parts can be implemented in different languages. For example, in [47] programs are considered, in which CPU code was implemented in C#, whereas GPU code was developed in C for CUDA, which is a special extension of C language.

In the GPU program model, we take into account this feature as follows. We will regard a program as a set of components; how-

ever, now each component belongs either to CPU or to GPU code. Thus, for CPU and GPU sets of basic operators can differ.

To construct the execution model, we need to define the interpretation of Glushkov algebra expressions that describe a program. Let V be the set of program variables. For simplicity, we assume that variables are typeless and take values in some universal set D . *Memory states* are partial mappings $b:V \rightarrow D$ from variables to their values. The *information environment* is the set of memory states: $B = \{b:V \rightarrow D\}$. Then, the operators of Glushkov algebra are interpreted as a function over the set B , and conditions are interpreted as predicates on the same set. The interpretation of basic operators and conditions is defined for each subject domain.

For compound expressions the interpretation is defined below, where $u \in Pr$, $v \in Pr$ are conditions; $P \in Op$, $Q \in Op$ are operators.

1. For logical operations:

$$\begin{aligned}(u \text{ AND } v)(b) &= (uv)(b) = u(b) \wedge v(b); \\(u \text{ OR } v)(b) &= (u + v)(b) = u(b) \vee v(b); \\(\text{NOT } u)(b) &= \bar{u}(b) = \neg u(b).\end{aligned}$$

2. For prediction operation (checking a condition after operator execution):

$$(u \text{ AFTER } P)(b) = (Pu)(b) = u(P(b)).$$

3. For sequential composition of operators:

$$(P \text{ THEN } Q)(b) = (PQ)(b) = Q(P(b)).$$

4. For branching:

$$(\text{IF } u \text{ THEN } P \text{ ELSE } Q)(b) = ((u)P, Q)(b) = \begin{cases} P(b), & u(b) = 1, \\ Q(b), & u(b) = 0. \end{cases}$$

5. For iteration:

$$(WHILE\ u\ P)(b) = (\{u\}P)(b) = \begin{cases} (\{u\}P)(P(b)), & u(b) = 1, \\ b, & u(b) = 0. \end{cases}$$

Now we can describe the transition system which models a sequential program execution. The states of the transition system have the form $s = (b, R, F)$, where $b \in B$ is the current memory state, $R \in Op$ is the current control state (residual program, i.e. operator describing a part of a function that has not been yet executed [5]), $F \in (P \rightarrow Op)^*$ is a function call stack, i.e. a sequence of function identifiers and operators describing control state of a given function. The initial state for the given program is $s_0 = (b_0, Op(P_m), (P_m \rightarrow \emptyset))$, where P_m is the main function (entry point of the program). The transition relation is described by the following transition rules:

- 1) $(b, yR, F) \rightarrow (y(b), R, F)$, where $y \in Op$ is a basic operator;
- 2) $(b, if(u, P, Q)R, F) \rightarrow \begin{cases} (b, PR, F), & u(b) = 1, \\ (b, QR, F), & u(b) = 0; \end{cases}$
- 3) $(b, while(u, P)R, F) \rightarrow \begin{cases} (b, Pwhile(u, P)R, F), & u(b) = 1, \\ (b, R, F), & u(b) = 0; \end{cases}$
- 4) $(b, call(P_j)R, (... , P_i \rightarrow \emptyset)) \rightarrow (b, Op(P_j), (... , P_i \rightarrow R, P_j \rightarrow \emptyset))$;
- 5) $(b, \varepsilon, (... , P_i \rightarrow R, P_j \rightarrow \emptyset)) \rightarrow (b, R, (... , P_i \rightarrow \emptyset))$.

Rule 1 defines the execution of basic operators. Rules 2 and 3 describe branching and loop operators. The execution of a function is described by rules 4 (function call) and 5 (return from a function).

Final states (from which no transition is possible) are $s_f = (b, \varepsilon, \emptyset)$.

The described transition system for sequential programs will be denoted as S^{seq} . Notice that this system is deterministic, as for each state there is an unambiguously determined transition.

5.4.3. Parallel program execution model. Based on the constructed sequential program model, we describe a similar model for the parallel programs executed on GPUs. In this model, we consider separately execution of CPU and GPU programs and model each of these cases as transition systems. Then the model of the entire program S^{gc} is built as a union of CPU and GPU models.

GPU program model: block level. The GPU code execution model S^{gpu} is built as a multilevel transition system [5]. The states of the multilevel system combine multiple lower-level states, which are themselves represented as a transition system. At the lowest level we model execution of separate threads using modified sequential system S^{seq} . Transitions of separate thread models are unified as a transition of higher-level transition system.

Our model S^{gpu} contains three levels of state hierarchy: a thread level, a block level and a grid level. Separate threads (the first level) are united in blocks (the second level), which in turn form the grid that describes GPU program as a whole (the third level).

The important feature of the model S^{gpu} compared to the sequential model S^{seq} is a memory hierarchy. In CUDA there are five kinds of memory: the registers, shared memory, constants cache, texture cache and global memory. In our model, we consider only the two most frequently used kinds of memory, namely, shared memory and global memory.

Various kinds of memory in the model can be represented as additional components of a state. For the lowest level (threads) the state is $s = (b_g, b_s, R, F)$, where $b_g \in B_g$ is a global memory state; $b_s \in B_s$ is a shared memory state. However, to simplify the model, we combine all kinds of memory into a single information environment $b = b_g \cup b_s$, $b \in B = B_g \times B_s$. We also assume that all GA operators and predicates operate over the combined set B .

Thread level transition relation is described by rules 1–5 (rule 4 has an additional restriction — the called function P_j should work on GPU, which is described by `__device__` modifier in C for CUDA).

Block level state is the set of threads with their states: $s^2 = \{T_i \rightarrow s_i^1\}$. A number of threads per block is fixed and determined by thread call parameters. Block level transitions are combinations of thread level transitions. This procedure is performed as follows: some subset of threads is selected; the transition is performed for each of the selected threads according to the thread level transition relation; a new state of the block is obtained as a combination of new states of individual threads. In this new state, the control states of individual threads are determined by the transition relation for S^{seq} . However, we need to merge individual memory states, as all threads operate over common memory. Therefore we use the function $merge: B \times B^* \rightarrow B$ that changes the value of each variable which has been changed in at least one of the threads. This function is defined as follows:

$$merge(b_0, b_1, \dots, b_k) = \left\{ v_i \rightarrow \begin{cases} !\{b_j(v_i) \mid b_j(v_i) \neq b_0(v_i), j = \overline{1, k}\} \\ b_0(v_i) \end{cases} \right\},$$

where $!A$ denotes an arbitrary element from the set A .

The *merge* function is applied to both kinds of memory b_g, b_s (or to the combined memory b).

There is an important restriction on block level transitions: all threads that perform transitions simultaneously should execute the same operator (although over different data). These models the hardware restriction of current NVIDIA devices, namely, that there is only one instruction unit per block, and so only threads that execute the same instruction can execute simultaneously [130].

For thread synchronization, we add the `_Barrier` operator. This operator is used to synchronize threads within the block: each thread that reaches this operator waits for all other threads. To model

execution of the `_Barrier` operator we add the following transition rules:

- 6) $(b, _Barrier; R, F) \rightarrow (inc(b), waiting_Barrier; R, F);$
- 7) $(b, waiting_Barrier; R, F) \rightarrow (zero(b), R, F),$
if $bc = threads.$

We use the additional variable $bc \in b_s$, which describes a number of waiting threads. Two operators $inc(b): bc \leftarrow bc + 1$ and $zero(b): bc \leftarrow 0$ increase the number of waiting threads and clear the waiting list. In C for CUDA the `_Barrier` operator is implemented as `__syncthreads()` primitive.

GPU program model: grid level. The grid level model is constructed from multiple block level models in the same way as the block level model is constructed from thread level models.

Grid level states are collections of blocks with their states: $s^3 = \{B_j \rightarrow s_j^2\}$. The number of blocks per grid is fixed and determined by thread call parameters, in the same way as in block level.

Grid level transitions are combined from block level transitions in the same way as for block level. One difference from the block level model is that shared memory is specific for each block, therefore *merge* function is only used for global memory b_g .

CUDA platform provides some synchronization facilities that work at the grid level (such as atomic operations). However, such facilities are not supported by all devices, have a negative effect on program performance and contradict CUDA best design practices that suggest independent execution of blocks [130]. Therefore we do not include these facilities in our model.

CPU program model. To model CPU code execution, we use the extended sequential model S^{seq} . We need to add the means of interaction with GPU to this model. Consider the following new operators:

- *init_gpu* is the GPU initialization;

- $copy_gpu(m_G, m_C)$ is copying the data into graphical memory;
- $copy_back(m_C, m_G)$ is copying of the results from graphical memory;
- $call_gpu(P_i, block, grid)$ is a call of GPU function (CUDA kernel).

Interaction of CPU and GPU programs is achieved from CPU code using the operators $copy_gpu$, $copy_back$ and $call_gpu$ (the operator $init_gpu$ is used once at the beginning of the program, and does not influence the further execution). The first two operators copy the data between CPU memory and GPU global memory. Formally we describe this as a mapping between certain subsets of CPU memory b_c and global memory b_g .

The $call_gpu$ operator actually starts the GPU code execution. Its work is described by two transition rules:

- 8) $(s^c, (b_g, \emptyset)) \rightarrow (s^c, s_0^g(P_i, block, grid));$
- 9) $(s^c, s_f^g) \rightarrow ((b, R, F), (b'_g, \emptyset)).$

Here we use the designation $s^c = (b, call_gpu(P_i, block, grid)R, F)$ for a current state of CPU model. Rule 8 describes the creation of the initial state of the model S^{gpu} , when the operator $call_gpu$ is executed. The parameters of this state, such as a number of blocks and threads and the kernel to execute, are taken from operator parameters. Notice that this rule does not change CPU control state: from the CPU point of view, the program execution is suspended during execution of GPU kernel. Rule 9 is applied when GPU computations are completed (i.e. GPU model achieves its final state s_f^g). It clears GPU control state and completes execution of the $call_gpu$ operator in CPU model. The state of CPU memory is not changed during GPU computations: results should be explicitly copied using the $copy_back$ operator.

All the components of the S^{gc} model are shown in Fig. 5.3.

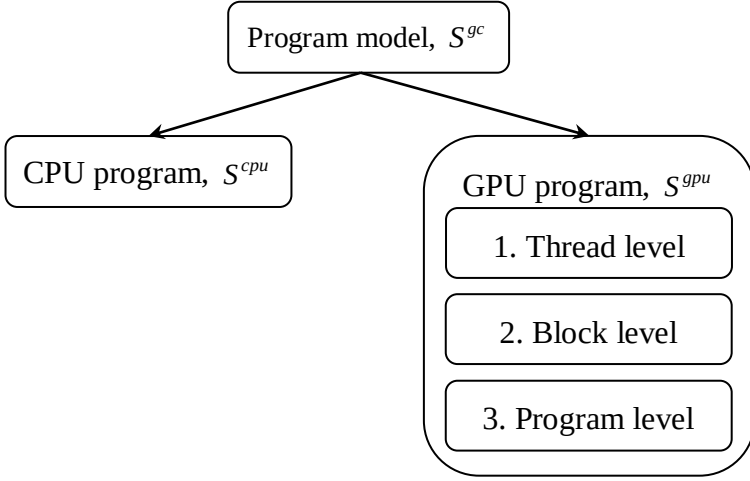


Fig. 5.3. The structure of the GPU program model

Applications of algebra-dynamic models. The described algebra-dynamic models of program execution can be used to perform a formal analysis of programs for GPUs. In particular, in [46] authors utilize algebra-dynamic models to prove program transformation correctness for multithreaded programs. Certain program properties (equivalence, absence of conflicts, and absence of deadlocks) can be formulated in terms of the developed models. Then, the correctness of transformation is proved for programs having such properties. The same approach is applicable to GPU.

One more possible application of the algebra-dynamic models of program execution is the estimation of program execution time. In [49], the general approach to estimation of complexity of an algorithm represented as Glushkov algebra expression is described. However, such estimation cannot take into account peculiarities of hardware that executes an algorithm (e.g. restriction on simultaneous execution of different instructions described in Subsection 5.4.3). On

the other hand, the considered algebra-dynamic models provide an accurate description of program execution, including software and hardware implementation details. Therefore they can be used to obtain more accurate estimates of program execution time.

Further, yet another application of algebra-dynamic models is described, namely, constructing program transformations that can increase performance by parallelizing and optimizing programs.

5.4.4. Program transformations based on rewriting rules.

To automate program transformations, we use TermWare rewriting rules system [38, 154] (see also Subsections 3.2.7, 3.3.7).

5.4.4.1. The transition from a sequential program to GPU program. Rewriting rules are used to automate program transformations, such as a transition from a sequential program for CPU to a parallel program executing on GPU. For this purpose, a sequential program is represented as a high-level model described in Subsection 5.4.2. The code $code_i$ of each component is modeled as an expression of Glushkov algebra which has a natural term representation. These terms are then transformed with TermWare rewriting rules.

Consider the parallelizing transformations for loop expressions. Let the fragment of the initial sequential program be the following:

$$Seq1 = for(i, 0, m, body(i)), \quad (5.2)$$

where the loop operator $for(var, min, max, body)$ with a counter is used, which can be expressed using the general *while* operator. The compound operator $body(i)$ describes the loop body. Consider the transformation of $Seq1$ to the parallel equivalent:

$$\begin{aligned} Gpu1 = & init_gpu; \\ & copy_gpu; \\ & call_gpu(gbody1, block1d, grid1d); \\ & copy_back, \end{aligned} \quad (5.3)$$

where GPU interaction operators described earlier are used. The loop body $body(i)$ is moved to the new GPU component $gbody1$:

$$\begin{aligned} gbody1 = & assign(i, _GetCoor(x)); \\ & _CpuToGpu(body(i)) \end{aligned} \quad (5.4)$$

The first operator computes the iteration number i from thread parameters using the $_GetCoor(x)$ function. Then, the loop body is executed with $_CpuToGpu$ used for transformation between CPU and GPU operators.

The transformation from the sequential program $Seq1$ to the parallel version for graphics processing units $Gpu1$ is described by the following rewriting rules:

- 1) $for(\$iter, 0, \$itlm, \$body) \rightarrow$
 $[init_gpu; copy_gpu;$
 $call_gpu(gbody1, block1d, grid1d(\$itlm)); copy_back]$
 $[_AddMethod(gbody1, _CreateKernel1d(\$iter, \$body))]$
- 2) $_CreateKernel1d(\$iter, \$body) \rightarrow$
 $assign(\$iter, _GetCoor(x)); _CpuToGpu(\$body)$
- 3) $grid1d(\$itlm) \rightarrow (\$itlm + block1d - 1) / block1d$

Rule 1 describes the transition of the fragment of the program from $Seq1$ to $Gpu1$. Notice that the rule contains the action which creates a new component of the program. Rule 2 generates the body of the new component. Rule 3 sets the size of a grid for kernel call based on the iteration count of the original loop.

Notice that the rewriting rules which specify the transition are simple enough and follow directly from algebraic equalities (5.2–4.4). This is possible because we use high-level algebraic program models. Similar transformations were implemented in [47] for low-level program model (a parse tree), however, they used more rules and were less comprehensible.

5.4.4.2. GPU program optimization. Rewriting rules can be also used to automate optimizing transformations. In this case, they are applied to the models of parallel GPU programs, either obtained manually or as a result of prior parallelizing transformations.

As an example of optimizing transformation, consider the transition from a global to a shared memory that improves memory access efficiency [130]. This transformation only affects GPU components of a program (kernels). The kernel *gbody1* (5.4) is transformed as follows:

```

gbody1.1 = assign(i, _GetCoor(x));
           copy_shared(i);
           _Barrier;
           _GlobalToShared(gbody(i));
           _Barrier;
           copy_global(i),

```

where the operator $gbody(i) = _CpuToGpu(body(i))$ is used to denote the computations performed in each GPU thread. The transformed kernel uses two operators $copy_shared(i)$ and $copy_global(i)$ to copy data between global and shared memory. These operators are similar to $copy_gpu$ and $copy_back$ operators that copy data between CPU and GPU memory. The difference is that $copy_gpu$ and $copy_back$ operators copy all data at once, whereas in $copy_shared(i)$ and $copy_global(i)$ each thread copies a part of data. Therefore it is necessary to synchronize threads by using the $_Barrier$ operator. The transformation also uses the function $_GlobalToShared$ for the transition from the operators working above global memory B_g to operators above shared memory B_s .

The rules implementing the transformation are similar to the rules used for the transformation of the CPU program *Seq1* to the GPU program *Gpu1*.

5.4.4.3. The transition between high-level and low-level program models. As mentioned before, high-level program models allow for a more brief and expressive representation of program transformations. However, there is a need for means for transition between program model and source code. When low-level program models (parse trees) are used, such transition is carried out using a parser and a code generator for a given programming language. Such approach has been used, for example, in [47]. However, to construct high-level models, additional knowledge of a subject domain is needed. Such knowledge can be expressed in the form of basic predicates and operators of Glushkov algebra. For the transition between source code and its high-level algebraic model, we use a special kind of rewriting rules called patterns, described in detail in Subsection 6.5.

An important feature of high-level models is language independence. A single high-level program model can be transformed into low-level models for multiple languages. For example, consider an implementation of the operator *init_gpu* in two languages, C and C#. In the first case, this operator is represented as a function call:

$$t_g^c = \text{FunctionCall}(\text{InitCUDA}, \text{NIL}),$$

which produces the code fragment `InitCUDA()`.

For C# the same operator results in the creation of an object:

$$t_g^{cs} = \text{DeclarationAssignment}(\text{cuda}, \text{CUDA}, \text{New}(\text{CUDA}, [0, \text{true}])).$$

The corresponding code fragment is `CUDA cuda = new CUDA(0, true)`.

Therefore, high-level models allow describing program transformations independently of implementation language.

Thus, in this subsection we presented the formal facilities for development of efficient parallel programs for graphics processing units. Rewriting rules enable automated program parallelization and optimization. Using high-level program models allows describing program transformations in a more concise and comprehensible fashion.

ion, and also enables using a single model to describe programs in different languages. Examples using the proposed approach are given in Subsection 7.3.

Control questions and exercises

1. Give the definitions of convolution, involution, reorientation, reinterpretation and transformation metarules. What are metarules intended for?
2. By means of the reinterpretation metarule, make the transition from the sort algorithm *Solute* (see Subsection 2.4, Example 2.8)

$$\begin{aligned} Solute = \{[d(Y_1, K)] ([l > r \mid Y_1] Transp(l, r \mid Y_1) * \\ * SET(Y_1, H), R(Y_1))\} \end{aligned}$$

to more efficient (in terms of execution time) shuttle sort algorithm *Shuttle* by replacing the basic operator $SET(Y_1, H)$ with the operator $L(Y_1)$ which shifts the pointer Y_1 by one element to the right. Demonstrate the process of sorting on array $M : H \ Y_1 \ 5 \ 2 \ 3 \ 1 \ 4 \ K$.

3. Demonstrate the use of reinterpretation metarule for a transition from the sort algorithm *Bubble* (see Subsection 2.1, Example 2.3):

$$\begin{aligned} Bubble = \{[Sorted(M)] \{[d(Y_1, K)] ([l > r \mid Y_1] \\ Transp(l, r \mid Y_1), E) * R(Y_1)\} * SET(Y_1, H)\} \end{aligned}$$

to its modification *Bubble2* more efficient in terms of execution time. For this purpose, include the additional pointer Y_2 to the array, so that $M : H \ Y_1 \ a_1 \ a_2 \ \dots \ a_n \ Y_2 \ K$. The modified scheme *Bubble2* is the following:

$$\begin{aligned} \text{Bubble2} = \{ & [\text{Sorted}(H, Y_2, M)] \{ [d(Y_1, Y_2)] ([l > r \mid Y_1] \\ & \text{Transp}(l, r \mid Y_1), E) * R(Y_1) \} * \text{SET}(Y_1, H) * L(Y_2) \}, \end{aligned}$$

where $\text{Sorted}(H, Y_2, M)$ is the predicate which takes the true value, if the fragment of the array M between H and Y_2 is sorted, and false otherwise; $d(Y_1, Y_2)$ is the predicate which takes the true value, if Y_1 reached Y_2 , and false otherwise; $L(Y_2)$ is the shift of the pointer Y_1 by one element to the left.

Demonstrate the process of sorting on the array $M : H \ Y_1 \ 5 \ 2 \ 3 \ 1 \ 4 \ K$.

4. Describe the main objects of the algorithmic language SAA/1. What is the difference between SAA schemes and regular schemes?
5. Build the SAA scheme of the sort algorithm *Bubble2* described in exercise 3.
6. Build the SAA scheme of the parallel sort algorithm *PBubble* considered in Subsection 2.3, Example 2.6.
7. Describe the abstract automaton model of the parameter-driven generator of texts.
8. Give the definition of a hyperscheme.
9. Give the definition of the algebra of hyperschemes. What is the difference between SAA-M and modified AHS?
10. Formulate the definition of structured design grammar. What are matrix SDGs?
11. Describe the process of designing hyperschemes representing derivation control algorithms in context-free SDGs.
12. Let $PS(n)$ be a parallel regular scheme which processes the data sequence $D : H \ d_1 \ d_2 \ \dots \ d_n \ K$, where H and K are markers fixing the beginning and the end of the sequence; d_i is data element. The scheme $PS(n)$ is the following:

$$\begin{aligned}
PS(n) = & (A(1) * CP(PROC_FIN(1)) // A(2) * \\
& * CP(PROC_FIN(2)) // A(n) * CP(PROC_FIN(n))) * \\
& * S(PROC_FIN),
\end{aligned}$$

where $A(i)$ is the operator processing the element d_i , $i = 1, \dots, n$; $CP(PROC_FIN(i))$ is the control point fixing the moment of computation completion in i -th thread; $S(PROC_FIN)$ is the synchronizer waiting for completion of computation in all threads.

Build SDG intended for designing the class of parallel regular schemes $\{PS(j) | j = 1, \dots, n\}$.

13. By using the design metarules, transform the parallel regular scheme $PS(n)$ from exercise 12 into the sort algorithm, which splits the input numerical array to subarrays, sorts the subarrays in parallel by means of the scheme *Shuttle* (see exercise 2 above) and merges them.
14. Give the definition of a transition system. What is a deterministic transition system? Formulate the definitions of multicomponent and multilevel transition systems.
15. What levels the parallel program execution model on GPU considered in Subsection 5.4 has?
16. What types of memory are used in CUDA? How are they defined in the GPU program execution model?
17. How the synchronization of threads is implemented in the GPU program execution model?
18. How the interaction of CPU and GPU programs is achieved in the GPU program execution model?
19. What are the applications of algebra-dynamic models of program execution?
20. Build rewriting rules which implement the transition from global to shared memory in the GPU program execution model.

Chapter 6

Software tools for design and synthesis of programs

The algebra-algorithmic approach is supported by tools developed within the framework of Kyiv algebraic-cybernetic school. The first of them was the system called MULTIPROCESSIST [112] developed in the 1980s in the programming automation department of V. M. Glushkov Institute of Cybernetics. The system provided the multilevel design of algorithms and data structures represented in the form of SAA schemes and synthesis of corresponding programs. The algebraic approach was also used at development of the system for transformation of algorithm schemes [138]. The method of controlling spaces based on the algebraic approach was used in the parallel programming technology PARUS [7].

In this chapter, we consider the developed tools for programming automation, which continue the aforementioned research, namely, the Integrated toolkit for design and synthesis of programs (IDS) [5, 11, 43, 67, 174] and the term rewriting system TermWare [38, 69, 154] (see also Subsections 3.2.7 and 3.3.7). IDS uses algebraic specifications, based on Glushkov algorithmic algebra (see Subsection 2.3) and represented in three forms: algebraic (formal language), natural-linguistic and graphical. The toolkit is based on the method of dialogue design of syntactically correct algorithm specifications (DSC-method) [5, 159], which eliminates syntax errors during the construction of algorithm schemes. Specialization to a subject domain is done by describing basic operators and predicates from this domain. Unlike the mentioned MULTIPROCESSIST system, which is focused on parsing SAA scheme (on the basis of which a program is generated), IDS excludes the possibility of occurrence of syntax error during the design of an algorithm. Program transformations, such as from sequential to parallel algorithm, are implemented as rewriting rules. For automation of transformations of algorithms and programs, IDS is applied together with TermWare.

6.1. The architecture of the integrated toolkit

The developed integrated toolkit [67] for design and synthesis of algorithms and programs consists of the following basic components (Fig. 6.1):

- the *DSC-constructor* intended for dialogue design of syntactically correct schemes of algorithms represented in SAA-M and synthesis of programs in C, C++, Java languages;
- the *flowgraph editor*, which is applied for editing a graphical representation of an algorithm;
- the *generator of SAA schemes* on the basis of higher-level schemes (hyperschemes);
- the *database* containing the description of SAA-M constructs, basic predicates and operators, and also their program implementations.

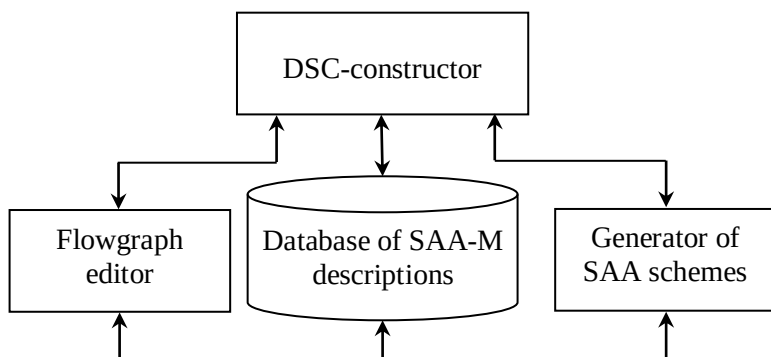


Fig. 6.1. The architecture of IDS toolkit

As it was already mentioned, the developed toolkit uses three forms of algorithm representation: analytic (a formula in the algebra of algorithms), natural-linguistic (SAA scheme) and a flowgraph (based on Kaluzhnin algebra).

The analytic representation is based on Glushkov algorithmic algebra and is a compact notation of an algorithm, focused on its further transformation, e.g. minimization, improvement by various criteria, based on usage of metarules. The natural-linguistic represen-

tation (an SAA scheme) is based on the algorithmic language SAA/1. The flowgraph representation is based on the algebra of flowgraphs.

The combination and interrelation of the analytical, natural-linguistic and visualized forms of representation of algorithmic schemes in IDS gives a comprehensive understanding of specifications and facilitates the achievement of demanded program quality. Algorithm projects created in the toolkit are invariant to a chosen programming language and can be considered as documentation for a developed software product and allow to generate programs in various programming languages.

Example 6.1. Let us illustrate the mentioned representation forms on the sequential sort algorithm *Solute* [159]. Let the input array to be sorted has the following marking:

$$M : H \ Y_1 \ a_1 \ a_2 \ \dots \ a_n \ K,$$

where H and K are the markers fixing the beginning and the end of the array M ; Y_1 is the pointer moving over the array during the processing.

The regular scheme (analytic representation) of an algorithm is the following:

$$\begin{aligned} Solute = & Start * SET(Y_1, H) * \\ & * \{ [d(Y_1, K)] ([l > r | Y_1] Transp(l, r | Y_1) * SET(Y_1, H), R(Y_1)) \} * Fin, \end{aligned}$$

where $Start$ is an initialization operator; $d(Y_1, K)$ is the predicate, which takes the true value, if the pointer Y_1 reached the marker K , and false otherwise; $l > r | Y_1$ is the predicate, which takes the true value, if the specified relation holds for the elements, located to the left and to the right of the pointer Y_1 ; $Transp(l, r | Y_1)$ is the operator of transposition of the elements adjacent to the pointer Y_1 ; $SET(Y_1, H)$ is the operator placing the pointer Y_1 in a position directly to the right of the marker H ; $R(Y_1)$ is the shift of the pointer

Y_1 over the array M by one symbol to the right; Fin is the final operator which outputs the sorted array M .

According to the scheme of the algorithm, the first unordered pair of array elements is searched with the help of the pointer Y_1 scanning in the direction from left to right. Then, after the transposition, the pointer Y_1 is placed at the beginning of the array and scanning is repeated. The process finishes when Y_1 reaches the marker K . The flowgraph of the *Solute* algorithm is given in Fig. 6.2.

