CZWARTA JESIENNA SZKOŁA PTI

WSPÓŁCZESNE KIERUNKI ROZWOJU INFORMATYKI

Organizowana przez POLSKIE TOWARZYSTWO INFORMATYCZNE

Do użytku
wewnętrznego

Mrągowo, 7 — 11 grudnia 1987 r.

CZWARTA JESIEŃNA SZKOŁA PTI

WSPÓŁCZESNE KIERUNKI ROZWOJU INFORMATYKI

Organizowana przez POLSKIE TOWARZYSTWO INFORMATYCZNE

Do użytku
wewnętrznego

Mrągowo, 7 — 11 grudnia 1987 r.

Druk. Wyd. "ALFA" nakł. 300 zam. 2305

S P I S T R E Ś C I

		Str.
1.	Dr Stanisław BUDKOWSKI, doc. Piotr DEMBINSKI (BULL, Francja) **An introduction to ESTELLE: a Specification Language for Distributed Systems**	1
2.	Dr Jacek IRLIK (Uniwerytet Śląski) **Wypowiedź w kwestii ochrony oprogramowania**	37
3.	Doc. Jan MADEY (Uniwersytet Warszawski) **Ockham, Edison, Joyce i inni, czyli o programowaniu współbieżnym**	41
4.	Mgr inż. Barbara OSUCHOWSKA (Wydawnictwa Naukowo-Techniczne) **O informatyce - po polsku**	59
5.	Prof. Andrzej SALWICKI (Uniwersytet Warszawski) **Tworzenie modułów programistycznych w oparciu o abstrakcyjne typy danych**	81
6.	Prof. Łukasz A. TURSKI (Polska Akademia Nauk) **Komputery w fizyce i fizyka w komputerach**	109
7.	Prof. Władysław M. Turski (Uniwersytet Warszawski) **Kanoniczny krok procesu programowania**	141
8.	Doc. Bogusław GAWLIK (Uniwerytet Jagielloński) **Ochrona programów komputerowych na gruncie prawa autorskiego**	169

Spis treści

1. Dr Stanisław WOŹNIAKOWSKI, doc. Piotr ŁUKASIEWICZ, Instytut Transportu
Automatyzacja ruchu kolejowego – specyfikacja wymagań dla wybranych systemów

2. Dr Jacek TRZCIK (Politechnika Śląska)
Wymagania a koszty wybranej optymalizacji

3. Doc. Jan NADZY (Politechnika Gdańska)
Ochrona, Bateria, Zapis i dane, zapis
o przyporządkowaniu współrzędnych

4. Mgr inż. Barbara GOŁĄBOWSKA (Politechnika Łódzka, Instytut Technologii)
O latowaniu – po polsku

5. Prof. Andrzej SKLAVIDI (Politechnika Warszawska)
Tworzenie modelu przepływowych w oparciu
o obserwacje ruchu danych

6. Prof. Łukasz K. TURSKI (Polska Akademia Nauk)
Komputery w fizyce i fizyka a komputerach

7. Prof. Wiesław M. Turski (Politechnika Warszawska)
Komputerowy pomiar prędkości przepływu

8. Doc. Zygmunt SZALIC (Politechnika Świętokrzyska)
Ochrona przyrody środowiskowych na granicy pomiaru
ostrości

SŁOWO WSTĘPNE

Jak powszechnie wiadomo, od dnia narodzin informatyki cena sprzętu komputerowego systematycznie maleje przy jednoczesnym wzroscie jego niezawodności, natomiast na oprogramowanie wydaje się coraz więcej pieniędzy, mimo że jego jakość jest najczęściej daleka od zadowalającej. Ocenia się, że - w globalnych nakładach na informatykę ponoszonych na świecie - oprogramowanie pochłania około 80% ze stałą tendencją wzrostową. Sytuacja ta doczekała się kilka lat temu - słusznie czy niesłusznie - miana "kryzysu oprogramowania" i doprowadziła do energicznych działań zarówno w sferze badań podstawowych, jak i w zakresie budowy różnorodnych narzędzi wspomagających proces tworzenia programów. Z rozwojem tych badań i narzędzi wiąże się nadzieje na przełamanie wielu niedostatków trapiących dzisiaj przemysł produkcji oprogramowania.

Na niniejszy tom składa się siedem wykładów, z czego cztery są poświęcone tematom związanym z wyżej opisanym problemem. Trzy pozostałe dotyczą zagadnień z pogranicza informatyki i, kolejno, fizyki, prawa i języka polskiego.

W grupie czterech wykładów związanych z tworzeniem oprogramowania jest mowa o nowych metodach oraz o wspomagających stosowanie tych metod językach i systemach komputerowych. Wykład Andrzeja Salwickiego dotyczy szeroko obecnie badanej i uprawianej na świecie techniki programowania modularnego. Autor pokazuje, jak - wychodząc od formalnej specyfikacji systemu oprogramowania - można w sposób systematyczny tworzyć jego moduły. Omawiana metoda jest ilustrowana przykładami budowania programów w języku Loglan. Władysław Turski przedstawia kompleksową metodę projektowania i konstruowania dużych systemów oprogramowania. Metoda ta doczekała się już na świecie narzędzia dla zastosowań w postaci bogatego

środowiska programistycznego. Jan Madey poświęcił swój wykład praktycznym aspektom programowania systemów współbieżnych. Jest to temat niezwykle ważny, współbieżność jest bowiem jednym z podstawowych atrybutów współczesnych systemów informatycznych. Innym ważnym atrybutem takich systemów jest ich rozproszenie na wiele współpracujących podsystemów. Dla zorganizowania wymiany informacji między podsystemami tworzy się protokoły komunikacyjne. Są to na ogół bardzo złożone programy o wielu współbieżnie wykonujących się modułach. Stanisław Budkowski, nasz tegoroczny wykładowca zagraniczny, przedstawia w swoim wykładzie specjalistyczny język Estelle służący do specyfikacji protokołów.

Nie jest tajemnicą, że dzisiejsza informatyka - zaprzęgnięta głównie do kieratu masowej biurotyki - narodziła się z zastosowania "szybkich elektronicznych maszyn liczących" do problemów związanych z obliczeniami w fizyce. Obecnie bardzo niewielki procent światowych mocy obliczeniowych jest zaangażowany w takie zastosowania. Mimo to na tych właśnie peryferiach informatycznej codzienności dzieją się stale rzeczy fascynujące. Superkomputery używane do obliczeń naukowych wyznaczają kierunki rozwoju informatyki na równi z innymi jej zastosowaniami. Zagadnieniom tym jest poświęcony wykład przygotowany przez fizyka Łukasza Turskiego.

Problemem nieinformatycznym, ale niezwykle ważnym dla informatyki, jest ochrona praw autorów programów. W Polsce w chwili bieżącej sytuacja autorów programów jest co najmniej niejasna - daleko bardziej niejasna niż w wielu innych krajach europejskich i pozaeuropejskich. Dlatego właśnie temat ten powraca do programu szkół PTI już po raz trzeci. Tym razem przedstawiamy dwugłos przygotowany przez Bogusława Gawlika - prawnika specjalizującego się w zagadnieniach ochrony praw autorskich oraz

Jacka Irlika - informatyka i prawnika w jednej osobie.

Szkoła nie byłaby szkołą, gdyby w jej programie zabrakło lekcji polskiego. Mimo że jesteśmy informatykami, naszym codziennym językiem nie jest ani Pascal, ani Loglan, ani - Boże uchowaj - Basic, ale ojczysty język polski. Niestety, pośpiech, z jakim informatyka wkracza do tego języka, nie zawsze wychodzi jemu i nam na dobre. O problemach poprawnej polszczyzny w informatyce, a także o sprawach terminologicznych, pisze Barbara Osuchowska, informatyk-wydawca.

Warszawa, w październiku 1987 Andrzej Blikle

Jesienna Szkoła PTI
Mrągowo, grudzień 1987 r.

AN INTRODUCTION TO ESTELLE: A SPECIFICATION LANGUAGE FOR DISTRIBUTED SYSTEMS

S. Budkowski - P. Dembiński
BULL S.A.
Corporate Networking and
Communication /DRCG/
Distributed System Architecture
and Standards /ARS/
68, route de Versailles,
78430 Louveciennes, France

Abstract: ESTELLE is a Formal Description Technique, defined within ISO /International Organization for Standardization/ for specification of distributed, concurrent information processing systems. In particular, ESTELLE can be used to describe the services and protocoles of the layers of Open Systems Interconection /OSI/ architecture defined by ISO. Its present ISO status is Draft International Standard /DIS 9074/. The article outlines syntactical and semantical aspects of this description technique.

Submitted for publication in "Computer Networks and ISDN Systems" /invited paper/

1. INTRODUCTION

The problem of specifying distributed systems is much more difficult than that of the classical (sequential) systems. The difficulties are related to the necessity of describing several sequential components which may then cooperate and execute in parallel.

In order to attain a good reliability level of produced software, their development should begin with formal specifications. Only on such specifications one may properly undergo verifications. It is a well known fact that the cost of error correction increases quickly during the software life-cycle. Therefore it is necessary to detect and correct errors during the initial phase of the software development process. This necessity is only reenforced by the design complexity when distributed systems are involved.

Estelle is a language for specifying distributed systems with a particular application in mind, namely that of communication protocols and services. The semantics for Estelle has been formally defined ([8], [18]), justifying the claim that Estelle is a Formal Description Technique.

An Estelle specification is aimed to describe a structure of communicating state automata whose internal actions are Pascal statements.

Estelle is one of the description techniques which all are to serve as means to remove ambiguities from ISO protocol standards which traditionally have been defined by a combination of a natural language prose, state tables, etc. But an unambiguous formal specification still may be far from any implementation. There is a vital need for formalized specifications of distributed systems in general, and communication protocols in particular which would, at the same time, indicate how implementations may be derived from them. We are convinced that this is precisely where the principal field of Estelle application is situated.

A considerable effort is presently being made in developing supporting tools for Estelle. In particular, such tools are developed within two projects of the European Esprit program: SEDOS (Software Environment for Design Open Systems - Nov.1984-Oct.1987) ,and SEDOS-ESTELLE-DEMONSTRATOR (June 1986-May 1989).

The prototype tools such as an (syntax driven) editor, a compiler, and a simulator/debugger will be the outcome of the SEDOS project. Then, these tools will be integrated and industrialized as an Estelle Work Station within the SEDOS-ESTELLE-DEMONSTRATOR project. For further details, see [6], [7], and [11].

The similar efforts in developing Estelle. tools are also conducted: in USA (National Bureau of Standards [13], Protocol Development Corporation [14]); in Canada (University of Montreal [15], University of British Columbia [16]), and in Japan [17]. At the time being at least four Estelle compiler prototypes are operational throughout the world.

Estelle has benefited from experiments in using predecessor description techniques (see [1], [2], [3], [4]) and from the cooperation with CCITT which defined SDL (Specification and Descriptions language - [5]) with which Estelle has some notions in common.

The development of Estelle was initiated within the International Organization for Standardization (ISO) in 1981 (ISO/TC97/SC21/WG1/FDT subgroup B chaired by Richard TENNEY). Its present ISO status is a Draft International Standard (DIS) the last step before becoming an International Standard. Its formal syntax and semantics are described in ISO DIS 9074 [17] which is available from the ANSI Secretariat and from the national standard organizations.

This paper presents the Estelle language by describing its principal features (section 2), the syntactical (section 3) and the semantical aspects (section 4), and by illustrating its use through an example (section 5).

2. A BRIEF OVERVIEW OF ESTELLE PRINCIPAL FEATURES

2.1 Tasks

A distributed system specified in Estelle is viewed as a collection of communicating components called in this paper tasks (or module instances). Each task has a number of input/output access points called interaction points. Two kinds of interaction points are distinguished: external and internal. A task will be represented graphically as an rectangle with points on its boundary (external interaction points) and/or inside of it (internal interaction points). In Fig.2 the reader can find examples of the above convention. The interaction points 1-13 of this figure are external, while points 14-16 are internal interaction points (of task C).

The internal behavior of a task is described in terms of a nondeterministic communicating state automaton (a transition system) whose transition actions are given in the form of Pascal statements.

A task is called active if its specification includes, in its transition part, at least one transition and otherwise, it is called inactive.

A task may have one of the following attributes

- systemprocess
- systemactivity
- process
- activity

The tasks attributed "systemprocess" or "systemactivity" are called system tasks.

2.2 Structuring

Tasks may be nested. The hierarchical (tree) structure of tasks may be depicted as in (Fig.1a) or as in (Fig.1b). The parent/children relationship is represented by edges or nested boxes respectively. The root of the tree (the enclosing box) is the main task representing the specified system.

The following five attributing principles must be observed within a hierarchy of tasks:

(1) Every active task must be attributed,
(2) System tasks cannot be nested within an attributed task,
(3) Tasks attributed "process" or "activity" must be nested within a system task,
(4) Tasks attributed "process" or "systemprocess" may be substructured only into tasks attributed either "process" or "activity",
(5) Tasks attributed "activity" or "systemactivity" may be substructured only into tasks attributed "activity".

Observe that inactive tasks can be attributed, that all tasks embodying a system task are inactive and nonattributed, and that those are the only nonattributed tasks within the hierarchy. Observe also that there are only one level of system tasks in the hierarchy (first attributed tasks on every path going down from the root of the tree).

Besides the hierarchical task structure a communication structure exists within a specified system. The elements of this structure will be represented graphically (Fig.2) by line segments which bind tasks' interaction points. Two types of binding are distinguished. When an external interaction point of a task is bound to an external interaction point of its parent task, we say that these interaction points are attached. In Fig.2 pairs of interaction points: (1,3),(2,8),(9,11) and (10,13) are attached. Two bound interaction points are said to be connected if both are external interaction points of two sibling tasks (e.g., (5,6),(4,7),(1,9) and (2,10) in Fig.2), or one is an internal interaction point of a task and the other is an external interaction point of one of its children tasks (e.g., (12,14) in Fig.2), or both are

internal interaction points of the same task (e.g., (15,16) in Fig.2).

It has to be noted that, at a given moment,

- an interaction point may be connected to at most one interaction point and it cannot be connected to itself,
- an external interaction point of a task may be attached to at most one external interaction point of its parent task and to at most one external interaction point of its children tasks,
- an external interaction point cannot be simultaneously attached and connected ,

The above restrictions are illustrated in Fig.3 (dotted lines represents forbiden bindings).

A line, composed of exactly one "connect" segment and zero or many "attach" segments, forms a, so-called, link between two interaction points. This link specify that two tasks, whose interaction points are the end-points of the link, can communicate exchanging messages (in both directions) through this linked interaction points (see Sec.2.3). In Fig.2, for example interaction points 3 and 11 or 8 and 13 are end-points of links between tasks D and E or F and H respectively. The points 1,9,2 and 10 are not end-points.

Within a specified system a fixed number of subsystems and communication links between them, are distinguished. Each subsystem is a subtree of tasks rooted in a system task (i.e.,a task attributed "systemprocess" or "systemactivity"). In particular, the whole specified system may form just one subsystem. In such a case, the specification itself (main task) is attributed "systemprocess" or "systemactivity".

For the clarity of pictures the following conventions are assumed:

- dotted lines are used for tasks enclosing subsystems,
- system tasks (subsystem roots) and communication links between them are in bold lines,
- "normal" lines are reserved for remaining tasks and links.

Fig.5 illustrates these conventions.

Since tasks enclosing subsystems are always inactive, the structure of subsystems and their communication links, once initialized, cannot be changed (static structure). In contrast, the internal structure of each subsystem may vary (dynamic structure). It is due to the fact that actions of a task may include statements creating and destroying its children tasks and their communication links .

2.3 Communication

Two communication mechanisms can be used in Estelle in order to enable a cooperation between tasks:

- message exchange
- restricted way of sharing variables

The tasks may exchange messages, called interactions. A task can send an interaction to another task through a previously established communication link between two interaction points of the tasks. An interaction received by a task at its interaction point is appended to an unbound FIFO queue associated to this interaction point. The FIFO queue either exclusively belongs to this single interaction point (so called individual queue) or it is shared with others interaction points (so called common queue).

A task can always send an interaction. This principle is sometimes known as the "non-blocking send" communication principle.

An interaction sent through an interaction point, which is an "end-point" of a communication link, directly arrives to the other "end-point" of this link and is always accepted by the receiving task, since the associated queue is unbound. An interaction sent through an interaction point, which is not an "end-point" of a communication link, is considered to be lost. Thus only "end-to-end" communication between tasks is possible.

Certain variables can be shared between a task and its parent task. These variables have to be declared as exported by the children task. This is the only way variables may be shared. The simultaneous access to these variables is excluded because the execution of the parent's actions has always priority (so called parent/children priority principle of Estelle).

2.4 Parallelism

Two kinds of parallelism between tasks can be expressed in Estelle:

- asynchronous parallelism
- synchronous parallelism

The asynchronous parallelism is authorized only between subsystems, or more precisely, between actions of different tasks belonging to the different subsystems.

The synchronous parallelism is authorized only within a subsystem, or more precisely, between actions of different tasks belonging to the same subsystem.

Intuitively speaking, each subsystem runs by its own "computation steps". A computation step begins by a selection of a set transitions (actions) among those "ready-to-fire", or "offered-to-fire" by the subsystem's tasks (at most one transition per task). Then these transitions are executed (in parallel) and, when all of them completed, the next computation step begins. That way the relative speed of tasks within a subsystem may be, in general, controlled (synchronized). That is why we say that the parallelism within a subsystem has a synchronous character.

How the set of transitions (actions) is selected for synchronous execution within one computation step of a subsystem, depends always on the parent/children priority principle and on the way the subsystem's tasks are attributed.

The parent/children priority principle, which extends to the ancestor/descendent priority principle, means that a "ready-to-fire" action of a task excludes the selection of actions of all its descendents tasks.