The natural-linguistic representation of the algorithm is the following:

```

SCHEME SOLUTE SORT ====
  "Sequential Solute sort"
  END OF COMMENTS

  "Solute"
  ==== "START"
    THEN
      "Place the pointer Y(1) at the beginning of
        the array (M)"
    THEN
      WHILE NOT 'The pointer Y(1) is at the end of
        the array (M)'
        LOOP
          IF 'The elements l > r at pointer Y(1) in
            the array (M)'
            THEN
              "Transpose the elements l, r at pointer Y(1) in
                the array (M)"
            THEN
              "Place the pointer Y(1) at the beginning of
                the array (M)"
            ELSE
              "Shift the pointer Y(1) by (1) element in
                the array (M) to the right"
            END IF
          END IF
        END IF
      END IF
    END IF
  END IF

```

```

THEN
  "Shift the pointer Y(1) by (1) element in
    the array (M) to the right"
END OF LOOP
THEN
  "FIN"
END OF SCHEME SOLUTE SORT

```

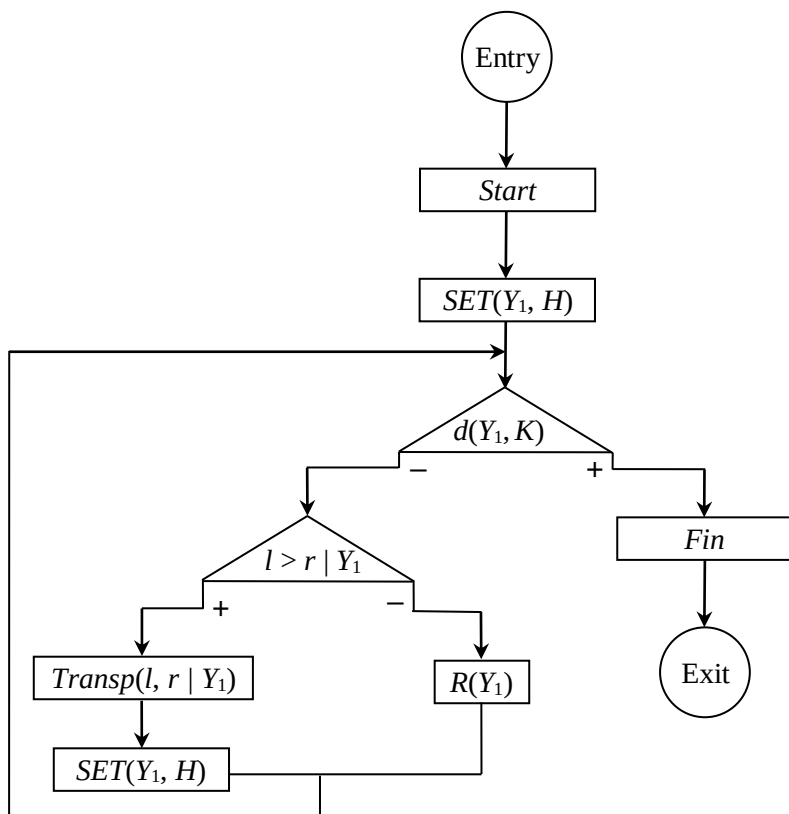


Fig. 6.2. The flowgraph of sort algorithm *Solute*

The alternative version of IDS called Synthesis was also developed [68]. The main difference between these tools is that the

Synthesis framework is based on using Web technologies and is suited for object-oriented programming, whereas the IDS toolkit is mainly aimed at imperative paradigm.

6.2. *Dialogue design of algorithm schemes*

One of the components of the IDS toolkit is the DSC-constructor intended for automated design and generation of schemes of algorithms and programs. The constructor is based on the method of dialogue design of syntactically correct programs (DSC-method) which is based on the interconnection of algebraic and grammatical descriptions of syntax of formal languages [159].

The basic idea of the DSC-constructor consists in level-by-level top-down designing of schemes by detailing language constructs of SAA-M. On each step of the design, the system allows a user to select only those constructs, the substitution of which into a scheme does not break syntactic correctness of an algorithm. The DSC-constructor uses the list of SAA-M constructs and an algorithm design tree. The list consists of logic and operator operations, the superposition of which allows to create algorithms in three forms mentioned earlier. The specifications of constructs are stored in the database of the toolkit. During the design of an algorithm, the SAA-M operations chosen by a user are displayed in the tree with further detailing of their variables. Depending on a type of the chosen variable, the system offers the corresponding list of SAA-M operations or basic concepts from the database. Note the level-by-level style of algorithm design and the possibility of transitions to various levels (tree nodes) during the design process.

The design of SAA schemes is carried out with the help of the algorithm design tree, which we will denote as T . The process of construction of the tree is represented by the following regular scheme:

$$\begin{aligned}
 DSC_process(T) = & INIT(T) * \\
 & * \{ [DSC_completed(T)] \text{ SelectNoninterpretedVariable}(v_n, T) * \\
 & \quad * \text{ SelectNoninterpretedConstruct}(c_n, v_n, T) * \\
 & \quad * \text{ ProcessNode}(c_n, v_n, T) * \text{ GenerateSchemeText}(T) \},
 \end{aligned}$$

where $INIT(T)$ is creation of the root node of the tree T with the name of the first compound operator, and also creation of its child node marked with a name of an operator variable; $DSC_completed(T)$ is a condition, which takes a true value, if all variables of the designed scheme are interpreted; $SelectNoninterpretedVariable(v_n, T)$ is the operator of selection of the node v_n of the tree T with necessary notinterpreted variable; $SelectNoninterpretedConstruct(c_n, v_n, T)$ is a selection of a necessary construct c_n (an operation of SAA-M or a name of basic or compound element) for detailing the selected node v_n in the tree T ; $ProcessNode(c_n, v_n, T)$ is the operator, which forms a child node for the current node v_n , marks it with a name of the chosen construct c_n , and also forms its child nodes with names of operator or logic variables of the construct. During the DSC designing, the operations $SelectNoninterpretedVariable(v_n, T)$ and $SelectNoninterpreted Construct(c_n, v_n, T)$ are carried out by a user. On the basis of the obtained current tree T , the operator $GenerateSchemeText(T)$ generates the text of an algorithm scheme.

The scheme $DSC_process(T)$ is implemented in the toolkit as follows. In the beginning, the algorithm design tree T contains the root node with the name of a first compound operator of a scheme, and also one child node marked with the name of an operator variable. After that, the user selects one of the operator structures from the list and thus defines the main construct of the scheme. The child node of the root node is marked with the text of this construct. Then its child nodes are added, which contain the names of variables included in this construct. Further the user chooses necessary operations for variables, basic or compound operators and predicates.

The SAA scheme corresponding to an algorithm design tree is displayed in a separate text window in an analytic or a natural-linguistic form depending on the options set. It is possible to add a

compound operator or a compound predicate to a scheme being designed. They are represented as an additional design tree.

During the work with the DSC-constructor, a user can edit the description of operator and predicate language constructs, and also basic operators and conditions which are stored in the database of the toolkit. The description of an element (an operation of SAA-M or a basic concept) in the database includes its representation in analytic and natural-linguistic form, and also an implementation in a target programming language.

Fig. 6.3 shows a screenshot of the DSC-constructor window with an SAA scheme of the Solute sort algorithm, which was considered in Example 6.1.

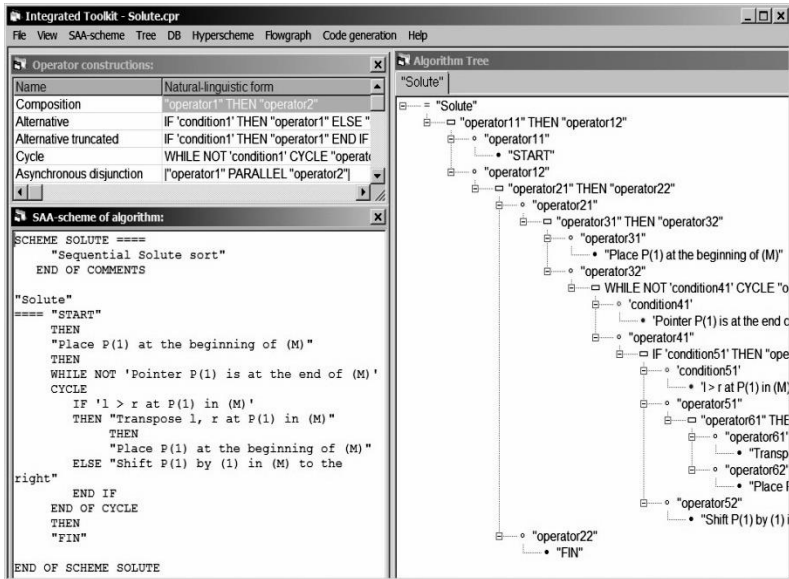


Fig. 6.3. The main window of the DSC-constructor

The window contains three subwindows: the left upper subwindow includes a list of SAA operations, the subwindow on the right side contains the algorithm design tree, and the third subwin-

dow shows the text of an SAA scheme. A flowgraph of an algorithm can be viewed and edited separately in a flowgraph editor.

Example 6.2. We will illustrate the process of designing of the SAA scheme of the Solute sort algorithm by means of the DSC-constructor. In the beginning, the user enters the name of the SAA scheme, a comment to it and the name of the first compound operator of the scheme (“Solute”). In the algorithm design tree, the root node with the name of the compound operator and a child node “operator1” (Fig. 6.4) is displayed.

In the window of a text representation of the SAA scheme its template is displayed:

```
SCHEME SOLUTE ====
  “Sequential Solute sort”
  END OF COMMENTS

“Solute”
===== “operator1”

END OF SCHEME SOLUTE
```

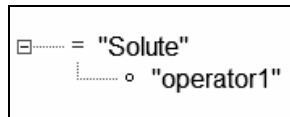


Fig. 6.4. The initial design tree of the sort algorithm “Solute”

Further from the list of operator constructs of SAA-M, displayed in a separate window, user has to choose a necessary construct for detailing the variable “operator1”. In this case, it is the composition of operators “operator11” THEN “operator12”. As a result of including this construct in the tree, the node “operator1” will be marked by the text of this construct, and two its child nodes will be marked with the text of variables included in the construct (Fig. 6.5).

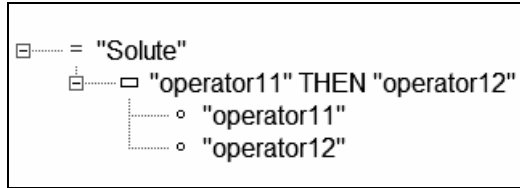


Fig. 6.5. The algorithm design tree after inclusion of the operation of composition of operators

The natural-linguistic representation of the SAA scheme is the following:

```

SCHEME SOLUTE =====
  "Sequential Solute sort"
  END OF COMMENTS

  "Solute"
  ===== "operator11"
            THEN
            "operator12"
  END OF SCHEME SOLUTE
  
```

Further, it is necessary to choose the basic operator "START" which initializes the data for the variable "operator11". As a result, a node with the text of the mentioned operator is added to the algorithm design tree (Fig. 6.6).

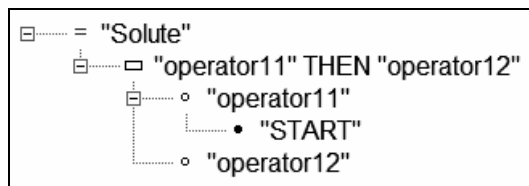


Fig. 6.6. The algorithm design tree after inclusion of the basic operator "START"

The text of the SAA scheme is the following:

```
SCHEME SOLUTE ====
  "Sequential Solute sort"
  END OF COMMENTS
```

```
"Solute"
===== "START"
        THEN
        "operator12"
```

```
END OF SCHEME SOLUTE
```

Further, the composition operation has to be chosen for detailing the variable "operator12". Then, for the first operand of the composition, the user has to select the basic operator "Place P(1) at the beginning of (D)", which places the pointer P with the number k at the beginning of the array D. The value of the parameter k has to be set to 1, and the value of the parameter D has to be set to M. The result of the mentioned actions is shown in Fig. 6.7.

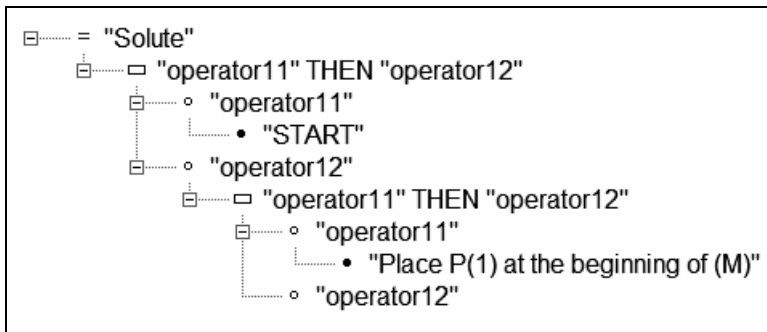


Fig. 6.7. The algorithm design tree after inclusion of the second composition operation and the basic operator of placing a pointer

The natural-linguistic representation of the SAA scheme is the following:

```
SCHEME SOLUTE ====
    "Sequential Solute sort"
    END OF COMMENTS

    "Solute"
    ==== "START"
        THEN
            "Place P(1) at the beginning of (M)"
            THEN
                "operator12"

    END OF SCHEME SOLUTE
```

After detailing the variable "operator12" by the inclusion of loop, branching and composition operations, and also necessary basic conditions and operators, we will obtain the resulting algorithm design tree shown in Fig. 6.8.

The natural-linguistic representation of the SAA scheme is the following:

```
SCHEME SOLUTE ====
    "Sequential Solute sort"
    END OF COMMENTS

    "SOLUTE"
    ==== "START"
        THEN
            "Place P(1) at the beginning of (M)"
            THEN
                WHILE NOT 'Pointer P(1) is at the end of (M)'
                LOOP
                    IF 'l > r at P(1) in (M)'
                    THEN
                        "Transpose l, r at P(1) in (M)"
```

```

      THEN
      "Place P(1) at the beginning of (M)"
    ELSE "Shift P(1) by (1) in (M) to the right"
    END IF
  END OF LOOP
  THEN
  "FIN"
END OF SCHEME SOLUTE

```

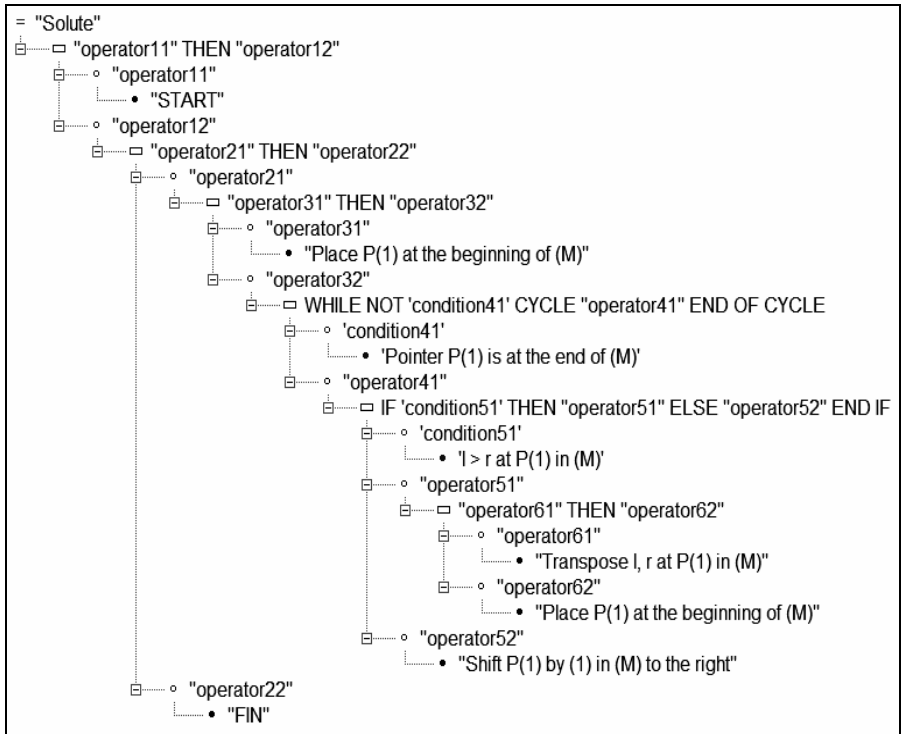


Fig. 6.8. The resulting algorithm design tree

The DSC-constructor can be applied both for designing algorithms and hyperschemes [174].

Schemes of algorithms designed by means of DSC-constructor can be represented in a flowgraph form with the usage of the flowgraph editor of the IDS system. The changes introduced during editing of a flowgraph, are correspondingly displayed on analytic and natural-linguistic representations of an algorithm in DSC-constructor. Fig. 6.9 shows the window of the flowgraph editor with a flowgraph representation of the above-considered Solute sort algorithm.

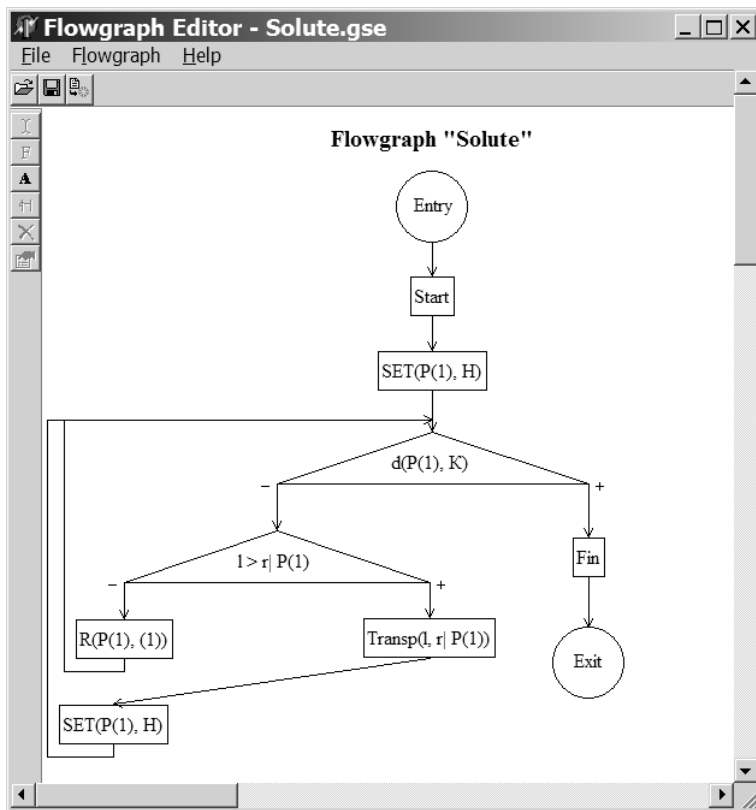


Fig. 6.9. The window of the flowgraph editor with the scheme of the Solute sort algorithm

The flowgraph editor gives a user the possibility to change an appearance of each component of a flowgraph representation, using the movement of flowgraph nodes (operators and recognizers) of a flowgraph with an extension of edges, change node text and color, a thickness and a style of lines, and also the size of arrows.

6.3. Generation of algorithms and programs

On the basis of an algorithm tree obtained as a result of designing, the IDS toolkit performs generation of programs. Besides designing SAA schemes, the toolkit supports also the automated construction of schemes of higher-level named hyperschemes (see Subsection 5.3), which are applied for generation of algorithm schemes. Notice that SAA schemes and hyperschemes have an identical syntax that allows to flexibly use parameters that control generation of schemes and are set at the level of basic operators and conditions. The approach to parameter-driven generation of algorithms considered in Subsection 5.3 was implemented in the component of IDS named the “Generator of SAA schemes”, allowing to interpret basic operators and conditions of hyperschemes. The prototype of the component was the MULTIPROCESSIST system [177]. Unlike the mentioned system, in the developed IDS toolkit designing of hyperschemes and SAA schemes is carried out in the mode of DSC constructing, which provides syntactic correctness of schemes.

Fig. 6.10 shows the sequence of program development in IDS. In the beginning, a hyperscheme is designed in the DSC-constructor. Further, the generator of SAA schemes carries out the generation of an algorithm scheme on the basis of the hyperscheme. Then DSC-constructor carries out code generation in target programming language on the basis of the SAA scheme.

In the following subsections, the process of generation of algorithm schemes and programs is considered in more detail.

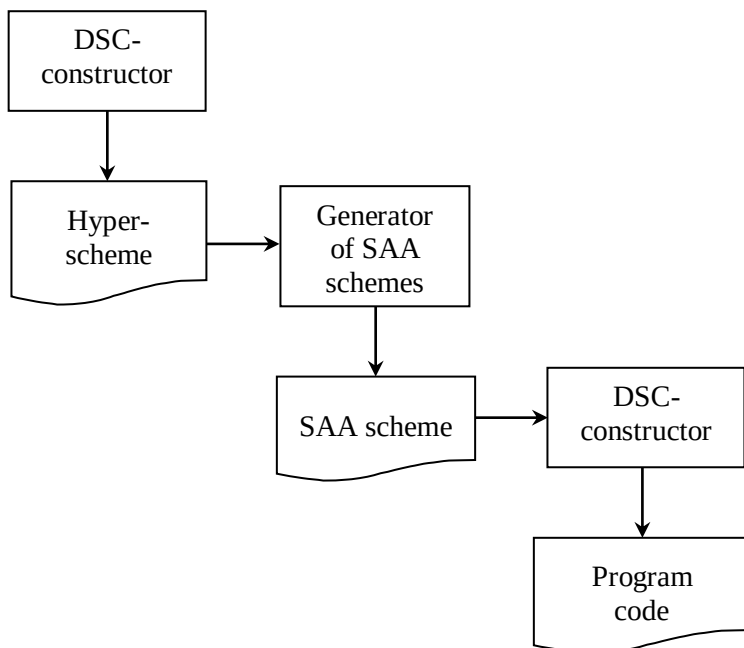


Fig. 6.10. The sequence of development of algorithms and programs in IDS

6.3.1. Generation of regular schemes on the basis of hyperschemes. As it was mentioned in Subsection 6.2, the process of algorithm design in DSC-constructor consists in level-by-level designing of an algorithm scheme by detailing SAA-M constructs and is presented in the form of a tree. Fig. 6.11 shows the window of the DSC-constructor with an example of a hyperscheme. On each step of construction of the hyperscheme, the system allows a user to choose only those constructs, the insertion of which into the scheme does not break its syntactic correctness. The hyperscheme designed DSC-constructor is the basis for generation of an SAA scheme, and the SAA scheme is used to synthesize a program in a chosen target programming language.

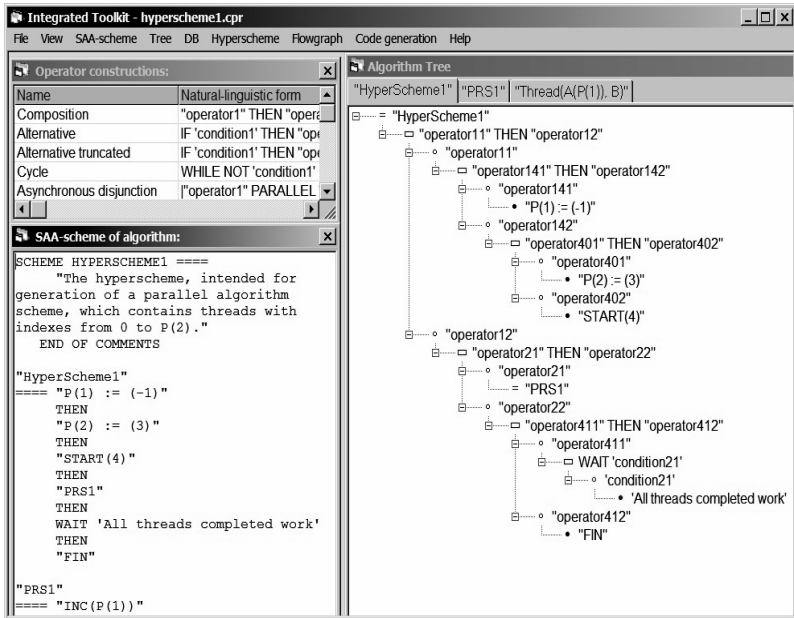


Fig. 6.11. The window of the DSC-constructor with an example of a hyperscheme

Consider the process of generation of schemes on the basis of hyperschemes. Before the beginning of the generation, a user can set the generation options: the name of a scheme to be generated, a comment to it, and also a name of an output file for writing the generated scheme. The hyperscheme is executed by an interpreter in interaction with a parser identifying the constructs of the algebra of hyperschemes (see Subsection 2.3.1) to be executed. The constructs are processed recursively according to the definition of function F . The text of basic operators and conditions of hyperschemes is parametrized. For simplification of processing, the parameters are designated in the text of basic and other elements of hyperschemes in the form of $P(i)$, where i is the number of the parameter. For example, the operator assigning the value n to the parameter $P(1)$ is the following: " $P(1) := (n)$ ". The execution of an elementary operator or

computation of an elementary condition during the generation of algorithm scheme consists in building of its text according to the values of parameters.

6.3.2. Synthesis of programs on the basis of algorithm schemes. On the basis of a tree obtained during the designing of an algorithm scheme, and also implementations of elementary operators and conditions in a target programming language, the DSC-constructor generates a program. During the synthesis, operator constructs of an algorithm scheme (an asynchronous disjunction, a composition, branching, a loop, etc.) are translated to corresponding operators of programming language, and basic operators and predicates are replaced with their implementations in the same language, which are stored in the database of IDS. Compound operators can be represented as subroutines (class methods). The input of a generator is also a file that contains a skeleton description of a basic class of a program (without implementations of methods), in which a substitution of the generated code is carried. Designing of program classes and their interrelations, and also generation of a skeleton program code can be carried out by means of Rational Rose [80].

Table 6.1 gives examples of description of the main operator operations in the database of IDS. Java is used as a target programming language.

Table 6.2 shows the examples of the description of basic elements for sorting algorithms. The natural-linguistic and analytic form of the description of a basic element includes the names of formal parameters indicated in brackets. Formal parameters specified in the text of program implementation of a basic concept are replaced by corresponding actual parameters (which are set in SAA schemes) during a program synthesis.

Implementations of basic elements are written with the usage of the Java class named `Sorting` (see Fig. 6.12), which is a reusable component developed for sorting tasks. This class contains the description of data — the processed array, pointers, markers, control points, and the methods for accessing these data. The formal parameter in the text of Java implementation of basic elements is marked with the symbol `%` followed by a number of parameter in a text of basic operator or predicate.

Table 6.1. The description of operator operations of SAA-M in the database of IDS.

Analytic form	Natural-linguistic form	Implementation in Java
“operator1” * “operator2”	“operator1” THEN “operator2”	<code>^operator1^;</code> <code>^operator2^</code>
([‘condition1’] “operator1”, “operator2”)	IF “condition1” THEN “operator1” ELSE “operator2” END IF	<code>if</code> <code>(^condition1^)</code> <code>{</code> <code> ^operator1^</code> <code>}</code> <code>else {</code> <code> ^operator2^ }</code>
{[‘condition1’] “operator1”}	WHILE NOT ‘condition1’ LOOP “operator1” END OF LOOP	<code>while</code> <code>(!(^condition1</code> <code>^))</code> <code>{</code> <code> ^operator1^</code> <code>}</code>

For example, the basic operator “Transpose l , r at $P(i)$ in (M) ” includes the parameter i , which is the pointer number. The value of this parameter will be substituted instead of a corresponding formal parameter in an implementation of this operator, which looks like `s.transp(%1)`, where `s` is an instance of the `Sorting` class; `transp` is the method of this class carrying out a transposition of elements of the array M , adjacent to the pointer $P(i)$. The implementations of other basic elements given in Table 6.2 are also the calls of methods of the `Sorting` class.

The mapping to a programming language for any element of the database can contain several variants, one of which can be chosen depending on a problem to be solved. For example, the basic operator of initialization (START) has various variants of implementation in target programming language for sorting and search algorithms.

Table 6.2. The examples of description in the database of basic operators and predicates for sorting algorithms.

Type	Natural-linguistic form	Analytic form	Implementation in Java
Operators	“Shift $P(i)$ by (n) in (M) to the right”	$R(P(i), (n))$	<code>s.moveR(%1, %2) ;</code>
	“Transpose l, r at $P(i)$ in (M) ”	$\text{Transp}(l, r P(i))$	<code>s.transp(%1) ;</code>
	“Place $P(i)$ at $P(j)$ in (M) ”	$\text{Place}(P(i), P(j))$	<code>s.place(%1, s.getPointerPos(%2)) ;</code>
Predicates	‘ $l > r$ at $P(i)$ in (M) ’	$l > r P(i)$	<code>s.compare(%1, ">")</code>
	‘Distance between $P(i)$ and $P(j)$ in (M) equal to (n) ’	$d(P(i), P(j)) = (n)$	<code>s.distance(%1, %2) == %3</code>
	‘Pointer $P(i)$ is at the end of (M) ’	$d(P(i), K)$	<code>s.atEnd(%1)</code>

In the DSC-constructor, there is a possibility to specify parameters of generation of a program on the basis of an algorithm scheme. The parameters are divided into two groups: the general parameters and parameters of generation for compound components of a scheme. The general parameters include a name of an input file containing a skeleton program code, the name of resulting generated file and parameters of substitution of the generated code in an input file. The code can be substituted instead of the specified string of symbols in a file (for example, `//<target>`) or in a body of the specified function, defined in an input file. In the generation parameters for compound operators and predicates, it is possible to set a name, access modifiers, formal parameters, type of return value for the generated subroutine (class method) for each compound element.

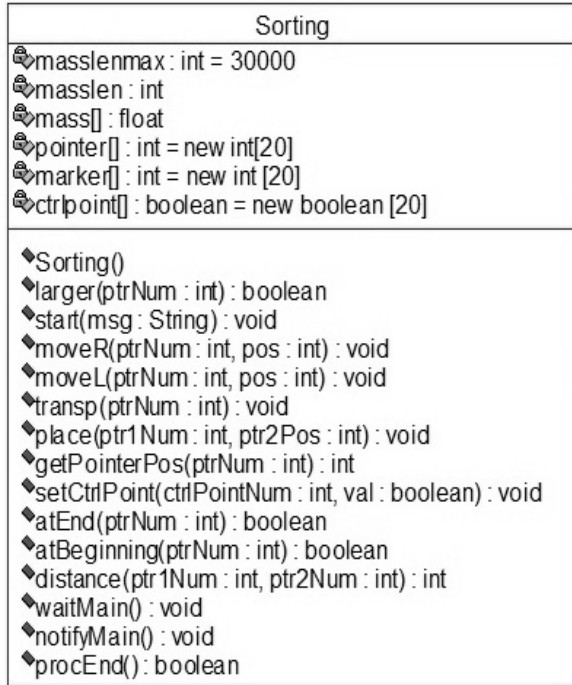


Fig. 6.12. The UML class diagram for `Sorting` class

The current version of IDS provides the generation of sequential and parallel code in Java, C, C++, C for CUDA, Cilk++ programming languages.