The role of attributes is best illustrated by two particular cases. The first when all subsystem's tasks are attributed "process" (the system task is attributed "systemprocess") and the second when all of them are attributed "activity" (the system task is attributed "systemactivity"). In the first case all "ready-to-fire" transitions (actions), which are not in the ancestor/descendent conflict, are selected (Fig.5a), while in the second case only one of them (Fig.5b). Therefore, in fact, there is no parallelism within a computation step of a "systemactivity" subsystem. The intermediate selections, between the above two extremes, are possible due to the fact that a "process" ("systemprocess") task may be substructured in both "processes" and "activities".

The subsystems, although they may exchange messages, runs asynchronously in that their computation steps are completely independent from each other. The relative speed of evolution of subsystems are not at all controlled (synchronized). Recalling the attributing principles of Sec.2.2 and the discussion there, this asynchronous behavior is possible since all ancestor (enclosing) tasks of a system task are inactive and nonattributed. Thus all means of control are absent.

2.5 Global behavior

In order to describe the behavior of the complete system specified in Estelle, the operational style (operational semantics) has been chosen. The

operational Estelle semantics considerably aids in specifying associated tools such as a simulator/debugger, and a compiler. It also reenforce a belief of the mutual compatibility of these tools, even if they are designed by different teams.

The global behavior of a system specified in Estelle is defined by the set of all possible sequences of, so called, "global situations" generated from the initial situation. Two consecutive global situations correspond to execution of one transition, i.e., transitions are consider atomic in that, conceptually, one cannot observe intermediate results of their execution.

The Estelle semantics describe the way these sequences are generated. i.e., the way the system's transitions (transitions of its tasks) may be interleaved to properly model the synchronous parallelism within subsystems combined with the asynchronous parallelism between them. This global semantics is described in more details in Sec.4.

The notion of time does not appear explicitly in Estelle. This is a consequence of the retained hypothesis that execution times of actions is not known. This knowledge is considered implementation dependent. However, to interpret properly "delays" (i.e., dynamic values assigned to some transitions which indicate a number of time units execution of these transitions must be delayed) an independent time process is assumed to exist. The computation model of Estelle outlined above is dependent on this time process only in that a relationship between progress of time and computation is defined and the "delay-timers" are observed in order to decide whether a transition can or can not be fired (see Sec.3.2.3). The way the "delay-timers" are interpreted are explained, by an example in Sec.5.2.

The formulated constraints specify a class of acceptable time processes. In each implementation or for simulation purpose any element of this class may be chosen. (see examples in [9] and [10]).

3. HIGHLIGHTS ON THE SYNTAX OF ESTELLE

3.1. Channels and interaction points

Channels in Estelle are "abstract" objects whose definitions specify sets of interactions (messages). The interaction points, through which interactions go out and come in, refer (in their declarations) to channels in a specific way. By such a reference a particular interaction point has a precisely defined set of interactions that can be respectively sent and received through this point.

Consider, for example, the following channel definition

```
channel  NAME (ROLE1, ROLE2);
by ROLE1:
   m1;
   m2;
    :
   mN;
by ROLE2:
   n1;
   n2;
    :
   nK;
```

where m1,..,mN,n1,..,nK are interaction declarations. Each interaction declaration consists of a name and possibly some typed parameters. Thus, an interaction declaration

REQUEST(x: integer, y: boolean)

specifies in fact a class of interactions with a common name REQUEST. Each of the interactions in the class is obtained by a substitution of actual parameters (values) for formal parameters x and y. Therefore,

(REQUEST,1,true) and (REQUEST,3,false)

are both interactions in the class specified by the interaction declaration of the above form. In absence of parameters the name represents itself.

Now, an interaction point p1 may be declared

p1 : NAME(ROLE1)

and another interaction point p2,

p2 : NAME(ROLE2)

In the first case, the set of interactions which can be send via p1 contains all interactions specified for ROLE1 in the channel definition (i.e., the interactions declared by m1,m2,..,mN), and the set of interactions which can be received contains all interactions specified for ROLE2 (i.e., the interactions declared by n1,n2,..,nK).

In the second case we have, as it easy to guess, exactly opposite assignment of sent and received interactions, i.e., those interactions which could be previously sent via p1 can now be received via p2 and vice versa.

We say that interaction points p1 and p2 above play <u>opposite roles</u>. Two interaction points both referring the same channel and the same role-identifier are said to play the <u>same role</u>.

Two interaction points which are connected or linked must play opposite roles since the exchange of

interactions take place between them (any interaction sent via one interaction point is received via the second and vice versa). Two interaction points which are attached must play the same role since the aim of attaching them is to "replace" one of them by the second.

Finally, to specify whether the interaction point p does or does not share its queue with other interaction points we respectively write

```
p : NAME(ROLE1) common queue
```

or

```
p : NAME(ROLE1) individual queue
```

3.2. Modules and tasks

The tasks are instances of modules. A module is specified in Estelle by a pair which consist of

- a module header definition, and
- a module body definition

A module header definition specifies the module type whose values are tasks with the same external visibility, i.e., with the same interaction points and exported variables, and the same class attribute. The definition of a module header begins with the keyword "module" followed by its name and optionally by a class attribute ("systemprocess", "process", "systemactivity" or "activity"), and declarations of interaction points (after the keyword "ip") and exported variables (after the keyword "export"). The definition ends with the keyword "end". The following is an example of a module header definition:

```
module A systemprocess;
 ip  p : T(S) individual queue;
     p1 : U(S) common queue;
     p2 : U(K) common queue;
 export   X,Y : integer; Z : boolean
end;
```

At least one module body definition is declared for each module header definition. A module body definition begins with the keyword "body" followed by: the body name, a reference to the module header name the body is associated with, and either a body definition followed by the keyword "end" or directly the keyword "external". For example, the two following bodies may be associated with the module header A:

```
body B for A;
"body definition"
end;
```

or

```
body D for A; external;
```

In fact, at a conceptual level, two modules have been defined: one of which may be identified by the pair (A,B), and the second - by the pair (A,C) . The modules thus defined have the same external visibility (same interaction points p,p1,p2 and same exported variables X,Y,Z) and the same class attribute (systemprocess). But their behaviors, defined by each of the body definitions are, in principle, different. This means that modules may have different behaviors and the same external visibility. A body defined as "external" does not denote any specific behavior of the module. It indicates that either the module body definition exists already elsewhere or will be provided later in the process of the specification refinement. This Estelle possibility of parametrizing specifications by "external" bodies may nicely serve to describe a system overall architecture without detailed description of the system components. This feature is illustrated by the example of Sec.5.1.

The body definition is composed of three parts :

- declaration part
- initialization part
- transition part

3.2.1. Declaration part

The declaration part of a body definition contains usual Pascal declarations (constants, variables, procedures and functions) and declaration of specific Estele objects, namely:

- channels
- modules
- module variables
- state and state-sets
- internal interaction points

Note that a body definition which is being declared may contain declarations of other modules (headers and bodies). This, applied repeatedly, leads to hierarchical tree structure of module definitions. The structure constitute a textual "pattern" for any dynamic hierarchy of tasks (module instances). Note that a number of tasks may change dynamically, but the hierarchical position of each task corresponds to the position of the task's definition in this pattern. For example, the body definition B declared below contains definitions of modules (A1,B1) and (A1,B2). These are children modules of the module (A,B), where the detailed definition of the module header A is that from the previous section. The above hierarchical structure of module definitions is depicted in Fig.6.

```
module  A...end;
body B for A;
:
   module  A1  process ;
   ip  p1 : T1(R1) individual queue;
       p2 : T1(R2) individual queue;
       p' : T(S)   individual queue;
   end;
   body B1 for A1; "body definition" end;
   body B2 for A1; "body definition" end;
:
end;
```

Module variables serve as references to module instances (tasks) of a certain module type. For example, the declaration

```
modvar  X,Y,Z : A1
```

says that X,Y and Z are variables of the module type specified by the module header named A1.

A module instance (task) may be created or destroyed by statements referencing module variables (init and release statements, see 3.2.2).

The internal behavior of each module (instance) is defined in terms of an state automaton whose control states are defined by enumeration of their names. For example,

```
state  IDLE, WAIT,OPEN, CLOSED
```

declares four control states IDLE,WAIT,OPEN and CLOSED. In other words, among the variables of a module, one variable is distinguished by the fact that it is introduced by the keyword "state". The "state" variable may assume values enumerated by the definition of the above form.

A group of control states are sometimes referenced using the group name which may be introduced by a declaration. For example,

```
stateset  IDWA = (IDLE, WAIT)
```

The internal interaction points may be declared to allow communication between a module instance (task) and its children module instances (tasks). They are declared in the same way as the external interaction points within a module header.

3.2.2. Initialization part

The initialization part of a module body specifies the values of some variables of the module with which every newly created instance of this module begins its

execution. In particular, local variables and the control variable "state" may have their values assigned, but also some module variables may be initialized which means that an "initial architecture" of the module children instances can be created.

To initialize Pascal variables Pascal statements are used (for example, X := 5) and to initialize the "state" variable to a control state, for example IDLE, we write

to IDLE

The initialization of a module variable results in creation of a new module instance (task) of the variable type. The variable is then a reference to the newly created task. To this end the init statement is used. In the initialization part, bindings may also be created between interaction points by the use of connect and attach statements. Assume the following is the initialization part of the module (A,B) from the previous section :

```
initialize
 begin
  init  X  with  B1;
  init  Y  with  B2;
  init  Z  with  B1;
  connect X.p1 to Y.p2;
  connect Y.p1 to Z.p2;
  attach p to X.p';
 end;
```

The above initialization part creates three module instances (tasks) referenced by the module variables X, Y and Z respectively. All tasks have the same external visibility defined by the module header A1 (the module variables X,Y and Z have been declared of module type A1). The tasks (referenced by) X and Z are instances of (A1,B1) and task (referenced by) Y is an instance of (A1,B2). The concrete hierarchy of tasks (module instances) of Fig.7 corresponds to the hierarchical "pattern" of module definitons from Fig.6. The initialization also establishes links between appropriate interaction points of newly created tasks. The result is presented in Fig.7. Recall that two interaction points must play "opposite" roles in order to be connected (because the exchange of interactions is made via these points), and must play the "same" role in order to be attached (because interaction point of a child task "replaces" that of the parent task).

3.2.3. Transition part

This part enables to describe, in detail, the internal behaviors of modules. It is composed of a collection of transitions declarations. Each transition begins with the keyword "trans". A transition may be either simple

or nested. A nested transition is a short hand notation for a collection of simple transitions (see below).

The following is an example of a nested transition:

```
trans when N.NI(p)
      from IDLE to WAIT
       provided p.present
        begin
         keep_copy(p,saved);
         output U.UC
        end;
       provided otherwise
        begin
         output N.NR
        end;
      from IDLE
       begin
       end;
```

Below we give the decomposition of the above nested transition into simple transitions:

```
trans when N.NI(p)
      from IDLE to WAIT
       provided p.present
        begin
         keep_copy(p,saved);
         output U.UC
        end;
trans when N.NI(p)
      from IDLE to WAIT
       provided not(p.present)
        begin
         output N.NR
        end;
trans when N.NI(p)
      from IDLE
       begin
       end;
```

In the above "decomposed" form one can see that with each keyword "trans" there is exactly one "begin-end" block associated. This characterizes simple transitions.

Each simple transition declaration is composed of two parts :

- the transition condition
- the transition action

The transition condition is composed of one or more clauses of the following categories:

- from-clause (from A1,..,An, where Ai is a control state or control state-set identifier)
- when-clause (when p.m, where p is an interaction point and m an interaction)

- provided-clause (provided B, where B is a boolean expression)
- priority-clause (priority n, where n is an non-negative constant)
- delay-clause (delay(E1,E2), where E1 and E2 are non-negative integer expressions)

Some clauses may be omitted and at most one of each category may appear in the condition of a simple transition . Presence of a "when-clause" excludes a "delay-clause" and vice versa. Transitions with a "when-clause" in their conditions are called input transitions. Transitions without a "when-clause" are called spontaneous. A spontaneous transition with a "delay-clause" is called delay transition.

A "from-clause" is said to be satisfied in a task state if the current value of its control variable "state" is among those listed by the from-clause. For example, if IDLE is the current control state of a task, then all three of the following from- clauses are satisfied

```
from IDLE,
from IDLE, OPEN, CLOSE,
from IDWA,
```

(recall that IDWA=(IDLE,WAIT)).

The "when p.m" clause is satisfied in a task state if the interaction m is at the head of the queue associated to the interaction point p.

The "provided B" clause is satisfied in a task state if the boolean expression B evaluates to "true" in that state.

The condition of a transition decides whether the transition is firable (or ready-to-fire) in a task state (and at a given moment of time if it concerns a delay transition). The action of one of those firable transitions eventually executes and the task will reach a new state.

A transition is said to be enabled in a task state if the "from", "when" and "provided" clauses, if present in the transition condition, are satisfied in this state.

A transition is said to be firable (or ready-to-fire) in a task state and at a given moment of time if:

(a) it is enabled in the state, and if it is a delay transition, with its delay clause "delay(E1,E2)", then it remained enabled for at least E1 time units, and

(b) it has the highest priority among transitions satisfying (a); the highest priority is represented by the least nonnegative integer.

The transition action is composed of :

- a to-clause (to A, where A is a control state identifier), and

- the "transition block", i.e., a piece of Pascal program, with specific Estelle extensions and restrictions, between "begin" and "end" keywords.

The "to-clause" (e.g. to OPEN) specifies the next control state that will be attained once the transition is fired. If omitted, the next state is the same as the current from-state.

The Pascal extensions consist of additional statements which make possible to create and destroy module instances (tasks), to create and destroy bindings between interaction points, and to send interactions.

The "create" statements are those described previously (init, connect, attach). The statements for the "destroy" counterparts are the following :

```
release X;
disconnect p;
disconnect X;
detach p;
```

The "release X" statement destroys the children task referenced by the module variable X.

The "disconnect p" statement disconnects the interaction point p from the interaction point to which it was connected , and "disconnect X" disconnects all the interaction points of the children task (referenced by) X.

The "detach p" statement detaches the interaction point p from the interaction point it was attached to (in fact the semantics of both attach and detach statements are slightly more complicated because the content of the queue of the interaction point referenced by an "attach" or "detach" is respectively: moved "down" to the queue of the end-point in the chain of attachments, or filled in with some of the interactions of this end-point queue).

If to interaction points p1 and p2 are connected (attached), then the result of "disconnect p1" ("detach p1") is the same as that of "disconnect p2" ("detach p2").

The "disconnect" and "detach" statements (similarly to "attach" and "connect" statements) may address either an interaction point of the task issuing the statement or an interaction point of one of its children tasks. In this second case the interaction point is accessed by the form "X.p" ("detach X.p" or "disconnect X.p") which means that the statements concerns the

interaction point p of the task currently referenced by the module variable X.

The capacity of executing these statements within a transition gives the possibility to change dynamically the hierarchical tree structure of tasks as well as the communication links between them.

There is also a special statement output which allows to send an interaction via a specified interaction point. For example, the statement "output p1.m" sends the interaction m via the interaction point p1.

As we explained earlier (see 2.3), if p1 and p2 are the two end-points of a communication link, then the "output p1.m" statement leads to putting interaction m in the FIFO queue associated with the interaction point p2.

Since an interaction point may be linked to at most one other interaction point there is a unique receiver of the interaction sent by a task.

The Pascal restrictions adopted in Estelle are mainly the following:

- all declared functions are without side effects (pure functions) and they may return values of an arbitrary type,
- pointers can be only used in a Pascal part,
- conformant arrays cannot be used,
- file type cannot be used,
- goto statements and labels are restricted in use,
- read and write statements cannot be used.

3.3. Specification module

All modules defined as described in the preceding sections are textually embodied in a principal module called "specification" module. This unique module is defined as follows :

```
specification SPEC-NAME [system-class];
[default-option]
[time-ptions]
"body definition"
end.
```

where system-class attribute (if any) is either "systemprocess" or "systemactivity", and default-option is either "individual queue" or "common queue".

The intent behind defining "common" or "individual" queue in a specification module definition is to give the default assignment of FIFO queues to those interaction points of the specification for which this assignment is not given in their declarations.

The time-options define a time unit (millisecond, second, etc.). A non-negative integer expression within a "delay-clause" indicate the number of time units the execution of a transition must (or may) be delayed (see Sec. 5.2 for interpretation of "delay-clauses").

The above specification definition is considered semantically equivalent to the following module definition (module header and module body declarations):

```
module ANY-NAME [system-class];
end;

body SPEC_NAME for ANY-NAME;
"body definition"
end;
```

where "ANY_NAME" may be chosen arbitrarily and "body definition" takes into account the "default-option".

Note that the specification module does not have neither interaction points nor exported variables. This means that an Estelle specification is not itself a module which communicates with other modules. In practice a specification body often constitutes a general "framework" for an open system being defined, i.e., it gives possibility of introducing some global context necessary for the system definition and initialization.

It is assumed that there exist only one instance of a specification module (main task).

4. AN OVERVIEW OF THE ESTELLE SEMANTICS

As we said earlier the semantics of Estelle is operational. This means that a, so called, next-state-relation is defined over the set of the system global states which are called here global situations. The next-state-relation (or rather next-situation-relation) specifies all possible situations which may be directly achieved from a given situation. The overall behavior of a system (a system defined by an Estelle specification) is then characterized by the set of all sequences of global situations which can be generated (by the next-situation-relation) from a certain initial situation.

Recall that the specified system is a collection of subsystems, that tasks within each subsystem may execute their transitions in a synchronous parallel way, and that tasks belonging to different subsystems may execute their transitions in an asynchronous parallel fashion.

The subsystems execute in a succession of "computation steps". Each "computation step" of a subsystem begins by selection of transitions (among those ready-to-fire and "offered" by the subsystem component tasks) that are to be executed (in parallel) within this "computation step". The selected transitions are then executed. The "computation step" ends when all of them completed. This execution of transitions within one "computation step" of a subsystem cannot be considered simultaneous since the result may depend on whether one or another completed first (recall that, for example, these transitions may send interactions into a common queue and that the order in which they are put in this queue depends on the their execution speed). All possible interleavings (permutations) of transitions selected in a computation step must therefore be taken into account in the proposed model. Nevertheless, the execution of these transitions is synchronized in that a new selection starts only if all of them completed.

As opposed to the above, it is assumed that the tasks of different subsystems cooperate in an asynchronous parallel way, therefore their relative speeds are not at all controlled.

To properly model the behaviors of a complete system specified in Estelle, both forms of parallelisms had to be expressed by the way the transition executions are interleaved. Adequacy of this interleaved model is assured in turn by the assumption of the transition atomicity.

4.1. Global situations

Each global situation of the transitions system is composed of current information on :

- the hierarchical structure of tasks within the specified system SP, the structure of bindings established between their interaction points, and the local state of each task. All this information is included in a, so called, global instantaneous description of SP (in short "GIDsp").

- the transitions that are "in parallel (synchronous) execution" within each subsystem; the set of these transitions for i-th subsystem is denoted by Ai (i=1,..n, where n is the number of subsystems).

Each global situation is denoted by :

sit = (GIDsp; A1,...,An)

The global situation is said to be initial if the "GIDsp" is initial and all sets Ai are empty. The

"GIDsp" is initial if it results from the initialization part of the specification SP.

If, in a global situation, Ai = 0, then we say that the i-th subsystem is in its management phase. During this phase a new set of transitions for parallel synchronous execution is selected. Otherwise, i.e., if Ai ≠ 0, the i-th subsystem is executing.

4.2. Next-situation-relation

This relation defines the successive situations of an arbitrary current situation

(GIDsp; A1,..,Ai,..,An).

It is defined in the following manner:

For every i = 1,2,..,n,

1) If, in the current situation, Ai=0, then the next situation is the following

(GIDsp; A1,...,AS(GIDsp/i),..,An)

where AS(GIDsp/i) is the set of transitions selected for execution by the i-th subsystem,

2) If, in the current situation, Ai ≠ 0, then for every transition t of Ai, the next situation is the following

(t(GIDsp); A1,..,Ai-(t),..,An)

i.e., the new GIDsp results from execution of t and t is removed from the set Ai.

Each transformation of a given global situation into a successive situation expresses the result of either a spontaneous evolution (case (1)) or an execution (or rather completing the execution) of a transition selected among those currently executing (case (2)). As any transition of Ai (for any i) may terminate before any other (the relative speed of execution of transitions is not known), therefore all the successive situations (for each t of Ai and for each i) have to be considered.

These transformations applied from the initial global situation, define all possible sequences of global situations (computations).

The execution of a transition t:

- may cause a change in its local state whose influence on a next global situation is expressed by t(GIDsp). In particular, this transition may "output", i.e., it may send an interaction which

is put into the FIFO queue of another task, it may create a new children task and a new communication link, etc.

- cannot influence neither the choice of transitions already selected by the other subsystems (the sets Aj, for j≠i, remain unchanged in the next situation) nor the choice of transitions within the same subsystem (the set Ai becomes Ai-(t) in the next situation).

The selection of transitions to be executed within one computation step, by an i-th subsystem, i.e., the choice of the set AS(GIDsp/i) is regulated by

- the principle of parent/children priority, and

- the tasks' attributes.

The rule applied to a task within a subsystem can be formulated as follows:

- if the task has a "ready-to-fire" transition to offer, then this one will be selected (parent/children priority),

- otherwise, depending on whether the task is attributed "process" ("systemprocess") or "activity" ("systemactivity") respectively, all or one (chosen nondeterministically) of those "ready-to-fire" transitions offered by its children tasks, will be selected.

This rule applied recursively, starting with the root (system) task of the i-th subsystem gives the selected set AS(GIDsp/i). Fig.5 illustrates the applications of this rule for two special cases.

It is worth to note that the "ready-to-fire" transition offered by a task cannot be selected if any of this task's ancestors had something to offer. This property excludes parallelism between tasks being in ancestor/descendent relation.

5. EXAMPLE

The example below is not presenting a specification of a real system. It should help the reader to get familiar with the syntax of Estelle and:

- to illustrate a way the overall structure of a system can be specified using "external" parameters which replace concrete module body definitions (Sec. 5.1),

- to illustrate and analyze the use and interpretation of the "delay-clauses" and "priority-clauses" (Sec. 5.2).

The body defined in Sec. 5.2 may replace the appropriate "external" body in the scheme of Sec. 5.1.

5.1. Specifying the overall structure of a system

The specification below declares and initialize a system which consists of three subsystems X,Y and Z (i.e., the subsystems are referenced by module variables X,Y and Z of types: USER,RECEIVER and NETWORK respectively) which exchange some messages through their interaction points connected as declared in the initialization part of the specification. While the RECEIVER_BODY is further specified in Sec.5.2. as one single task, the other subsystems remain "external" and they definitions can be differently substructured in subsequent refinments of the specification. The scheme of the specified EXAMPLE system is presented in Fig.8.

```
specification EXAMPLE;
 default individual queue;
 timescale second;

 channel UCH(User,Provider);
  by Provider: DATA_INDICATION;

 channel NCH(User,Provider);
  by User: DATA_INDICATION;
  by Provider: SEND_AK;

 module USER systemactivity;
  ip U: UCH(User);
 end;
 body USER_BODY for USER; external;

 module RECEIVER systemactivity;
  ip U: UCH(Provider); N: NCH(Provider);
 end;
 body RECEIVER_BODY for RECEIVER; external;

 module NETWORK systemprocess;
  ip N: UCH(User);
 end;
 body NETWORK_BODY for NETWORK; external;

 modvar X: USER; Y: RECEIVER; Z: NETWORK;

 initialization
  begin
   init X with USER_BODY;
   init Y with RECEIVER_BODY;
   init Z with NETWORK_BODY;
   connect X.U to Y.U;
   connect Y.N to Z.N
  end;

end.
```

5.2. Interpretation of delay-clause

The computational model for Estelle is formulated, as far as possible in time-independent terms. But some Estelle spontaneous transitions may contain a delay-clause of the form "delay(E1,E2)". The intent of this clause is to indicate that execution of the transition (if it is enabled) must be delayed. The minimum time the transition must be delayed and the maximum time it may be delayed are initially specified by the values of integer expressions E1 and E2 respectively. An implementation may choose a concrete delay value in the closed interval determined by these expressions. Let us note finally that "delay(E1)" means the same as "delay(E1,E1)".

This above intuitive interpretation of the delay-clauses is illustrated below by an example (taken from [19] with slight changes). The body definition which we describe in this example can replace the "external" body parameter of the RECEIVER module of a simple communication scheme defined in Sec.5.1. The necessary channel definitions and declaration of interaction points are then defined at the specification level of the EXAMPLE in Sec.5.1.

The following narrative description, assume a protocol in which:

- multiple protocol data units (messages) may be acknowledged in a single acknowledgment;
- each protocol data unit shall be aknowledged after some maximum time (max);
- it is desirable to acknowledge each protocol data unit received after a minimum time (min), but when more than four are received an acknowledgment must be sent after this minimum delay (min);
- the maximum number of unacknowledged protocol data units is seven;
- when the system remains too long inactive (inactive period) a "dummy" acknowledgments are generated to provide minimal activity to prevent disconnections.

The following module body describes a solution in Estelle. The specification in Sec.5.1 assumes time scale resolution of 1 second (see "timescale" option there).

```
body RECEIVER_BODY for RECEIVER;
(* declaration part *)
type
 time_period: integer;
```

```
const
 one_second = 1;
 twenty_seconds = 20;
 sixty_seconds = 60;
 high = 0;
 medium = 1;
 low = 2;
state
 IDLE, AK_SENT;
var
 ak_no: integer;
 min, max, inactive_period: time_period;
initialize
(* initialization part *)
 to IDLE
  begin
   min := one_second ;
   max := twenty_seconds;
   inactive_period := sixty_seconds;
   ak_no := 0;
  end;
trans
(* transition part *)
 from IDLE
  to IDLE
   priority medium
    when N.DATA_INDICATION
     1: begin
         output U.DATA_INDICATION;
         ak_no := ak_no + 1
        end;
  to AK_SENT
   provided (ak_no > 0) and (ak_no <= 4)
    priority low
     delay(min,max)
      2: begin
          output N.SEND_AK(ak_no)
         end;
   provided (ak_no > 4) and (ak_no < 7)
    priority high
     delay(min)
      3: begin
          output N.SEND_AK(ak_no)
         end;
   provided ak_no =7
      4: begin
          output N.SEND_AK(ak_no)
         end;
   provided otherwise (* ak_no = 0 *)
    priority low
     delay (inactive_period)
      5: begin
          output N.SEND_AK(ak_no)
         end;
 from AK_SENT
  to IDLE
      6: begin
          ak_no = 0
         end;
end;
```

The above transition part of the RECEIVER_BODY consists of one nested transition representing in fact six simple transitions. The outmost level of factorization (from_clauses) devides these six transitions into two groups : 1-5 and 6. To_clauses separate the first transition and the nested transition which groups simple transitions 2-5. These transitions in turn, are factorized by their provided_clauses.

The module begins in its IDLE control state, with the delay values appropriatly initialized and the number of messages which remained to acknowledge is set to 0. Note that immediately after this initialization the delay-timer associated with transition 5 begins to run since the transition becomes enabled.

If a DATA_INDICATION message arrived during the inactive period (60 seconds), then the message is transmitted to USER and the number of messages to acknowledge increases by 1 (transition 1). Otherwise, i.e., when 60 seconds passed without any message, an "artifficial" acknowledgment is sent to prevent disconnection (transition 5).

Note that imediately after the first message arrived and was served the delay-timer of transition 2 starts to run, but the transition may be fired only if: there is no new message to be served, when 1 second already passed, and the number of messages to be acknowledged is not greater than 4. This transition certainly will be fired if the obove situation has remained unchanged for 20 seconds. If the transition was fired or more than 4 messages have been served in between (the transition becomes no more enabled), then its timer is canceled.

Imediately after the 5-th consecutive message were served (transition 1) but all five have not yet been acknowledged, the delay-timer of transition 3 is turned on. The timer will run exactly 1 second and the transition will be executed if the number of unacknowledged messages remained during this time less than 7. Otherwise, when the 7-th message was served (transition 1) during this period, the transition will not be executed and its timer will be canceled. The transition 4 will be executed instead.

It is worth to note the role which is played by priorities assigned to transitions. Due to them, in a havy trafic of comming messages (more than 7 in less than one second), acknowledgments are sent always for blocks of 7 messages. In a less havy trafic (more than 4 but less than 7 messages in one second), the acknowledgments are sent always for blocks of 5-6 messages. If the messages arrive in quantities 1 to 4 every 20 seconds they may be acknowledged one by one or in blocks of 2-4 messages depending on message distribution in that time and the real delay value in the time interval <min, max> (the input transition 1

have a higher priority than transition 2). If no more than one message arrives every 20 seconds, the messages certainly (and in any implementation) will be acknowledged one by one.

5. REFERENCES

[1] ANSART J.P., RAFIQ O.,SHARI V., Protocol Description and Implementation Language (PDIL), Proceedings IFIP 2nd International Workshop on Protocol Specification, Verification and Testing, C. SUNSHINE (ed), North Holland,1982.

[2] AYACHE J.M., COURTIAT J.P., DIAZ M;, LC/1 A Specification and Implementation Language for Protocols, Proc IFIP 3rd International Workshop on Protocol Specification, Verification and Testing, H. RUDIN and C.H. WEST (ed), North Holland, 1983.

[3] BOCHMAN G.V., Finite Description of Communication Protocols" Computer Network, Vol.2, Oct. 1978.

[4] TENNEY R.L., BLUMER T.P., A Formal Specification Technique and Implementation Method for Protocols, ICST Report n° ICST/HLNP 81-15, National Bureau of Standards, Gaitherburg, MD 20899, 1981.

[5] CCITT/SGXI Recommendation Z101 à Z104, Functional Specification and Description Language, 1985.

[6] DIAZ M., SEDOS : Un environnement logiciel pour la conception des systèmes distribués, 3ème Congrès "De nouvelles architectures pour les communications", Paris 28-30 Oct. 1985.

[7] COURTIAT J.P., DEMBINSKI P., GROZ R., JARD C., ESTELLE : un langage pour les algorithmes distribués et les protocoles. Technique et Science Informatiques, vol. 6, N° 2, 1987.

[8] DEMBINSKI P., ESTELLE Semantics, SEDOS Rep. n° SEDOS;054, June 1986.

[9] DEMBINSKI P., BUDKOWSKI S., Defining delays in a time-independent model for ESTELLE, Report FDT 7581, Agence de l'Informatique, 1985.

[10] COURTIAT J.P., Une vue réseaux de Pétri du langage ESTELLE, 3ème Congrès de Nouvelles Architectures pour les Communications, Paris 28-30 Oct.1986.

[11] ANSART J.P. at all, Software tools for ESTELLE, Sixth Int. Workshop on Protocol Specification, Testing and Verification, Montreal, June 10-13, 1986.

[12] ANSART J.P., CHARI V., DEMBINSKI P., De la Spécification à l'Implantation en ESTELLE, 9ème Journée Francophone sur l'Informatique, Liège, 20-21 Janv., 1987.

[13] User Guide for the NBS Prototype Compiler for ESTELLE, Rep. N° ICST/APM 87-1, NBS Institute for Computer Science and Technology, 1986.

[14] ESTELLE Development system, Users Manuel, Protocol Development Corporation, 1330 Beacon Str. Suite 320, Brookline, MA 02146, 1986.

[15] G. BOCKMAN, Usage of Protocol Development Tools : the results of a survey, Proc. VII IFIP Intern. Workshop on Protocol Specification Testing and Verification, Zurich 1987.

[16] S.T. VUONG A.C. LAN, Semi-automatic Implementation of Protocols, Proc. IEEE INFOCOM 87, San Francisco 1987.

[17] T. KATO, T. HASEGARA, H. HORINCHI, Design of Translator from ESTELLE with ASN.1 to ADA : design concept; KDD R & D Labs, 1987.

[18] ISO/TC97/SC21/WG1/DIS9074 ESTELLE - A formal Description Technique Based on an Extended State Transition Model, 1987.

[19] R.J. LINN, Jr. A tutorial on the Features and Facilities of ESTELLE, ICST report, National Bureau of Standards, Gaithersburg, MD 20899, 1986.