Example 6.3. As an illustration, we will consider a process of development of the parallel program of address sort in Java language [45]. For designing and generation of the mentioned program, IDS and Rational Rose are used.

The essence of the address sort consists in computation for each element of an input array M_0 , its index (an address) in an output array M . Thus the computation of an index for each element of an input array is independent. In the considered algorithm, the array M_0 of length n is split into m subarrays $M_0(i)$ ($i=1, \dots, m$), which are processed by m parallel threads ($m \geq 1$). The thread with

the number i computes output indexes of elements of the subarray $M_0(i)$ and inserts them into the array M . The parallel regular scheme $Adrsort(M_0, M, m)$ of address sort is the following:

$$Adrsort(M_0, M, m) = START * (\prod_{i=1}^m Adr_thread(i)) * \\ * S(PROC_FIN) * FIN;$$

$$Adr_thread(i) = CalcBounds(i, m_1, m_2) * (j := m_1) * \\ * \{ [j > m_2] Adr_insert(j) * (j := j + 1) \} * CP(PROC_FIN(i),$$

where $START$ is an operator of data initialization (an input of an array to be sorted); $Adr_thread(i)$ is a thread which implements an insertion of elements of the subarray $M_0(i)$ into output array M ; $CalcBounds(i, m_1, m_2)$ is a calculation of bounds (m_1, m_2) of the subarray $M_0(i)$; $Adr_insert(j)$ is a compound operator computing an index of j -th element of the array M_0 ($j = 1, \dots, m$) in the output array M , and inserting this element into the array M ; $CP(PROC_FIN(i))$ is a control point fixing the moment of completion of processing in i -th thread; $S(PROC_FIN)$ is a synchronizer with a condition of achievement of control points in all m parallel threads; FIN is a final operator (an output of the sorted array).

The essence of the given algorithm consists in parallel functioning of m threads of calculations $Adr_thread(i)$ each inserting the elements of the subarray $M_0(i)$ into the array M by means of the operator $Adr_insert(j)$. The compound operator $Adrsort(M_0, M, m)$ includes the synchronizer $S(PROC_FIN)$ which delays the calculations until all threads finish processing. After the synchronizer completes its execution, we will have the sorted file M .

Fig. 6.13 shows the UML class diagram with the basic classes of the developed application. The `Adrsort` class is the basic

class of the address sort program. The execution of the program begins with the method `main`, in which the input array is initialized, m parallel threads are called and synchronized, and also the result of sorting is outputted. The method `ins` is an implementation of the compound operator $Adr_insert(j)$ of the scheme $Adrsort(M_0, M, m)$. This method is called by parallel threads in the process of sorting. The class `SortThread` inherits the class `Thread` and has a code for i -th parallel thread ($i = 1, \dots, m$). The classes `Adrsort` and `SortThread` use the methods of the `Sorting` class (a reusable component for writing sort programs).

In Rational Rose, on the basis of the class `Adrsort`, a file with a skeleton program code not containing implementations of the methods `main` and `ins` was generated. The further development of algorithms of these methods, and also synthesis and insertion of code implementing these functions, was done in IDS. Generation of the class `SortThread` was completely carried out in IDS.

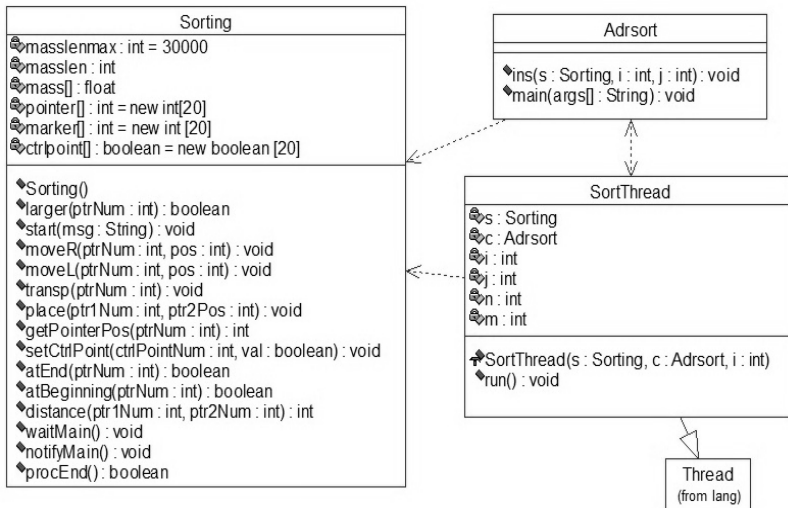


Fig. 6.13. The UML class diagram for parallel address sort

In the process of synthesis of program code, the DSC-constructor uses templates of program implementations of SAA-M constructs and basic concepts from the database. Further, the examples of program implementations of operations intended for the formalization of parallel computations are considered: an asynchronous disjunction, a synchronizer, and a control point. The natural-linguistic form of representation of each of these operations and its mapping to a text template in Java language are given.

The natural-linguistic form of representation of **m-place asynchronous disjunction** $\bigvee_{i=1}^m A$ is

```
PARALLEL(i = 1, ..., m)
( "operator1" )
```

For the address sort problem being considered, the following variant of a template of implementation of this operation in Java language was chosen:

```
int m = s.getThreadNum();
for (int i = 1; i <= m; i++)
{
    SortThread st = new SortThread(s, c,
                                    i);
}

// $classes

class SortThread extends Thread
{
    Sorting s;
    Adrsort c;
    int n, m, i, j;

    /* Class constructor */
    SortThread(Sorting s, Adrsort c,
               int i)
```

```

    {
        this.s = s;
        this.c = c;
        this.i = i;
        n = s.getDataLen();
        m = s.getThreadNum();
        /* Start a thread */
        new Thread(this).start();
    }

    public void run()
    {
        ^operator1^
    }
}

```

The given fragment of program implementation is divided into two parts by the `// $classes` string. The first part describes the loop, in which `m` parallel threads are created (the objects of the `SortThread` class). The second part of the fragment contains the `SortThread` class (inheriting the `Thread` class), in which the template of a thread class (without the implementation of the `run` method) is given. The constructor of the `SortThread` class performs initialization of data and launching a thread by means of the `start` method. The parameters of this constructor are the following: `s` is the object of the class `Sorting`; `c` is the instance of the main class (`Adrsort`) of the program; `i` is the index of a thread being created. The method `run` of the class `SortThread` contains the string `^operator1^` instead of which the implementation of an operand of an asynchronous disjunction is substituted. In the process of the synthesis, in that place of Java program which corresponds to an asynchronous disjunction, the loop of creation of threads will be inserted. The text of the class `SortThread` (with a text of implementation of *i*-th thread) is included after the text of the basic class of the program.

For an implementation of the **synchronizer** operation (the natural-linguistic form of this operation is WAIT 'condition1') in Java language, the following template was chosen:

```
if (! (^condition1^))  
s.waitMain(); // Suspend the main thread
```

Here waitMain is the method of the class Sorting which delays computation in the main thread of the program in case if the condition ^condition1^ is not satisfied. The string ^condition1^ in the process of generation is replaced with the call of the method s.procEnd(), which returns the true value in case if all *m* parallel threads finished computation, and false otherwise. The description of the waitMain in Sorting class is the following:

```
public void waitMain()  
{  
  
    // Set the flag that the main thread  
    // is in waiting state  
    notified = false;  
    synchronised (mainThread)  
    {  
        try { // Suspend the main thread  
            mainThread.wait ();  
        }  
        catch (InterruptedException e) {  
            e.printStackTrace(); }  
    }  
}
```

Here mainThread is the variable of the class Thread, containing the information about the main thread of the program. The thread mainThread in the given fragment is turned into a waiting state by means of a call of a standard method wait().

For an implementation of the **control point** operation (the natural-linguistic form of this operation is CP 'condition1') in Java language, the following variant of a Java template was chosen:

```
^condition1^
if (s.procEnd())
// Resume the main thread
s.notifyMain();
```

In the process of program generation, the string `^condition1^` is replaced with an implementation of the basic predicate *PROC_FIN*(*i*), namely, a call of the method `s.setCtrlPoint(i, true)` of the class *Sorting*. This method assigns a true value to a control point with the index *i*. After the execution of this method, all control points associated with threads are checked by means of the method call `s.procEnd`. After the completion of computation in all threads (the function `s.procEnd` returned the true value), the method `notifyMain` of *Sorting* class resumes the main thread. The description of this method is the following:

```
public void notifyMain()
{
    if (! (notified))
    {
        synchronized (mainThread)
        {
            /* Resume the mainThread */
            mainThread.notify();
            // Set the flag that the main
            // thread was resumed
            notified = true;
        }
    }
}
```

In the given function, the work of the thread `mainThread` is resumed by means of calling a standard method `notify()`.

6.4. The rewriting rules system

To automate the transformation of programs, the above considered IDS toolkit is applied together with the developed symbolic computation system TermWare based on rewriting rules technique [38, 154].

6.4.1. The language of the TermWare system. The TermWare language is based on terms, i.e. expressions of the form $f(x_1, \dots, x_n)$ with variables and data types. Variables (which are written as $\$var$) and constants of certain types of data (numerical, logic, string and atomic — unchangeable strings) are used as atomic terms. For simplification of notation and perception, the simplification reductions for many terms are used, for example, $x + y$ is used for $plus(x, y)$; $[x : y]$ — for $cons(x, y)$ (the list with the first element x and the rest of the list y); $x ? y : z$ — for $ifelse(x, y, z)$. The set of all terms is a term written with the use of these reductions. Table 6.3 shows the basic notations of TermWare terms.

The TermWare rules have the following general form:

$$source [condition] \rightarrow destination [action],$$

where four terms are used: *source* is an input sample; *destination* is a target sample; *condition* is a condition defining the applicability of the rule; *action* is an operation executed when the rule triggers.

Actions being executed and conditions being checked are optional components of a rule, which can call an imperative code. For this purpose the developer should implement the “fact base” — a class implementing the *IFacts* interface and providing methods that can be called from rewriting rules. Thus, the connection between declarative rules in TermWare language and an imperative code in traditional object-oriented languages (such as Java or C#) is established. Besides, it is possible to write custom strategy (in the form of

a class implementing the interface *ITermRewritingStrategy*) defining an order of application of rules.

Table 6.3. The reduced notations of TermWare terms.

Notation	Term
$x \rightarrow y$	<code>rule(x, y)</code>
$x[c] \rightarrow y$	<code>if_rule(x, c, y)</code>
$x[c] \rightarrow y[r]$	<code>if_rule(x, c, action(y, r))</code>
$x[c0] \rightarrow y0 \mid$ $[c1] \rightarrow y1 \mid \dots \mid$ $! \rightarrow z$	<code>if_else_rule(x, c0, y0,</code> <code>[else_rule(c1, y1), ...],</code> <code>fail_rule(z))</code>
$x.y$	<code>apply(x, y)</code>
$[x, y, \dots, z]$	<code>cons(x, cons(y, ..., cons(z, NIL)))</code>
$\{x, y, \dots, z\}$	<code>set(x, y, ..., z)</code>
$x + y, x - y,$ $x * y, x / y$	<code>plus(x, y), minus(x, y),</code> <code>multiply(x, y), divide(x, y)</code>
$x == y, x != y,$ $x > y, x >= y,$ $x < y, x <= y$	<code>eq(x, y), neq(x, y),</code> <code>greater(x, y), greater_eq(x, y),</code> <code>less(x, y), less_eq(x, y)</code>
$x \&\& y, x \parallel y, !x$	<code>logical_and(x, y), logical_or(x, y),</code> <code>logical_not(x)</code>

The TermWare system was initially implemented in the form of Java library intended for embedding in applications [154]. A command-line interface for interactive execution of simple rewriting rules was also developed. Later, the TermWare system was transferred to Microsoft.NET platform, which allowed using rewriting rules technique for transformation of programs written in C# and other languages supported by .NET platform. A parser and a code generator for C# language were developed, which enabled the TermWare system to work with programs represented in the form of source code in this language. The graphical user interface for more convenient creation and application of rewriting rules was also implemented [178]. The developed components constitute the toolkit

intended for transformation of programs on Microsoft.NET platform, which is described in the following subsection.

6.4.2. The structure of the rewriting rules toolkit. The developed software toolkit Termware.NET [69] is intended for automated transformation of programs with the use of rewriting rules technique. The toolkit consists of the following basic components:

- the parser for translating a program code from high-level languages (such as C#, C++, Java) to a model of a program in the form of terms;
- the system of application of rewriting rules for program transformation;
- the generator for translation of a program model in the form of terms to a programming language;
- the graphical user interface for viewing and editing terms and rewriting rules, and also managing other components.

The interaction between the components is schematically presented in Fig. 6.14.

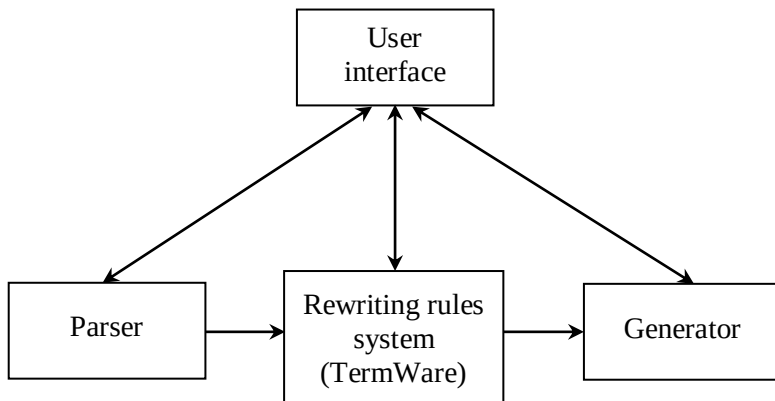


Fig. 6.14. The basic components of the Termware.NET toolkit

The parser carries out a transition from a source text of a program in a high-level programming language (for example, C#) to its model in the form of terms. Similarly, the code generator carries out a transition from a program model to source code in a target lan-

guage. Notice that the given model is low-level and explicitly describes all syntactic constructs of the programming language. For convenience of a user, the toolkit supports a transition to a high-level algebraic model. At present, the toolkit supports parsers and code generators for Java, C# 2.0 and Fortran.

The toolkit provides the possibility of addition of new parsers and generators for other programming languages. For this purpose, it is necessary to provide the classes implementing the interfaces *IParser* and *ICodeGenerator*, which carry out transformation between a text representation of source code in a specified language and a syntactic model (a parse tree), represented in the form of terms.

For automatic application of rewriting rules to specified terms (in particular, program models), the toolkit uses the TermWare library for the Microsoft.NET platform. The toolkit supports loading and saving of systems of rules in TermWare language. Besides, there is a possibility of implementation of facts databases (containing the methods accessible for calling from rewriting rules) or additional rewriting strategies in C# language or in other languages of Microsoft.NET platform.

The graphical interface of the developed toolkit is presented in Fig. 6.15. In the left side of the toolkit window, the treelike representation of a current term (program model) is placed. The right part of the interface supports the editing of rules in text or graphic (tree-like) representation. The graphical interface allows a user to look through and edit terms in visual treelike representation, instead of a text. Standard functions for working with tree elements, such as folding separate subtrees, are supported. Terms are edited by means of the contextual menu supporting addition and removal of nodes, change of contents of nodes, work with a clipboard. The same possibilities are accessible also at editing rules in treelike representation. Thus, the graphical interface simplifies understanding of difficult models of programs and allows working with them without studying specific syntax of the rewriting rules system. At the same time, experienced users can work with terms in a text representation as well. In particular, leaf nodes of the tree can contain not only an atomic symbol, but also a subterm of any complexity represented in the text

form. This allows to quickly add difficult elements of a model, such as arithmetic expressions.

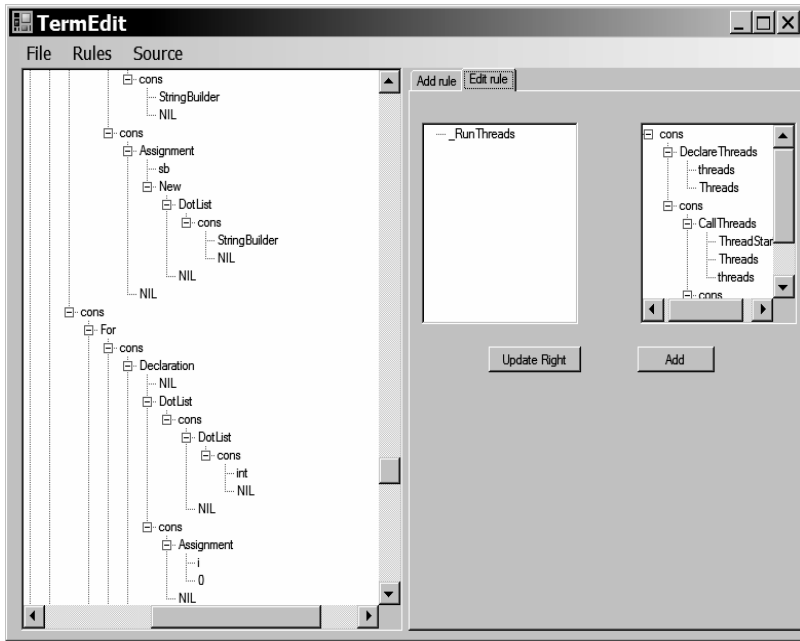


Fig. 6.15. The graphical user interface of the Termware.NET toolkit

6.4.3. Additional facilities of the rewriting rules toolkit.

Besides the basic possibilities of the TermWare system [154], the developed toolkit supports the additional facilities focused on simplification of work of a user with the system. Such facilities concern:

- labels;
- TODO-terms;
- multisystems;
- rule templates;
- modifiers;
- patterns.

TODO-terms and labels are the terms of a special kind which can be used in rewriting rules together with other terms, but also can be additionally processed by the toolkit.

Labels look like *_MARK_...*, i.e. the term name begins with the symbols *_MARK _* followed by any other symbols and/or sub-terms. Labels are used for selection of separate elements (subterms) of a model and have no independent significance in a model. Labels are used for indication of fragments of code (model) to which the transformations are applied. Besides, labels can be used to prevent repeated application of rules to terms that have been already transformed. The toolkit processes the labels in a special way: there is a command for removing all labels from a model; besides, the generator ignores labels at creation of code based on a model. Thus, labels do not influence generated code, but allow rules to work correctly.

As an example of use of labels, consider the rules intended for addition of the element *Field(b, int)* (the integer field of a class with the name *b*) directly after the element *Field(a, int)*. The simplest rule implementing the mentioned transformation is the following:

$$[Field(a,int):\$next] \rightarrow [Field(a,int):[Field(b,int):\$next]] \quad (6.1)$$

Here the variable *\$next* is used for a designation of a list tail. However, the rule represented in such a way will lead to an infinite loop: after its first triggering, in the model there is the element *Field(a, int)* to which the same rule is applicable. To avoid an infinite loop, it is possible to mark the element *Field(a, int)* in the right part of a rule with a label *_MARK_Processed*:

$$\begin{aligned} [Field(a,int, \textit{_MARK_Target}):\$next] \rightarrow \\ [Field(a,int):[Field(b,int):\$next]] \end{aligned} \quad (6.3)$$

In practice, both described approaches can be applied depending on problem features. If it is necessary to transform only one element of a model (for example, the field with a name *a* only in one class), a label on an initial term is used. If the transformation should work with all elements of the given kind (to add a new field after all

fields with a name a), it makes sense to use labels on a resulting term.

TODO-terms are used for defining rules of the general kind, in which the part of resulting terms is in advance unknown. Similarly to labels, TODO-terms look like $_TODO_...$, where symbols $_TODO_$ are followed by any symbols and subterms. The toolkit supports the list of all terms of such kind and suggests a user to replace them with specific elements of a model. Automatic replacement of a TODO-term by a term obtained from it by removal of symbols $_TODO_$ is also supported.

As an example of use of TODO-terms, consider slightly changed transformation (6.1). We will assume that it is necessary to replace the element $Field(a, int)$ with an element of the form $Field(b, int)$, but with a name which should be set by a user. For this purpose, the following rule is used:

$$Field(a, int) \rightarrow Field(_TODO_b, int) \quad (6.4)$$

In this case, a user can choose an element $_TODO_b$ from the list of TODO-terms and replace it with the necessary name. If a model contains several terms of the form $_TODO_b$, all of them will be replaced. If a user does not change the given term, it is automatically replaced with a term b during code generation. It is also possible to use the rule of the following kind:

$$Field(a, int) \rightarrow _TODO_Field(b, int) \quad (6.5)$$

In this case, the whole term in the right part of the rule can be changed, though the default value remains the same.

A more difficult example of use of TODO-terms for defining optimizing transformations of multithreaded programs is given in Subsection 6.2.

Multisystems allow to apply several systems of rules in the specified order; each of the systems is executed only once. For example, the rule (6.2) can be united with the rule

$$Field(a, int, _MARK_Processed) \rightarrow Field(a, int) \quad (6.6)$$

The rules (6.2) and (6.6) cannot be united in one system, as they will lead to an infinite loop (after the application of the rule (6.2) and the rule (6.6), the rule (6.2) is applicable again). However, they can be united into one multisystem. In this case, the rule (6.2) is applied at first, then the rule (6.6) is executed, but after its application the rule (6.2) is already inapplicable. Thus, it is possible to implement transformations, which contain identical elements in initial and resulting terms, thus avoiding cycling of rules.

Rule templates allow simplifying the creation of composite sets of rules which are often used for solving certain problems.

As an example, consider the addition of an element to a list immediately after a given element. For creation of such a rule, the user selects a term corresponding to an element of the list and chooses the command “Create Special Rule — List Insert After” from the contextual menu. The result of application of this command to the term $T(x_1, \dots, x_n)$ is the following rule:

$$\begin{aligned} & [T(x_1, \dots, x_n):\$next] \rightarrow \\ & [T(x_1, \dots, x_n, _MARK_Processed):[_TODO_Add: \$next]] \end{aligned} \quad (6.7)$$

This rule inserts the term $_TODO_Add$ after the term $T(x_1, \dots, x_n)$. Besides, the term T is marked with the label $_MARK_Processed$. Rule (6.2) was created by means of the rule template (6.7). Rule templates can contain not only separate rules, but also systems and even multisystems of rules.

The list of rule templates can be extended by a user. For this purpose, it is necessary to specify the name of a rule and create a rule template containing the subterm $_CURRENT$. This subterm, after executing the “Create Special Rule” command, will be replaced by the current subterm (i.e. the subterm for which the contextual menu was called). At creation of a rule template, it is possible to use special functions $_AddMark$ for addition of a label and $_ClearMark$ for removal of labels. For example, rule (6.7) described above is defined by the rule template

$$[_CURRENT: \$next] \rightarrow$$

$$\begin{aligned} &[_{AddMark}(_{CURRENT}, Processed): \\ &[_{TODO_Add}: \$next]] \end{aligned} \quad (6.8)$$

Modifiers are used for reduction of the size of the treelike representation of a term. Modifiers of two kinds are used: modifiers-terms and modifiers-lists.

Modifiers-terms have the form $m_t = m_t(f) = \langle f, string_f \rangle$, where $f \in \Sigma_t$ is some terminal symbol, $string_f : T_f \rightarrow STRING$ is a function which transforms the terms from the set $T_f = \{f(t_1, \dots, t_n) \mid t_i \in T_v\}$ (the terms with the name f) to a string representation. The function $string_f$ can simply return the string representation of a term (in the form of $f(t_1, \dots, t_n)$). Besides, it can use reductions for arithmetic and other operations, similar to described in Subsection 6.4.1. For example, the string representation of the term $t = plus(a, b)$ is $string_{plus}(t) = a + b$.

Modifier-list is a pair $m_l = m_l(f) = \langle f, R_f \rangle$, where the symbol $f \in \Sigma_t$ is used for designation of a list, and the rules R_f translate the list to a standard form $cons(t_1, cons(t_2, \dots, cons(t_n, NIL) \dots))$. As a result of the application of the modifier-list, the standard list takes the form of a single-level tree with the top f and the nodes $t_1, t_2, \dots, t_n$. The nodes with the symbols $cons$ and NIL , which have technical character and interfere with the perception of a model, are removed.

Patterns also facilitate a reduction of a size of a tree representing a model, however, unlike the modifiers, patterns change the model itself, and not just its graphic representation. Patterns describe conformity between often occurring combinations of elements of a model and their reduced designation. The set of patterns allows to carry out a transition between a low-level (more detailed) model of a program and its high-level variant.

As it is was described in Subsection 6.4.2, the developed toolkit contains the parser and the generator of C# code, which allows building a model of a code in the form of terms on the basis of

source code, and also transit from a model to source code. However, such model is low-level, as it contains the description of all syntactic constructs of the programming language. For example, consider a simple and widespread construct of C# language, a for-loop. This loop corresponds to a code fragment

$$\text{for}(\text{int } \$var = \$start; \$var < \$end; \$var++) \{ \$body \} \quad (6.9)$$

Here variables $\$var$, $\$start$, $\$end$, $\$body$ are used for indicating the sections of a code, which vary depending on a program. The fragment of the code (6.9) corresponds to the following low-level (syntactic) model:

$$\begin{aligned} & \text{For}(\text{cons}(\text{Declaration}(\text{NIL}, \text{DotList}(\text{cons}(\text{DotList}(\text{cons}(\text{int}, \\ & \quad \text{NIL})), \text{NIL})), \text{cons}(\text{Assignment}(\$var, \$start), \text{NIL})), \text{NIL}), \quad (6.10) \\ & \quad \text{less}(\$var, \$end), \text{cons}(\text{PostIncrement}(\$var), \text{NIL}), \$body) \end{aligned}$$

This term is quite cumbersome and inconvenient for understanding and change. Therefore, more high-level (algebraic) representation is used instead of it:

$$\text{ForCnt}(\$var, \$start, \$end, \$body) \quad (6.11)$$

A high-level model is represented in the form of terms, as well as low-level program model. However, unlike the low-level, the high-level describes not syntactic constructs of a programming language, but the operators of Glushkov algorithmic algebra.

The advantage of use of high-level program models consists in the possibility of shorter and more expressive notation of program transformations. However, there is a necessity of transition between a program model and a source code. For low-level (syntactic) models, such transformation is carried out with the use of a parser and a generator for a given programming language. However, for construction of models of higher level, it is necessary to have additional knowledge about a subject domain, which can be expressed in a form of sets of basic operators and predicates of Glushkov algebra.

For automated transition from source code to a high-level program model and in the opposite direction, we use the rewriting rules technique. The transition is made in two stages: between source code and a low-level model (a parse tree), and then between a low-level model and a high-level model (operators of Glushkov algebra). At the first stage, the parser and the generator of a given language are used. The second stage is carried out with the use of rewriting rules: as both kinds of models are representable in a form of terms, the transformations between them are written in the form of rules. The rules are represented in the form of TermWare patterns. In the general case, the pattern is defined as a pair of systems of rules:

- R_p is for extraction of a pattern from a given term;
- R_g is for expanding the pattern into its corresponding low-level term.

In a more specific case, the pattern is defined by a pair of terms:

- t_p is a designation of the pattern (an element of a high-level model);
- t_g is an implementation of the pattern (an element of a low-level model).

In this case, $R_p = \{t_g \rightarrow t_p\}$ and $R_g = \{t_p \rightarrow t_g\}$.

The pattern consisting of the operator t_p and its implementation in terms of the low-level model t_g are defined for each high-level operator. For creating a high-level model from a low-level one, rule sets R_p are applied for each pattern. Similarly, rule sets R_g are applied for transformation from a high-level to a low-level model.

For example, in the case of for-loop, it is possible to use the expression (6.11) as a term t_p , and the expression (6.10) as a term t_g . The pattern obtained in such a way can be used for an automatic transition from a syntactic to an algebraic representation of the loop.

As a more specific example of pattern use, consider the function `_GetCoor`, which is applied for computing a number of an ini-

tial iteration of a loop on the basis of parameters of a thread and a block in CUDA platform. In this case

$$t_p = _GetCoor(\$c),$$

$$t_g = Dot(blockIdx, \$c) * Dot(blockDim, \$c) + Dot(threadIdx, \$c).$$

Thus, the element of a high-level model $_GetCoor(x)$ can be transformed into an element of a low-level model

$$Dot(blockIdx, x) * Dot(blockDim, x) + Dot(threadIdx, x),$$

which then will be transformed into a fragment of a source code

$$blockIdx.x * blockDim.x + threadIdx.x.$$

The transformation in an opposite direction is also possible, in which the fragment of source code is translated to an element of a low-level model with the use of the parser, and then the rule R_p of a pattern is used for selection of an element of a high-level model.

Another important feature of high-level models is their independence from an implementation language. One high-level program model can correspond to low-level programs in various languages (or with the use of various platforms). To support additional language, we need to add low-level model support (i.e. parser and code generator) and implement patterns for this language.