Fig.1

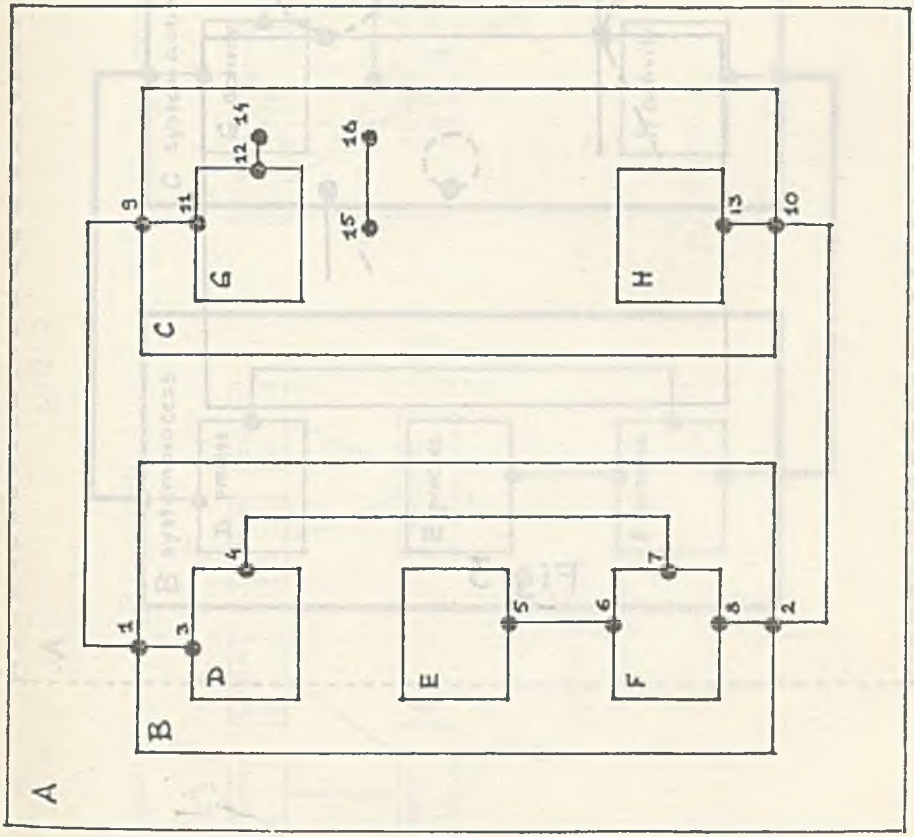


Fig.2

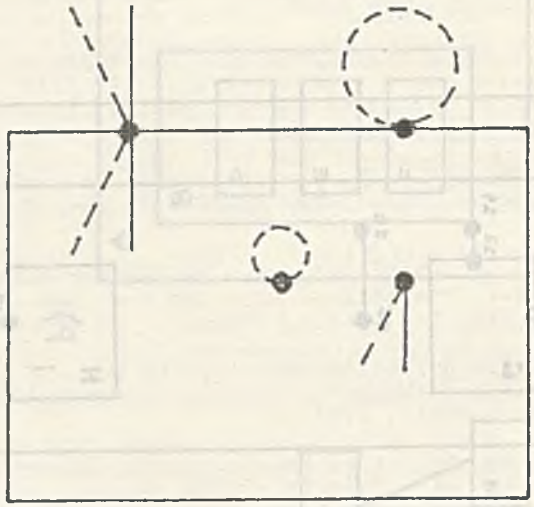


Fig. 3

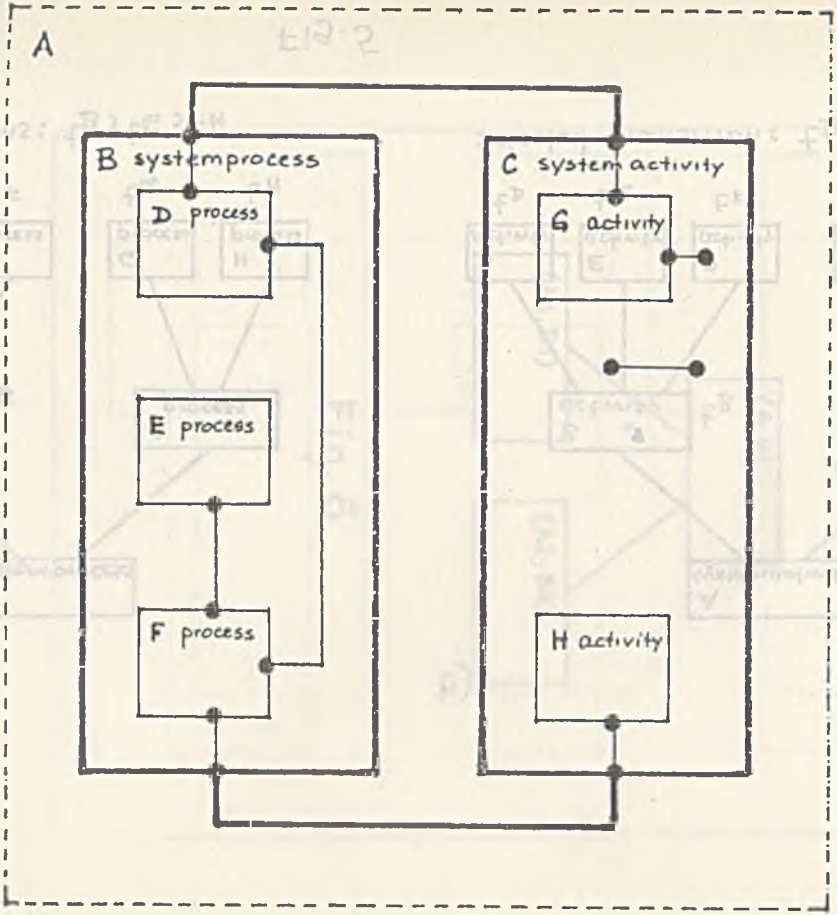
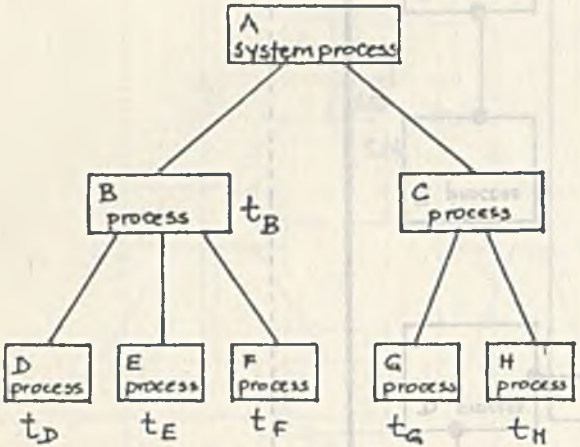

A
B systemprocess
D process
E process
F process
C system activity
G activity
H activity

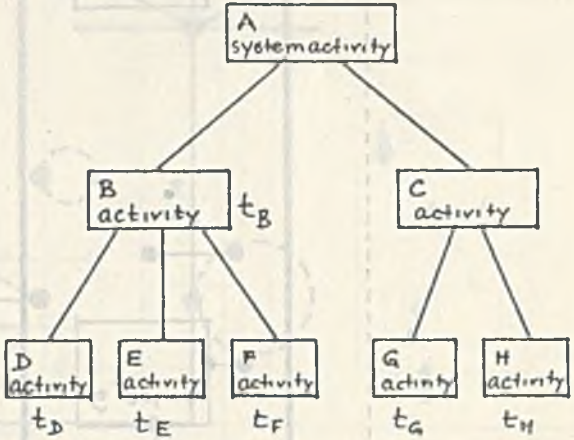
Fig. 4

a)



selected transitions: t_B, t_G, t_H

b)



selected transition: t_B or t_G or t_H

Fig. 5

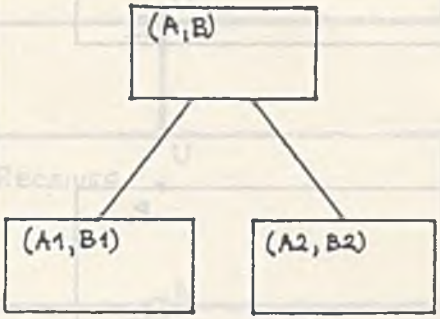


Fig. 6

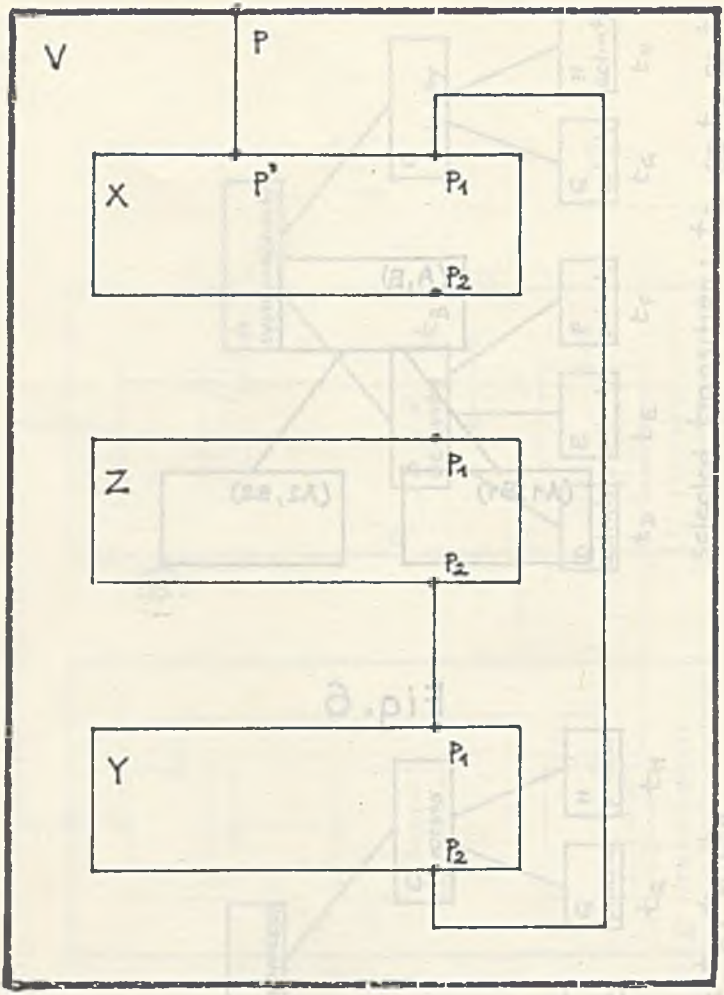


Fig. 7

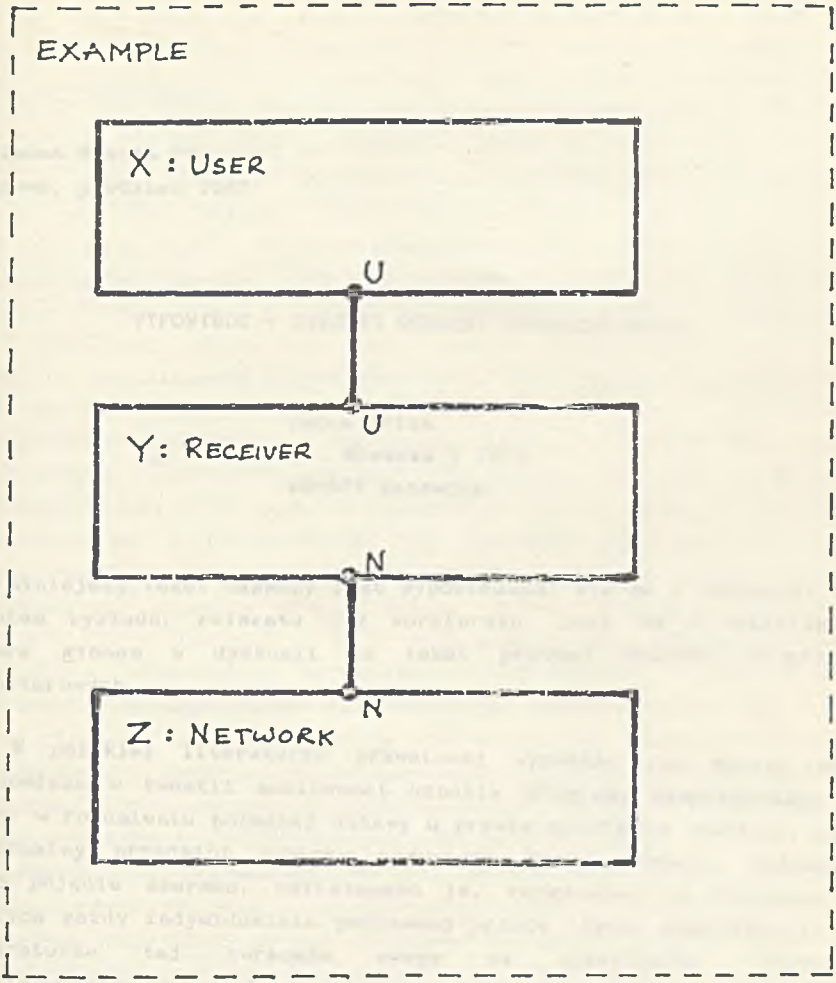


Fig. 8

Fig. 7

Fig. 8

Jesienna Szkoła PTI
Mrągowo, grudzień 1987

WYPOWIEDZ W KWESTII OCHRONY OPROGRAMOWANIA

Jacek Irlik
ul. Mieszka I 15/8
40-877 Katowice

Niniejszy tekst nazwany jest wypowiedzią. Nie ma z założenia być tekstem wykładu, referatu czy koreferatu. Jest on w zamierzeniu autora głosem w dyskusji na temat prawnej ochrony programów komputerowych.

W polskiej literaturze prawniczej wyrazano juz szereg razy stanowisko w kwestii mozliwości uznania programu komputerowego za utwór w rozumieniu polskiej ustawy o prawie autorskim. Sciślej, jako ewentualny przedmiot ochrony rozwazano oprogramowanie, rozumiane jako pojęcie szersze, definiowano je, rozkładano na elementy, z których każdy indywidualnie poddawano próbie takiej kwalifikacji. W literaturze tej zwracano uwagę na doktrynalny charakter publikowanych rozwazań - a to w związku z brakiem polskiego orzecznictwa sądowego w tym zakresie. Prezentowane poglądy nie były zgodne w kwestii, czy program komputerowy mozna uznać za utwór w powyzszym znaczeniu. Prezentowano tez rozmaite stanowiska w odniesieniu do innych, związanych z tym kwestii. Rozwazano na przykład problemy, jakie elementy oprogramowania lub które z jego postaci mozna uznać za posiadające cechy utworu, które z osób uczestniczących w procesie jego tworzenia należy uznać za podmioty prawa autorskiego czy wreszcie, jakie działania stanowią naruszenie tego prawa w przypadku, jeśli dotyczy ono oprogramowania. Przegląd literatury w tym zakresie dokonany został w ramach publikowanego w materiałach niniejszej Szkoły wykładu docenta Gawlika.

Autor niniejszej wypowiedzi uchyla się dziś od zajęcia stanowiska w kwestiach wspomnianych wyzej, proponując równocześnie dość zasadniczą zmianę toku rozumowania w ich rozstrzyganiu. Moze nie warto zastanawiać się nad tym, czy oprogramowanie lub jego elementy można uznać za utwór w rozumieniu prawa autorskiego, prawa, które konstruowano. historycznie stosunkowo niedawno, mając na myśli wytwory intelektualne zupełnie inne, mające już w czasie konstruowania tego prawa dość bogatą historię swoich gatunków. Moze warto zastanowić się najpierw nad tym, co w zakresie oprogramowania należy uznać za godne ochrony i w oparciu o wynik takich rozważań konstruować odpowiednie prawo, nie wykluczając posłużenia się tradycyjnym prawem autorskim jako "materiałem pomocniczym". W jednym z ostatnich numerów Biuletynu PTI profesor Turski pisze na temat akcji pod hasłem "RESPECT COPYRIGHT - ENCOURAGE CREATIVITY", proponując odwrócenie kolejności: "POPIERAJ TWORCZOSC - SZANUJ PRAWO AUTORSKIE". Nawiązując do tej propozycji autor niniejszej wypowiedzi proponuje zatem zastanowienie się nad tym, co w tej twórczości chcemy popierać. Nie należy też zapominać o tym, ze wynik takiego zastanowienia jest w istotny sposób zależny od tego, jak rozumiany jest w takim przypadku zaimek MY.

W pierwszych latach po zbudowaniu pierwszego komputera nie istniała mozliwość bezpośredniego wykorzystania tej techniki przez jakąkolwiek osobę spoza kręgu profesjonalistów. Obecnie zastosowanie tych urządzeń staje się coraz bardziej powszechne. Coraz częściej, coraz większa liczba osób ma bezpośredni kontakt z coraz większymi systemami mając przed sobą jedynie terminal (jako zasadnicze narzędzie kontaktu - ekran i klawiaturę). Osoby te coraz częściej nie są profesjonalistami-informatykami. Coraz częściej więc spotkać się można - w odniesieniu do systemów uzytkowych - z wezwaniem do tworzenia systemów "przyjaznych dla uzytkownika". Czy ta "przyjazność" nie jest przypadkiem tym właśnie, co należy najbardziej popierać? Jeśli tak jest, to w konsekwencji najbardziej godne ochrony (zapewnionej przez odpowiednie ukształtowanie prawa i jego przestrzeganie) jest to, co składa się na tę "przyjazność".

Jakie cechy oprogramowania mozna by uznać jako stanowiące o "przyjazności"? Co na nią się składa?

Autorowi wydaje się, że składają się na nią:

- wygoda uzytkowania sytemu, na co składają się przejrzystość i czytelność uzytych menu, prostota zasad komunikowania się z systemem, mozliwość łatwego opanowania umiejętności formułowania bardziej złozonych poleceń, przejrzystość udzielanych odpowiedzi,

- szybkość reakcji systemu, na co składają się zarówno organizacja jego struktury, jak i wykorzystane narzędzia oraz uzyte algorytmy obliczeniowe,

- walor estetyczny, szata graficzna tego, co system przedstawia uzytkownikowi w odpowiedzi.

Uzytkownika dziś - a chcemy chyba, aby tak właśnie było w stopniu coraz to większym - nie obchodzi (nie powinien) typ komputera, który dla niego pracuje (zwłaszcza w sieci), nie obchodzą wykorzystane osiągnięcia inzynierii oprogramowania. Uzytkownik system widzi z zewnątrz i z zewnątrz go ocenia. Jeśli myślimy o informatyce "dla ludzi", to powinniśmy popierać to, co prowadzi do uzyskania mozliwie najlepszej takiej oceny zewnętrznej.

Wydaje się słuszne stwierdzenie, ze uzytkownik ocenia system i widzi jego indywidualność w sposób podobny do tego, w jaki czyni to odbiorca dzieła literackiego, plastycznego czy muzycznego.
Odbiorcy obrazu nie interesuje technologia jego wytworzenia. Nie interesuje go sposób gruntowania płótna, uzyte farby i sposób ich zmieszania. Nie obchodzi go to nawet wówczas, gdy to właśnie ta technologia pozwoliła na ten, a nie inny efekt artystyczny.
Twórca dzieła muzycznego równiez często w technologii szuka źródła nowych efektów artystycznych, szuka w wykorzystaniu coraz to nowych instrumentów, w ich zestawieniu i sposobie uzycia. Słuchacza interesuje to, co odbiera.
W tradycyjnym prawie autorskim ochronie podlega to, co podlega społecznej percepcji, zewnętrznie obserwowalny efekt. Technologia ochronie nie podlega, a jeśli to właśnie ona daje ten efekt, a twórca chce mieć nań monopol - pozostaje mu jedynie chronić jej tajemnicę.

Wynik toku powyższego argumentowania wydaje się nieco zaskakujący. Odrzucając a priori prawo autorskie jako konstrukcję stworzoną dla zupełnie innej rzeczywistości niż ta, o której mowimy i próbując stworzyć - poprzez pytanie, co społecznie jest godne ochrony - podstawę dla konstruowania jakiegoś nowego prawa mającego za przedmiot oprogramowanie, dochodzimy do wniosku, że to, co chcemy chronić - chronione jest według takiej samej zasady, jak w prawie autorskim.

Przedmiotem ochrony jest przy takim podejściu zewnętrznie obserwowalne zachowanie systemu oprogramowania.
To co widać jest MOJE.
Użytkownik woli MOJE od TWOJEGO, bo jest mu bardziej przyjazne.
Nawet, jeśli wbrew mojej woli skorzystasz z moich narzędzi, to wymyśl coś istotnie nowego zewnętrznie, coś nowego, co jest lepsze i jako lepsze będzie preferowane przez odbiorców. Lepsze - w klasyfikacji określonych powyżej cech.

Autor oświadcza na koniec raz jeszcze, że prezentowana wypowiedź jest wyrazem jego osobistego stanowiska w kwestii ochrony oprogramowania i ma charakter postulatu, a nie stwierdzenia.

Autor dziękuje organizatorom seminarium JULIN'87 za zaproszenie do uczestnictwa w nim, bo to właśnie to seminarium pomogło istotnie w sformułowaniu głównych tez niniejszej wypowiedzi.

Jesienna Szkoła PTI
Mrągowo, grudzień 1987

OCKHAM, EDISON, JOYCE I INNI, CZYLI O PROGRAMOWANIU WSPOŁBIEŻNYM

Jan Madey
Instytut Informatyki
Uniwersytet Warszawski
PKiN pok.850
00-901 Warszawa
tel. 268-258

1. Wprowadzenie

1.1. Współbieżność

Programowanie współbieżne jest bardzo ważną i rozwijającą się dyscypliną informatyczną. *Współbieżność* jest tu rozumiana jako *potencjalna równoległość*. Mówimy, że dwa sekwencyjne procesy są współbieżne, jeżeli ich wykonywanie nie jest rozłączne w czasie. Nie ustala się przy tym, czy procesy te (lub ich fragmenty) są wykonywane jednocześnie na oddzielnych procesorach, czy też operacje procesów są wykonywane naprzemiennie, na jednym procesorze. Co więcej, program opisujący kilka procesów współbieżnych musi działać poprawnie w każdej z tych sytuacji, tzn. niezależnie od liczby czynnych procesorów.

Współbieżność jest więc pewną *abstrakcją równoległości*. Prace badawcze prowadzone w tej dziedzinie mają bardzo szeroki zakres, od czysto teoretycznych, do typowo aplikacyjnych. W ostatnich latach pojawiły się nowe rozwiązania sprzętowe, które umożliwiają rzeczywiście równoległe wykonywanie wielu procesów współbieżnych. Wiele z wyników teoretycznych znajduje teraz praktyczne zastosowanie.

1.2. Synchronizacja

Procesy współbieżne *komunikują się* ze sobą w celu wymiany informacji oraz dla *synchronizacji*. Języki oraz techniki programowania sekwencyjnego nie były przystosowane do opisywania takich zagadnień. Okazało się, że nawet stosunkowo proste żądania synchronizacyjne mogą być bardzo trudne do wyspecyfikowania aparatem sekwencyjnym. Rozważmy bowiem typowy schemat procesu współbieżnego:

```
repeat
    operacje lokalne
    protokół wstępny
    rejon krytyczny
    protokół końcowy
forever
```

Operacje lokalne oznaczają tę fazę procesu, która jest nieistotna z punktu widzenia synchronizacji i nie ma wpływu na inne procesy. W szczególności proces może się w tej fazie nawet zakończyć. Natomiast *rejon krytyczny* oznacza te operacje procesu, które nie mogą być wykonywane jednocześnie z operacjami rejonu krytycznego jakiegokolwiek innego procesu. Mówimy, że musi być zagwarantowane *wzajemne wykluczanie się* rejonów krytycznych procesów. Rejon krytyczny opisuje zwykle dostęp do wspólnych, dzielonych zasobów. *Protokoły* reprezentują instrukcje mające na celu właśnie zagwarantowanie spełnienia zasady wzajemnego wykluczania. Otóż zapisanie tych protokołów przy użyciu jedynie tradycyjnych konstrukcji programowania sekwencyjnego jest skomplikowane i mało czytelne.

Łatwo popełnić błąd, trudno wykazać poprawność rozwiązania. Dlatego też zaczęto rozwijać języki i techniki specjalnie przeznaczone do opisu procesów współbieżnych.

Pierwsze badania w tej dziedzinie podjęto jeszcze wtedy, gdy komputery miały tradycyjną architekturę: jeden centralny procesor, kilka procesorów wejścia/wyjścia, jedna wspólna pamięć operacyjna. Stąd proponowane konstrukcje i rozwiązania opierały się na następującym założeniu: procesy komunikują się i synchronizują poprzez *dzielone zmienne globalne*. Zmienne te reprezentują na naszym poziomie abstrakcji wspólną pamięć. Klasycznym mechanizmem synchronizacyjnym tego typu jest *semafor*, zaproponowany przez Dijkstrę w 1965 roku [6]. Używając semaforów można banalnie prosto zapisać protokoły otaczające rejon krytyczny procesu: każdy protokół, to pojedyncza operacja na zmiennej semaforowej. Okazało się jednak, że przy zapisywaniu skomplikowanych procesów, w których musi wystąpić wiele różnych semaforów, badanie poprawności programu staje się bardzo trudne, znacznie trudniejsze niż w wypadku programowania sekwencyjnego.

1.3. Poprawność

W programowaniu sekwencyjnym problem poprawności jest dobrze określony. Mówimy, że program jest *częściowo poprawny*, jeżeli daje szukany wynik, o ile się kończy. Gdy dodatkowo wykażemy, że program ten nie zapętla się, to jest on *całkowicie poprawny*. Projektując program musimy we właściwy sposób określać zarówno specyfikację całego programu, jak i jego części składowych. Wówczas można starać się udowodnić poprawność programu. W praktyce jednak postępuje się na ogół inaczej: formalny sposób wykazywania poprawności programu jest wspomagany (a niekiedy całkowicie zastąpiony) tradycyjnym *testowaniem*. Aczkolwiek z pomocą testowania możemy jedynie wykryć błąd, a nie wykazać brak błędów, to jest ono bardzo ważnym, praktycznym narzędziem. Zauważmy teraz, że programowanie sekwencyjne charakteryzuje się następującą cechą: każdorazowe wykonanie programu (poprawnego lub błędnego) z tymi samymi danymi wejściowymi daje te same wyniki.

Jedynie dzięki temu testowanie może być użyte do zlokalizowania "chorego" miejsca w programie.

W programowaniu współbieżnym sytuacja jest znacznie bardziej skomplikowana. Każdorazowe wykonanie programu współbieżnego błędnego może dać różne wyniki dla tych samych danych wejściowych -- tym samym więc testowanie traci rację bytu. Sytuacja taka wynika z faktu, że jednostkowe operacje procesów wchodzących w skład programu mogę być dowolnie przeplecione. Ciąg wykonawczy programu współbieżnego jest więc klasą *przeplecionych ciągów wykonawczych* procesów składowych. Chcąc wykazać poprawność programu należałoby zatem zbadać wszystkie możliwe elementy tej klasy i to dla każdego układu danych wejściowych. Jest to oczywiście niewykonalne. Ale dla wykazania *niepoprawności* programu wystarczy podać jeden scenariusz przeplatania operacji procesów, dla którego otrzymujemy niepoprawne wyniki.

Stosując bardziej formalne podejście do poprawności programów współbieżnych określa się dwa rodzaje własności programów: *zapewniania* (ang. *safety*) i *żywotności* (ang. *liveness*). Pierwsze z nich są pewną analogią do częściowej poprawności w programowaniu sekwencyjnym, a drugie -- do całkowitej poprawności. Własności zapewniania są to takie, które wynikają z jawnej i statycznej specyfikacji programowanego problemu. Należy więc w ten sposób skonstruować program, aby taka specyfikacja była spełniona. Przykładem specyfikacji tej klasy jest żądanie wzajemnego wykluczania rejonów krytycznych procesów. Jest to bowiem warunek bezwzględny, nie zmieniający się w czasie wykonywania programu. Natomiast własności żywotności dotyczą dynamicznych aspektów. Chodzi o zagwarantowanie, że każde oczekiwane zdarzenie nastąpi w skończonym czasie. Najbardziej typowym przykładem naruszenia warunków żywotności jest doprowadzenie programu do stanu *blokady* (*zastoju, zakleszczenia*; ang. *deadlock*), kiedy to procesy oczekują zdarzenia, które nigdy nie zajdzie. Lokalną formą zastoju jest tzw. *zagłodzenie* procesu -- jeden z procesów nie może być kontynuowany z powodu "złego zachowywania się" pozostałych.

Chcąc więc udowodnić poprawność programu współbieżnego

należy wykazać, że są spełnione zarówno własności zapewniania, jak i żywotności. Nie można przy tym robić żadnych założeń co do bezwzględnych i względnych szybkości wykonywania się procesów.

Rozważania o poprawności mają sens jedynie wtedy, gdy założy się pewne warunki o procesach i o środowisku, w którym są one realizowane. Na naszym poziomie abstrakcji przyjmujemy, że procesy są *słabo powiązane* (ang. *loosly connected*). Oznacza to, że proces tylko okazjonalnie i krótko przebywa w swoim rejonie krytycznym, a znaczną większość czasu poświąca na operacje lokalne, nieistotne z punktu widzenia synchronizacji. Co więcej, jeśli w fazie lokalnych operacji proces się zakończy, to nie może mieć to wpływu na działanie inych procesów. Natomiast zakładamy, że każdy proces opuści swój rejon krytyczny. Wymagać będziemy również aby protokoły były krótkie i efektywne.

1.4. Systemy scentralizowane a rozproszone

W punkcie 1.2 wspomnieliśmy o *semaforze* jako o mechaniźmie synchronizacyjnym dla procesów komunikujących się poprzez *zmienne globalne* reprezentujące wspólną pamięć operacyjną w scentralizowanym systemie komputerowym. Semafor jest mechanizmem niestrukturalnym i trudno wykazać poprawność dużych, złożonych programów, w których procesy korzystają z wielu semaforów. Dlatego poszukiwano innych mechanizmów, choć ciągle jeszcze dla systemów scentralizowanych. W latach siedemdziesiątych opracowano konstrukcję zwaną *monitorem*, do czego przyczynili się głównie Brinch Hansen, Dijkstra oraz Hoare. Termin "monitor" został zaproponowany przez Hoare'a w pracy [8]. Monitor jest strukturalnym mechanizmem, który spełnia kilka funkcji. Po pierwsze, nadaje się do definiowania abstrakcyjnych typów danych -- możemy określać z jego pomocą jakie są jedyne dopuszczalne operacje na zbiorze pewnych obiektów. Po drugie, ułatwia modularne programowanie. Po trzecie wreszcie, umożliwia synchronizację w dostępie do dzielonych zmiennych. Jest to bowiem moduł grupujący zarówno zmienne reprezentujące zasoby dzielone przez procesy, jak i procedury opisujące jedyne operacje wykonalne na tych zasobach. Co więcej, z

definicji procedury te wykluczają się wzajemnie. Jest więc automatycznie zagwarantowane, że w danej chwili tylko jeden proces ma dostęp do dzielonych zasobów ("jest wewnątrz monitora"). Jeżeli zatem dysponujemy językiem programowania współbieżnego z monitorami, to istotną część problemów rozwiązali już za nas twórcy kompilatora tego języka i znacznie łatwiej jest wykazać poprawność programu oraz wykryć niektóre błędy już w fazie kompilacji. Zauważmy jeszcze na koniec, że semafory i monitory są w pewnym sensie równoważnymi mechanizmami. Za pomocą semaforów można symulować monitory, a z kolei typ semaforowy można zdefiniować jako monitor.

Pojawienie się rozproszonych systemów komputerowych zmusiło informatyków do poszukiwania nowych mechanizmów synchronizacyjnych. W systemach rozproszonych mamy bowiem zestaw procesorów, z których każdy ma własną pamięć lokalną, a pamięć wspólna na ogół nie występuje. Dlatego też mechanizm oparty o zmienne dzielone jest nieodpowiedni. Fundamentalną pracą w zakresie nowego typu synchronizacji jest artykuł [9], w którym Hoare proponuje model języka programowania współbieżnego, zwanego skrótowo *CSP* od tytułu pracy (*Communicating Sequential Processes*). Procesy komunikują się w CSP poprzez specjalne operacje wejścia/wyjścia. Operacje te są w pełni synchroniczne: w komunikacji biorą udział zawsze dokładnie dwa procesy i obydwa muszą być do tego gotowe. Zatem zwykle jeden z procesów jest automatycznie opóźniany, do chwili gotowości drugiego. Dopiero wówczas następuje wymiana informacji między tymi procesami, jako jednostkowa akcja. Ten model (w pewnych odmianach) stał się podstawą większości rozwiązań przyjętych w nowszych językach, bądź w rozważaniach teoretycznych.

1.5. Języki programowania współbieżnego

Monitory występują w kilku językach programowania. Pierwszym z nich był *Współbieżny Pascal*, opracowany przez Brinch Hansena [2] i zastosowany przez niego do opisu kilku prostych systemów operacyjnych. Ważnym i stosunkowo nowym językiem tej klasy jest

Współbieżny Euklides autorstwa Holta [10]. Został on wykorzystany do opracowania systemu operacyjnego *Tunis*, który jest funkcjonalnym odpowiednikiem popularnego systemu *Unix*. Brinch Hansen w swoich nowszych pracach odszedł od koncepcji z lat siedemdziesiątych. W szczególności uznał, że celowe jest zastąpienie monitora bardziej prymitywnymi konstrukcjami, ale za to już jednofunkcyjnymi (jak wspominaliśmy w uprzednim punkcie, monitor spełnia właściwie trzy zadania). To podejście znalazło odbicie w języku *Edison* [3], prezentowanym skrótowo w dalszej części niniejszej pracy.

Idee z CSP zostały zastosowane w wielu różnych językach. W szczególności mechanizm komunikacji procesów w *Adzie* [15] jest inspirowany przez CSP. Bardzo zbliżone do CSP są dwa omawiane dalej języki, a mianowicie *occam* [11] i *Joyce* [5]. Pierwszy z nich ma duże praktyczne znaczenie, gdyż istnieje ścisły związek pomiędzy tym językiem, a nowym rozwiązaniem technologicznym, tzw. *transputerem* [14].

Poza wskazaną literaturą źródłową można znaleźć informacje o programowaniu współbieżnym i wybranych językach w bardzo wielu pracach, w szczególności w [1] i [12].

2. Język *occam*

Wilhelm Ockham, właściwie William of Occam, ur. ok. 1300 w Occham (hrabstwo Surrey, Anglia), zm. ok. 1349 w Monachium, filozof i teolog franciszkański, jeden z najwybitniejszych przedstawicieli późnej scholastyki. M.in. sformułował słynną zasadę (zwaną brzytwą okhamowską), wg. której "bytów nie należy mnożyć bez konieczności". (Wg [16])

2.1. Założenia i cechy języka

Język programowania occam [11] został opracowany przez brytyjską firmę INMOS około pięciu lat temu. Jest on szczególnie

dostosowany do architektury rozproszonej opartej na dużej liczbie jednakowych i prostych mikroprocesorów, które mogą się ze sobą komunikować. Firma INMOS wprowadziła ostatnio na rynek takie wyspecjalizowane mikroprocesory z lokalną pamięcią i portami komunikacyjnymi, zwane *transputerami* (por. 2.4). Również kompilatory języka occam są już komercyjnie dostępne. W języku tym można zarówno rozwiązywać typowe problemy, jak i projektować systemy oparte na transputerach.

Occam jest inspirowany przez CSP Hoare'a, zwłaszcza w aspekcie podejścia do współbieżności. Podstawowym pojęciem jest *proces*. Pary procesów komunikują się ze sobą poprzez *kanały* i komunikacja ta jest w pełni *synchroniczna*. Warto od razu zauważyć, że w CSP komunikacja jest synchroniczna i także dotyczy par procesów, ale nie występuje pojęcie kanału; obydwa procesy uczestniczące w wymianie informacji muszą jawnie wskazać z kim zamierzają się komunikować.

Program w occamie jest zbudowany z *procesów*. Proces wykonuje pewną sekwencję *akcji* i zatrzymuje się. Każda akcja odpowiada jednemu z następujących trzech *procesów prymitywnych*: *przypisanie*, *wejście* (lub *czytanie* z kanału), *wyjście* (lub *pisanie* do kanału). W każdej chwili proces może być *gotowy* i oczekiwać komunikacji poprzez jeden lub kilka kanałów. Kanały są jednokierunkowe. Gdy proces wejściowy i proces wyjściowy są gotowe do komunikacji poprzez ten sam kanał, następuje przekazanie wartości i procesy kontynuują swoje akcje. Jeżeli proces jest gotów i oczekuje czytania z kilku kanałów, to wartość jest pobrana z pierwszego kanału użytego do pisania przez inny proces.

Procesy mogą być nazwane i mieć parametry. W szczególności parametrami mogą być kanały. *Wartościami* w occamie są pojedyncze słowa, które mogą reprezentować *liczby, znaki, wartości logiczne*. Jedyne złożone struktury danych, to *jednowymiarowe tablice* słów, ale można indeksować także bajty. W *wyrażeniach* nie uwzględnia się pierwszeństwa operatorów. Zmienne są *lokalne* dla procesów -- współbieżne procesy mogą się komunikować wyłącznie przez kanały.

2.2. Procesy prymitywne i konstruktory

Procesy prymitywne, jak już wspomniano, to:

- *przypisanie* zmienna := wyrażenie
- *wejście* kanał ? zmienna
- *wyjście* kanał ! wyrażenie

Do tej grupy zalicza się także proces *pusty* oraz *oczekiwanie* opóźniające wykonywanie o wskazany okres czasu:

- *puste* SKIP
- *oczekiwanie* WAIT AFTER czas

Konstruktory służą do komponowania bardziej złożonych procesów. Rozróżniamy następujące konstruktory (czyli *procesy złożone*):

- *sekwencyjny* SEQ (odpowiadający instrukcji złożonej)
- *równoległy* PAR (opisujący współbieżne akcje)
- *alternatywny* ALT (wybierający spełniony *dozór*)
- *warunkowy* IF (odpowiadający instrukcji "jeśli")
- *iteracyjny* WHILE (odpowiadający instrukcji "dopóki")
- *replikacyjny* FOR (opisujący podobne procesy)

Przy zapisie procesów złożonych nie używa się tradycyjnych nawiasów zdaniowych *begin-end*, ale strukturę opisuje się przez *wcięcia* tekstu.