Control questions and exercises

1. Name the main components of the IDS toolkit. What is the main difference of IDS from the MULTIPROCESSIST system?
2. What is the method underlying the design of algorithms in IDS?
3. Describe the processes of generating schemes on the basis of hyperschemes and generation of programs in IDS. How the constructs of SAA-M are mapped to a target programming language in the toolkit? Give examples of templates for implementing SAA-M constructs in Java.

4. Design the sort algorithm *Solute* (see Subsection 2.4, Example 2.8) and generate corresponding code in one of the programming languages C++, C#, Java using IDS [67].
5. Construct the sort algorithm *Shuttle* (see exercise 2 at the end of Chapter 5) and generate the corresponding program in one of languages C++, C#, Java by means of IDS.
6. Design the sort algorithm *Bubble* (see Subsection 2.1, Example 2.3) and generate corresponding code in one of the programming languages C++, C#, Java using IDS.
7. Using IDS, construct the modified sort algorithm *Bubble2* (see exercise 2 at the end of Chapter 5) and generate corresponding code in one of the programming languages C++, C#, Java. Compare the execution time of programs implementing the algorithms *Solute*, *Shuttle*, *Bubble*, *Bubble2*.
8. Construct the parallel sort algorithm *PBubble* (see Subsection 2.3, Example 2.6) generate corresponding code in one of the programming languages C++, C#, Java using IDS.
9. Describe the main features of the rewriting rules system TermWare.
10. What are the main components of the Termware.NET framework?
11. Describe the additional facilities which Termware.NET has in comparison with the basic possibilities of the TermWare system.
12. Apply Termware.NET [69] for doing exercise 16 given at the end of Chapter 3.
13. Use Termware.NET for transforming a simple C# program according to the instructions given in exercise1_en.pdf [69]. The program contains declarations of two variables (*p1* and *tmp*) and operators printing their values. The transformation of the program consists in adding new variable *p2* after variable *p1*, and also the operator printing the value of *p2*.
14. Apply Termware.NET for a transition between C# programs implementing various sort algorithms. Transform a program implementing the algorithm *Solute* (see Subsection 2.4, Example 2.8) to a program implementing *Shuttle* (see exercise 2 at the end of Chapter 5), and also the transformation of *Bubble*

(see Subsection 2.1, Example 2.3) to *Bubble2* (see exercise 2 at the end of Chapter 5).

Chapter 7

Automated development of efficient parallel programs

In this chapter, the examples of application of IDS and TermWare for design, generation and transformation of programs for multicore processors and graphics processing units are given. Formalization and verification of parallel programs using a proof assistant software Mizar is considered.

7.1. Design and parallelization of programs for multicore processors

As an example of usage of IDS and TermWare.NET systems, consider the problem of finding the quantity of prime numbers in the range from 1 to some number [49]. The computations are made by checking each odd number, whether is it divided by smaller odd factors. The given problem belongs to a class of problems in which some independent calculations are carried out and then the result of calculations is saved to the shared memory. Such problems are quite easy for parallelization, however, there are some features influencing the performance of a multithreaded program.

Below we give the fragment of the SAA scheme (namely, the compound operator `SimulationRun`) of the sequential algorithm intended for finding prime numbers. The scheme was designed with the help of the IDS toolkit.

```
“SimulationRun”  
==== Locals  
  (  
    “Declare a variable (number) of type (long)”;  
    “Declare a variable (start) of type (long)”;  
    “Declare a variable (end) of type (long)”;  
    “Declare a variable (stride) of type (long)”;  
    “Declare a variable (factor) of type (long)”  
  )  
  THEN  
    “start := 1”  
  THEN
```

```

“end := Number”
THEN
“stride := 2”
THEN
IF ‘start = 1’
THEN “start := start + stride”
END IF
THEN
“number := start”
THEN
WHILE NOT ‘number >= end’
LOOP
    “factor := 3”
    THEN
    WHILE NOT ‘Remainder from division of (number)
        by (factor) = (0)’
    LOOP
        “factor := factor + 2”
    END OF LOOP
    THEN
    IF ‘factor = number’
    THEN “Primes[PrimeCount] := number”
        THEN
        “PrimeCount := PrimeCount + 1”
    END IF
    THEN
    “number := number + stride”
END OF LOOP

```

On the basis of the given scheme, the IDS toolkit generated the fragment of the sequential program in C# language (the method `SimulationRun`), which is given in Subsection A.1 of Appendix A.

The sequential program is parallelized by splitting the main loop (which searches prime numbers) into `NUM_THREADS` loops, where `NUM_THREADS` is the number of threads. In the elementary case, each thread processes consecutive numbers. The transformation of the program consists of the following actions:

1) the replacement of a call of the SimulationRun function by operators of creation of NUM_THREADS threads;

2) the modification of the loop parameters;

3) the addition of synchronization means (a critical section).

All these actions are implemented using TermWare rules.

With the help of the Termware.NET system, the source code of the program is transformed to a low-level model with use of the parser of C# language, and then the low-level model is transformed to high-level model with use of patterns. As a result we obtain the following term:

```
Method(SimulationRun, void, NIL,
[protected, override],
[Declaration(number, long),
Declaration(start, long),
Declaration(end, long),
Declaration(stride, long),
Declaration(factor, long),
Assignment(start, 1),
Assignment(end, Number),
Assignment(stride, 2),
If(Equals(start, 1),
Assignment(start, start + stride)),
While(Not(number >= end), [
Assignment(factor, 3),
While(Not(Equals(number % factor, 0)),
[Assignment(factor, factor + 2)]),
If(Equals(factor, number),
[Assignment(ArrayElement(Primes,
PrimeCount), number),
Assignment(PrimeCount,
PrimeCount + 1)])],
Assignment(number,
number + stride)])])]
```

The given term is transformed by applying the following system of rules to it:

1. `cons(Method(SimulationRun, void, NIL, $x1, $x2), $x0)`
 $\rightarrow$ `cons(Method(SimulationRun, void, NIL, $x1,`
`_Parallel_For_Call (RunThreadNum, [_Index],`
`Threads), _MARK_1), cons(Method(RunThreadNum,`
`void, [Parameter(ThreadNum, int)], [protected, virtual],`
`$x2), $x0));`
2. `Assignment(start, 1)  $\rightarrow$  Assignment(start, ThreadNum * BLOCKSIZE + 1);`
3. `Assignment(end, Number)  $\rightarrow$  Assignment(end,`
`ThreadNum * BLOCKSIZE + BLOCKSIZE);`
4. `If(Equals(factor, number), $x0)  $\rightarrow$  If(Equals(factor,`
`number), Lock(This, $x0)).`

This rule system works as follows: rule 1 replaces the method `SimulationRun` with a call of the new method `RunThreadNum` with the defined number of threads `ThreadNum`. Thus the code of the method `RunThreadNum` after application of rule 1 appears the same as an initial code of the method `SimulationRun`. Further, the rules 2–4 modify a code taking into account an introduction of multithreading. The rules 2 and 3 change the limits of the loop, which depend now on an index of a thread. Rule 4 adds blocking for protection of operation of access to shared memory (namely, the variable `PrimeCount` and the array `Primes`). Notice that for creation of rule 1, the command “Create Special Rule — List Insert After”, described in Subsection 6.4.3 was used.

The given system of rules is applied to the term of the sequential program and as a result we obtain the term of the parallel multithreaded program named `BlockTasksThread`. The new method `RunThreadNum` is added, limits of loops are modified and blocking of critical section is added. The term corresponding to the new method `RunThreadNumWe` is the following:

```
Method(RunThreadNum, void,
[Parameter(ThreadNum, int)],
    [protected, virtual],
    [Declaration(number, long),
```

```

Declaration(start, long),
Declaration(end, long),
Declaration(stride, long),
Declaration(factor, long),
Assignment(start,
            ThreadNum * BLOCKSIZE + 1),
Assignment(end, ThreadNum *
            BLOCKSIZE + BLOCKSIZE),
Assignment(stride, 2),
If(Equals(start, 1), Assignment(start,
                                start + stride)),
While(Not(number >= end), [
Assignment(factor, 3),
While(Not(Equals(number % factor, 0)),
    [Assignment(factor, factor + 2)]),
    If(Equals(factor, number),
        Lock(This,
            [Assignment(ArrayElement(Primes,
                PrimeCount), number),
              Assignment(PrimeCount,
                PrimeCount + 1)])),
Assignment(number, number +
            stride)]])

```

On the basis of the obtained term, the code of the parallel program was generated, which is given in Subsection A.1 of Appendix A.

In the obtained program `BlockTasksThread`, the complexity of one iteration of the loop increases depending on iteration number and therefore different threads carry out various amounts of work. To increase the performance, it is possible to modify loop variables so that consecutive numbers are processed by different threads; thus the amount of work of each thread will be approximately identical. This transformation is implemented in the form of the following TermWare rules:

1. `Assignment(start, plus(multiply(ThreadNum, BLOCKSIZE), 1)) → Assignment(start, 2 * ThreadNum + 1);`
2. `Assignment(end, plus(multiply(ThreadNum, BLOCKSIZE), BLOCKSIZE)) → Assignment(end, Number);`
3. `Assignment(stride, 2) → Assignment(stride, 2 * Threads).`

The result of the application of these rules to the `BlockTasksThread` program is a more efficient `InterleavedTasksThread` program. The obtained program can be improved due to use of other means of synchronization, in particular, atomic functions of `Interlocked` type (which carry out an atomic change of an argument) [72]. In this connection, the new patterns for separate operations are used, for example:

$$\begin{aligned} _AtomicIncrement(x) &= lock(cs); \\ &\quad Increment(x); \\ &\quad unlock(cs). \end{aligned}$$

This pattern can be implemented by means of the combination of standard means, i.e. as a fragment of the code `lock(cs){x++;}`, and in the form of one call of `Interlocked` function (that corresponds to the fragment `Interlocked.Increment(x)`). As a result of the transition from initial critical sections to atomic functions, we obtain even more efficient program `InterlockedThread` (though performance change in comparison with the previous program `InterleavedTasksThread` appears insignificant).

Fig. 7.1 and Fig. 7.2 show the results of performance comparison of the obtained programs executed on Intel Core2 Duo E6550 CPU processor (2 cores, 2.33 GHz), 4 GB of memory. The dependencies of execution time on the quantity of numbers and the number of threads are given.

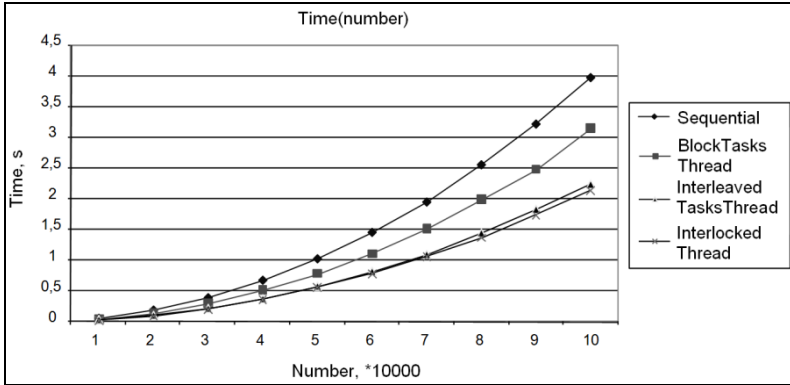


Fig. 7.1. The dependence of execution time on the quantity of numbers (2 threads).

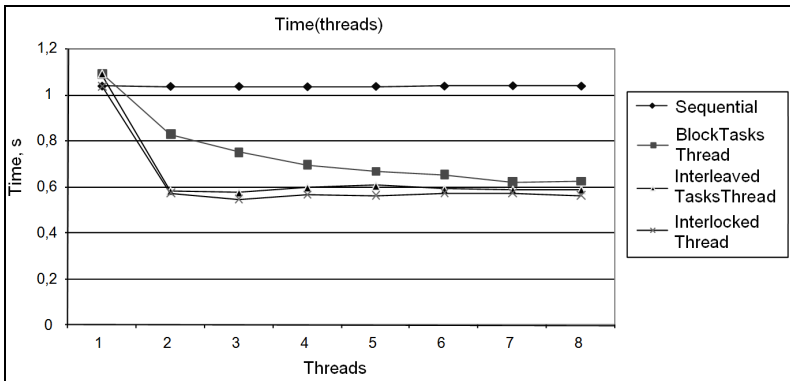


Fig. 7.2. The dependence of execution time on the quantity of threads (50000 numbers).

The graphs show that the elementary way of parallelization in the initial parallel program BlockTasksThread allows using hardware resources of the processor, but not completely. Usage of more correct scheme of parallelization in the InterleavedTasksThread program allows to speed up the performance practically twice in comparison with the sequential case, i.e. to completely use the hardware resources. We will notice also that at the use of larger quantities

of threads, the performance of BlockTasksThread increases, verging towards more efficient InterleavedTasksThread and InterlockedThread. Thus, for problems of smaller size the correct distribution of computation between threads becomes not so essential.

7.2. Transformation of coordination constructs in parallel programs

In this subsection, we consider the application of TermWare.NET for transformation of coordination constructs on the example of the multithreaded program intended for simulation of one-dimensional Brownian motion [49]. The problem is formulated as follows: it is necessary to simulate the behavior of a set of particles in a one-dimensional crystal of certain length. On each step of the simulation, each particle moves to the right or to the left with a specified probability. The behavior of each particle is simulated by a separate thread. The feature of this problem is rather small amount of calculations in comparison with synchronization operations (necessary for writing a state of the crystal to shared memory). We start with the program named SmallLockThread, which is already parallel. The high-level model of the program is represented in the form of a term, which is given in Subsection A.2 of Appendix A.

Consider the transformations of SmallLockThread program, which modify the synchronization constructs. In the elementary case, the transformation can be directed on a change of the critical section. Two transformations are possible: the extension of the critical section (when the critical section joins a code which initially does not belong to it) and splitting of the critical section to several critical sections.

The extension of the critical section is implemented with use of the system of rules R1:

1. THEN(CriticalSection(\$x0, \$x1), \$x2) →
CriticalSection(\$x0, THEN(\$x1, \$x2));
2. THEN(\$x0, CriticalSection(\$x1, \$x2)) →
CriticalSection(\$x1, THEN(\$x0, \$x2));

3. $\text{IF}(\$x0, \text{CriticalSection}(\$x1, \$x2)) \rightarrow \text{CriticalSection}(\$x1, \text{IF}(\$x0, \$x2)).$

The first two rules define the extension of the critical section at linear fragments of code. The third rule defines the inclusion of IF constructs in the critical section (similar rules can be also created for other control constructs). The application of R1 leads to the transformation of SmallLockThread program to BigLockThread program (the term corresponding to this program is given in Subsection A.2 of Appendix A).

Splitting of the critical section into some smaller critical sections is made by means of the following system of rules R2:

1. $\text{CriticalSection}(\$name, \text{THEN}(\$x, \$y)) \rightarrow \text{THEN}(\text{CriticalSection1}(\text{TODO_Name}(\$name, 0), \$x), \text{CriticalSection1}(\text{TODO_Name}(\$name, 1), \$y));$
2. $\text{CriticalSection1}(\text{TODO_Name}(\$name, \$n), \text{THEN}(\$x, \$y)) \rightarrow \text{THEN}(\text{CriticalSection1}(\text{TODO_Name}(\$name, \$n), \$x), \text{CriticalSection1}(\text{TODO_Name}(\$name, \$n+1), \$y));$
3. $\text{CriticalSection1}(\$name, \text{NIL}) \rightarrow \text{NIL}.$

The first rule allocates the first operator from the critical section in a separate critical section. The second rule describes the consecutive allocation of elements of the critical section in separate critical sections, and the third rule describes the stop condition. The terms describing new critical sections receive the special name CriticalSection1 instead of CriticalSection. Further, it allows using the rules which work only on such transformed critical sections.

Let us notice that special term TODO_Name is used in the given rules. The terms of such kind (TODO_Term) specify the necessity of creation of additional rules, which extend a general rule with information specific to a given problem. For instance, the critical sections in the above-considered rules can have various names. The user can specify the names which should be used in a given problem. In particular, it makes sense to use an array of critical sections and block each operation by means of the critical section corre-

sponding to an index of a changeable element of a crystal. In this case, the system of rules R3 is applied:

1. `TODO_Name(Admixture_CS, 0) → ArrayElement(Admixture_CS, x);`
2. `TODO_Name(Admixture_CS, 1) → ArrayElement(Admixture_CS, nextx);`
3. `CriticalSection1($x, $y) → CriticalSection($x, $y).`

The first two rules define specific names of critical sections. Rule 3 translates the term with a special name `CriticalSection1` to the standard term `CriticalSection`. We will notice that instead of `CriticalSection1` it was possible to use a marked term `CriticalSection($id, $body, _MARK_1)`. Thus the toolkit is capable to define the presence of terms demanding additional transformations, and also automatically create an analog of rule 3 for removing the label `_MARK_1`.

The result of application of these rules to the initial program is the `SeparatedLockThread` program (see Subsection A.2 of Appendix A). Thus, application of elementary rules for modifying critical sections allows receiving new implementations of the given problem on the basis of the initial program. We will notice that both obtained programs are not optimal. In general, the considered transformations are more likely to worsen the properties of the program. The extension of critical sections reduces a quantity of code, which can be carried out in parallel (though simplification of the structure of the program can lead to use of compiler optimizations, which improves the performance). Splitting a critical section increases the synchronization overhead.

Thus, the modification of critical sections itself cannot promote the program optimization. However, the application of such transformations allows introducing additional means of synchronization which can be more efficient.

As an example, we will consider the use of the `Interlocked` class. An automatic application of this means of synchronization is based on use of the `SeparatedLockThread` program (which itself is

not so efficient). We apply the following system of rules R4 to a term corresponding to this program:

1. CriticalSection(\$x0, Decrement(\$x1)) → AtomicDecrement(\$x1);
2. CriticalSection(\$x0, Increment(\$x1)) → AtomicIncrement(\$x1).

The use of similar rules for other methods of the Interlocked class is also possible. The result of application of the above rules is the InterlockedThread program (see Subsection A.2 of Appendix A). This program is more efficient than the initial. We will notice that similar transformations could be applied without the preliminary use of rules R3 as the additional information on specific objects of synchronization appears to be not necessary for the transformations R4.

In addition to the use of standard means of synchronization implemented in .NET Framework, the use of the specialized means developed manually is also possible. So, the problem of simulation of Brownian motion belongs to the class of problems in which data are processed in a loop, where the size of the loop can be quite large and the number of operations in one iteration of the loop is rather insignificant. For such problems, it makes sense to unite some iterations of the loop in one. It can be implemented by means of the LoopManipulation class (the source code of essential methods of the class is given in Subsection A.2 of Appendix A).

The following rule R5 is applied to the BigLockThread program for transition to usage of user-defined synchronization construct:

```
FOR(Parameters($0, 0, $1), CriticalSection($2, $3)) →
THEN(Assignment(Identifier(LoopManipulation, lck), $2),
THEN(DelegateAssignment(LoopManipulationDelegate,
loopBody, $3), CALL(Identifier(LoopManipulation,
RunSafeLoop), Parameters(loopBody, $1))))).
```

The result of application of this rule is the LoopManipulationThread program (see Subsection A.2 of Appendix

A). This program has the same lacks, as `BigLockThread` (i.e. the most part of the code is in the critical section and it can be carried out only sequentially). Nevertheless, the synchronization overhead is reduced essentially due to the reduction of the number of critical sections, and therefore the `LoopManipulationThread` program is more efficient.

Let us notice that the rule R5 is difficult enough as it applies the means of `LoopManipulation` class with the creation of objects and delegates, call of methods, etc. The developed toolkit allows simplifying similar rules by means of introduction of patterns. It is possible to introduce the term `_BlockingFOR(Parameters($varname, $begin, $end), $cs, $body)` which is equivalent to the fragment of code given above. Then the rule R5 can be rewritten as R5':

$$\text{FOR}(\$params, \text{CriticalSection}(\$cs, \$body)) \rightarrow \\ _ \text{BlockingFOR}(\$params, \$cs, \$body).$$

The description of the specific implementation of the operator `_BlockingFOR` is taken out to the description of a pattern, which is created once and further the new operator can be used at implementation of any rules or program models.

Thus, `TermWare` rules can be applied for a transition from the standard synchronization means (lock blocks) to specialized constructs, which are built into .NET Framework (for example, `Interlocked` class) or implemented manually (the `LoopManipulation` class).

The programs obtained until now used the general rules that allowed applying them to a wide class of problems. However, the performance of such programs appeared low. Further we consider the transformations, which essentially use knowledge about a specific problem for increasing the efficiency of program implementation.

The basic problem at developing an efficient parallel implementation of the program for simulation of Brownian motion is the necessity of execution of a considerable quantity of simple iterations, which leads to a large synchronization overhead. In the `LoopManipulationThread` program, the synchronization is used not so often, but it is achieved due to blocking considerable fragments of

code (practically all the operations are executed sequentially). For increasing the performance, it is necessary to reduce the number of synchronizations (similarly to LoopManipulationThread) and not to extend a critical section at the same time.

One of the variants of implementation of the mentioned approach consists in keeping the state of each iteration within corresponding thread with writing of this state to a shared memory after a certain quantity of iterations. The result program (InvisibleMovesThread) can be obtained from the SmallLockThread with use of the following rules:

1. CriticalSection(\$x0, \$x1) $\rightarrow$ IF(iteration_counter % TODO_BlockSize == 0,
THEN(TODO_SaveLocalState(\$x0, \$x1),
THEN(TODO_InitLocalState, NIL)),
TODO_UpdateLocalState(\$x1));
2. FOR(Parameters(iteration_counter, \$x1, \$x2), \$x0) $\rightarrow$
THEN(TODO_InitLocalState,
TODO_FOR(Parameters(iteration_counter, \$x1, \$x2),
\$x0)).

The first rule transforms the critical section to a local saving of changes or introducing all changes into shared memory. The second rule adds the initialization of the local state before the main loop.

After application of these rules, it is necessary to specify a specific way of initialization of the local state, its updating and saving to a shared memory. These data are specified by the developer according to the knowledge about the specific problem. The toolkit specifies what terms are necessary for redefining (proceeding from the names of the kind TODO_Term). In this case, the rules are the following:

1. TODO_InitLocalState $\rightarrow$ Assignment(firstx, x);
2. TODO_UpdateLocalState(\$x0) $\rightarrow$ NIL;
3. TODO_SaveLocalState(\$x0,
THEN(Decrement(ArrayElement(Crystal, x)), \$x1)) $\rightarrow$

- ```
CriticalSection($x0,
```
- ```
  THEN(Decrement(ArrayElement(Crystal, firstx)), $x1));
```
4. TODO_BlockSize $\rightarrow$ blockIterations;
 5. TODO_FOR($\$x0$, $\$x1$) $\rightarrow$ FOR($\$x0$, $\$x1$).

Each rule defines a specific value of a corresponding term. In the problem under consideration, the initialization consists in storing the current position of a particle. Updating of the local state is not required, as the combination of several movements of a particle is reduced to moving from the initial position to the final. Saving the local state to shared memory is similar to writing one movement, but with the use of a starting position of a particle instead of a previous. The quantity of iterations in the block is set in the blockIterations variable. Rule 5 is created automatically, if the rule for transformation of the TODO_FOR term is not specified.

The results of performance measurements of the developed programs are shown in Fig. 7.3 and Fig. 7.4. The programs were executed on Intel Core2 Duo E6550 CPU processor (2 cores, 2.33 GHz), 4 GB of memory.

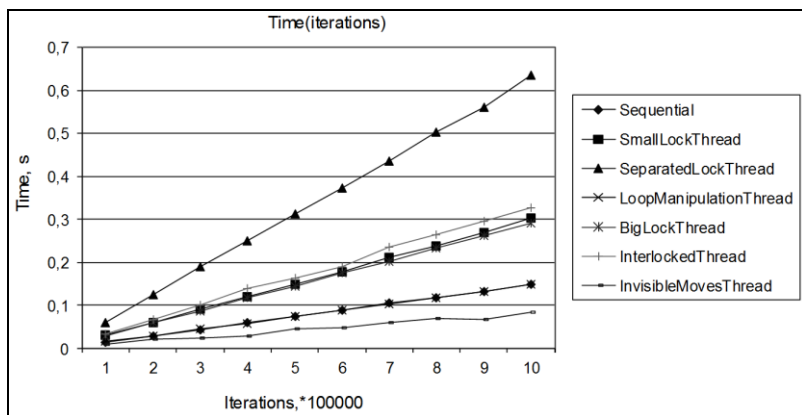


Fig. 7.3. The dependency of the execution time of different implementations of the program simulating one-dimensional Brownian motion on the number of iterations (2 particles, 2 threads)

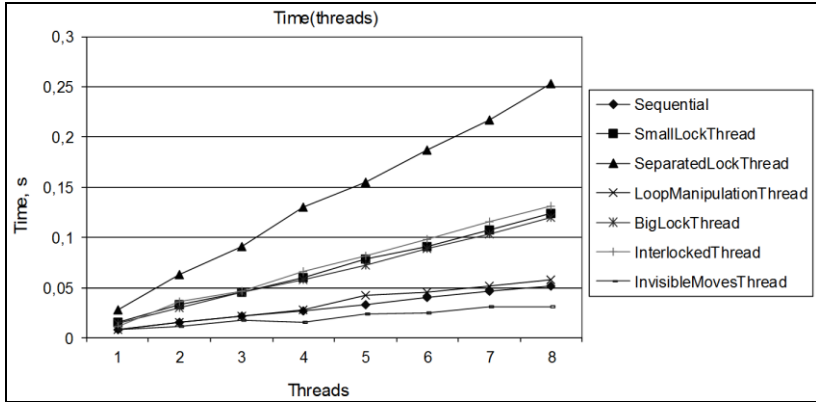


Fig. 7.4. The dependency of the execution time of different implementations of the program simulating one-dimensional Brownian motion on the number of threads (particles), 100000 iterations

As can be seen from the figures, most programs are actually less efficient than the sequential program: only InvisibleMovesThread program achieves some speedup.

A comparison of execution times demonstrates that the main reason for poor parallel performance is indeed synchronization overhead. Therefore, the worst program is SeparatedLockThread that contains two critical sections per iteration. The InterlockedThread program is better because of more efficient synchronization constructs. The SmallLockThread and BigLockThread programs have almost the same performance, because they have the same number of critical sections (one per iteration). The LoopManipulationThread program is almost identical to the Sequential program, because almost all computations are forced to be run by a single thread at a time. Finally, the most efficient parallel program is InvisibleMovesThread, because it is able to reduce the number of synchronization constructs without losing benefits of parallelism.

7.3. Development of parallel programs for graphics processing units

In this subsection, we give the examples of use of the developed algebra-dynamic models (see Subsection 5.3) and TermWare for parallelization and optimization of programs for NVIDIA graphics processing units supporting the CUDA technology. The mentioned examples include the problem of summation of vector elements and the Game of Life. The input data for each problem is source code of a sequential program in C# language. In the beginning, the source code is used to construct a low-level model of a program by using the parser of the Termware.NET system. Then a high-level model is constructed with the use of patterns. For the transition from a sequential to a parallel multithreaded program, the parallelizing transformations are applied to a program model. Then, optimizing transformations are applied to a parallel program in order to increase its efficiency. For each of the obtained versions of the program, the source code is generated, which is compiled and executed at various sizes of initial data. The execution times of a sequential program T_1 and a parallel program T_n (where n is a number of GPU cores) are measured, and also the multiprocessor speedup $Sp = T_1 / T_n$ is computed. All the programs being considered in this subsection were executed on the computer with the following parameters: Intel Core2 Duo E6550 CPU (2 kernels, 2.33 GHz), 4 GB of memory, NVIDIA GeForce GTS 250 GPU (128 cores, 1 GB of graphics memory).

7.3.1. Parallel summation of elements of a numeric vector. Consider the optimizing transformations for a problem of finding the sum of the large data array [47]. The hierarchical algorithm of summation is used: at each stage, the array elements are split into pairs and are summarized independently from each other, which allows receiving a high degree of parallelism. The elementary implementation of this algorithm in CUDA is given in Subsection A.3 of Appendix A.

Often the performance of CUDA programs depends more on a correct implementation of memory access than on computation efficiency. In particular, in the parallel summation problem, all the op-

erations are made with a global memory, which is accessible to all processors. However the access speed to global memory is significantly slower than to faster types of memory. The faster alternative of global memory is shared memory. It is located on each processor and therefore has a maximum access speed. The disadvantages of shared memory are its small size and that access to it is possible only within one block, which demands the synchronization. Usually work with shared memory is carried out as follows: data from a processed segment of global memory are copied to shared memory (this process is also parallel: each thread copies its part of data). After that, a synchronization of threads is carried out with the help of the `__syncthreads()` command. Further all the work is done in shared memory. When it is necessary to write results to global memory, synchronization is carried out and the results are copied by parts from a shared to global memory.