2.3. Proste przykłady

(a) Opis procesu *bufor*, który w pętli pobiera wartość z kanału *bufor.we* i przekazuje ją do kanału *bufor.wy*:

```
WHILE TRUE
  VAR x :
  SEQ
    bufor.we ? x
    bufor.wy ! x
```

(b) Rozwinięcie powyższego przykładu dla opisania bufora dwuelementowego (reprezentowanego przez dwa anonimowe procesy):

```
CHAN łącznik :
PAR
  WHILE TRUE
    VAR x :
    SEQ
      bufor.we ? x
      łącznik ! x
  WHILE TRUE
    VAR x :
    SEQ
      łącznik ? x
      bufor.wy ! x
```

(c) Bufor podwójny, pozwalający na współbieżne wejście i wyjście:

```
WHILE TRUE
  VAR x, y :
  SEQ
    PAR
      bufor.we ? x
      bufor.wy ! y
    PAR
      bufor.we ! x
      bufor.wy ? y
```

2.4. Transputery

Jak wspominaliśmy, istnieje istotny związek między occamem a transputerami. *Transputer* jest to wyspecjalizowany prosty mikrokomputer, składający się z procesora, pamięci, zegara, pewnej liczby portów oraz łącza do urządzeń peryferyjnych. Procesor transputera *realizuje bezpośrednio procesy occamu*; można więc uznać, że occam jest asemblerem dla transputerów.

Od niedawna transputery są już dostępne komercyjnie. Na przykład transputer IMS T424, będący płatkiem o powierzchni paru centymetrów kwadratowych, ma następujące parametry:

* pełna 32 bitowa architektura, 10 MIPS
* zredukowany zbiór instrukcji
* 4 KB pamięci statycznej RAM
* multipleksowane łącze do pamięci, 4 GB
* 8 bitowe łącze do urządzeń
* cztery porty komunikacyjne

3. Język *Edison*

Thomas Alva Edison, ur. 1847 w Milan (Ohio, USA), zm. 1931 w West Orange (New Jersey, USA), elektrotechnik, wszechstronny wynalazca. Był samoukiem, jedynie 3 miesiące uczęszczał do szkoły powszechnej. Opatentował ponad 1000 wynalazków, m.in. fonograf, żarówkę, elektrodynamiczny hamulec kolejowy. (Wg [16])

3.1. System Edison

Autor systemu Edison, Per Brinch Hansen, od kilkunastu lat zajmuje się systemami operacyjnymi i językami programowania, zarówno od strony badawczej i metodycznej, jak i praktycznej. Jest on głównym projektantem systemu operacyjnego komputera *RC 4000*, którego prototyp był zainstalowany w Zakładach Azotowych w Puławach, a który to system jest cytowany w każdym poważnym podręczniku o systemach operacyjnych. Rozwijając dalej swoje pomysły Brinch Hansen opracował różnego rodzaju narzędzia do programowania systemowego i współbieżnego. Jest on m.in twórcą tzw. *warunkowych rejonów krytycznych*, z których powstały później *monitory*. Języki programowania, w których występują monitory są *bezpieczne* w tym sensie, że dostarczają wyrafinowanych i złożonych narzędzi oraz narzucają duży rygor na programistę. Dzięki temu wiele błędów można wykryć już na poziomie kompilacji i znacznie łatwiej jest wykazywać poprawność programów. Wadą tego podejścia jest spora złożoność i objętość tych języków. Parametry

pierwszych prostych mikrokomputerów w sposób naturalny narzuciły ograniczenia na złożoność ich podstawowego oprogramowania. Brinch Hansen ujrzał w tym szansę całkowitego zerwania z przeszłością i zastosowania nowego podejścia, którego podstawową cechą jest *prostota*. Cytując z [4], str.2 :
System oprogramowania komputera osobistego jest prosty, jeżeli można się nauczyć jego używania w 1 dzień, poznać jego szczegóły w 1 miesiąc, a skonstruować go w 1 rok.

Dla spełnienia tego postulatu Brinch Hansen zaproponował użycie jednego języka wysokiego poziomu do opracowania kompletnego podstawowego oprogramowania systemowego. Język ten, *Edison*, jest zdaniem jego twórcy "mniejszy i prostszy niż Pascal, ale mocniejszy od kombinacji języków Pascal i Concurrent Pascal".

System Edison był opracowany pierwotnie dla mikrokomputera PDP 11/23, a następnie zainstalowany też na IBM PC. W skład oryginalnej wersji systemu wchodzą następujące programy napisane w języku Edison:

* *Kompilator języka Edison*	4200 wierszy
* *System operacyjny*	1200 wierszy
* *Edytor ekranowy*	500 wierszy
* *Redaktor tekstu*	400 wierszy
* *Program drukujący*	400 wierszy
* *Asembler Alva dla PDP 11*	1600 wierszy
Razem ok.	8300 wierszy

oraz

* *jądro* napisane w asemblerze	1800 wierszy
Razem ok.	10100 wierszy

3.2. Cechy języka Edison

Edison ma swoje korzenie w następujących językach: *Pascal, Concurrent Pascal* oraz *Modula 2*. Prostota języka jest uzyskana przez eliminację tych konstrukcji, które nie były niezbędne, oraz przez zastąpienie pewnych złożonych konstrukcji bardziej elementarnymi. Tym samym więc Brinch Hansen odszedł od swoich uprzed-

nich idei języka rygorystycznego i przez to bezpiecznego. Wynika to po części z rozwoju metodyki programowania współbieżnego, a po części z konieczności ograniczenia języka do niezbędnego minimum. W efekcie więc w języku Edison nie występują:

- *typy okrojone*
- *typ real*
- *warianty w rekordach*
- *pliki*
- *wskaźniki*
- *instrukcje skoku*
- *instrukcje wyboru* (case)
- *instrukcje "powtarzaj" i "dla"* (repeat, for)
- *instrukcje wiążące* (with)
- *monitory, klasy*
- ...

A oto lista podstawowych pojęć języka Edison:

* *Typy elementarne:*
  - *standardowe* (int, bool, char)
  - *wyliczeniowe* (enum)
* *Rekordy, tablice, zbiory*
* *Stałe, zmienne, wyrażenia*
* *Konstruktory*
* *Procedury*
* *Moduły i obiekty eksportowane*
* *Instrukcje puste* (skip), *przypisania, procedur*
* *Instrukcje warunkowe* (if)
* *Instrukcje iteracyjne* (while)
* *Instrukcje współbieżne* (cobegin)
* *Instrukcje synchronizujące* (when)

3.3. Przykład

Dla porównania z opisem buforów w języku occam (p.2.3) przedstawimy teraz prosty przykład modułu w Edisonie definiującego abstrakcyjny typ *bufor* i operacje na nim wykonywalne: *put*, *get*. Bufor jest jednoelementowy. Operacja *put* jest wykonalna

tylko gdy bufor jest pusty, natomiast operacja *get* tylko gdy bufor jest zapełniony. Należy ponadto zagwarantować wyłączność dostępu do bufora (w danej chwili tylko jeden proces może wykonywać operację *put* lub *get*). Chodzi więc w rzeczywistości o uzyskanie efektu *monitora*.

```
module
  var slot : char; full : bool
 *proc put(c : char)
  begin
    when not full do
      slot := c; full := true
    end
  end
 *proc get(var c : char)
  begin
    when full do
      c := slot; full := false
    end
  end
begin full := false end
```

Bufor jest reprezentowany przez lokalną zmienną (*slot*), która jest używana wyłącznie wewnątrz modułu i nie jest znana na zewnątrz. Podobnie jest ze zmienną roboczą *full*. Natomiast na zewnątrz modułu korzystamy z procedur *put* i *get*, które są *eksportowane* (fakt ten jest zaznaczony symbolem "*"). W module inicjuje się także niektóre obiekty lokalne (zmienna *full*). Semantyka instrukcji synchronizującej zapewnia wzajemne wykluczanie przy dostępie do zmiennej dzielonej -- mamy więc rzeczywiście pełną zgodność z monitorem.

3.4. Uwagi końcowe

Edison był pomyślany oryginalnie jako język do programowania systemów rozproszonych. Procesy są opisywane poprzez *instrukcję współbieżną* postaci:

```
cobegin 1 do SL1
        also 2 do SL2
        ...
        also n do SLn
end
```

Etykiety procesów, stałe 1, 2, ..., n, były wprowadzone po to, aby można było przyporządkowywać procesor procesowi. Jednakże wzorcowe realizacje Edisona były wykonane przez Brinch Hansena na komputerach jednoprocesorowych i tym samym równoległość jest symulowana.

W Instytucie Informatyki UW skonstruowano system wielomikroprocesorowy z językiem Edison, oparty na kilku mikrokomputerach ZX Spectrum połączonych w sieć. Wybór typu komputera był podyktowany ówczesną sytuacją sprzętową Instytutu. Opis tego systemu jest zamieszczony w [7]. Obecnie są prowadzone prace nad językiem o roboczej nazwie *Edison Plus*, który jest inspirowany zarówno przez język Edison, jak i język Joyce [5] (por. następny punkt).

4. Język *Joyce*

James Augustine Aloysius Joyce, ur. 1882 w Dublinie, zm. 1941 w Zurychu, pisarz irlandzki piszący po angielsku, jeden z najwybitniejszych nowatorów prozy XX wieku. Dzieła Joyca, prześladowane przez cenzurę i nawet palone przez władze różnych krajów (Ulisses, napisany w 1922 r.), były przez wiele lat przedmiotem zażartych polemik krytyków. (Wg [16])

Język programowania Joyce [5] został opracowany i zaimplementowany (na mikrokomputerze typu IBM PC) przez Brinch Hansena w 1986 roku. Jak podaje autor, "Joyce jest semantycznym wariantem CSP z notacją Pascala". W języku występuje większość pojęć i konstrukcji pascalowych, z różnymi rozszerzeniami i modyfikacjami w kierunku CSP.

Program w języku Joyce składa się z zagniezdżonych procedur definiujących abstrakcyjne maszyny zwane *agentami* (są to odpowiedniki procesów). Wykonanie programu oznacza uaktywnienie inicjalnego agenta, który może dynamicznie uaktywniać podagentów. Agent i jego podagenci wykonują się *współbieżnie*. Zmienne agenta są lokalne, niedostępne dla innych agentów. Agenci komunikują się ze sobą poprzez *kanały dwukierunkowe*. Komunikacja ta jest synchroniczna, jak w CSP. Kanały są tworzone dynamicznie i są dostępne poprzez *zmienne portowe*.

Kompilator języka ma trzy przebiegi i został napisany w Pascalu. Cały tekst kompilatora, to zaledwie 3300 wierszy.

BIBLIOGRAFIA

1. Ben-Ari, M. *Principles of Concurrent Programming*, Prentice-Hall, Englewood Cliffs, New Jersey (1982).

2. Brinch Hansen, P. "The Programming Language Concurrent Pascal", *IEEE Trans. on Software Engineering*, 1, 2, 199-207 (1975).

3. Brinch Hansen, P. "Edison - A Multiprocessor Language", *SOFTWARE -- Practice and Experience*, 11, 4, 325-361 (1981).

4. Brinch Hansen, P. *Programming a Personal Computer*, Prentice-Hall, Englewood Cliffs, New Jersey (1983).

5. Brinch Hansen, P. "Joyce - A Programming Language for Distributed Systems", *SOFTWARE -- Practice and Experience*, 17, 1, 29-55 (1987).

6. Dijkstra, E.W. "Cooperating Sequential Processes", *Programming Languages*, F.Genuys (editor), Academic Press, NY (1968).

7. Dubnicki, C., Madey, J., Wyglądała, W. "Edison-N -- An Edison Implementation for a Network of Microcomputers", *Sprawozdania Instytutu Informatyki UW*, nr 155, Warszawa (1986); ukaże się w *SOFTWARE -- Practice and Experience*.

8. Hoare, C.A.R. "Monitors: An Operating System Structuring Concept", *Communications of the ACM*, 17, 10, 549-557 (1974).

9. Hoare, C.A.R. "Communicating Sequential Processes", *Communications of the ACM*, 21, 8, 666-677 (1978).

10. Holt, R.C. *Concurrent Euclid, The Unix System and Tunis*, Addison-Wesley, Reading, Massachusetts (1983).

11. INMOS Limited, *Occam Programming Manual*, Prentice-Hall, Englewood Cliffs, New Jersey (1984).

12. Iszkowski, W., Maniecki, M. *Programowanie współbieżne*, Wydawnictwa Naukowo-Techniczne, Warszawa (1982).

13. Madey, J. "Systemy operacyjne dla mikrokomputerów", w materiałach drugiej Szkoły Jesiennej PTI *Współczesne kierunki rozwoju informatyki*, Mrągowo (1985).

14. May, D., Shepherd, R. "Occam and the Transputer", *Concurrent Languages in Distributed Systems*, G.L.Reijns, E.I.Dagless (editors), Elsevier Science Publishers, North-Holland (1985).

15. Pyle, I.C. *Ada*, Wydawnictwa Naukowo-Techniczne, Warszawa (1986).

16. *Wielka Encyklopedia Powszechna*, Państwowe Wydawnictwo Naukowe, Warszawa (1965).

Jesienna Szkoła PTI
Mrągowo, grudzień 1987

O INFORMATYCE - PO POLSKU

mgr inż. Barbara Osuchowska
Wydawnictwa Naukowo-Techniczne
Redakcja Informatyki i Podstaw Techniki
ul. Mazowiecka 2/4
00-048 Warszawa, tel. 267271 w. 276 lub 254

1. Wprowadzenie

Na treść tekstu nieliterackiego - czy to naukowego, zawodowego, czy to podręcznika lub nawet pracy popularnonaukowej - składają się różnego rodzaju elementy, m.in. pojęcia i definicje, twierdzenia, dowody, wnioski, dyskusje, a także wszelkie opisy zjawisk i rzeczy. Aby podać je w postaci jasnej i zrozumiałej dla czytelnika, trzeba się posłużyć, i to równocześnie, rozmaitymi środkami wyrazu. Zagadnienia te - w odniesieniu do literatury z dziedziny informatyki - są tematem niniejszego wykładu.

Do podstawowych środków wyrazu, którymi posługuje się autor pisząc książkę, artykuł, sprawozdanie z pracy badawczej lub jeszcze inny tekst, należą:

(1) język literacki;

(2) wyrazy i zwroty będące metodycznym narzędziem służącym do nadania treści odpowiedniej postaci, czyli tzw. pomocniczy aparat

językowy:

(3) wyrazy i zwroty o ściśle określonym znaczeniu, czyli terminologia specjalistyczna właściwa danej dyscyplinie naukowej.

Pierwszym ze środków wyrazu jest język literacki. Wbrew przyjętej tutaj, wobec braku innej, ogólnej nazwie "nieliterackie", teksty naukowe, zawodowe itp. powinny być pisane w sposób z natury rzeczy właściwy utworom literackim, a więc oczywiście poprawnie pod względem stylistyczno-językowym, leksykalnie urozmaicone, żywe, stylistycznie lekkie, wzbogacone ciekawymi związkami frazeologicznymi, obrazowymi porównaniami, a jeżeli się da, to i przenośnią, żartem, grą słów. Warstwa literacka jest tkanką łączną, ważnym spoiwem każdego tekstu. Ułatwia czytelnikowi korzystanie z książki lub artykułu i przyswojenie zawartej w nich wiedzy. Tekst nieliteracki wcale nie musi być napisany nudno; jeśli się chce (i potrafi), można mu nadać ciekawą formę.

Niestety jednak nawet najbogatsza i najpiękniejsza polszczyzna nie wystarczy, aby w dziele naukowym, podręczniku lub dziele nieliterackim jeszcze innego gatunku wyłożyć właściwe treści w sposób maksymalnie jednoznaczny. Są do tego niezbędne jeszcze dwa, już przez nas wymienione, inne środki wyrazu: wyrazy i zwroty będące metodycznym narzędziem niezbędnym do nadania treści odpowiedniej postaci, czyli pomocniczy aparat językowy, oraz wyrazy i zwroty o ściśle określonym, specyficznym znaczeniu, a więc wszelkie terminy specjalistyczne z różnych dziedzin nauki i techniki.

Przy operowaniu tymi obydwoma środkami wyrazu, a więc pomocniczym aparatem językowym i terminami specjalistycznymi, obowiązują autora reguły wręcz przeciwne niż przy pisarstwie

niejako potocznym, polegającym na wypełnianiu dzieła literacką tkanką łączną. Tutaj bogactwo językowe okazuje się nie zaletą, lecz wadą, a zbędne słowotwórstwo jest wręcz zakazane. Zachowanie pełnej jasności i jednoznaczności wywodów jest możliwe tylko wówczas, gdy pomocniczym aparatem językowym i słownictwem specjalistycznym autor posługuje się konsekwentnie w całym tekście - od jego pierwszej stronicy do ostatniej.

2. Zagadnienia językowe

Autora tekstu nieliterackiego obowiązują wszelkie ogólne (i szczególne) normy poprawnościowe dotyczące zarówno składni, jak i gramatyki języka polskiego. Autorom tekstów specjalistycznych, m.in z dziedziny informatyki, zagadnienia te sprawiają często wiele kłopotów. Dlatego w tym wykładzie sporo miejsca poświęcamy poprawności językowej tekstu.

Na początku rozważań, aby uniknąć ewentualnych nieporozumień, od razu wyjaśnijmy, że styl autora i poprawność językowa dzieła - to dwie całkiem odmienne sprawy. Każdy autor ma indywidualny sposób wyrażania myśli, swój własny styl, do którego ma pełne prawo. Styl autora nie może być jednak w żadnym wypadku w niezgodzie z ogólnie obowiązującymi normami językowymi. Nie może usprawiedliwiać błędów językowych, składniowych, niewłaściwej budowy zdań, braku logiki sformułowań itp.

Językowe wady tekstów specjalistycznych, m.in. z dziedziny informatyki, nie są niestety rzadkością. Przyczyny tego stanu rzeczy bywają, jak sądzę, różne.

Przyczyną pierwszą jest zwykła nieznajomość norm i reguł

językowych. Ludzka to sprawa, jako że na studiach specjalistycznych ani później, podczas pracy zawodowej, na poprawność formułowania myśli i sprawy językowe nikt najczęściej nie zwraca uwagi (ani tego nie uczy), a wiadomości szkolne dawno uległy zapomnieniu.

Przyczyną drugą błędów językowych jest autorska niestaranność, brak dbałości o poprawne wyrażanie myśli, i, co gorsze, nieprzywiązywanie do tego większego znaczenia, połączone niekiedy z pewną dozą nonszalancji: nie muszę być purystą językowym, jestem przecież specjalistą w swym zawodzie, a nie filologiem.

Dalsze przyczyny są głębsze i bardziej niepokojące. Wśród wielu autorów panuje bowiem niczym nie uzasadnione przekonanie, że język dzieła powinien być nie literacki, lecz "naukowy". W praktyce pod tym określeniem kryje się (a raczej niestety ujawnia wobec czytelników) język trudny, o zawiłych konstrukcjach, schematyczny, pełen neologizmów, słów żywcem zaczerpniętych z języków obcych, z żargonu zawodowego. Powstaje język sztuczny i napuszony; im dzieło jest napisane trudniej, tym przecież brzmi mądrzej, bardziej "naukowo" ... I następne, także niczym nie uzasadnione przekonanie, iż w owym języku "naukowym" reguły poprawnościowe właściwe językowi literackiemu autora nie obowiązują. Wiem z własnej praktyki, że takie poglądy nie jest łatwo zwalczać.

Mówiąc o normach i regułach językowych, nie mam tutaj oczywiście na myśli żadnych sztywnych, skostniałych kanonów. Dobrze przecież wiemy, że język się zmienia i zmieniać się będzie. I nikomu nie zależy na tym, żeby język polski mumifikować. Chodzi o coś zupełnie innego - o to, aby środki językowe służące

do przekazywania określonych treści były jak najbardziej skuteczne. Niewłaściwe środki wyrazu zmniejszają czytelność wypowiedzi, utrudniają ich zrozumienie, a często nawet je zniekształcają. Nie trzeba chyba nikomu udowadniać, że prostota i poprawność językowa, jasność i przejrzystość tekstu powinny być atrybutami monografii naukowej i książki zawodowej, nie mówiąc już o podręczniku i książce popularnonaukowej, które z założenia są trudne, jako że niosą nowe dla odbiorcy treści; trzeba zrobić wszystko, żeby ułatwić czytelnikowi ich przyswojenie.

Podczas opracowywania tekstu nieodzowną pomocą dla autorów są różne poradniki i słowniki języka polskiego. Zachęcam gorąco do korzystania z nich na co dzień. Jak wykazuje wieloletnia praktyka redaktora tekstów z dziedziny nauk ścisłych i techniki, m.in. z informatyki, istnieje pewna, niestety dość liczna grupa błędów i usterek językowych stale się powtarzających i dla tych tekstów typowych. W tym wykładzie ograniczymy się z konieczności tylko do wybranych spraw językowych.

Strona bierna

Oto przykłady stosowania strony biernej:

Partie szachowe rozgrywane z użyciem komputera są z zainteresowaniem studiowane przez mistrzów tej królewskiej gry.

Następnie zostaje opracowany system wyszukiwania informacji.

Praca jest oparta na wynikach badań poprzedników.

Na każdej stronicy maszynopisu tekstu z dziedziny informatyki można znaleźć zawsze co najmniej kilka tak zbudowanych zdań. Są ciężkie, niezgrabne, grzeszą brakiem zwięzłości. Mimo że w języku polskim unika się strony biernej, autorzy i tłumacze używają jej z

lubością.

Przypominamy: Zgodnie z tendencjami języka polskiego, zamiast strony biernej - wtedy, kiedy nie zależy na uwydatnieniu postawy biernej - lepiej używać:

- formy osobowej strony czynnej, np.

Partie szachowe rozgrywane z użyciem komputera z zainteresowaniem studiują mistrzowie tej królewskiej gry.

- formy zwrotnej, np.

Następnie opracowuje się system wyszukiwania informacji.

- innej formy bezosobowej, np.

Pracę oparto na wynikach badań poprzedników.

Słowo posiłkowe *zostać*

Jak wiemy, stronę bierną tworzy się za pomocą słów posiłkowych *być* lub *zostać* oraz imiesłowu biernego, np. *był programowany, został opracowany.* Autorzy nadużywają nie tylko strony biernej, lecz także słowa posiłkowego *zostać* w tych wszystkich kontekstach, w których nie ma potrzeby uwydatnienia chwili początkowej czynności lub stanu. Zamiast więc np.

Zostaną napisane efektywne algorytmy.

powinno się napisać

Będą napisane efektywne algorytmy (lub, lepiej: *Napisze się efektywne algorytmy*).

Zdanie zaś

Uruchamianie systemu zostało rozpoczęte w lecie ubiegłego roku.

jest poprawne.

Szyk przestawny

Oto następna, nie najnowsza zresztą maniera językowa, której ulegają autorzy i tłumacze. Ciągle czytamy takie np. zdania:

W buforze przechowywany jest tylko pojedynczy znak.

W książce omawiane są podstawowe właściwości Ady.

Wykryte zostały oba błędy w programie.

W zdaniach tych zastosowano szyk przestawny. Nie ma żadnego powodu, aby rezygnować z szyku prostego, właściwego językowi polskiemu. Teksty, w których wprowadzono szyk przestawny, nie brzmią ani na jotę bardziej naukowo. Lepiej więc napisać:

W buforze jest przechowywany tylko jeden znak.

W książce są omawiane podstawowe właściwości Ady.

Zostały wykryte oba błędy w programie.

Byłoby oczywiście jeszcze lepiej, gdyby autorzy tych zdań pozbyli się w nich strony biernej.

Podobna uwaga odnosi się do czasu przyszłego. Forma *czytał będę* jest niepoprawna, a szyk wyrazów można zmienić tylko wówczas, gdy operuje się bezokolicznikiem; wyrażenie *czytać będę* jest dopuszczalne.

Dysonansowe konteksty

Duża grupa błędów i niezręczności językowych powstaje w wyniku prawie bezmyślnego łączenia różnych wyrazów nie mających ze sobą niemal nic wspólnego w jedno pozornie spójne wyrażenie. Tę grupę niedostatków językowych ilustruje następujący, wręcz humorystyczny przykład:

Wzrasta moc zainstalowanych komputerów, spadają ceny sprzętu informatycznego, obniża się koszt czasu pracy maszyn, które są wysoko wydajne.

Moc się *zwiększa*, ceny *maleją* lub się *zmniejszają*, koszt *maleje* lub się *zmniejsza*, maszyny są *bardzo wydajne* lub po prostu *wydajne*. Tak to powinno brzmieć poprawnie. Wyrazy *wysoki*, *niski*, *podwyższać*, *obniżać* itp. mają w języku polskim wyraźnie określone znaczenie i nie należy nimi zastępować wyrazów w danym kontekście właściwych. Do natrętnych zbitek wyrazów, dysonansowych kontekstów, należą m.in. *wysoka wydajność* (zamiast *duża*, *dobra*), *niska jakość* (zamiast *zła*), *podwyższanie kwalifikacji* (zamiast *zwiększanie*), *niskie zaspokojonie potrzeb* (zamiast *małe*).

Dysonanse kontekstów bywają znacznie jaskrawsze. Trafiają się *wachlarze potrzeb i świadczonych usług*, *pakiety zagadnień*, *poważne problemy*, *pracujące soboty*, *odgrywanie wpływu*, a nawet humorystyczne, już przez innych cytowane wyrażenia, takie jak *sytuacja w jajach i mięsie*, *reforma szkolna potknie się na plastelinie i szybko przemijający ząb czasu, który niejedną łzę już osuszył*.

Imiesłów a podmiot zdania głównego

Do dość często spotykanych błędnych konstrukcji językowych należą zdania typu: *Idąc ulicą padał deszcz*. Nie wszystkie tego rodzaju błędy aż tak rzucają się w oczy, niektóre z nich są trudne do wychwycenia. Przyjrzyjmy się takim oto przykładowym zdaniom:

Zakładając x ≠ 0, nasz wzór przyjmie inną postać.

Tworząc bazę danych, informacje o publikacjach naukowo-technicznych są gromadzone na taśmach.

Postępując dwuetapowo, urządzenie do kopiowania może najpierw służyć do sporządzania matrycy z oryginału dokumentu, a następnie tę matrycę można użyć do wykonania wielu kopii.

Przytoczone zdania są bezsensowne, gdyż imiesłów nie odnosi się w nich do podmiotu, a powinien! Być może zdania te należałoby

sformułować np. tak:

Jeżeli założymy x = 0, nasz wzór przyjmie inną postać.

Aby utworzyć bazę danych, informacje o publikacjach naukowo-technicznych gromadzi się na taśmach.

Postępujemy dwuetapowo: najpierw na urządzeniu do kopiowania sporządzamy matrycę z oryginału dokumentu, a następnie używamy tej matrycy do wykonania wielu kopii.

Zapewne zdania te można by sformułować lepiej; powinien to jednak uczynić sam autor.

Zależności logiczne

W języku polskim, podobnie zresztą jak w innych językach, występują pary wyrazów służące do wyrażania różnych zależności logicznych. Do takich par należą m.in.: *jeżeli - to, wówczas - gdy, wtedy - kiedy, nie tylko - lecz także, dopóty - dopóki, o ile - o tyle, zarówno - jak i*. Przyjrzyjmy się takim oto zdaniom:

Do opisu systemu są nie tylko niezbędne opisy wejść, ale i zbioru stanów.

O ile jakaś wartość parametru nie znajduje się w programie, to można ją wziąć z generatora liczb losowych.

Pomiary trzeba wykonywać tak długo, dopóki nie otrzyma się pożądanego wyniku.

Zdania te nie są skonstruowane poprawnie, gdyż ich autorzy nie przestrzegali zasady, że użycie pierwszego wyrazu (tj. zapowiednika) z jakiejś pary wymaga użycia drugiego wyrazu (tj. następnika) koniecznie z tej samej pary, jako że są one nierozłączne. Zdaniom tym nadamy np. taką postać:

Do opisu systemu są niezbędne opisy nie tylko wejść, lecz także zbioru stanów.

Jeżeli jakaś wartość parametru nie znajduje się w programie, to można ją wziąć z generatora liczb losowych.

Pomiary trzeba wykonywać dopóty, dopóki nie otrzyma się pożądanego wyniku.

Zwróćmy uwagę na to, że w pierwszym zdaniu trzeba było także zmienić szyk wyrazów.

Zaimki względne jako wskaźniki zespolenia

Zaimki względne będące wskaźnikami zespolenia powinny być zawsze pierwszym członem zdania podrzędnego. Zamiast więc np.

Jest to funkcja, określenie której jest ważne.

należy pisać

Jest to funkcja, której określenie jest ważne.

Przeniesienie wyrazu *który* na dalsze miejsce w zdaniu podrzędnym jest dopuszczalne tylko wówczas, gdy na początku tego zdania znajduje się wyrażenie przyimkowe, tj. wyrażenie będące połączeniem przyimka z rzeczownikiem, przymiotnikiem, przysłówkiem, liczebnikiem lub zaimkiem. Wyrażeniami przyimkowymi są np. *za pomocą, w skład, w środku, na wprost, na przeciąg, w miarę, w pobliżu, na skutek, na dół, za granicą* itp.

Można więc np. pisać albo tak

Mamy tekst programu napisanego w Adzie, w którego skład wchodzą dwa rodzaje słów i różne znaki interpunkcyjne.

albo tak

Mamy tekst programu napisanego w Adzie, w skład którego wchodzą dwa rodzaje słów i różne znaki interpunkcyjne.

Redundancja

Nikomu z informatyków pojęcie redundancji nie jest obce. Z redundancją mamy do czynienia w tekstach autorskich wówczas, gdy w zdaniu występuje nadmiar określeń, np. wyrażeń jednoznacznych lub bliskoznacznych, nie wnoszących do wypowiedzi nic nowego,

będących powtórzeniem tego, co już raz zostało powiedziane (np. *cofnąć się do tyłu*).

W pewnych szczególnych przypadkach powtarzanie treści jednego wyrazu w uzupełniającym go określeniu (lub określeniach) może być stosowane celowo do ubarwienia wypowiedzi, wzbogacenia języka lub jako figura retoryczna (np. *ład i porządek*). Najczęściej jednak redundancja jest wadą, niepotrzebnym obciążeniem tekstu powodującym jego rozwlekłość, zmniejszającym czytelność wypowiedzi.

Do redundancji należą m.in. dwa bliskie znaczeniowo środki stylistyczne: tautologie i pleonazmy. Oto przykłady tego rodzaju wyrażeń wartych korekty:

Osiągnięcia realne i rzeczywiste
Problemy i zagadnienia
Kolejno jeden po drugim
Na stronicach od 1 do 4 włącznie
Jako dodany ekstra element
Zgrupowane razem
Wzajemnie powiązane między sobą
Dwie alternatywy
Odmienne warianty
Kontynuować dalej
Ponadto zaznaczono także powiązania
Potencjalna możliwość
Realizacja wykonania zadań
Planowanie przyszłościowe

Personifikacja pojęć i rzeczy

W ostatnich latach coraz częściej obserwuje się u autorów tendencję do traktowania rzeczy tak, jakby to one myślały i miały

wolną wolę, mogły spontanicznie działać. Jaskrawym tego przykładem jest ulubiony przez dziennikarzy zwrot *komputer oszukał rencistę*. Zawinili ludzie, a nie maszyna, którą widocznie źle zaprogramowano lub do której wprowadzono niewłaściwe dane. W tekstach pisanych przez specjalistów z dziedziny informatyki nie ma oczywiście tak daleko idących uproszczeń, występują w nich jednak zdania świadczące o braku precyzji formułowania myśli, o niedbałości językowej. Oto grupa przykładów wymagających interwencji:

Sprzęt komputerowy przedstawia sobą dobrą jakość.

Programy komputerowe dążą do szybkiego uzyskania potrzebnych wyników.

Etap trzeci zajmuje się opracowaniem numerycznego sterowania procesami technologicznymi.

Są to wskaźniki oceniające pracę zespołu programistów.

Poprzednie ustalenia nic nie mówiły na temat porządku, w którym należy wartościować węzły.

Głównym zadaniem pierwszej fazy prac jest poszukiwanie koncepcji statku jako środka realizującego sformułowane zadanie transportowe.

Biorąc dodatkowe dane z podsystemu opisu podstacji i odcinków, algorytm zestawiania pociągów tworzy rozkład jazdy adekwatny do potrzeb przewozowych.

Artykuł pisze o bazach danych.

Omawiana praca formułuje podstawowe warunki prowadzenia tego procesu.

W referacie [17] badano możliwości zastosowania tego algorytmu.

Wszyskie te wyrażenia trzeba, oczywiście, przeredagować, gdyż *sprzęt komputerowy odznacza się dobrą jakością, w programowaniu dąży się do szybkiego uzyskania potrzebnych wyników, etap trzeci - to opracowanie ..., pracę zespołu programistów ocenia się na podstawie wskaźników* itp.

Na tym kończymy omawianie wybranych zagadnień językowych - usterek i wad spotykanych m.in. w tekstach z dziedziny informatyki.

3. Pomocniczy aparat językowy

Aparat ten tworzą wyrazy i zwroty wzięte wprawdzie z języka potocznego, ale używane w sposób ograniczony w porównaniu z funkcją, którą pełnią w języku literackim. To ograniczenie widać wyraźnie na przykładzie homonimów, tj. wyrazów o jednakowym brzmieniu, lecz o różnych znaczeniach w języku potocznym. Rekrutując wyraz będący homonimem do pomocniczego aparatu językowego, pozbawia się go tym samym wieloznaczności, gdyż spośród różnych znaczeń wybiera się tylko jedno i wyrazu tego tylko w tym znaczeniu używa.

Na przykład z wyrazu *funkcja* korzysta się tylko powiedzmy w znaczeniu "zależność", odrzucając pozostałe znaczenia, takie jak "odwzorowanie", "rola", "stanowisko", a z wyrazu *operacja* korzysta się w danym tekście tylko w znaczeniu "czynność jednostkowa" i nie używa się go w znaczeniu "czynność w ogóle", "działanie", "zabieg", np. chirurgiczny, "ogół działań bojowych prowadzących do osiągnięcia celu strategicznego", "specjalna akcja", np. zbrojna. Dzięki takiemu postępowaniu unika się wieloznaczności wypowiedzi tak bardzo niepożądanej lub wręcz szkodliwej w tekstach omawianej kategorii. Przy nieprzestrzeganiu tej zasady w dziele poświęconym bankowości wyrażenie *rachunek wykazuje saldo dodatnie* jest dwuznaczne, bo oznacza albo to, że "na koncie są pieniądze", albo to, że "na skutek wykonania obliczeń wynikła wartość dodatnia". Być może szerszy kontekst pozwoli czytelnikowi poprawnie odczytać

sens, ale tego utrudnienia można by uniknąć decydując się na używanie homonimu *rachunek* tylko w jednym znaczeniu.

I podobnie, z grupy synonimów, tj. wyrazów bliskoznacznych, których w języku potocznym celowo używa się wymiennie, aby nie było nużących powtórzeń, do pomocniczego aparatu językowego wprowadza się tylko jeden z nich - ten, który autor uważa za najwłaściwszy - i w danym fragmencie tekstu lub w całym tekście nim konsekwentnie operuje. Sumienny autor nie pisze np. raz o *fazie* pierwszej, a potem o *kroku* drugim i *etapie* czwartym tego samego eksperymentu. Tego, co raz było u niego *metodą*, nie nazywa dalej ani *techniką*, ani *sposobem*. Tego, co raz nazwał *drganiami*, nie nazywa dalej *wibracjami*, *oscylacjami*, *wahaniami*. Jeżeli zdecydował się na *warunek wstępny*, to nie nazywa go potem *warunkiem wejściowym*, *początkowym* ani *wyjściowym* (od potocznego wyrażenia *punkt wyjścia*).