For transition to usage of shared memory, the transformation described in Subsection 4.4.2 is used. For implementation of this transformation, the following system of rules is used:

1. `Function($x, $y, parallelAdd, $z, [$b1:$b2]) →`
`Function($x, $y, parallelAdd1, $z,`
`[SharedDecl:[$b1:CopyBack($b2)]]);`
2. `SharedDecl → ArrayDeclaration (localData, int,`
`BLOCK_SIZE, __shared__);`
3. `Assignment (ArrayElement (outData, i),`
`ArrayElement(inData, i)) →`
`Assignment (ArrayElement (localData,`
`tid),ArrayElement(inData, i));`
4. `PlusAssignment(ArrayElement (outData, i),`
`ArrayElement(outData, i + s)) →`
`PlusAssignment (ArrayElement (localData, tid),`
`ArrayElement (localData, tid + s));`
5. `CopyBack([$x:$y]) → [$x:CopyBack($y)];`
6. `CopyBack(NIL) → If (Equals(tid, 0),`
`Assignment(ArrayElement(outData, Dot(blockIdx, x)),`
`ArrayElement(localData, 0))).`

Rules 1 and 2 add the declaration of the local array of shared memory. Rule 3 replaces copying directly to a target array with copying to a local array. Similarly, rule 4 replaces the summation in a global memory with a summation in shared memory. Rules 5 and 6 add copying a result of calculation to an output array (at the end of a kernel). The modified kernel obtained after application of these rules is given in Subsection A.3 of Appendix A.

Further, some transformations specific to the given problem are applied. One more lack of the initial implementation of the parallel summation algorithm is the considerable quantity of branching. In the code, the following loop condition is used: `if (tid % (2*s) == 0)`. The value of this condition varies often for adjacent threads (in particular, for $s = 1$ all adjacent threads go at different branches). As the architecture of GPU provides the larger quantity of computing blocks in comparison with control blocks, such branching compels to start threads sequentially instead of in parallel.

The reduction of branching degree is possible based on redistribution of computations between threads. In the initial implementation, calculations are carried out in threads with numbers that are multiples of 2^n (even numbers at the first step, multiple of 4 at the second step, etc.). We consider the transformation, which transfers the computations to consecutive threads (the first half of threads at the first step, the first quarter at the second step and so on). The transformation consists of one rule:

```
If(Equals(tid % (2 * s), 0),
  PlusAssignment(ArrayElement(localData, tid),
    ArrayElement(localData, tid + s))) →
[DeclarationAssignment(index, int, 2 * s * tid), If(index <
  Dot(blockDim, x),
  PlusAssignment(ArrayElement(localData, index),
    ArrayElement(localData, index + s)))].
```

The given rule replaces the previous conditions with the following code:

```
int index = 2 * s * tid;
```

```

if (index < blockDim.x)
data[index] += data[index + s];

```

After the application of the previous rule, the quantity of branching goes down essentially. However, now the disadvantage is access conflicts to shared memory. The feature of this memory is that it is divided into 16 banks, the access of adjacent threads to one bank is carried out sequentially and to various banks — in parallel [130]. In previously obtained variant of implementation, all data are stored in first elements of the array, which lay in one bank. Therefore all references to array elements appear to be sequential. To avoid such situation, it is possible to apply the transformation consisting in a change of the summation order. Earlier, the adjacent elements were summarized at the first stage, the elements with distance 2 were summarized at the next stage, etc. Further we pass to another summation order: at the first stage, we summarize maximally distant elements and at the next stages, we reduce the distance between the elements. For this purpose, the following system of rules is used:

1. For(DeclarationAssignment(s, int, 1), s < Dot(blockDim, x), MulAssignment(s, 2), \$body) →
For(DeclarationAssignment(s, int, Dot(blockDim, x) / 2), s > 0, RShiftAssignment(s, 1), \$body);
2. If(Equals(tid % (2 * s), 0), \$body) → If(tid < s, \$body).

This system of rules is applied to the program which uses shared memory. The first rule replaces the parameters of the loop: the distance between summarized elements decreases, not increases. The second rule replaces the summation condition. The transformed code is the following:

```

for (int s = blockDim.x / 2; s > 0;
    s >>= 1) {
    if (tid < s)
        localData[tid] += localData[tid + s];
    __syncthreads ();
}

```

Let us notice that at the use of the given transformation two optimizations are introduced: the reduction of conflicts at accessing memory banks and a decrease in quantity of branching. In this sense, this transformation is more efficient, than considered in the previous stage. However, the given transformation uses more knowledge about the features of the problem, i.e. about the possibility to carry out the summation in another order.

The considered transformation allowed to get rid of branching and access memory conflicts, but the efficiency of the usage of threads has thus decreased. The half of all threads do not perform computations, as the condition $tid < s$ is not satisfied for them. Therefore, it is possible to modify the kernel by excluding the threads, which do not carry out computations. In the new implementation, the first summation is made at the stage of copying of data to shared memory and therefore all threads participate in it. The rules for carrying out this transformation are the following:

1. $\text{DeclarationAssignment}(i, \text{int}, \text{Dot}(\text{blockIdx}, x) * \text{Dot}(\text{blockDim}, x) + \text{Dot}(\text{threadIdx}, x)) \rightarrow \text{DeclarationAssignment}(i, \text{int}, 2 * \text{Dot}(\text{blockIdx}, x) * \text{Dot}(\text{blockDim}, x) + \text{Dot}(\text{threadIdx}, x));$
2. $\text{Assignment}(\text{ArrayElement}(\text{localData}, \text{tid}), \text{ArrayElement}(\text{inData}, i)) \rightarrow \text{Assignment}(\text{ArrayElement}(\text{localData}, \text{tid}), \text{ArrayElement}(\text{inData}, i) + \text{ArrayElement}(\text{inData}, i + \text{Dot}(\text{blockDim}, x)));$
3. $\text{CallKernel}(\text{Shape}(\text{ARR_SIZE}/\text{BLOCK_SIZE}, \text{BLOCK_SIZE}), \text{parallelSum}, [\text{inData}, \text{outData}]) \rightarrow \text{CallKernel}(\text{Shape}(\text{ARR_SIZE} / (2 * \text{BLOCK_SIZE}), \text{BLOCK_SIZE}), \text{parallelSum}, [\text{inData}, \text{outData}]).$

Rule 2 defines that at copying to shared memory, the summation is also carried out. Thus one block can process twice the amount of information. Rule 1 specifies the modification of an index in the input array and rule 3 defines the change of parameters of the

kernel call (as each block processes twice the amount of data, the size of the grid is reduced by a half). As a result, the code of the kernel is modified in the following way:

```
int i = 2 * blockIdx.x * blockDim.x +
      threadIdx.x;
localData[tid] = inData[i] +
                  inData[i + blockDim.x];
```

One of the disadvantages of the initial implementation, which was not eliminated after all the transformations, is a large number of synchronizations (at each step of summarizing). These synchronizations are necessary for the first iterations, as different threads can be executed on different warps. However, on the last iterations, when the number of threads that perform the computations is not greater than 32 (the size of the warp), the synchronizations are done automatically. Therefore we can get rid of the explicit call of `__syncthreads()`. Besides, in this case we can extend the loop and remove the branching at the condition `tid < s`. For this, we use the following system of rules:

1. `For(DeclarationAssignment(s, int, Dot(blockDim, x) / 2), s > 0, RShiftAssignment(s, 1), $body) → [For(DeclarationAssignment(s, int, Dot(blockDim, x) / 2), s > 32, RShiftAssignment(s, 1), $body), _Unroll($body)];`
2. `_Unroll(If(tid < s, $body)) → If(tid < 32, _Unroll($body));`
3. `_Unroll[$x:$y] → [_Unroll($x):_Unroll($y)];`
4. `_Unroll(PlusAssignment($x, $y)) → _Unroll(PlusAssignment($x, $y), 32);`
5. `_Unroll(PlusAssignment($x, $y), $n) → [ PlusAssignment( $x, _Replace($y, $n)), _Unroll( PlusAssignment( $x, $y), $n / 2) ] [ $n > 1 ];`
6. `_Unroll(PlusAssignment($x, $y), 1) →`

- PlusAssignment(\$x, _Replace(\$y, 1));
7. $_Replace(\text{ArrayElement}(\text{localData}, \text{tid} + s), \$n) \rightarrow \text{ArrayElement}(\text{localData}, \text{tid} + \$n);$
 8. $_Unroll(\text{SyncThreads}) \rightarrow \text{NIL}.$

In this system, rule 1 replaces the limits of the loop from 0 to 32 and adds the template of the extended variant after the loop. Rule 2 removes the check $\text{tid} < s$ and adds the check $\text{tid} < 32$ for the extended loop. Rules 3–7 define the replacement of the loop body (summarizing) to the extended variant, which is repeated for the values 32, 16, 8, 4, 2, 1. As a result, rule 8 removes explicit calls of the synchronization function.

The final code of the kernel after application of all the rules is given in Subsection A.3 of Appendix A.

Table 7.1 shows the results of the comparison of different variants of implementation of the program of summation of vector elements. The time of 100 executions of each program was measured. The execution time of a sequential program is also given.

Table 7.1. The comparison of different program optimization means for the problem of parallel summation of elements of a numeric vector

Variant	Execution time, ms	Speedup
Initial	625	0.92
Shared memory	300	1.9
Branching eliminated	186	3.1
Conflicts eliminated	77	7.5
Idleness eliminated	44	13.1
Extended	30	19.3
Sequential	578	1

As can be seen from the results, the initial parallel implementation was not efficient and even had lower performance than the sequential version. Each transformation gradually increased performance. In total, after application of all the transformations, we ob-

tained the parallel program which is 20 times more efficient than the initial.

7.3.2. The Game of Life problem. As another example of usage of the proposed approach, we consider the C# implementation of the Game of Life [29, 39]. First, we apply parallelizing transformations from Subsection 5.4.4.1 to the initial sequential program. Then, the parallel program is optimized by usage of transformations considered in Subsection 5.4.4.2. The parallelization was applied to nested loops that compute the new state of the game field. After the transformation, each game cell was computed on a separate GPU thread. The optimizing transformation consisted in using shared memory instead of global memory. To improve the efficiency of shared memory use, another transformation was applied that increased the number of cells computed by each thread. In the transformed version, each thread computed the line of 16 cells. Kernel call parameters were also modified: from 16×16 block size (corresponding to 16×16 field fragment) to 256×1 block size (256×16 field fragment). Block size was selected automatically using the knowledge of limit of 16 KB of shared memory.

The program transformation was carried out in an automated way using developed rewriting rules and Termware.NET system. The source code of the initial program was automatically transformed into the low-level model using the parser of the system and then into the high-level model using patterns. After that, we manually selected the loop to be parallelized and the transformation (i.e. the set of rules) to be applied. The transformation was executed automatically and resulted in a high-level model of parallel GPU program that was transformed back to the low-level model and then to source code. The optimizing transformations were applied in the same way. Therefore the transformation process is automated and user input is required only to select which of the developed rules should be applied and which part of code they should affect. To estimate the effect of transformations, we measured the execution time for three program versions: initial sequential program; program parallelized without optimization; parallelized and optimized program. The execution time of each program was measured for 1000 iterations on the 512×512 field. The results of measurements are given in Table 7.2.

Table 7.2. The results of performance measurements for programs for the Game of Life problem.

Program	Execution time, s	Speedup
Sequential	15,66	1
Parallel, no optimization	1,89	8,3
Parallel, optimized	0,614	25,5

As can be seen from the table, even the non-optimized parallel program results in 8 times speedup and optimization results in 25 times speedup compared to the initial version.

7.4. Formalization and verification of parallel programs using a proof assistant

In this section we will illustrate one of the approaches to formal verification of parallel programs using an example of a proof of the mutual exclusion property of Peterson's algorithm [137], the correctness of which can be checked automatically using a proof assistant software.

7.4.1. A simple parallel programming language. Consider a simple programming language which allows one to write programs which implement parallel algorithms. A program in this language describes one or more processes which run in parallel. Each process executes sequentially, but different processes can perform some actions simultaneously. Data exchange between different processes can be performed using shared variables (shared memory). A description of each process consists of its header and body. The header includes:

- the process name;
- the list of declarations of local variables, i.e. variables which can be accessed only from the process which declares them;
- the list of declarations of shared variables, i.e. variables, the values of which can be read or changed by any process.

A variable declaration consists of an identifier and a type. A list of shared variable declarations does not necessarily contain all shared variables of the program, but it includes all shared variables

used by the given process. The body of a process is a list of statements. The concrete syntax of a language can be described by a grammar given below, in which the symbols which use italic font shape denote nonterminals; symbols which use bold font denote terminals; $\{x\}$ denotes 0 or more repetitions of x ; $[x]$ denotes that x is optional.

```

program ::= {process}
process ::= header { stmt_list }
header ::= process pname : [local_def] [shared_def]
local_defs ::= local (lvardecl {; lvardecl})
shared_defs ::= shared (svardecl {; svardecl})
lvardecl ::= loc_id : type
svardecl ::= sh_id : type
stmt_list ::= statement {; statement}
statement ::= [label :]
                | [label :] loc_id ::= expr
                | [label :] if bexpr then stmt_list else stmt_list
                               end
                | [label :] while bexpr do stmt_list end
                | [label :] loc_id ::= sh_id
                | [label :] sh_id ::= expr
                | [label :] wait wc_list do stmt_list end
wc_list ::= waitcond { || waitcond }
waitcond ::= sh_id = expr | sh_id != expr
bexpr ::= expr = expr | expr != expr
expr ::= loc_id | boolean | natural
type ::= bool | nat
boolean ::= true | false
natural ::= 0 | 1 | 2 | ...

```

In this grammar the nonterminals *pname*, *label*, *loc_id*, *sh_id* are not defined. Informally, they denote the process names, labels of statements, local identifiers, and shared identifiers respectively. We assume that the sets of permissible local identifiers and shared identifiers are disjoint.

The language includes the following constructs: sequential execution (**;**), the assignment (**:=**), the “if” operator (**if**), the loop operator (**while**). There is also the empty operator which does not perform any action. Any statement can have a label. Label does not change its action.

There are 3 types of assignment operators: assignment of the value of an expression which can contain constants and can reference local variables to a local variable; assignment of the value of a local variable to a shared variable; assignment of the value of a shared variable to a local variable.

The “if” operator and the loop operator depend on Boolean expressions (*bexpr*) which have the form of an equality or of inequality of two expressions which contain constants and can reference local variables.

Moreover, there is a **wait** operator which includes a list of Boolean expressions which denote condition predicates (*wc_list*), and a statement list (*stmt_list*) called the body of the **wait** operator. Informally, the body of **wait** is executed when at least one of the condition predicates becomes true.

A syntactically valid program in this language must satisfy several additional conditions: the names of processes must be distinct; local (shared) variables declared in the header of the same process must be distinct; each local (shared) variable used in the body of a process must be declared in the header of this process; if a shared variable is declared in the header of several processes, its type must be same in all these declarations; types of values of the left-hand side and the right-hand side of an assignment, equality or inequality must be the same; each local variable must be initialized before use.

A program which implements Peterson’s algorithm for two processes *p1*, *p2* and guarantees that the processes cannot simultaneously reach the critical sections denoted by the labels *critical_section_p1* and *critical_section_p2*, can be described as follows:

```
process p1: shared (flag1: bool; flag2:
bool; turn: bool)
{
```

```

    flag1 := true;
    turn := false;
    wait flag2 = false || turn = true do
        critical_section_p1:
    end;
    flag1 := false
}
process p2: shared (flag1: bool;
                    flag2: bool;
                    turn: bool)
{
    flag2 := true;
    turn := true;
    wait flag1 = false || turn = false
    do
        critical_section_p2:
    end;
    flag2 := false
}

```

7.4.2. Formalization of Peterson's algorithm and the proof of the mutual exclusion property. Let us give a formal semantics and a proof of the mutual exclusion property of Peterson's algorithm in Mizar system [115]. Mizar system is based on the classical first-order logic and Tarski-Grothendiek axiomatic set theory [115]. The users of the system can write articles with mathematical definitions and proofs in a special formal language called the Mizar language. Correctness of proofs written in this language can be checked automatically by a software. Mizar includes a large library of formalized notions and proofs from different areas of mathematics which is called Mizar Mathematical Library. A description of the Mizar language can be found in [115], and a description and comparison of other computer proof assistant systems can be found in [168].

Formalization in Mizar system, given below, is described in [83], and is based on the event semantics proposed in [2]. The text of the latest version of the formalization with full proofs is available in [54].

The formalization introduces the notions of an event structure (`Events_structure`) of a parallel program with shared memory, which is parameterized by an algebraic structure on values, which can be stored in the shared memory (`Values`). In the case of the Peterson's algorithm, it is sufficient that the shared variables can store two distinct values, so in the formalization one can restrict attention to value structures with two distinguished values (constants) `True`, `False`. Such a structure is called `Values_with_True_and_False`. An event structure is a multisorted algebraic structure with the following sorts:

- events which can occur during a program execution (`events`);
- processes (`processes`);
- shared variable names which can be used by the processes (`locations`);
- program execution traces (`traces`), which describe sequences of actions which can be performed during a program execution.

The main relation in this structure is the binary precedence relation on the set of events ($\leq$), which turns the set of events into a linear preorder (`LinearPreorder`): $e_1 \leq e_2$ means that an event e_1 in a program execution (non-strictly) precedes an event e_2 . In the general case it is possible that $e_1 \leq e_2$ and $e_2 \leq e_1$. In this case it is assumed that the events e_1, e_2 are simultaneous.

Besides, an event structure includes:

- the functions `procE`, `traceE` on processes and traces respectively, which take values in the set of events;
- the functions `readE`, `writeE` on names of shared variables, which take values in the set of events;
- a partial function `value_of` on events which takes values in the carrier set of the `Values` structure.

Event structures can be used to formalize and prove the mutual exclusion property of Peterson's algorithm: in the event structures which satisfy certain conditions (axioms) which express general properties of parallel programs and the properties of the shared memory, and also satisfy additional properties which express that the

actions of processes follow Peterson's algorithm, the events which correspond to execution of the critical sections cannot be simultaneous.

The definition of the event structure (Events_structure) in Mizar language is given below. Before the main text of the definition (struct Events_structure) there are several auxiliary definitions.

```

definition
  struct (1-sorted)
    Values_with_True_and_False
    (#
      carrier -> set,
      True -> Element of the carrier,
      False -> Element of the carrier
    #);
end;

definition
  let A be Values_with_True_and_False;
  attr
    A is consistent
  means
    the True of A <> the False of A;
end;

registration
  cluster consistent for
    Values_with_True_and_False;
  existence
  proof
    ...
  end;
end;

definition
  mode

```

```

        Values_with_Bool
    is
        consistent
        Values_with_True_and_False;
        correctness;
    end;

definition
    let A be RelStr;
    attr
        A is strongly_connected
    means
        the InternalRel of A
        is_strongly_connected_in the carrier
        of A;
    end;

registration
    cluster non empty reflexive transitive
    strongly_connected for RelStr;
    existence
    proof
        ...
    end;
end;

definition
    mode
        LinearPreorder
    is
        reflexive transitive
        strongly_connected RelStr;
        correctness;
    end;

definition
    let Values be Values_with_Bool;
    struct Events_structure over Values

```

```

(#
  events -> non empty LinearPreorder,
  processes -> non empty set,
  locations -> non empty set,
  traces -> non empty set,
  procE -> Function of
    the processes,
    bool the carrier of the
    events,
  traceE -> Function of
    the traces,
    bool the carrier of the
    events,
  readE -> Function of
    the locations,
    bool the carrier of the
    events,
  writeE -> Function of
    the locations,
    bool the carrier of the
    events,
  value_of -> PartFunc of
    the carrier of the
    events,
    the carrier of Values
#);
end;

```

Below, we introduce a set of short notations for the types of elements of the carriers of the event structure: if S is an event structure, then $\text{Process of } S$, $\text{Event of } S$, $\text{EventSet of } S$, $\text{Location of } S$, $\text{Trace of } S$ are the types of processes, events, sets of events, shared variable names, and traces of S respectively.

```

definition
  let Values be Values_with_Bool,

```

```

        S be Events_structure over Values;
mode
    Process of S
is
    Element of the processes of S;
mode
    Event of S
is
    Element of the carrier of the
    events of S;
mode
    EventSet of S
is
    Subset of the carrier of the
    events of S;
mode
    Location of S
is
    Element of the locations of S;
mode
    Trace of S
is
    Element of the traces of S;
end;

```

For convenience, below we introduce a set of identifiers which are used later to denote values in the shared memory (a, a1, a2), processes (p, p1, p2), shared variable names (x, x1, x2), traces (tr, tr1, tr2), events (e, e0-e5) and an event set (E).

```

reserve Values for Values_with_Bool;
reserve a, a1, a2 for Element of the
carrier of Values;
reserve S for Events_structure over
Values;
reserve p, p1, p2 for Process of S;
reserve x, x1, x2 for Location of S;

```



```

reserve tr, tr1, tr2 for Trace of S;
reserve e, e0, e1, e2, e3, e4, e5 for
Event of S;
reserve E for EventSet of S;

```

Also, below we introduce the following notations:

- e reads x means that in the event e (in some process of the program during some execution) the program reads the shared variable x ;
- e writesto x means that in the event e the program writes to the shared variable x ;
- E reads x means that the set of events E contains an event during which the program reads x ;
- E writesto x means that the set of events E contains an event during which the program writes a value to x ;
- e in tr means that the event e occurs during an execution of the program which corresponds to the trace tr ;
- e in p means that the event e occurs in the process p during some execution of the program;
- $value\ e$ denotes the value which is read from or written to a shared variable by the program in the event e , if e is an event during which the program performs a one-time read or write access to some shared variable;
- e in p, tr means that the event e occurs in the process p during an execution of the program which corresponds to the trace tr ;
- e writesto x, a means that in the event e the program writes the value a to x ;
- e reads x, a means that in the event e the program reads the value a from x .

```

definition
  let Values, S, e, x;
  pred
    e reads x
  means

```

```

        e in (the readE of S) . x;
    pred
        e writesto x
    means
        e in (the writeE of S) . x;
end;

definition
    let Values, S, x, E;
    pred
        E reads x
    means
        ex e st e in E & e reads x;
    pred
        E writesto x
    means
        ex e st e in E & e writesto x;
end;

definition
    let Values, S, e, tr;
    pred
        e in tr
    means
        e in (the traceE of S) . tr;
end;

definition
    let Values, S, e, p;
    pred
        e in p
    means
        e in (the procE of S) . p;
end;

definition
    let Values, S, e;
    func

```

```

        value e
    equals
        (the value_of of S) . e;
    correctness;
end;

definition
    let Values, S, e, p, tr;
    pred
        e in p, tr
    means
        e in p & e in tr;
end;

definition
    let Values, S, e, x, a;
    pred
        e writesto x, a
    means
        e writesto x & value e = a;
    pred
        e reads x, a
    means
        e reads x & value e = a;
end;

```

Below we define some properties of the event structure. An event structure is

- process-complete, if each event occurs in some process;
- process-ordered, if any two distinct events in one process cannot be simultaneous;
- read-write-ordered, if any two distinct events which entail (read or write) access to the same shared variable are not simultaneous;
- read-write-consistent, if for any event e which occurs in an execution which corresponds to a trace tr , such that

during e the program reads the value a from x , there exists another event e_0 , which precedes it in the same trace, during which the program writes the value a to x , and, moreover, each event which occurs in an execution which correspond to tr , which precedes e and entails write access to x , also precedes e_0 ;

- read-write-exclusive, if no event can entail both read access and write access to some shared variable(s);
- consistent, if it has 5 above mentioned properties.

definition

```

let Values, S;
attr
    S is process-complete
means
    for  $tr, e$  st
         $e$  in  $tr$ 
    holds
         $ex\ p\ st\ e\ in\ p$ ;
attr
    S is process-ordered
means
    for  $p, e_1, e_2$  st
         $e_1$  in  $p$  &
         $e_2$  in  $p$ 
    holds
        ( $e_1 \leq e_2$  &  $e_2 \leq e_1$  implies
          $e_1 = e_2$ );
attr
    S is read-write-ordered
means
    for  $x, e_1, e_2$  st
        ( $e_1$  reads  $x$  or  $e_1$  writesto  $x$ )
        &
        ( $e_2$  reads  $x$  or  $e_2$  writesto  $x$ )
    holds
        ( $e_1 \leq e_2$  &  $e_2 \leq e_1$  implies
          $e_1 = e_2$ );

```

```

attr
  S is read-write-consistent
means
  for tr, x, e, a st
    e in tr & e reads x &
    value e = a
  holds
    ex e0 st
      e0 in tr & e0 < e & e0
      writesto x & value e0 = a &
      for e1 st
        e1 in tr & e1 <= e & e1
        writesto x
      holds
        e1 <= e0;
attr
  S is read-write-exclusive
means
  for e, x1, x2 holds
    not (e reads x1 & e writesto
        x2);
end;

definition
  let Values, S;
  attr
    S is consistent
  means
    S is process-complete process-
    ordered read-write-ordered
      read-write-consistent read-
      write-exclusive;
end;

```

Below we check that consistent event structures exist and give a name to such structures. More specifically, such structures are called `Distributed_system_with_shared_memory`.

```

registration
  let Values;
  cluster consistent for
    Events_structure over Values;
  existence
  proof
    ...
  end;
end;

definition
  let Values;
  mode

Distributed_system_with_shared_memory of
Values
is
consistent Events_structure over Values;
  correctness;
end;

```

We will consider that consistency of an event structure holds for all parallel programs, i.e. this will be our assumption about the computational model of parallel programs and about the memory model.

Now let us consider consistent event structures. For convenience, below we introduce notations which will be used later for processes (p , p_1 , p_2), shared variable names (x , x_1 , x_2 , $flag_1$, $flag_2$, $turn$), traces (tr , tr_1 , tr_2), events (e , e_0 – e_5) and a set of events (E) in a consistent event structure.

```

reserve DS for
Distributed_system_with_shared_memory
of Values;
reserve  $p$ ,  $p_1$ ,  $p_2$  for Process of DS;
reserve  $x$ ,  $x_1$ ,  $x_2$ ,  $flag_1$ ,  $flag_2$ ,  $turn$ 
for Location of DS;

```

```

reserve tr, tr1, tr2 for Trace of DS;
reserve e, e0, e1, e2, e3, e4, e5 for
Event of DS;
reserve E for EventSet of DS;

```

Below we introduce the following notations:

- $e1 \ll e2$ means that the event $e1$ strictly precedes $e2$, i.e. $e1$ precedes $e2$ and is not simultaneous with $e2$;
- $(e1, e2)$ interval denotes the set of events between $e1$ and $e2$, i.e. the set of events such that $e1$ strictly precedes them and such that these events strictly precede $e2$;
- $(e1, e2)$ interval_in (p, tr) denotes the set of events between $e1$ i $e2$ which occur in the process p during a program execution which corresponds to the trace tr .

definition

```

  let Values, DS, e1, e2;
  pred  $e1 \ll e2$  means  $e1 \leq e2$  & not ( $e2 \leq e1$ );
end;

```

definition

```

  let Values, DS, e1, e2;
  func
     $(e1, e2)$  interval -> EventSet of DS
  equals
    {  $e : e1 < e$  &  $e < e2$  };
  correctness
  proof
    ...
  end;
end;

```

definition

```

  let Values, DS, e1, e2, p, tr;
  func
     $(e1, e2)$  interval_in  $(p, tr)$  ->

```

```

      EventSet of DS
equals
  { e : e1 < e & e < e2 & e in p,
    tr };
correctness
proof
  ...
end;
end;

```

Below we formulate several auxiliary statements.

```

theorem (e1,e2) interval_in (p,tr) c=
(e1,e2) interval
proof
  ...
end;

```

```

theorem thLinPreordEvents: e1 <= e2 or
e2 <= e1
proof
  ...
end;

```

```

theorem e in p,tr & e1 < e & e < e2
implies e in (e1,e2) interval_in (p,tr);

```

```

theorem e1 < e2 implies e1 <= e2 by
ORDERS_2:def 6;

```

```

theorem thEvStrictPrec: e1 in p & e2 in
p & e1 < e2 implies e1 << e2
proof
  ...
end;

```

```

theorem e1 in p,tr & e2 in p,tr & e1 <
e2 implies e1 << e2

```



```

by thEvStrictPrec;

theorem e1 << e2 implies e1 < e2 by
ORDERS_2:def 6;

theorem e1 in p & e2 in p implies e1 =
e2 or e1 << e2 or e2 << e1
by thLinPreordEvents, thEvStrictPrec,
ORDERS_2:def 6;

theorem thEvTrans: e1 <= e2 & e2 <= e3
implies e1 <= e3
proof
...
end;

theorem thEvStrictTrans1: e1 <= e2 & e2
<< e3 implies e1 << e3 by thEvTrans;

theorem thEvStrictTrans2: e1 << e2 & e2
<= e3 implies e1 << e3 by thEvTrans;

theorem e1 << e2 & e2 << e3 implies e1
<< e3 by thEvTrans;

```

Below we introduce the notation for the relation of simultaneity of two events (e_1, e_2 are_simultaneous) and for the complement of this relation (e_1, e_2 are_not_simultaneous).

```

definition
  let Values, DS, e1, e2;
  pred e1, e2 are_simultaneous means e1
  <= e2 & e2 <= e1;
end;

notation

```

```

    let Values, DS, e1, e2;
    antonym e1, e2 are_not_simultaneous
    for e1, e2 are_simultaneous;
end;

```

Below we formulate auxiliary statements about the properties of the strict precedence relation.

```

theorem not e1, e2 are_simultaneous
implies e1 << e2 or e2 << e1
by thLinPreordEvents;

theorem lemwbefr:
  e1 in tr & e2 in tr &
  e1 reads x,a1 & e2 reads x,a2 & e1<=e2
  & a1<>a2
implies
  ex e st
    e in tr & e1 << e & e << e2 & e
    writesto x,a2
proof
...
end;

```

Now let us consider among all consistent event structures those which correspond to the behavior of programs which implement Peterson's algorithm. To describe such structures, let us introduce a special predicate

```

e
is_Peterson_critical_section_with_respec
t_to p,x1,x2,turn,a1,a2,tr

```

which takes the true value, if e is an event, such that three other events e_1 , e_2 , e_3 precede e in the process p and an execution which corresponds to the trace tr , where $e_1 < e_2$, $e_2 < e_3$, $e_3 < e$, and

- during e_1 the program writes the value `True` to the shared variable x_1 ;
- between e_1 and e in p and tr there is no event which entails write access to x_1 ;
- during e_2 the program writes the value `a2` to the shared variable `turn`;
- between e_2 and e in p and tr there is no event which entails write access to `turn`;
- during e_3 the program either reads the value `False` from the shared variable x_2 , or reads the value `a1` from the shared variable `turn`.

```

definition
  let Values, DS, p, tr, e, x1, x2,
      turn, a1, a2;
  pred e
  is_Peterson_critical_section_with_
    respect_to
    p,x1,x2,turn,a1,a2,tr means
    ex e1, e2, e3 st
      e1 in p,tr & e2 in p,tr & e3
      in p,tr &
      e1 < e2 & e2 < e3 & e3 < e &
      e1 writesto x1,the True of
      Values &
      not (e1,e) interval_in (p,tr)
      writesto x1 &
      e2 writesto turn,a2 &
      not (e2,e) interval_in (p,tr)
      writesto turn &
      (e3 reads x2,the False of
      Values or e3 reads turn,a1);
end;

```

In accordance with Peterson's algorithm for 2 processes, a program which implements has two processes p_1 , p_2 and shared Boolean variables `flag1`, `flag2`, `turn`.

For any event e in the critical section of p_1 there exist three events in p_1 which (strictly) precede it and occur in the following order:

- 1) the program writes the value `True` to the shared variable `flag1`,
- 2) the program writes the value `False` to the shared variable `turn`,
- 3) the program reads the value `False` from the shared variable `flag2`, or reads the value `True` from the shared variable `turn`.

Moreover,

- between the 1st event and e in p_1 there is no event which entails write access to `flag1`;
- between the 2nd event and e in p_1 there is no event which entails write access to `turn`.

For any event e in the critical section of the process p_2 there exist three events in p_2 which (strictly) precede e and occur in the following order:

- 1) the program writes the value `True` to the shared variable `flag2`,
- 2) the program writes the value `True` to the shared variable `turn`,
- 3) the program reads the value `False` from the shared variable `flag2`, or reads the value `False` from the shared variable `turn`.

Moreover,

- between the 1st event and e in p_2 there is no event which entails write access to `flag2`;
- between the 2nd event and e in p_2 there is no event which entails write access to `turn`.

Thus, in a (consistent) event structure DS of a program which implements Peterson's algorithm the following statements have to hold:

- 1) if e is an event in a critical section of p_1 , then

```
e
is_Peterson_critical_section_with_
respect_to
p1, flag1, flag2, turn, the False of
Values, the True of Values, tr
```

2) if e is an event in a critical section of $p2$, then

```
e is_Peterson_critical_section_with_
_respect_to
p2, flag2, flag1, turn, the True of
Values, the False of Values, tr
```

We will take these statements as the definition of the notion of an event in critical sections of processes $p1$, $p2$ for a event structure of an implementation of Peterson's algorithm.

Now let us consider how the mutual exclusion property of Peterson's algorithm can be formulated and proven.

Let DS be a consistent event structure. Below we introduce a predicate

```
Es are_Peterson_critical_sections_in tr
```

which, informally, has the following sense: if DS is an event structure of a program which implements Peterson's algorithm, then this predicate is true, if Es consists of events which occur simultaneously in critical sections of both processes of DS during an execution which corresponds to the trace tr .

In more details, this predicate is true, if:

1) in DS there exist processes $p1$, $p2$ such that each process of DS is either $p1$ or $p2$.

2) in DS there exist shared variable $flag1$, $flag2$, $turn$ such that

- the process $p1$ does not write to $flag2$ and does not write the value `False` to $turn$;

- the process p2 does not write to flag1 and does not write the value True to turn;
- each element of Es belongs to the critical section of p1;
- each element of Es belongs to the critical section of p2.

The definition in Mizar language is given below.

```

definition
  let Values, DS, tr;
  let Es be set;
  pred Es
are_Peterson_critical_sections_in tr means
  ex p1, p2 st
    ((for p being Process of DS
holds p = p1 or p = p2) &
      ex flag1, flag2, turn st
        (for e st e in p1, tr holds
          not e writesto flag2 &
          not e writesto turn, the
False of Values) &
          (for e st e in p2, tr holds
            not e writesto flag1 &
            not e writesto turn, the
True of Values) &
            for e st e in Es holds
              e
is_Peterson_critical_section_with_
respect_to
              p1, flag1, flag2, turn,
              the False of
Values, the True of Values, tr &
              e
is_Peterson_critical_section_with_
respect_to
              p2, flag2, flag1, turn,
              the True of Values, the
False of Values, tr);
end;
```

Now a variant of the statement about the mutual exclusion property of Peterson's algorithm can be given as follow: if the events e_1 , e_2 in some event structure DS of a program which implements Peterson's algorithm correspond to the executions of the critical sections of both processes of DS during a program execution which corresponds to a trace tr , then either e_1 and e_2 coincide, or they are not simultaneous.

The formulation of this statement in Mizar language is given below.

```
theorem
e1 in tr & e2 in tr & {e1, e2}
are_Peterson_critical_sections_in tr
implies
e1 = e2 or e1 << e2 or e2 << e1
```

The full text of the formal proof of this statement in Mizar language is given in Appendix B. Correctness of this proof can be checked automatically by the Mizar system.

Exercises

1. Apply TermWare.NET [69] for parallelization and optimization of the C# program finding prime numbers. Beginning from the Sequential program, obtain BlockTasksThread, InterleavedTasksThread and InterlockedThread using rewriting rules. Execute all the programs on a multicore processor and build a chart showing the dependence of execution time on the quantity of numbers (at number of threads equal to number of processor cores), and also the chart showing the dependence of execution time on the number of threads (for 50000 numbers).
2. Use TermWare.NET for transforming the coordination constructs in the multithreaded C# program simulating a one-dimensional Brownian motion. Using rewriting rules, make a transition from Sequential program to SmallLockThread, SeparatedLockThread, LoopManipulationThread, BigLockThread, InterlockedThread and InvisibleMovesThread.

Execute all the programs on a multicore processor and build a chart showing the dependence of execution time on number of iterations at a number of particles (threads) equal to the number of processor cores, and also a chart showing the dependence of execution time on number of particles (threads) at 100000 iterations.

3. Apply TermWare.NET for transforming sequential bubble sort C# program based on the scheme *Bubble* (see Subsection 2.1, Example 2.3) into a parallel program based on the scheme *PBubble* (see Subsection 2.3, Example 2.6). Execute the sequential and parallel programs on a multicore processor and compare their execution time.
4. Use TermWare.NET for transforming sequential shuttle sort C# program based on the scheme *Shuttle* (see exercise 2 at the end of Chapter 5) into a parallel program. The parallel should split the input array to subarrays, sort them in parallel and merge the subarrays to a sorted array. Execute the sequential and parallel programs on a multicore processor and compare their execution time.
5. Apply TermWare.NET to optimize the parallel CUDA program for summation of elements of a numeric vector. Execute the obtained versions of programs on GPU and compare their execution time.
6. Use TermWare.NET for parallelization of the Bitonic sort program [18, 47] intended for execution on a GPU. Execute the obtained versions of programs and compare their execution time.
7. Apply TermWare.NET for parallelization of matrix multiplication program intended for execution on a GPU. Execute the obtained versions of programs and compare their execution time.
8. Enter the formal proof described in Subsection 7.4 in the Mizar system and verify its correctness.

REFERENCES

1. A theory of clones and formalized design of programs / A. Doroshenko, G. Tseytlin, O. Yatsenko, L. Zachariya // Proc. Int. Conf. "Concurrency, Specification and Programming" (CS&P'2006), 27–29 September 2006. — Wandlitz, Germany. — 2006. — P. 328–339.
2. *Abraham U.* Proving behavioral properties of distributed algorithms using their compositional semantics / U. Abraham, Ie. Ivanov, M. Nikitchenko // Proc. of the First Int. Seminar Specification and Verification of Hybrid Systems, October 10–12, 2011, Taras Shevchenko National University of Kyiv. — 2011. — P. 9–19.
3. *Akhter S.* Multi-core programming. Increasing performance through software multi-threading / S. Akhter, J. Roberts. — Intel Press, 2006. — 336 p.
4. *Alalfi M.* SQL2XMI: Reverse engineering of UML-ER diagrams from relational database schemas / M. Alalfi, J. R. Cordy, T. R. Dean // Proc. 15th Working Conference on Reverse Engineering (WCRE 2008), Antwerp, Belgium, October 2008. — P. 187–191.
5. Algebra-algorithmic models and methods of parallel programming / P. I. Andon, A. Yu. Doroshenko, K. A. Zhreb, O. A. Yatsenko. — Kyiv: Akademperiodyka, 2018. — 192 p.
6. *Andon P. I.* Programming high-performance parallel computations: formal models and graphics processing units / P. I. Andon, A. Yu. Doroshenko, K. A. Zhreb // Cybernetics and Systems Analysis. — 2011. — Vol. 47, Issue 4. — P. 659–668.
7. *Anisimov A. V.* Programming of parallel processors in control spaces / A. V. Anisimov, P. P. Kulyabko // Cybernetics. — 1984. — Vol. 20, Issue 3. — P. 404–418.
8. APS and IMS are best for rewriting and modelling [Online]. — Available from : <http://apsystems.org.ua>
9. *Arnoldus J.* Repleo: a syntax-safe template engine / J. Arnoldus, J. Bijpost, M. van den Brand // Proc. 6th Int. Confer-

- ence on Generative Programming and Component Engineering. — Salzburg, Austria, October 1–3, 2007. — P. 25–32.
10. Automated generation of parallel programs for graphics processing units based on algorithm schemes / A. Yu. Doroshenko, O. G. Beketov, R. B. Ivaniv et al. // *Problems in Programming*. — 2015. — Issue 1. — P. 19–28. (in Ukrainian)
 11. Automated program design — an example solving a weather forecasting problem [Online] / Doroshenko A., Ivanenko P., Ovdii O., Yatsenko O. // *Open Physics*. — 2016. — Vol. 14, Issue 1. — P. 410–419. — Available from : <https://doi.org/10.1515/phys-2016-0048>
 12. Automated Refactoring of Object Oriented Code into Aspects / D. Binkley, M. Ceccato, M. Harman et al. // *Proc. ICSM 2005, IEEE 21st Int. Conference on Software Maintenance*. — Budapest, Hungary, September 2005. — P. 27–36.
 13. *Baader F.* Term rewriting and all that / F. Baader, T. Nipkow. — Cambridge : Cambridge University Press, 1999. — 316 p.
 14. *Bagge O. S.* Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs / O. S. Bagge, K. T. Kalleberg, M. Haverlaen, E. Visser // *Third IEEE Int. Workshop on Source Code Analysis and Manipulation (SCAM 2003)*. — Amsterdam, The Netherlands, September 2003. — P. 65–74.
 15. *Barney B.* Introduction to parallel computing [Online] / B. Barney. — Available from : https://computing.llnl.gov/tutorials/parallel_comp/#ModelsShared
 16. *Basarab I. A.* Compositional databases / I. A. Basarab, M. S. Nikitchenko, V. N. Redko. — Kyiv: Lybid. — 1992. — 191 p.
 17. *Bird R. S.* Lectures on constructive functional programming / R. S. Bird // *Constructive Methods in Computer Science* (M. Broy, ed.). — NATO ASI Series F. — Springer Verlag, 1993. — Vol. 55. — P. 151–218.
 18. Bitonic sort [Online]. — Available from : <http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm>

19. *Blelloch G. E.* Programming parallel algorithms / G. E. Blelloch // Communications of the ACM. — 1996. — Vol. 39, Issue 3. — P. 85–97.
20. *Bouwers E.* Grammar engineering support for precedence rule recovery and compatibility checking / E. Bouwers, M. Bravenboer, E. Visser // Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007). — Braga, Portugal, March 2008. — P. 85–101.
21. *Bravenboer M.* Preventing injection attacks with syntax embeddings. A host and guest language independent approach / M. Bravenboer, E. Dolstra, E. Visser // Generative Programming and Component Engineering (GPCE 2007). — New York, NY, USA, October 2007. — P. 3–12.
22. *Bulyonkov M.* Mixed computation and compilation: new approaches to old problems / M. Bulyonkov // Theor. Comput. Sci. — 1990. — Vol. 71. — P. 209–226.
23. *Buyya R.* High performance cluster computing: programming and applications / R. Buyya. — Vol. 2. — Upper Saddle River, New Jersey: Prentice Hall, 1999. — 664 p.
24. *Carriero N.* Coordination languages and their significance / N. Carriero, D. Gelernter // Communications of the ACM. — 1992. — Vol. 35, Issue 4. — P. 96–107.
25. *Casl* — the common algebraic specification language / Mossakowski T., Haxthausen A.E., Sanella D., Tarlecki A. // Logics of Specification Languages. — Berlin: Springer, 2008. — P. 241–298.
26. *Chandy K. M.* Parallel program design: a foundation / K.M. Chandy, J. Misra. — New York: Addison Wesley, 1988. — 516 p.
27. *Cole M. I.* Algorithmic skeletons: structured management of parallel computation / M. I. Cole. — Cambridge: MIT Press, 1989. — 208 p.
28. *Compiling language definitions: the ASF+SDF compiler* / M. van den Brand, J. Heering, P. Klint, P. Olivier // ACM Transactions on Programming Languages and Systems. — 2000. — Vol. 24. — P. 334–368.

29. Conway's Game of Life [Online]. — Available from : https://en.wikipedia.org/wiki/Conway's_Game_of_Life
30. *Cordy J. R.* The TXL source transformation language / J. R. Cordy // Science of Computer Programming. — 2006. — Vol. 61, Issue 3. — P. 190–210.
31. *Craciun A.* Logic programming [Online] / A. Craciun. — 2012. — 75 p. — Available from : <http://web.info.uvt.ro/~acraciun/lectures/lp/pdf/logicProgrammingNotes.pdf>
32. *Crnkovic I.* Component-based software engineering — new paradigm of software development [Online] / I. Crnkovic, M. Larsson ; Malardalen University, Sweden. — Available from : <http://www.mrtc.mdh.se/publications/0293.pdf>
33. *Czarnecki K.* Generative programming: methods, tools, and applications / K. Czarnecki, U. Eisenecker. — Boston: Addison-Wesley, 2000. — 864 p.
34. Data model reverse engineering in migrating a legacy system to Java / M. Ceccato, T. R. Dean, P. Tonella, D. Marchignoli // Proc. 15th Working Conference on Reverse Engineering (WCRE 2008), Antwerp, Belgium, October 2008. — P. 177–186.
35. *Dershowitz N.* Rewriting. In: Robinson A. J., Voronkov A. Handbook of automated reasoning / N. Dershowitz, D. A. Plaisted. — Elsevier, 2001. — 2128 p.
36. *Dijkstra E. W.* A discipline of programming / E.W. Dijkstra. — New Jersey: Prentice Hall, 1976. — 217 p.
37. *Dolstra E.* Building interpreters with rewriting strategies / E. Dolstra, E. Visser // Workshop on Language Descriptions, Tools and Applications (LDTA 2002). — Grenoble, France, April 2002. — P. 57–76.
38. *Doroshenko A.* A rewriting framework for rule-based programming dynamic applications / A. Doroshenko, R. Shevchenko // Fundamenta Informaticae. — 2006. — Vol. 72, Issue 1–3. — P. 95–108.

39. *Doroshenko A.* Formal facilities for designing efficient GPU programs / A. Doroshenko, K. Zhreb, O. Yatsenko // Proc. Int. Workshop “Concurrency, Specification, and Programming”, CS&P’2010. — Hellenenau, Germany, 27–29 September 2010. — Berlin: Humboldt University, 2010. — P. 142–153.
40. *Doroshenko A.* Formalized development of parallel programs with ontologies and algebra of algorithms / A. Doroshenko, O. Yatsenko // Proc. Int. Workshop “Concurrency: Specification and Programming” (CS&P 2008), Gross Vaeter (near Berlin), 29.09–1.10.2008, Germany. — 2008. — P. 108–119.
41. *Doroshenko A.* Intensional aspects of algebra of algorithmics / A. Doroshenko, G. Tseytlin, O. Yatsenko, L. Zachariya. — Proc. Int. Workshop “Concurrency, Specification and Programming” (CS&P’2007), 27–29 September 2007. — Lagow, Poland. — 2007.
42. *Doroshenko A.* Parallelizing legacy Fortran programs using rewriting rules technique and algebraic program models / A. Doroshenko, K. Zhreb // In ICT in Education, Research, and Industrial Applications, V. Ermolayev, H. C. Mayr, M. Nikitchenko et al. ; Eds. CCIS 347. — Springer Berlin Heidelberg, 2013. — P. 39–59.
43. *Doroshenko A.* Developing and optimizing parallel programs with algebra-algorithmic and term rewriting tools / A. Doroshenko, K. Zhreb, O. Yatsenko // Communications in Computer and Information Science. Information and Communication Technologies in Education, Research, and Industrial Applications. — Springer, Heidelberg, 2013. — Vol. 412. — P. 70–92.
44. *Doroshenko A.* Using ontologies and algebra of algorithms for formalized development of parallel programs / A. Doroshenko, O. Yatsenko // Fundamenta Informaticae. — 2009. — Vol. 93, Issue 1–3. — P. 111–125.
45. *Doroshenko A. Yu.* About the synthesis of Java programs by algebra-algorithmic specifications / A. Yu. Doroshenko, O. A. Yatsenko // Problems in programming. — 2006. — Issue 4. — P. 58–70. (in Russian)

46. *Doroshenko A. Yu.* Algebra-dynamic models for program parallelization / A. Yu. Doroshenko, K. A. Zhreb // Problems in programming. — 2010. — Issue 1. — P. 39–55. (in Russian)
47. *Doroshenko A. Yu.* Development of highly-parallel applications for graphical processing units using rewriting rules / A. Yu. Doroshenko, K. A. Zhreb // Problems in programming. — 2009. — Issue 3. — P. 3–18. (in Russian)
48. *Doroshenko A. Yu.* Mathematical models and methods of organization of highly productive parallel computations. The algebra-dynamic approach / A. Yu. Doroshenko. — Kyiv: Naukova dumka, 2000. — 177 p. (in Russian)
49. *Doroshenko A. Yu.* On complexity and coordination of computation in multithreaded programs / A. Yu. Doroshenko, K. A. Zhreb, O. A. Yatsenko // Problems in programming. — 2007. — Issue 2. — P. 41–55. (in Russian)
50. *Doroshenko A. Yu.* Ontological and algebra-algorithmic tools for automated design of parallel programs for cloud platforms / A. Yu. Doroshenko, O. M. Ovdii, O. A. Yatsenko // Cybernetics and Systems Analysis. — 2017. — Vol. 53, Issue 2. — P. 323–332.
51. *Doroshenko A.* Models and parallel programming abstractions to enhance concurrency of parallel programs / A. Doroshenko, G. Tseytlin // Fundamenta Informaticae. — 2004. — Vol. 60, Issue 1–4. — P. 99–111.
52. *Dybvig R. K.* From macrogeneration to syntactic abstraction / R. K. Dybvig // Higher-Order and Symbolic Computation. — 2000. — Vol. 13. — P. 57–63.
53. *El-Ramly M.* An experiment in automatic conversion of legacy Java programs to C# / M. El-Ramly, R. Eltayeb, H. A. Alla // Proc. IEEE 2006 Int. Conference on Computer Systems and Applications (AICCSA'06), Dubai, March 2006. — P. 1037–1045.
54. Event-based proof of the mutual exclusion property of Peterson's algorithm / Ie. Ivanov, M. Nikitchenko, U. Abraham // Formalized Mathematics. — Vol. 23(4). — 2015. — Walter de Gruyter GmbH, Berlin. — P. 325–331.

55. *Fabry J.* ReLax: implementing KALA over the Reflex AOP kernel / J. Fabry , E. Tanter, T. D'Hondt // Proc. 2nd Workshop on Domain-Specific Aspect Languages (DSAL 2007). — Vancouver, Canada, March 2007. — P. 3–12.
56. *Flynn M. J.* Some computer organizations and their effectiveness / M. J. Flynn // IEEE Transactions on Computers. — 1972. — Vol. 21, Issue 9. — P. 948–960.
57. Flynn's taxonomy [Online]. — Available from : https://en.wikipedia.org/wiki/Flynn%27s_taxonomy#cite_note-1
58. Formal analysis of Java programs in JavaFAN / A. Farzan, F. Chen , J. Meseguer, G. Rosu // Proc. Int. Conf. on Computer Aided Verification, Boston, Mass., 2004. — P. 501-505.
59. *Friedman-Hill E.* Jess in action / E. Friedman-Hill. — Greenwich: Manning Publications Co. — 2003. — 480 p.
60. *Garrido A.* Algebraic semantics of the C preprocessor and correctness of its refactorings. Technical report UIUCDCS-R-2006-2688 [Online] / A. Garrido, J. Meseguer, R. Johnson. — Univ. of Illinois at Urbana-Champaign, 2006. — Available from : <http://www.ideals.illinois.edu/handle/2142/11162>
61. *Garrido A.* Formal specification and verification of Java refactorings / A. Garrido, J. Meseguer // Proc. of 6th IEEE Int. Workshop on Source Code Analysis and Manipulation (SCAM'06). — Philadelphia, PA, USA, 27–29 September 2006. — P. 165–174.
62. *Glushkov V. M.* Algebra. Languages. Programming. 3rd edition / V. M. Glushkov, G. O. Tseytlin, K. L. Yushchenko. — Kyiv: Naukova dumka, 1989. — 376 p. (in Russian)
63. *Glushkov V. M.* Controlling spaces in asynchronous parallel computations / V. M. Glushkov, A. V. Anisimov // Cybernetics. — 1980. — Vol. 16, Issue 5. — P. 633–641.
64. *Glushkov V. M.* Methods of symbolic multiprocessing / V. M. Glushkov, G. O. Tseytlin, K. L. Yushchenko. — Kyiv: Naukova dumka, 1980. — 252 p. (in Russian)
65. *Glushkov V. M.* The automatization of computers design / V. M. Glushkov, Yu. V. Kapitonova, A. A. Letichevskii. — Kyiv: Naukova dumka, 1975. — 232 p. (in Russian)

66. *Gnedenko B. V.* The elements of programming / B. V. Gnedenko, V. S. Korolyuk, K. L. Yushchenko. — Moscow: Fizmatgiz, 1961. — 348 p. (in Russian)
67. Google Disk. IDS toolkit [Online]. — Available from : http://bit.ly/ids_toolkit
68. Google Disk. Synthesis toolkit [Online]. — Available from : http://bit.ly/synthesis_toolkit
69. Google Disk. Termware.NET (TermEdit) [Online]. — Available from : http://bit.ly/termware_dotnet
70. *Grant S.* An Interactive interface for refactoring using source transformation / S. Grant, J. R. Cordy // Proc. REFACE'03, 1st Int. Workshop on Refactoring: Achievements, Challenges, Effects. — Victoria, Canada. — 2003. — P. 30–33.
71. *Groenewegen D.* Declarative access control for WebDSL: combining language integration and separation of concerns / D. Groenewegen, E. Visser // Eighth Int. Conference on Web Engineering (ICWE 2008), July 2008. — P. 175–188.
72. *Guptha S.* Multithreaded programming in a Microsoft Win32* Environment [Online] / S. Guptha. — Available from : <http://eng.harran.edu.tr/~nbesli/SP/senkronizasyon.pdf>
73. *Hamey L.* Implementing a domain-specific language using Stratego/XT / L. Hamey, S. Goldrei // Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA'07). — Braga, Portugal, March 2007. — P. 32–46.
74. *Harris M.* GPGPU: general-purpose computation on GPUs / M. Harris. — In SIG-GRAPH 2005 GPGPU COURSE, 2005. — 277 p.
75. *Harrison J.* Introduction to functional programming / J. Harrison. — Cambridge: Cambridge university, 1997. — 168 p.
76. *Hemel Z.* Code generation by model transformation. A case study in transformation modularity / Z. Hemel, L. C. L. Kats, E. Visser // Theory and Practice of Model Transformations. First Int. Conference on Model Transformation (ICMT 2008). — Heidelberg, July 2008. — P. 183–198

77. *Hockney R. W.* Parallel computers: architecture, programming and algorithms / R. W. Hockney, C. R. Jeshope. — Bristol: Adam Hilger Ltd., 1981. — 423 p.
78. How to design classes [Online] / M. Felleisen, M. Flatt, R. B. Findler et al. — 2011. — Available from : <http://www.ccs.neu.edu/home/matthias/HtDC/htdc.pdf>
79. *Huet G.* Confluent reductions: abstract properties and applications to term rewriting systems / G. Huet // Proc. 18th IEEE Symposium on Foundations of Computer Science. — J. ACM. — Vol. 27, Issue 4. — 1980. — P. 797–821.
80. IBM — Rational Rose Enterprise [Online]. — Available from : <http://www-03.ibm.com/software/products/en/enterprise>
81. Insertion modeling in distributed system design / A. A. Letichevsky, Yu. V. Kapitonova, V. P. Kotlyarov et al. // Problems in programming. — 2008. — Issue 4. — P. 13–38.
82. Insertion modeling system and constraint programming / Letichevsky A., Letichevskiy O., Peschanenko V. et al. // Proc. 7th Int. Conf. ICTERI 2011, Kherson, Ukraine (May 4–7, 2011). — 2011. — P. 51–64.
83. *Ivanov Ie.* Event-based proof of the mutual exclusion property of Peterson’s algorithm [Online] / Ie. Ivanov, M. Nikitchenko, U. Abraham. — Available from : <http://fm.mizar.org/miz/peterson.miz>
84. *Ivanov Ie. V.* Operational semantics of programs over complex-named data / Ie. V. Ivanov // Bulletin of Taras Shevchenko National University of Kyiv. Series Physics & Mathematics. — 2011. — Issue 1. — P. 133–136. (in Ukrainian)
85. Javachecker [Online]. — Available from : http://www.gradsoft.kiev.ua/products/javachecker_eng.html
86. Jess, the rule engine for the Java platform [Online]. — Available from : <http://www.jessrules.com/>
87. *Jing X.* Rule-based automatic software performance diagnosis and improvement / X. Jing // Proc. 7th Int. Workshop on Software and performance (WOSP’08). — Princeton, New Jersey, USA, June 24–26, 2008. — P. 1–12.

88. *Kapitonova Yu. V.* Algebraic programming: methods and tools / Yu. V. Kapitonova, A. A. Letichevsky // *Cybernetics and Systems Analysis*. — 1993. — Vol. 29, Issue 3. — P. 307–312.
89. *Kapitonova Yu. V.* Basic paradigms of programming / Yu. V. Kapitonova, A. A. Letichevsky // *Cybernetics and Systems Analysis*. — 1994. — Vol. 30, Issue 6. — P. 793–806.
90. *Kapitonova Yu. V.* Mathematical theory of computing systems design / Yu. V. Kapitonova, A. A. Letichevsky. — Moscow: Nauka, 1988. — 296 p. (in Russian)
91. *Karp R. M.* A survey of parallel algorithms for shared memory machines / R. M. Karp, V. Ramachandran // *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.). — Amsterdam: North Holland, 1989. — P. 869–941.
92. *Kats L. C. L.* Mixing source and bytecode. A Case for Compilation by Normalization / L. C. L. Kats, M. Bravenboer, E. Visser // *Proc. 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)*. — New York, NY, USA, October 2008. — P. 91–108.
93. *Keller R. M.* A fundamental theorem of asynchronous parallel computation / R. M. Keller // *Lect. Notes Comput. Sci.* — New York : Springer Verlag, 1975. — Vol. 24. — P. 102–112.
94. *Knuth D. E.* Simple word problems in universal algebra / D. E. Knuth, P. B. Bendix // In J. Leech, editor, *Computational Problems in Abstract Algebra*. — Pergamon Press, Oxford, 1970. — P. 263–297.
95. *Kordic V.* Petri Net: theory and applications / V. Kordic. — I-Tech Education and Publishing, 2008. — 534 p.
96. *Kryvolap A.* Extending Floyd-Hoare logic for partial pre- and postconditions / A. Kryvolap, M. Nikitchenko, W. Schreiner // *ICTERI 2013*. — *Communications in Computer and Information Science*. — 2013. — Vol. 412. — Springer, Heidelberg. — P. 355–378.
97. *Letichevsky A. A.* A model for interaction of agents and environments / A. A. Letichevsky, D. R. Gilbert // *Recent trends in algebra's development technique Language*. — 2000. — P. 311–329 p.

98. *Letichevsky A. A.* A rewriting machine and optimization of strategies of term rewriting / A. A. Letichevsky, V. V. Khomenko // *Cybernetics and Systems Analysis*. — 2002. — Vol. 38, Issue 5. — P. 637–649.
99. *Letichevsky A. A.* Algebraic programming is APS system / A. A. Letichevsky, Yu. V. Kapitonova // *Proc. of ISSAC'90*, Tokyo, Japan (August, 1990). — New York: ACM Press, 1990. — P. 68–75.
100. *Letichevsky A. A.* Computations in APS / A. A. Letichevsky, J. V. Kapitonova, S. V. Konozenko // *Theoretical Computer Science*. — 1993. — Vol. 119. — P. 145–171.
101. *LogP: towards a realistic model of parallel computation* / Culler D., Karp R., Patterson D. et al. // *Proc. 4th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPOPP'93)*, San Diego, California, USA (May 19–22, 1993). *ACM SIGPLAN Notices*. — 1993. — Vol. 28, Issue 7. — P. 1–12.
102. *Macroconveyor computations of functions on data structures* / V. M. Glushkov, Yu. V. Kapitonova, A. A. Letichevskii, S. P. Gorlach // *Cybernetics*. — 1981. — Vol. 17, Issue 4. — P. 439–449.
103. *Maltsev A. I.* Algebraic systems / A. I. Maltsev. — Moscow: Nauka, 1970. — 392 p. (in Russian)
104. *Marrer G.* Fundamentals of programming: with object oriented programming / G. Marrer. — Laptop Press, LLC, 2009. — 358 p.
105. *Maude as a wide-spectrum framework for formal modeling and analysis of active networks* / J. Meseguer, P. C. Olveczky, M.-O. Stehr, C. Talcott // *DARPA Active Networks Conference and Exposition (DANCE'02)*. — 2002. — P. 494–510.
106. *Maude System* [Online]. — Available from : <http://maude.cs.uiuc.edu>
107. *Meta-Environment* [Online]. — Available from : <http://www.meta-environment.org>

108. *Miklosko J.* Algorithms, software and hardware of parallel computers / J. Miklosko, V. J. Kotov. — Berlin: Springer, 1984. — 395 p.
109. Model driven architecture: how far have we come, how far can we go? / G. Miller, A. Evans, I. Jacobson et al. // Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications OOPSLA'03. — Anaheim, CA, USA, October 2003. — P. 273–274.
110. Modeling programs over complex-named data by term rewriting systems / Ie. Ivanov, M. Nikitchenko, L. Feraud, M. Strecker // Proc. 7th Int. Scientific Conference TAAPSD'2010. — P. 40–49.
111. MPI: a message-passing interface standard [Online]. — Available from : <http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf>
112. Multilevel structured design of programs: theoretical bases, tools / K. L. Yushchenko, G. O. Tseytlin, V. P. Hrytsay, T. K. Terzyan. — Moscow: Finansy i statistika, 1989. — 208 p. (in Russian)
113. Multi-threaded programming with POSIX threads [Online]. — Available from : <http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/multi-thread.html>
114. *Naudin P.* Algorithmique algébrique avec exercices corrigés / P. Naudin, C. Quitté. — Paris: Masson, 1992. — 469 p. (in French)
115. *Naumowicz A.* A brief overview of Mizar / A. Naumowicz, A. Kornilowicz // Lecture Notes in Computer Science. — Springer, 2009. — Vol. 5674. — P. 67–72.
116. *Nikitchenko M.* Basics of intensionalized data: presets, sets, and nominats / M. Nikitchenko, A. Chentsov // Computer Science Journal of Moldova. — 2012. — Vol. 20, Issue 3 (60). — P. 334–365.
117. *Nikitchenko M.* Programming with nominative data / M. Nikitchenko, Ie. Ivanov // Proc. of CSE2010 Int. Scientific

- Conference on Computer Science and Engineering, 20–22 September 2010. — Kosice, Slovakia. — P. 30–39.
118. *Nikitchenko M. S. Applied logic : a textbook / M. S. Nikitchenko, S. S. Shkilniak. — Kyiv: Publishing house of Taras Shevchenko National University of Kyiv, 2013. — 278 p. (in Ukrainian)*
 119. *Nikitchenko M. S. Composition-nominative approach to clarification of the concept of a program / M. S. Nikitchenko // Problems in programming. — 1999. — Issue 1. — P. 16–31. (in Russian)*
 120. *Nikitchenko M. S. Composition-nominative aspects of address programming / M. S. Nikitchenko // Cybernetics and Systems Analysis. — 2009. — Vol. 45, Issue 6. — P. 864–874.*
 121. *Nikitchenko M. S. Composition-nominative languages of programs with associative denaming / M. S. Nikitchenko, Ie. V. Ivanov / Bulletin of the Lviv University. Series Applied Mathematics and Informatics. — 2010. — Issue 16. — P. 124–139. (in Russian)*
 122. *Nikitchenko M. S. Composition-nominative logics in rigorous development of software systems / M. S. Nikitchenko, V. G. Tymofieiev // Lecture Notes in Business Information Processing. — 2013. — Vol. 137. — P. 140–151.*
 123. *Nikitchenko M. S. Intensional aspects of the concept of a program / M. S. Nikitchenko // Problems in programming. — 2001. — Issue 3–4. — P. 5–13. (in Russian)*
 124. *Nikitchenko M. S. Mathematical logic and theory of algorithms : a textbook / M. S. Nikitchenko, S. S. Shkilniak. — Kyiv: Publishing house of Taras Shevchenko National University of Kyiv, 2008. — 528 p. (in Ukrainian)*
 125. *Nikitchenko M. S. Satisfiability in composition-nominative logics / M. S. Nikitchenko, V. G. Tymofieiev // Central European Journal of Computer Science. — 2012. — Vol. 2, Issue 3. — P. 194–213.*
 126. *Nikitchenko M. S. Stability and monotony of programs regarding structural transformations of data / M. S. Nikitchenko, Ie. V. Ivanov / Problems in programming. — 2010. — Issue 2–3. — P. 58–67. (in Russian)*

127. *Nikitchenko M. S.* A composition-nominative approach to program semantics / M. S. Nikitchenko. — Technical report IT-TR: 1998-020. — Technical University of Denmark. — 1998. — 103 p.
128. *Nikitchenko M. S.* Abstract computability of non-deterministic programs over various data structures / M. S. Nikitchenko // *Lecture Notes in Computer Science*. — 2001. — Vol. 2244. — P. 468–481.
129. Normalizing metamorphic malware using term rewriting // A. Walenstein, R. Mathur, M. Chouchane, A. Lakhotia // *Proc. IEEE 6th Int. Workshop on Source Code Analysis and Manipulation (SCAM 2006)*. — Philadelphia, USA, September 2006. — P. 75–84.
130. NVidia CUDA technology [Online]. — Available from : <http://www.nvidia.com/cuda>
131. NVIDIA. CUDA C Programming Guide [Online]. — Available from : http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
132. *Olmos K.* Strategies for source-to-source constant propagation / K. Olmos, E. Visser // *Workshop on Reduction Strategies (WRS 2002)*. — Copenhagen, Denmark, July 2002. — P. 20.
133. OpenMP Application Programming Interface. Version 4.5 [Online]. — Available from : <http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>
134. *Paige R.* Towards model transformation with TXL / R. Paige, A. Radjenovic // *Proc. Metamodelling for MDA Workshop 2003*, York, U.K., November 2003. — P. 162–177.
135. *Parasyuk I.* Architecture of high-performance fuzzy multi-agent system / I. Parasyuk, S. Yershov. — Third Int. Conference “High Performance Computing” HPC-UA 2013, October 7–11, 2013, Kyiv, Ukraine. — P. 292–296.
136. *Parasyuk I. N.* Categorical approach to the construction of fuzzy graph grammars / I. N. Parasyuk, S. V. Yershov // *Cybernetics and Systems Analysis*. — 2006. — Vol. 43, Issue 4. — P. 570–581.

137. *Peterson G.* Myths about the mutual exclusion problem / G. Peterson // Inf. Process Lett. — Vol. 12. — 1981. — P. 1133–1145.
138. *Petrushenko A.N.* An approach to automation of optimizing transformations of algorithms and programs / A.N. Petrushenko // Cybernetics and Systems Analysis. — 1991. — Vol. 27, Issue 5. — P. 744–753.
139. *Pogorilyy S.D.* A conception for creating a system of parametric design of parallel algorithms and their software implementations / S. D. Pogorilyy, I. Yu. Shkulipa // Cybernetics and System Analysis. — 2009. — Vol. 45, Issue 6. — P. 952–958.
140. Protégé [Online]. — Available from : <http://protege.stanford.edu>
141. Protégé-OWL editor [Online]. — Available from : <http://protegewiki.stanford.edu/wiki/Protege-OWL>
142. Proving correctness of compiler optimizations by temporal logic / D. Lacey, N. D. Jones , E. van Wyk, C. C. Frederiksen // Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02). — Portland, Oregon, January 16–18, 2002. — P. 283–294.
143. Recursive machines and computing technology / V. M. Glushkov, M. B. Ignatiev, V. A. Myasnikov, V. A. Torgashev // Proc. IFIP Congress (Stockholm, 1974). — Amsterdam: North Holland, 1974. — P. 65–71.
144. *Redko V. N.* Compositions of programs and composition programming / V. N. Redko // Programming. — 1978. — Issue 5. — P. 3–24. (in Russian)
145. *Roy C. K.* NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization / C. K. Roy, J. R. Cordy // Proc. ICPC 2008, IEEE Int. Conference on Program Comprehension. — Amsterdam, June 2008. — P. 172–181.
146. *Sannella D.* Foundations of algebraic specification and formal software development / D. Sannella, A. Tarlecki. — Berlin, Heidelberg: Springer, 2012. — 584 p.

147. *Sidiroglou S.* Countering network worms through automatic patch generation / S. Sidiroglou, A. D. Keromytis // IEEE Security & Privacy. — 2005. — Vol. 3, Issue 6. — P. 41–49.
148. *Skillicorn D. B.* Models for practical parallel computation / D. B. Skillicorn // Int. Journal of Parallel Programming. — 1991. — Vol. 20, Issue 2. — P. 133–158.
149. *Skobelev V. G.* On Algebraic properties of nominative data and functions / V. G. Skobelev, M. Nikitchenko, Ie. Ivanov // Communications in Computer and Information Science (CCIS). — Springer, 2014. — Vol. 469. — P. 117–138.
150. Source transformation for concurrency analysis / T. Cassidy, J. R. Cordy, T. Dean, J. Dingel // Proc. LDTA 2005, ACM 5th Int. Workshop on Language Descriptions, Tools and Applications. — Edinburgh, Scotland, April 2005. — P. 26–43.
151. Stratego program transformation language [Online]. — Available from : <http://strategoxt.org>
152. Stratego/XT 0.17. A language and toolset for program transformation / M. Bravenboer, K. T. Kalleberg, R. Vermaas, E. Visser // Science of Computer Programming. Special issue on experimental software and toolkits. — 2008. — Vol. 72, Issue 1–2. — P. 52–70.
153. Systems specification by basic protocols / A. A. Letichevsky, Yu. V. Kapitonova, V. A. Volkov et al. // Cybernetics and Systems Analysis. — 2005. — Vol. 41, Issue 4. — P. 479–493.
154. TermWare system [Online]. — Available from : http://www.gradsoft.com.ua/products/termware_rus.html
155. The syntax definition formalism SDF — reference manual / J. Heering, P. R. Hendriks, P. Klint, J. Rekers // SIGPLAN Not. — 1989. — Vol. 24, Issue 11. — P. 43–75.
156. *Tip F.* A slicing-based approach for locating type errors // F. Tip, T. Dinesh // ACM Trans. Softw. Eng. Methodol. — 2001. — Vol. 10, Issue 1. — P. 5–55.
157. Transformation of legacy software into client/server applications through pattern-based rearchitecturing / S. Hunold, M. Korch, B. Krellner et al. // Proc. 32nd Annual IEEE Int. Conference on Computer Software and Applications

- (COMPSAC'08). — Turku, Finland, July 2008. — P. 303–310.
158. *Tseytin G. S.* Toward assembly programming / G. S. Tseytin // Programming. — 1990. — Issue 1. — P. 23–33.
 159. *Tseytlin G. O.* Introduction to algorithmics / G. O. Tseytlin. — Kyiv: Sfera, 1998. — 310 p. (in Russian)
 160. TXL programming language [Online]. — Available from : <http://www.txl.ca/>
 161. Unified Modeling Language specification [Online]. — Available from : <http://www.uml.org>
 162. *Valiant L. G.* A bridging model for parallel computation / L. G. Valiant // Communications of the ACM. — 1990. — Vol. 33, Issue 8. — P. 103–111.
 163. *Van den Brand M.* Control flow normalization for COBOL/CICS legacy system / M. van den Brand, A. Sellink, C. Verhoef // 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98). — 1998. — P. 1–12.
 164. *Visser E.* WebDSL: a case study in domain-specific language engineering / E. Visser // Int. Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'2007). — Heidelberg, October 2008. — P. 291–373.
 165. *Visser E.* Building program optimizers with rewriting strategies / E. Visser, Z.-e.-A. Benaissa, A. Tolmach // Proc. third ACM SIGPLAN Int. Conference on Functional Programming (ICFP 1998). — Baltimore, Maryland, USA, September 1998. — P. 13–26.
 166. *Waddington D. G.* High fidelity C++ code transformation / D. G. Waddington, B. Yao // Proc. 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005), Electronic Notes in Theoretical Computer Science. — Edinburgh University, UK, April 3, 2005.
 167. *Wang L.* Enhancing security using legality assertions / L. Wang, J. R. Cordy, T. Dean // Proc. IEEE 12th Int. Working Conference on Reverse Engineering (WCRE'05). — Pittsburgh, USA, November 2005. — P. 35–44.

168. *Wiedijk F.* The seventeen provers of the world. Foreword by Dana S. Scott / F. Wiedijk (ed.) // *Lecture Notes in Artificial Intelligence*. — 2006. — Vol. 3600.
169. *Winkler T.* Programming in OBJ and Maude / T. Winkler // *Functional Programming, Concurrency, Simulation and Automated Reasoning*, Int. Lecture Series 1991–1992. — Hamilton : Springer Verlag, 1993. — P. 229–277.
170. *Wirth N.* Algorithms and data Structures / N. Wirth. — Upper Saddle River, NJ: Prentice Hall, 1985. — 288 p.
171. *Yakovyna V.* Discrete and continuous time high-order Markov models for software reliability assessment [Online] / V. Yakovyna, O. Nytrebych // *ICTERI 2015 Int. Workshop on Theory of Reliability for Modern Information Technologies (WS TheRMIT 2015)*. — P. 419–431. — Available from : <http://ceur-ws.org/Vol-1356/>
172. *Yakovyna V.* Software reliability assessment using high-order Markov chains / V. Yakovyna, D. Fedasyuk, O. Nytrebych et al. // *Int. Journal of Engineering Science Invention*. — 2014. — Vol. 3, Issue 7. — P. 1–6.
173. *Yatsenko O.* On application of machine learning for development of adaptive sorting programs in algebra of algorithms // *Proc. Int. Workshop “Concurrency: Specification and Programming” (CS&P’2011)*. — 2011. — P. 577–588.
174. *Yatsenko O.* On parameter-driven generation of algorithm schemes / O. Yatsenko // *Proc. Int. Workshop “Concurrency: Specification and Programming” CS&P’2012*, Berlin, Germany (September 26–28, 2012). — 2012. — P. 428–438.
175. *Yershov A. P.* Introduction to theoretical programming / A. P. Yershov. — Moscow: Nauka, 1977. — 288 p. (in Russian)
176. *Yershov A. P.* On essence of translation / A. P. Yershov // *Programming*. — 1977. — Issue 5. — P. 21–39. (in Russian)
177. *Yushchenko K. L.* Algebraic-grammatical specifications and synthesis of structured program schemas / K. L. Yushchenko, G. O. Tseytlin, A. V. Galushka // *Cybernetics and Systems Analysis*. — 1989. — Vol. 25, Issue 6. — P. 713–727.

178. *Zhereb K. A.* The rule-based software toolkit for automation of development of applications on Microsoft.NET platform / K.A. Zhereb // Upravl. Sistemy i Mashiny. — 2009. — Issue 4. — P. 51–59.

APPENDIX A

Models and source codes of examples

In the following subsections, C# source codes and program models represented in the form of terms for examples considered in Subsections 7.1, 7.2 and 7.3, are given.

A.1. Finding the quantity of prime numbers

The fragment of the sequential program in C# language (the method `SimulationRun`):

```
protected override void SimulationRun()
{
    long number;
    long start;
    long end;
    long stride;
    long factor;
    start = 1;
    end = Number;
    stride = 2;

    if (start == 1)
    {
        start = start + stride;
    };
    number = start;
    while (!(number >= end))
    {
        factor = 3;
        while (!(number % factor) == 0))
        {
            factor = factor + 2;
        };
        if (factor == number)
        {
            Primes[PrimeCount] = number;
```

```

        PrimeCount = PrimeCount + 1;
    };
    number = number + stride;
}
}

```

The fragment (the method `SimulationRun`) of transformed code of the parallel program generated in Termware.NET system (the changes in comparison with the sequential version are highlighted in bold font):

```

protected virtual void RunThreadNum(int
ThreadNum)
{
    long number;
    long start;
    long end;
    long stride;
    long factor;
    start = ThreadNum * BLOCKSIZE + 1;
    end = ThreadNum * BLOCKSIZE +
        BLOCKSIZE;
    stride = 2;
    if (start == 1) {start = start +
        stride;};
    number = start;
    while (!(number >= end))
    {
        factor = 3;
        while (!(number % factor) == 0))
        { factor = factor + 2; };
        if (factor == number)
        {
            lock (this)
            {
                Primes[PrimeCount] = number;
                PrimeCount = PrimeCount + 1;
            }
        }
    }
}

```

```

        };
        number = number + stride;
    }
}

protected override void SimulationRun()
{
    Thread[] threads = new
    Thread[Threads];
    for (int i = 0; i < Threads; i++)
    {
        Thread t = new Thread(new

        ParameterizedThreadStart
        (RunThreadNum) ;
        threads[i] = t;
        t.Start(i);
    }
    for (int i = 0; i < threads.Length;
        i++)
    {
        threads[i].Join();
    }
}

```

A.2. Simulation of one-dimensional Brownian motion

The SmallLockThread program represented in the form of a term:

```

Root (Globals,
    Main (
        THEN (BeginTimeCount,
            THEN (
                InitCriticalSection (Admixture_CS),
                THEN (ReadParameters,
                    THEN (

```

```

        Assignment(NUM_THREADS,
AdmixtureCount),
        THEN(
            Assignment(
                ArrayElement(Crystal, 0),
AdmixtureCount),
            THEN(PrintCrystalState,
                THEN(
                    PARALLEL(
                        Parameter(thread, 0,
                            minus(NUM_THREADS, 1)),
                        CALL(MoveParticle,
                            Parameters(thread))),
                    THEN(
                        WAIT(
ProcessedAll(NUM_THREADS)),
                        THEN(PrintCrystalState,
                            THEN(EndTimeCount,
NIL))))))))) ,
        MoveParticle(
            Parameters(ThreadNum),
            THEN(LocalVariables,
                THEN(InitRandom,
                    THEN(
                        FOR(
                            Parameters(iteration_counter, 0,
                                IterationAmount),
                        THEN(
                            Assignment(nextx,
                                CalculateNewPosition(
                                    minus(CrystalLen, 1), x,
                                    ProbabilityLimit, ThreadNum)),
                        THEN(
                            IF(
                                logical_not(
                                    eq(nextx, x)),
                                THEN(

```

```

CriticalSection(Admixture_CS,
                THEN(
                    Decrement(

ArrayElement(Crystal, x)),
                THEN(
                    Increment(

ArrayElement(Crystal, nextx)), NIL))),
                THEN(
                    Assignment(x, nextx),
NIL))), NIL))), NIL))),
    CalculateNewPosition(
        Parameters(right_margin, pos, limit,
ThreadNum),
    THEN(LocalVariables,
        THEN(
            AssignRandom(r),
            THEN(
                IF(
                    logical_and(
                        greater_eq(r, limit),
                        neq(pos, right_margin)),
                    Assignment(res,
                        plus(pos, 1))),
            THEN(
                IF(
                    logical_and(
                        less(r, limit),
                        neq(pos, 0)),
                    Assignment(res,
                        minus(pos, 1))), NIL))))))

```

The given term contains the following coordination constructs:

1) `PARALLEL(Parameter($var_name, $begin, $end), $body)` is launching of some number ($\$end - \$begin$) of tasks at separate threads. Each task executes the code `$body`, the index of the task is passed as the parameter `$var_name`;

2) `WAIT(ProcessedAll($number))` is waiting for completion of work in all threads;

3) `CriticalSection($id, $body)` is a critical section (the lock block), which corresponds to the object `$id` and contains the code `$body`.

The fragment of the `BigLockThread` program (the `MoveParticle` function):

```
MoveParticle(  
  Parameters(ThreadNum),  
  THEN(LocalVariables,  
    THEN(InitRandom,  
      THEN(  
        FOR(  
          Parameters(iteration_counter, 0,  
            IterationAmount),  
          CriticalSection(Admixture_CS,  
            THEN(  
              Assignment(nextx,  
                CalculateNewPosition(  
                  minus(CrystalLen, 1), x,  
                  ProbabilityLimit, ThreadNum)),  
            THEN(  
              IF(  
                logical_not(  
                  eq(nextx, x)),  
              THEN(  
                THEN(  
                  Decrement(  
                    ArrayElement(Crystal,  
                      x)),  
                THEN(  

```

```

Increment(
ArrayElement(Crystal, nextx)), NIL)),
    THEN(
        Assignment(x, nextx),
        NIL))), NIL))))) , NIL)))))

```

The SeparatedLockThread program:

```

MoveParticle(
    Parameters(ThreadNum),
    THEN(LocalVariables,
        THEN(InitRandom,
            THEN(
                FOR(
                    Parameters(iteration_counter, 0,
                        IterationAmount),
                THEN(
                    Assignment(nextx,
                        CalculateNewPosition(
                            minus(CrystalLen, 1), x,
                            ProbabilityLimit, ThreadNum)),
                THEN(
                    IF(
                        logical_not(
                            eq(nextx, x)),
                    THEN(
                        CriticalSection(
                            ArrayElement(Admixture_CS,
                                x),
                            Decrement(
                                ArrayElement(Crystal,
                                    x))),
                        THEN(
                            CriticalSection(
                                ArrayElement(Admixture_CS, nextx),
                                Increment(

```

```

        ArrayElement(Crystal,
                      nextx))),
    THEN(
        Assignment(x, nextx),
        NIL))))) , NIL))) , NIL)))))

```

The InterlockedThread program:

```

MoveParticle(
    Parameters(ThreadNum),
    THEN(LocalVariables,
        THEN(InitRandom,
            THEN(
                FOR(
                    Parameters(iteration_counter, 0,
                               IterationAmount),
                THEN(
                    Assignment(nextx,
                               CalculateNewPosition(
                                   minus(CrystalLen, 1), x,
                                   ProbabilityLimit, ThreadNum)),
                    THEN(
                        IF(
                            logical_not(
                                eq(nextx, x)),
                            THEN(
                                AtomicDecrement(
                                    ArrayElement(Crystal, x)),
                                THEN(
                                    AtomicIncrement(
                                        ArrayElement(Crystal,
                                                       nextx)),
                                    THEN(
                                        Assignment(x, nextx),
                                        NIL))))) , NIL))))) , NIL)))))

```

The LoopManipulation class (user-defined synchronization construct):

```

public class LoopManipulation
{
    public static object lck=new object();
    public static int blockIterations = 1000;
    public static void
RunSafeLoop(LoopBodyDelegate loopBody,
            int
iterations)
    {
        for (int i = 0; i < iterations /
blockIterations; i++)
        {
            lock (lck)
            {
                for (int j = 0;
                    j < blockIterations; j++)
                { loopBody(); }
            }
        }
        lock (lck)
        {
            for (int i = 0;
                i < iterations%blockIterations;

                i++)
            { loopBody(); }
            {
                loopBody();
            }
        }
    }
}
public delegate void LoopBodyDelegate();

```

The method RunSafeLoop is used instead of a loop with a critical section:

```

LoopManipulation.lck = Admixture_CS;
LoopBodyDelegate loopBody = delegate()
{
    // loop body code
};
LoopManipulation.RunSafeLoop(loopBody,
IterationAmount);

```

The LoopManipulationThread program:

```

MoveParticle(
    Parameters(ThreadNum),
    THEN(LocalVariables,
        THEN(InitRandom,
            THEN(
                Assignment(
                    Identifier(LoopManipulation, lck),
                    Admixture_CS),
                THEN(
DelegateAssignment(LoopManipulationDelegate,
                    loopBody,
                        THEN(
                            Assignment(nextx,
                                CalculateNewPosition(
                                    minus(CrystalLen, 1), x,
                                    ProbabilityLimit, ThreadNum)),
                            THEN(
                                IF(
                                    logical_not(
                                        eq(nextx, x)),
                                    THEN(
                                        Decrement(
                                            ArrayElement(Crystal,
                                                            x)),
                                        THEN(
                                            Increment(
                                                ArrayElement(Crystal,

```

```

                                nextx)),
                                THEN (
                                    Assignment(x, nextx),
                                    NIL))))) , NIL)) ,
THEN (
    CALL (
        Identifier (LoopManipulation,
                    RunSafeLoop),
        Parameters (loopBody,
                    IterationAmount)), NIL)))))

```

A.3. Parallel summation of elements of a numeric vector

The source code of the initial kernel function:

```

extern "C" __global__ void parallelAdd (int
*inData, int *outData)
{
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x +
        threadIdx.x;
    outData[i] = inData[i];
    __syncthreads ();
    for (int s = 1; s < blockDim.x; s *= 2)
    {
        if (tid % (2 * s) == 0)
            outData[i] += outData[i + s];
        __syncthreads();
    }
}

```

The kernel function using a shared memory:

```

extern "C" __global__ void parallelAdd1 (int
*inData, int *outData)
{
    __shared__ int localData[BLOCK_SIZE];

```

```

int tid = threadIdx.x;
int i = blockIdx.x * blockDim.x +
        threadIdx.x;
localData[tid] = inData[i];
__syncthreads();
for ( int s = 1; s < blockDim.x; s *= 2 )
{
    if (tid % (2*s) == 0)
        localData[tid] += localData[tid + s];
    __syncthreads();
}
if (tid == 0)
    outData[blockIdx.x] = localData[0];
}

```

The result of application of all the optimizing transformations:

```

extern "C" __global__ void parallelAdd5 (int
*inData, int *outData)
{
    __shared__ int localData[BLOCK_SIZE];
    int tid = threadIdx.x;
    int i = 2 * blockIdx.x * blockDim.x +
            threadIdx.x;
    localData[tid] = inData[i] +
                    inData[i + blockDim.x];
    __syncthreads();
    for (int s = blockDim.x / 2; s > 32;
        s >>= 1)
    {
        if (tid < s)
            localData [tid] += localData [tid + s];
        __syncthreads ();
    }
    if (tid < 32)
    {
        localData[tid] += localData[tid + 32];
    }
}

```

```
        localData[tid] += localData[tid + 16];  
        localData[tid] += localData[tid + 8];  
        localData[tid] += localData[tid + 4];  
        localData[tid] += localData[tid + 2];  
        localData[tid] += localData[tid + 1];  
    }  
    if (tid == 0)  
        outData[blockIdx.x] = localData[0];  
}
```


APPENDIX B

The text of the formal proof of a theorem in Mizar language

The text of the formal proof of the main theorem of Section 7.4.2 in Mizar language is given below.

```
theorem
  e1 in tr & e2 in tr & {e1, e2}
  are_Peterson_critical_sections_in tr
implies
  e1 = e2 or e1 << e2 or e2 << e1
proof
  assume U0: e1 in tr & e2 in tr;
  assume {e1, e2}
  are_Peterson_critical_sections_in tr;
  then consider p1, p2 such that
  U1: for p being Process of DS holds
    p = p1 or p = p2
  and
  U2: ex flag1, flag2, turn st
    (for e st e in p1, tr holds
    not e writesto flag2 &
    not e writesto turn,
    the False of Values) &
    (for e st e in p2, tr holds
    not e writesto flag1 &
    not e writesto turn,
    the True of Values) &
    for e st e in {e1, e2} holds
    e is_Peterson_critical_section_with_
      respect_to
    p1, flag1, flag2, turn,
    the False of Values, the True of
    Values, tr & e
    is_Peterson_critical_section_with_
    respect_to
```

```

p2, flag2, flag1, turn,
the True of Values,
the False of Values, tr;
consider flag1, flag2, turn such that
U3nw: (for e st e in p1, tr holds
not e writesto flag2 &
not e writesto turn,
the False of Values)
& (for e st e in p2, tr holds
not e writesto flag1 &
not e writesto turn,
the True of Values)
and
U3: for e st e in {e1,e2} holds
e is_Peterson_critical_section_with_
respect_to
p1, flag1, flag2, turn,
the False of Values,
the True of Values, tr & e
is_Peterson_critical_section_with_
respect_to
p2, flag2, flag1, turn,
the True of Values,
the False of Values, tr by U2;
{e1} c= {e1,e2} & {e2} c= {e1,e2} by
ZFMISC_1:36;
then e1 in {e1,e2} & e2 in {e1,e2} by
ZFMISC_1:31;
then U4:
e1 is_Peterson_critical_section_with_
respect_to
p1, flag1, flag2, turn,
the False of Values,
the True of Values, tr & e1
is_Peterson_critical_section_with_
respect_to
p2, flag2, flag1, turn,
the True of Values,

```

```

    the False of Values, tr & e2
is_Peterson_critical_section_with_
respect_to
    p1, flag1, flag2, turn,
    the False of Values,
    the True of Values, tr & e2
is_Peterson_critical_section_with_
respect_to
    p2, flag2, flag1, turn,
    the True of Values,
    the False of Values, tr by U3;

assume Aneq: not e1 = e2;
W1: (e1 in p1 & e2 in p1) or
    (e1 in p2 & e2 in p2)
implies
e1 << e2 or e2 << e1
by Aneq, thLinPreordEvents,
    thEvStrictPrec, ORDERS_2:def 6;
DS is consistent;
then DS is process-complete;
then W2: (ex p being Process of DS st
    e1 in p) &
    (ex p being Process of DS st e2 in p)
by U0;

W3: e1 in p1 & e2 in p2 & e1, e2
are_simultaneous implies
contradiction
proof
assume A0: e1 in p1 & e2 in p2;
assume A0s: e1, e2 are_simultaneous;
consider u1 being Event of DS,
w1 being Event of DS,
r1 being Event of DS
such that
V1: u1 in p1, tr & w1 in p1, tr & r1 in
p1, tr & u1 < w1 & w1 < r1 & r1 < e1 &

```

u1 writesto flag1,the True of Values &
 not (u1,e1) interval_in (p1,tr)
 writesto flag1 &
 w1 writesto turn,the True of Values &
 not (w1,e1) interval_in (p1,tr)
 writesto turn &
 (r1 reads flag2,the False of Values or
 r1 reads turn,the False of Values)
 by U4;
 V1o: u1 << w1 & w1 << r1 & r1 << e1
 by V1, A0, thEvStrictPrec;
 consider u2 being Event of DS,
 w2 being Event of DS,
 r2 being Event of DS
 such that
 V2: u2 in p2,tr & w2 in p2,tr & r2 in
 p2,tr &
 u2 < w2 & w2 < r2 & r2 < e2 &
 u2 writesto flag2,the True of Values &
 not (u2,e2) interval_in (p2,tr)
 writesto flag2 &
 w2 writesto turn,the False of Values &
 not (w2,e2) interval_in (p2,tr)
 writesto turn &
 (r2 reads flag1,the False of Values or
 r2 reads turn,the True of Values)
 by U4;
 V2o: u2 << w2 & w2 << r2 & r2 << e2
 by V2, A0, thEvStrictPrec;
 RR1: r1 <= r2 implies contradiction
 proof
 assume D0: r1 <= r2;
 D01: u1 << r2 & r2 << e1
 by V1o,thEvTrans,D0,A0s,V2,
 A0,thEvStrictPrec,thEvStrictTrans2;
 D1: r2 in tr & r2 reads turn,the True
 of Values
 proof

```

r2 reads flag1,the False of Values
implies
contradiction
proof
assume Y1: r2 reads flag1,
the False of Values;
DS is consistent;
then DS is read-write-consistent;
then consider r2w being Event of DS
such that
Y2: r2w in tr & r2w < r2 & r2w
writesto flag1 &
value r2w = the False of Values &
for e st e in tr & e <= r2 & e
writesto flag1
holds e <= r2w by Y1,V2;
u1 = r2w
proof
DS is consistent;
then DS is process-complete;
then ex p st r2w in p by Y2;
then TA01: r2w in p1,tr or r2w in
p2,tr
by U1,Y2;
TA02: u1 < r2w & r2w < e1
implies
r2w in (u1,e1) interval_in (p1,tr)
by U3nw,Y2,TA01;
TB01: r2w << e1
by D01,thEvStrictTrans1,ORDERS_2:def
6,Y2;
u1 in p1,tr & r2w in p1,tr & not (u1
<< r2w)
by
TB01, Y2, TA02, TA01, U3nw, V1,
ORDERS_2:def 6;
then u1 = r2w or r2w << u1
by thLinPreordEvents, thEvStrictPrec,

```

```

ORDERS_2: def 6;
hence thesis by Y2,V1,D01;
end;
then value r2w = the True of
Values & Values
is consistent
by V1;
hence contradiction by Y2;
end;
hence thesis by V2;
end;
D2: r1 reads flag2, the False of Values
proof
r1 reads turn, the False of Values
implies
contradiction
proof
assume Y1: r1 reads turn, the False of
Values;
Values is consistent;
then consider r2w being Event of DS
such that
Y2: r2w in tr & r1 << r2w & r2w << r2
&
r2w writesto turn, the True of Values
by lemwbefr,V1,D1,D0,Y1;
Y3: r2w in p1,tr
proof
DS is consistent;
then DS is process-complete;
then ex p st r2w in p by Y2;
then (r2w in p1,tr or r2w in p2,tr) &
r2w writesto turn, the True of Values &
Values is consistent by U1,Y2;
hence thesis by U3nw;
end;
r2w << r2 & r2 <= e2 by ORDERS_2: def
6,Y2,V2;

```

```

then IA0: r2w << e2 by thEvTrans;
IB0: w1 <= r1 & r1 << r2w by Y2, V1,
ORDERS_2:def 6;
w1 < r2w & r2w < e1
by thEvTrans,A0s,IA0,IB0,ORDERS_2:def
6;
then r2w in (w1,e1) interval_in
(p1,tr) by Y3;
hence contradiction by V1,Y2;
end;
hence thesis by V1;
end;
DS is consistent;
then DS is read-write-consistent;
then consider wr2 be Event of DS such
that
H1: wr2 in tr & wr2 < r2 & wr2
writesto turn &
value wr2 = the True of Values &
for e1 st
e1 in tr & e1 <= r2 & e1 writesto turn
holds
e1 <= wr2 by D1;
H2: wr2 in p1,tr
proof
DS is consistent;
then DS is process-complete;
then ex p st wr2 in p by H1;
then (wr2 in p1,tr or wr2 in p2,tr) &
wr2 writesto turn,the True of Values &
Values is consistent by U1,H1;
hence thesis by U3nw;
end;
M1: wr2 << r1
proof
r2 << e1
by V2, A0, thEvStrictPrec, A0s,
thEvStrictTrans2;

```

```

then J1: wr2 << e1
by ORDERS_2:def 6,H1,thEvStrictTrans1;
J2: not r1 << wr2
proof
assume r1 << wr2;
then w1 << wr2 & wr2 < e1
by ORDERS_2:def 6, J1, V1,
thEvStrictTrans1;
then w1 < wr2 & wr2 < e1 by
ORDERS_2:def 6;
then wr2 in (w1,e1) interval_in
(p1,tr) by H2;
hence contradiction by V1,H1;
end;
DS is consistent;
then J30: DS is process-ordered;
DS is consistent;
then DS is read-write-exclusive;
then not r1 = wr2 by D2,H1;
hence wr2 << r1 by J2, H2, V1,
thLinPreordEvents,J30;
end;

```

```

Q0: not (u2 <= r1 & r1 <= e2)
proof
assume Q0a: u2 <= r1 & r1 <= e2;
DS is consistent;
then DS is read-write-consistent;
then consider r1w being Event of DS
such that
Y2: r1w in tr & r1w < r1 & r1w
writesto flag2 &
value r1w = the False of Values &
for e st e in tr & e <= r1 & e
writesto flag2
holds e <= r1w by V1,D2;
u2 = r1w
proof

```



```

DS is consistent;
then DS is process-complete;
then ex p st r1w in p by Y2;
then TA01: r1w in p1,tr or r1w in p2,
tr by U1, Y2;
TA02: u2 < r1w & r1w < e2
implies
r1w in (u2,e2) interval_in (p2,tr)
by TA01,U3nw,Y2;
r1w <= r1 & r1 << e1
by V1, A0, thEvStrictPrec, Y2,
ORDERS_2:def 6;
then r1w <= r1 & r1 << e2 by A0s,
thEvTrans;
then u2 in p2,tr & r1w in p2,tr & not
(u2 << r1w)
by Y2,TA01,U3nw,V2,TA02,thEvTrans,
ORDERS_2:def 6;
then u2 = r1w or r1w << u2
by thLinPreordEvents, thEvStrictPrec,
ORDERS_2:def 6;
hence thesis by Y2,V2,Q0a;
end;
then value r1w = the True of Values &
Values is consistent by V2;
hence contradiction by Y2;
end;
M20: r1 << e1 by V1,A0,thEvStrictPrec;
u2 << w2 & w2 <= wr2 & wr2 << r1
by M1, V2, thEvStrictPrec,
H1,ORDERS_2:def 6;
then u2 << wr2 & wr2 << r1 by
thEvTrans;
hence contradiction by
M20,Q0,A0s,thEvTrans;
end;
RR2: r2 <= r1 implies contradiction
proof

```

```

assume D0: r2 <= r1;
D01: u2 << r1 & r1 << e2
by V2o,thEvTrans,D0,A0s,V1,A0,
thEvStrictPrec,thEvStrictTrans2;
D1: r1 in tr & r1 reads turn,the False
of Values
proof
r1 reads flag2,the False of Values
implies
contradiction
proof
assume Y1: r1 reads flag2,the False of
Values;
DS is consistent;
then DS is read-write-consistent;
then consider r2w being Event of DS
such that
Y2: r2w in tr & r2w < r1 & r2w
writesto flag2 &
value r2w = the False of Values &
for e st e in tr & e <= r1 & e
writesto flag2
holds e <= r2w by Y1,V1;
u2 = r2w
proof
DS is consistent;
then DS is process-complete;
then ex p st r2w in p by Y2;
then TA01: r2w in p2,tr or r2w in
p1,tr
by U1,Y2;
TA02: u2 < r2w & r2w < e2
implies
r2w in (u2,e2) interval_in (p2,tr)
by U3nw,Y2,TA01;
TB01: r2w << e2
by D01,thEvStrictTrans1,ORDERS_2:def
6,Y2;

```

u_2 in p_2, tr & r_2w in p_2, tr & not (u_2
 $\ll r_2w$)
 by $TB01, Y_2, TA02, TA01, U3nw, V_2$,
 $ORDERS_2: def\ 6$;
 then $u_2 = r_2w$ or $r_2w \ll u_2$
 by $thLinPreordEvents, thEvStrictPrec$,
 $ORDERS_2: def\ 6$;
 hence thesis by $Y_2, V_2, D01$;
 end;
 then value $r_2w =$ the True of Values &
 Values
 is consistent
 by V_2 ;
 hence contradiction by Y_2 ;
 end;
 hence thesis by V_1 ;
 end;
 $D2: r_2$ reads $flag_1$, the False of Values
 proof
 r_2 reads $turn$, the True of Values
 implies
 contradiction
 proof
 assume $Y_1: r_2$ reads $turn$, the True of
 Values;
 Values is consistent;
 then consider r_2w being Event of DS
 such that
 $Y_2: r_2w$ in tr & $r_2 \ll r_2w$ & $r_2w \ll r_1$
 &
 r_2w writesto $turn$, the False of Values
 by $lemwbefr, V_2, D1, D0, Y_1$;
 $Y_3: r_2w$ in p_2, tr
 proof
 DS is consistent;
 then DS is process-complete;
 then ex p st r_2w in p by Y_2 ;
 then (r_2w in p_2, tr or r_2w in p_1, tr) &

```

r2w writesto turn,the False of Values
&
Values is consistent by U1,Y2;
hence thesis by U3nw;
end;
r2w << r1 & r1 <= e1 by ORDERS_2:def
6,Y2,V1;
then IA0: r2w << e1 by thEvTrans;
IB0: w2 <= r2 & r2 << r2w by Y2, V2,
ORDERS_2:def 6;
w2 < r2w & r2w < e2
by thEvTrans,A0s,IA0,IB0,
ORDERS_2:def 6;
then r2w in (w2,e2) interval_in
(p2,tr) by Y3;
hence contradiction by V2,Y2;
end;
hence thesis by V2;
end;
DS is consistent;
then DS is read-write-consistent;
then consider wr2 be Event of DS such
that
H1: wr2 in tr & wr2 < r1 & wr2
writesto turn &
value wr2 = the False of Values &
for e2 st
e2 in tr & e2 <= r1 & e2 writesto turn
holds
e2 <= wr2 by D1;
H2: wr2 in p2,tr
proof
DS is consistent;
then DS is process-complete;
then ex p st wr2 in p by H1;
then (wr2 in p2,tr or wr2 in p1,tr) &
wr2 writesto turn,the False of Values
&

```

```

Values is consistent by U1,H1;
hence thesis by U3nw;
end;
M1: wr2 << r2
proof
r1 << e2
by
V1, A0, thEvStrictPrec, A0s,
thEvStrictTrans2;
then J1: wr2 << e2
by ORDERS_2:def 6,H1,thEvStrictTrans1;
J2: not r2 << wr2
proof
assume r2 << wr2;
then w2 << wr2 & wr2 < e2
by ORDERS_2:def 6, J1, V2,
thEvStrictTrans1;
then w2 < wr2 & wr2 < e2 by
ORDERS_2:def 6;
then wr2 in (w2,e2) interval_in
(p2,tr) by H2;
hence contradiction by V2,H1;
end;
DS is consistent;
then J30: DS is process-ordered;
DS is consistent;
then DS is read-write-exclusive;
then not r2 = wr2 by D2,H1;
hence wr2 << r2 by J2, H2, V2,
thLinPreordEvents, J30;
end;
Q0: not (u1 <= r2 & r2 <= e1)
proof
assume Q0a: u1 <= r2 & r2 <= e1;
DS is consistent;
then DS is read-write-consistent;
then consider r1w being Event of DS
such that

```

```

Y2: r1w in tr & r1w < r2 & r1w
writesto flag1 &
value r1w = the False of Values &
for e st e in tr & e <= r2 & e
writesto flag1
holds e <= r1w by V2,D2;
u1 = r1w
proof
DS is consistent;
then DS is process-complete;
then ex p st r1w in p by Y2;
then TA01: r1w in p2,tr or r1w in
p1,tr by U1,Y2;
TA02: u1 < r1w & r1w < e1
implies
r1w in (u1,e1) interval_in (p1,tr)
by TA01,U3nw,Y2;
r1w <= r2 & r2 << e2
by
V2,A0,thEvStrictPrec,Y2,
ORDERS_2:def 6;
then r1w <= r2 & r2 << e1 by A0s,
thEvTrans;
then u1 in p1,tr & r1w in p1,tr & not
(u1 << r1w)
by Y2,TA01,U3nw,V1,TA02,thEvTrans,
ORDERS_2:def 6;
then u1 = r1w or r1w << u1
by thLinPreordEvents, thEvStrictPrec,
ORDERS_2:def 6;
hence thesis by Y2,V1,Q0a;
end;
then value r1w = the True of Values &
Values is consistent by V1;
hence contradiction by Y2;
end;
M20: r2 << e1 by V2, A0s, A0,
thEvStrictPrec, thEvStrictTrans2;

```

```

u1 << w1 & w1 <= wr2 & wr2 << r2
by
M1,thEvStrictPrec,V1,H1,
ORDERS_2:def 6;
then u1 << wr2 & wr2 << r2 by
thEvTrans;
hence contradiction by
M20,Q0,thEvTrans;
end;
thus contradiction by
thLinPreordEvents, RR1, RR2;
end;
W4: e1 in p2 & e2 in p1 & e1, e2
are_simultaneous implies
contradiction
proof
assume A0: e1 in p2 & e2 in p1;
assume A0s: e1, e2 are_simultaneous;
consider u1 being Event of DS, w1
being Event of DS,
r1 being Event of DS
such that
V1: u1 in p2,tr & w1 in p2,tr & r1 in
p2,tr &
u1 < w1 & w1 < r1 & r1 < e1 &
u1 writesto flag2,the True of Values &
not (u1,e1) interval_in (p2,tr)
writesto flag2 &
w1 writesto turn,the False of Values &
not (w1,e1) interval_in (p2,tr)
writesto turn &
(r1 reads flag1,the False of Values or
r1 reads turn,the True of Values)
by U4;
V1o: u1 << w1 & w1 << r1 & r1 << e1
by V1, A0, thEvStrictPrec;
consider u2 being Event of DS, w2
being Event of DS,

```

r2 being Event of DS
 such that
 V2: u2 in p1,tr & w2 in p1,tr & r2 in
 p1,tr &
 u2 < w2 & w2 < r2 & r2 < e2 &
 u2 writesto flag1,the True of Values &
 not (u2,e2) interval_in (p1,tr)
 writesto flag1 &
 w2 writesto turn,the True of Values &
 not (w2,e2) interval_in (p1,tr)
 writesto turn &
 (r2 reads flag2,the False of Values or
 r2 reads turn,the False of Values)
 by U4;
 V2o: u2 << w2 & w2 << r2 & r2 << e2
 by V2, A0, thEvStrictPrec;
 RR1: r1 <= r2 implies contradiction
 proof
 assume D0: r1 <= r2;
 D01: u1 << r2 & r2 << e1
 by V1o,thEvTrans,D0,A0s,V2,A0,
 thEvStrictPrec,thEvStrictTrans2;
 D1: r2 in tr & r2 reads turn,the False
 of Values
 proof
 r2 reads flag2,the False of Values
 implies
 contradiction
 proof
 assume Y1: r2 reads flag2,the False of
 Values;
 DS is consistent;
 then DS is read-write-consistent;
 then consider r2w being Event of DS
 such that
 Y2: r2w in tr & r2w < r2 & r2w
 writesto flag2 &
 value r2w = the False of Values &


```

for e st e in tr & e <= r2 & e
writesto flag2
holds e <= r2w by Y1,V2;
u1 = r2w
proof
DS is consistent;
then DS is process-complete;
then ex p st r2w in p by Y2;
then TA01: r2w in p2,tr or r2w in
p1,tr
by U1,Y2;
TA02: u1 < r2w & r2w < e1
implies
r2w in (u1,e1) interval_in (p2,tr)
by U3nw,Y2,TA01;
TB01: r2w << e1
by D01,thEvStrictTrans1,ORDERS_2:def
6,Y2;
u1 in p2,tr & r2w in p2,tr & not (u1
<< r2w)
by
TB01,Y2,TA02,TA01,U3nw,V1,
ORDERS_2:def 6;
then u1 = r2w or r2w << u1
by thLinPreordEvents,
thEvStrictPrec, ORDERS_2:def 6;
hence thesis by Y2,V1,D01;
end;
then value r2w = the True of Values &
Values is consistent by V1;
hence contradiction by Y2;
end;
hence thesis by V2;
end;
D2: r1 reads flag1,the False of Values
proof
r1 reads turn,the True of Values
implies

```

```

contradiction
proof
assume Y1: r1 reads turn,the True of
Values;
Values is consistent;
then consider r2w being Event of DS
such that
Y2: r2w in tr & r1 << r2w & r2w << r2
&
r2w writesto turn, the False of Values
by lemwbefr,V1,D1,D0,Y1;
Y3: r2w in p2,tr
proof
DS is consistent;
then DS is process-complete;
then ex p st r2w in p by Y2;
then (r2w in p2,tr or r2w in p1,tr) &
r2w writesto turn,the False of Values
&
Values is consistent by U1,Y2;
hence thesis by U3nw;
end;
r2w << r2 & r2 <= e2 by ORDERS_2:def
6,Y2,V2;
then IA0: r2w << e2 by thEvTrans;
IB0: w1 <= r1 & r1 << r2w by Y2, V1,
ORDERS_2:def 6;
w1 < r2w & r2w < e1
by thEvTrans,A0s,IA0,IB0,ORDERS_2:def
6;
then r2w in (w1,e1) interval_in
(p2,tr) by Y3;
hence contradiction by V1,Y2;
end;
hence thesis by V1;
end;
DS is consistent;
then DS is read-write-consistent;

```

```

then consider wr2 be Event of DS such
that
H1: wr2 in tr & wr2 < r2 & wr2
writesto turn &
value wr2 = the False of Values &
for e1 st
e1 in tr & e1 <= r2 & e1 writesto turn
holds
e1 <= wr2 by D1;
H2: wr2 in p2,tr
proof
DS is consistent;
then DS is process-complete;
then ex p st wr2 in p by H1;
then (wr2 in p2,tr or wr2 in p1,tr) &
wr2 writesto turn,the False of Values
&
Values is consistent by U1,H1;
hence thesis by U3nw;
end;
M1: wr2 << r1
proof
r2 << e1
by V2,A0,thEvStrictPrec,A0s,
thEvStrictTrans2;
then J1: wr2 << e1
by ORDERS_2:def 6,H1,thEvStrictTrans1;
J2: not r1 << wr2
proof
assume r1 << wr2;
then w1 << wr2 & wr2 < e1
by ORDERS_2:def 6,
J1,V1,thEvStrictTrans1;
then w1 < wr2 & wr2 < e1 by
ORDERS_2:def 6;
then wr2 in (w1,e1) interval_in
(p2,tr) by H2;
hence contradiction by V1,H1;

```

```

end;
DS is consistent;
then J30: DS is process-ordered;
DS is consistent;
then DS is read-write-exclusive;
then not  $r1 = wr2$  by D2,H1;
hence  $wr2 \ll r1$  by
J2,H2,V1,thLinPreordEvents,J30;
end;
Q0: not ( $u2 \leq r1 \ \& \ r1 \leq e2$ )
proof
assume Q0a:  $u2 \leq r1 \ \& \ r1 \leq e2$ ;
DS is consistent;
then DS is read-write-consistent;
then consider  $rlw$  being Event of DS
such that
Y2:  $rlw$  in  $tr$  &  $rlw < r1$  &  $rlw$ 
writesto flag1 &
value  $rlw =$  the False of Values &
for  $e$  st  $e$  in  $tr$  &  $e \leq r1$  &  $e$ 
writesto flag1
holds  $e \leq rlw$  by V1,D2;
 $u2 = rlw$ 
proof
DS is consistent;
then DS is process-complete;
then  $ex \ p \ st \ rlw$  in  $p$  by Y2;
then TA01:  $rlw$  in  $p2, tr$  or  $rlw$  in
 $p1, tr$  by U1,Y2;
TA02:  $u2 < rlw \ \& \ rlw < e2$ 
implies
 $rlw$  in  $(u2, e2)$  interval_in  $(p1, tr)$ 
by TA01,U3nw,Y2;
 $rlw \leq r1 \ \& \ r1 \ll e1$ 
by
V1,A0,thEvStrictPrec,Y2,
ORDERS_2: def 6;
then  $rlw \leq r1 \ \& \ r1 \ll e2$  by A0s,

```

```

thEvTrans;
then u2 in p1, tr & r1w in p1, tr & not
(u2 << r1w)
by Y2, TA01, U3nw, V2, TA02, thEvTrans,
ORDERS_2: def 6;
then  $\bar{u2} = r1w$  or  $r1w << u2$ 
by thLinPreordEvents, thEvStrictPrec,
ORDERS_2: def 6;
hence thesis by Y2, V2, Q0a;
end;
then value r1w = the True of Values &
Values is consistent by V2;
hence contradiction by Y2;
end;
M20: r1 << e1 by V1, A0, thEvStrictPrec;
u2 << w2 & w2 <= wr2 & wr2 << r1
by M1, V2, thEvStrictPrec, H1,
ORDERS_2: def 6;
then  $\bar{u2} << wr2$  &  $wr2 << r1$  by
thEvTrans;
hence contradiction by
M20, Q0, A0s, thEvTrans;
end;
RR2: r2 <= r1 implies contradiction
proof
assume D0: r2 <= r1;
D01: u2 << r1 & r1 << e2
by
V2o, thEvTrans, D0, A0s, V1, A0,
thEvStrictPrec,
thEvStrictTrans2;
D1: r1 in tr & r1 reads turn, the True
of Values
proof
r1 reads flag1, the False of Values
implies
contradiction
proof

```

```

assume Y1: r1 reads flag1,the False of
Values;
DS is consistent;
then DS is read-write-consistent;
then consider r2w being Event of DS
such that
Y2: r2w in tr & r2w < r1 & r2w
writesto flag1 &
value r2w = the False of Values &
for e st e in tr & e <= r1 & e
writesto flag1
holds e <= r2w by Y1,V1;
u2 = r2w
proof
DS is consistent;
then DS is process-complete;
then ex p st r2w in p by Y2;
then TA01: r2w in p1,tr or r2w in
p2,tr
by U1,Y2;
TA02: u2 < r2w & r2w < e2
implies
r2w in (u2,e2) interval_in (p1,tr)
by U3nw,Y2,TA01;
TB01: r2w << e2
by D01,thEvStrictTrans1,ORDERS_2:def
6,Y2;
u2 in p1,tr & r2w in p1,tr & not (u2
<< r2w)
by TB01,Y2,TA02,TA01,U3nw,V2,
ORDERS_2:def 6;
then u2 = r2w or r2w << u2
by thLinPreordEvents, thEvStrictPrec,
ORDERS_2:def 6;
hence thesis by Y2,V2,D01;
end;
then value r2w = the True of Values &
Values

```

```

is consistent
by V2;
hence contradiction by Y2;
end;
hence thesis by V1;
end;
D2: r2 reads flag2,the False of Values
proof
r2 reads turn,the False of Values
implies
contradiction
proof
assume Y1: r2 reads turn,the False of
Values;
Values is consistent;
then consider r2w being Event of DS
such that
Y2: r2w in tr & r2 << r2w & r2w << r1
&
r2w writesto turn,the True of Values
by lemwbefr,V2,D1,D0,Y1;
Y3: r2w in p1,tr
proof
DS is consistent;
then DS is process-complete;
then ex p st r2w in p by Y2;
then (r2w in p1,tr or r2w in p2,tr) &
r2w writesto turn,the True of Values &
Values is consistent by U1,Y2;
hence thesis by U3nw;
end;
r2w << r1 & r1 <= e1 by
ORDERS_2:def 6,
Y2,V1;
then IA0: r2w << e1 by thEvTrans;
IB0: w2 <= r2 & r2 << r2w by Y2, V2,
ORDERS_2:def 6;
w2 < r2w & r2w < e2

```

```

by thEvTrans,A0s,IA0,IB0,
ORDERS_2:def 6;
then r2w in (w2,e2) interval_in
(p1,tr) by Y3;
hence contradiction by V2,Y2;
end;
hence thesis by V2;
end;
DS is consistent;
then DS is read-write-consistent;
then consider wr2 be Event of DS such
that
H1: wr2 in tr & wr2 < r1 & wr2
writesto turn &
value wr2 = the True of Values &
for e2 st
e2 in tr & e2 <= r1 & e2 writesto turn
holds
e2 <= wr2 by D1;
H2: wr2 in p1,tr
proof
DS is consistent;
then DS is process-complete;
then ex p st wr2 in p by H1;
then (wr2 in p1,tr or wr2 in p2,tr) &
wr2 writesto turn,the True of Values &
Values is consistent by U1,H1;
hence thesis by U3nw;
end;
M1: wr2 << r2
proof
r1 << e2
by
V1,A0,thEvStrictPrec,A0s,
thEvStrictTrans2;
then J1: wr2 << e2
by ORDERS_2:def 6,H1,thEvStrictTrans1;
J2: not r2 << wr2

```



```

proof
  assume r2 << wr2;
  then w2 << wr2 & wr2 < e2
  by ORDERS_2:def
  6,J1,V2,thEvStrictTrans1;
  then w2 < wr2 & wr2 < e2 by
  ORDERS_2:def 6;
  then wr2 in (w2,e2) interval_in
  (p1,tr) by H2;
  hence contradiction by V2,H1;
end;
DS is consistent;
then J30: DS is process-ordered;
DS is consistent;
then DS is read-write-exclusive;
then not r2 = wr2 by D2,H1;
hence wr2 << r2 by
J2,H2,V2,thLinPreordEvents,J30;
end;
Q0: not (u1 <= r2 & r2 <= e1)
proof
  assume Q0a: u1 <= r2 & r2 <= e1;
  DS is consistent;
  then DS is read-write-consistent;
  then consider rlw being Event of DS
  such that
  Y2: rlw in tr & rlw < r2 & rlw
  writesto flag2 &
  value rlw = the False of Values &
  for e st e in tr & e <= r2 & e
  writesto flag2
  holds e <= rlw by V2,D2;
  u1 = rlw
  proof
    DS is consistent;
    then DS is process-complete;
    then ex p st rlw in p by Y2;
    then TA01: rlw in p1,tr or rlw in

```

```

p2, tr by U1, Y2;
TA02: u1 < r1w & r1w < e1
implies
r1w in (u1, e1) interval_in (p2, tr)
by TA01, U3nw, Y2;
r1w <= r2 & r2 << e2
by V2, A0, thEvStrictPrec, Y2,
ORDERS_2: def 6;
then r1w <= r2 & r2 << e1 by A0s,
thEvTrans;
then u1 in p2, tr & r1w in p2, tr & not
(u1 << r1w)
by Y2, TA01, U3nw, V1, TA02, thEvTrans,
ORDERS_2: def 6;
then u1 = r1w or r1w << u1
by thLinPreordEvents, thEvStrictPrec,
ORDERS_2: def 6;
hence thesis by Y2, V1, Q0a;
end;
then value r1w = the True of Values &
Values is consistent by V1;
hence contradiction by Y2;
end;
M20: r2 << e1 by V2, A0s, A0,
thEvStrictPrec,
thEvStrictTrans2;
u1 << w1 & w1 <= wr2 & wr2 << r2
by
M1, thEvStrictPrec, V1, H1,
ORDERS_2: def 6;
then u1 << wr2 & wr2 << r2 by
thEvTrans;
hence contradiction by
M20, Q0, thEvTrans;
end;
thus contradiction by
thLinPreordEvents, RR1, RR2;
end;

```

```
      thus thesis by thLinPreordEvents, W1,  
      W2, U1, W3, W4;  
end;
```