Czasami dwom lub większej liczbie wyrazów z tej samej grupy synonimów trzeba przypisać w tekście dwa lub większą liczbę znaczeń. I wówczas sumienny autor, po przemyśleniu sprawy, wprowadza niezbędne rozróżnienia (podając, jeśli trzeba, dokładne definicje) i ich przestrzega. Pisze więc świadomie raz o *zestawie*, a raz o *zespole*, wyraźnie te pojęcia rozgraniczając i ich konsekwentnie używając, gdyż to, co było nazwane *zestawem* w rozdziale pierwszym jego książki, nie może stać się *zespołem* w rozdziale piątym.

Niektóre z wyrazów i zwrotów należących do pomocniczego aparatu językowego są terminami specyficznymi dla danej dziedziny nauki lub techniki. Granica między nimi nie zawsze jest ostra. W tekście matematycznym wyraz *zbiór* jest terminem, w tekście zaś o kolekcjonerach - tylko słowem z pomocniczego aparatu językowego.

Do określenia zbioru kolekcjonerskiego w pracy matematycznej trzeba się zatem posłużyć innym wyrazem, gdyż nie jest to zbiór w sensie matematycznym, jako że mogą się w nim powtarzać te same elementy, np. występować dwa egzemplarze tej samej książki.

4. Terminologia specjalistyczna

Terminów specjalistycznych trzeba także używać konsekwentnie i jednolicie w obrębie całego tekstu. Zdarza się przy tym, że do wyrażenia dwóch bliskich sobie, aczkolwiek ze względów merytorycznych dwóch różnych pojęć, autorowi są potrzebne odmienne terminy. Jeśli tak się szczęśliwie składa, że w języku potocznym istnieje grupa synonimów, w których wspólnym zakresie znaczeniowym te pojęcia się znajdują, jeden synonim rezerwuje się do nazywania jednego pojęcia, inny zaś - drugiego. W biologii mamy zarówno *rodzaje*, jak i *gatunki*. W fizyce i technice obok siebie występują *moment* (np. bezwładności) i *chwila* (czasu). W elektrotechnice przymiotnik *stały*, będący członem terminu technicznego, jest czymś innym niż *trwały* lub *ustalony*. W informatyce terminy *zbiór*, *plik* (ang. file), *stos* i *kolekcja* odpowiadają zupełnie różnym pojęciom.

Jeżeli w literaturze z danej dziedziny do określenia tego samego pojęcia używa się wymiennie kilku różnych terminów, autor ma swobodę działania. Może wybrać z nich jeden - albo ten, który uznaje za najwłaściwszy, albo ten, który wydaje się mu najpowszechniejszy. Może więc kierować się bądź przesłankami merytorycznymi, bądź chęcią lepszego dotarcia do czytelnika. Wybranego terminu używa się konsekwentnie i jednolicie w obrębie całego tekstu; pozostałe można dla porządku przytoczyć jako uzupełnienie w nawiasach, w pierwszym miejscu występowania terminu

w tekście. Jeśli żaden z obiegowych terminów nie jest jeszcze utrwalony, warto - nie tylko w przekładach, lecz także, jeśli to możliwe, w pracach oryginalnych - podać w nawiasach obcojęzyczny pierwowzór.

Zarówno jednak autorzy, jak i tłumacze, zwłaszcza tekstów z nowych i rozwijających się dyscyplin naukowych i technicznych, a więc także z informatyki, są często zmuszeni do nazywania nowych pojęć - wymyślania nowych polskich terminów lub doboru spośród już istniejących w języku potocznym, a nawet specjalistycznym. I naukowiec - twórca nowego pojęcia, i tłumacz - autor polskiego odpowiednika obcojęzycznej nazwy pojęcia, są w takiej samej, trudnej sytuacji. Jak powinni sobie radzić?

Po pierwsze, należy sprawdzić w opublikowanych pracach, czy naprawdę dane pojęcie nie zostało już po polsku nazwane; dość często tak właśnie bywa. Ale jeśli nie, to co wtedy?

Trzeba postarać się dobrać już istniejące polskie słowo; jeśli to się nie uda, warto spróbować je utworzyć na wzór już istniejącego, korzystając ze źródłosłowu rodzimego, a jeśli i to nie wyjdzie, na podstawie terminu obcojęzycznego. W tym ostatnim wypadku warto sprawdzić, jaka jest etymologia tego terminu, i ew. sięgnąć do jego źródłosłowu łacińskiego lub greckiego, gdyż język polski łatwiej toleruje słowa pochodzące z tych właśnie języków niż z nowożytnych. Słowem, które lepiej spolszczyć przez łacinę niż przez angielski, jest np. *kwerenda* (ang. query, łac. quaerenda). Trzeba przy tym zwracać uwagę na brzmienie nowego słowa i jego walory estetyczne. Zawsze warto również od razu przemyśleć konsekwencje płynące z wprowadzenia nowego terminu, m.in. wziąć pod uwagę możliwość tworzenia wyrazów pochodnych; tak więc dobierając rzeczownik warto spróbować utworzyć pochodzący od

niego czasownik, przymiotnik lub przeciwnie. Na przykład termin *komputer*, który wyparł już niemal całkowicie poprzednio używany termin *maszyna cyfrowa*, okazał się lepszy nie tylko dlatego, że krótszy, lecz także dlatego, że daje możność operowania wyrazami pochodnymi, takimi jak *komputerowy*, *komputeryzacja*, *komputeryzować*, *mikrokomputer*. Inny przykład: dla anglosaskiego terminu *hash* informatycy znaleźli polski odpowiednik: *kodowanie mieszające*. Niestety, zgrabnych form pochodnych utworzyć się nie udało i wszyscy się posługują czasownikiem *haszować* (nie *hashować*! - przed takimi dziwolągami przestrzegamy). W tym miejscu pozostaje nam tylko pozazdrościć Anglosasom: every noun can be verbed; od każdego rzeczownika tworzą czasownik, i po kłopocie.

Jedne z wymyślonych terminów przyjmują się od razu, inne dopiero po dłuższym czasie uporczywego lansowania, jeszcze inne prawa obywatelstwa nie uzyskują. Przy nazywaniu nowych pojęć są bowiem próby udane i nieudane.

W matematyce istnieją świetne, czysto polskie wyrazy: *całka* i *różniczka*. Ich twórcy, Śniadeccy, mieli jeszcze inny pomysł: *sinus* i *cosinus* nazwali *wstawą* i *dostawą*; okazał się to całkowity niewypał. Profesor Tadeusz Banachiewicz nie mógł się pogodzić z myślą o tym, że przymiotnik *kwadratowy*, oznaczający potocznie "kształt", np. stołu, jest równocześnie używany w znaczeniu "podniesiony do drugiej potęgi". Wymyślił więc nowy: *kwadrasty* (!). Równocześnie jednak zawdzięczamy Banachiewiczowi piękne *krakowiany* i *rachunek krakowianowy*. Współczesną, udaną próbą terminologiczną jest wprowadzona przez młodą informatyczkę *krotka*, odpowiednik anglosaskiego *tuple*. Autorzy *tuple* wyjaśniają, że dobrali to słowo jako rym do *couple* (para, stadło); nasza *krotka* wzięła się od *stokrotki*. Prawda, że ładnie?

Oczywiście, w żadnym wypadku nie wprowadza się nowego terminu wówczas, gdy już istnieje ogólnie przyjęty. Na zwisy *meshie* zamiast *krawatów* się nie zgodziliśmy. Jeżeli jednak wprowadzenie nowego terminu jest konieczne, autor musi go zawsze dokładnie zdefiniować i, co więcej, w pierwszym miejscu występowania terminu w tekście. Należy to do podstawowych powinności autora objętych opublikowanym przez UNESCO w roku 1968 *Kodeksem dobrych obyczajów w dziedzinie publikacji naukowych* (Warszawa, WNT 1984).

Czasami autor i tłumacz tekstu z jednej dziedziny wiedzy muszą się posłużyć aparatem pojęciowym z innej. Na przykład w książce traktującej m.in. o nowoczesnej diagnostyce medycznej i stosowanych do tego urządzeniach występuje wiele pojęć z dziedziny informatyki. W takim wypadku nie powinno się wyważać otwartych drzwi: zamiast tworzyć wyrazy lub kalkować obcojęzyczne lepiej zajrzeć do wydanych w kraju książek z informatyki i posłużyć się już istniejącą polską terminologią. Nie jest to, niestety, częsta praktyka. Zdarza się nam spotykać w literaturze - nie tylko z dziedziny nauk przyrodniczych, lecz także technicznych - takie wyrażenia, jak np. żywcem przeniesione anglosaskie *hardware* i *hardware'owy* zamiast od dawna przyjętych i używanych w informatyce polskich odpowiedników *sprzęt* i *sprzętowy*.

Na marginesie naszych rozważań o słownictwie. Zauważmy, że coraz częściej w przekładach, a nawet w oryginalnych dziełach naszych autorów, powtarzają się takie wyrażenia, jak np. *zorientowany* lub *ukierunkowany*, będące dosłownym tłumaczeniem angielskiego *oriented*. Warto sobie uprzytomnić, że ten wyraz, powiedzmy w takim kontekście, jak *problem-oriented language*, został na angielszczyźnie niejako wymuszony, sztucznie dodany, gdyż samo *problem language* oznacza "trudny język", podobnie jak *problem child* - "trudne dziecko". Dlatego w polskim przekładzie

wyraz *oriented* należy w takich kontekstach najczęściej w ogóle pomijać i - w naszym przykładzie - zamiast *język zorientowany problemowo* lub *język ukierunkowany problemowo* pisać *język problemowy*.

Sprawą budzącą często emocje, będącą kością niezgody między autorami i tłumaczami a redaktorami tekstów nieliterackich, jest miejsce podawania przymiotników (określeń) w wyrażeniach składających się z dwu lub większej liczby wyrazów. Jak pisać? *Strukturalne programowanie* czy *programowanie strukturalne*? *Obliczeniowa złożoność* czy *złożoność obliczeniowa*?

Jedni autorzy uważają, że język polski lubi, aby przymiotniki były zawsze przed rzeczownikiem, i taki właśnie szyk słów w swych tekstach forsują. Inni wyrażają pogląd, że reguły obowiązujące w tej mierze w języku literackim nie są regułami wymagającymi przestrzegania w językach specjalistycznych (tj. w odniesieniu do pomocniczego aparatu językowego i terminów specjalistycznych), w których z kolei faworyzuje się kolejność przeciwną - przymiotnik po rzeczowniku. Sprawa wzajemnego stosunku rzeczowników i przymiotników nie jest prosta i nie ma rygorystycznych przepisów, które by ją normowały. Niemniej jednak wybór miejsca przymiotnika powinien być świadomy i umotywowany. Kolejność rzeczownik + przymiotnik czy przymiotnik + rzeczownik zależy od tego, co i jak chce się wyrazić, od znaczenia, które się chce nadać danemu wyrażeniu. *Teatralny gest* to coś zupełnie innego niż *gest teatralny*, *książka literacka* nie musi być *literacką książką*, wyrażenie *krytyczna ścieżka* jest pozbawione większego sensu, podczas gdy *ścieżka krytyczna* swój sens ma. Wszystko zależy od kontekstu, a więc od otoczenia zdaniowego, w którym te wyrażenia się znajdują. Przymiotnik podajemy po rzeczowniku wówczas, gdy ma on mieć znaczenie wyróżniające, gdy chcemy uwydatnić jakąś cechę zasadniczą, trwałą, gatunek lub - na tle

jakiegoś ogólniejszego wyobrażenia - jakąś cechę szczególną.

A jak postępować wówczas, gdy przymiotników towarzyszących rzeczownikowi jest więcej? Na to pytanie odpowiada nam Kazimierz Zbierski - autor "Przeglądu językowego", swego czasu publikowanego na łamach czasopisma *Normalizacja* (1975, XXV, 7-8, s. 74):

..."Aby nazwa była zgodna z polskim słowotwórstwem oraz obowiązującymi zasadami gramatyki i pisowni polskiej, m.in. termin wielowyrazowy powinien być utworzony we właściwej kolejności, przy czym: [...] Jeżeli nazwa rzeczownikowa ma dwa określenia, to najczęściej jedno następuje przed rzeczownikiem, drugie po rzeczowniku, np. *Polska Rzeczpospolita Ludowa, porcelanowy sprzęt laboratoryjny*. Jeśli nazwa rzeczownikowa ma więcej niż dwa określenia, to najczęściej wszystkie następują po rzeczowniku, ale z uwzględnieniem usystematyzowania cech nadrzędnych, współrzędnych i podrzędnych, np. *izolatory liniowe odciągowe szpulowe porcelanowe, wozy kopalniane średnie nieresorowane*."

Oto jeszcze inne zasady ogólne dotyczące słownictwa w tekście specjalistycznym.

W odniesieniu do wyrazów i zwrotów zarówno tworzących pomocniczy aparat językowy, jak i wchodzących w skład terminów specjalistycznych, należy przestrzegać symetrii językowej w antonimach, tj. parach wyrazów o przeciwstawnych znaczeniach. Nie można równocześnie pisać *absolutny* i *względny*, *dodatni* i *negatywny*, *aktywny* i *bierny* itp. Tego rodzaju parom należy nadać symetryczną postać, używając ogólnie przyjętych wyrazów polskich, a nie pochodzenia obcego. Podane tutaj przykładowo antonimy powinny więc wyglądać tak: *bezwzględny* i *względny*, *dodatni* i *ujemy*, *czynny* i *bierny*.

Tej ostatniej zasady należy przestrzegać w odniesieniu do wszystkich używanych wyrazów i zwrotów. Zamiast np. *produkt kartezjański* lepiej pisać *iloczyn kartezjański*, zamiast *realizacja* - wykonanie, zamiast *realizowalny* - *wykonalny*, zamiast *kompatybilny* - *zgodny*, zamiast *kalkulacja* - *obliczenie*, zamiast *kompleks* - *zespół*, *zestaw* itp., zamiast *operand* - *argument*, zamiast *inwariant* - *niezmiennik*, zamiast *relewantny* - *istotny*, zamiast *system ekspertowy* - *system doradczy* itp.itp.

BIBLIOGRAFIA

Barras R.: *Scientists must write*. London, Chapman and Hall 1979.

Osuchowska B.: *Poradnik redaktora i autora. Nauki ścisłe i techniki*. Warszawa, Wyd. Polskiego Towarzystwa Wydawców Książek (w druku)

Osuchowska B.: Redagowanie dzieła. Słownictwo. *Prasa Techniczna* 1986, 2, s. 20-22.

Jesienna Szkoła PTI
Mrągowo, grudzień 1987

TWORZENIE MODUŁÓW PROGRAMISTYCZNYCH W OPARCIU O ABSTRAKCYJNE TYPY DANYCH

Andrzej Salwicki
Instytut Informatyki
Uniwersytet Warszawski
PKiN pok.850
00-901 Warszawa

STRESZCZENIE
Chcemy przekazać następujący komunikat: programowanie może być procesem znacznie pewniejszym, bezpieczniejszym i bardziej wydajnym gdy twórca oprogramowania zacznie posługiwać się dwoma narzędziami: logiką algorytmiczną w procesie specyfikacji oprogramowania i językiem Loglan w procesie układania programów.Nasze rozważania rozpoczynamy od wyliczenia przypadków (sytuacji) ważnych z punktu widzenia metodologii programowania. Następnie przedstawiamy wybrane cechy języka programowania Loglan i uzasadniamy tezę, że narzędzia jakie Loglan dostarcza są użyteczne we wszystich wymienionych sytuacjach. By wspomnieć kilka z nich: abstrakcyjne struktury danych, hierarchie podsystemów, protokoły współdziałania procesów współbieżnych etc. Wskazujemy na towarzyszący Loglanowi oryginalny system dedukcyjny logiki algorytmicznej AL. System ten umożliwia analizę i weryfikację modułów programistycznych. Przykład omawiający większy program symulacji oddziału banku zamyka to opracowanie.

1. Wstęp

W niniejszym opracowaniu pragniemy wykorzystać wyniki tymczasowe programu badawczego, który co prawda nie jest jeszcze zakończony, ale jego dotychczasowe wyniki pozwalają już sformułować wnioski o praktycznym znaczeniu. Głównymi celami pogramu są uzyskanie narzędzi dla szybkiej produkcji oprogramowania i wprowadzenie formalnych metod naukowych do tej produkcji , co pozwoli, mamy nadzieję, uczynić produkcję oprogramowania rzeczywistym procesem technologicznym.

Szybkość w oferowaniu nowych produktów programistycznych, możliwość szybkiego reagowania na potrzebę ulepszeń są ważne. Ewentualne zyski mają znaczenie ekonomiczne, technologiczne a nawet strukturalne czy strategiczne. Dla obecnego stanu produkcji oprogramowania charakterystyczny jest "manufakturowy" sposób wytwarzania software'u. Większość dużych systemów programistycznych jest dostarczana z dużymi opóźnieniami, co gorsza systemy te nie są niezawodne w działaniu. Zjawisko to jest najlepszym argumentem na rzecz naszej tezy, że nie nadeszła jeszcze era przemysłowej produkcji oprogramowania. Naszym zdaniem na to by można było mówić o takiej erze powinny być najpierw spełnione dwa warunki: 1° produkcja oprogramowania powinna być oparta na naukach podstawowych, tak jak inżynieria oparta jest o matematykę i fizykę, 2° powinniśmy mieć możliwość składania dużych systemów programistycznych z modułów (być może pochodzących od różnych producentów), tak jak samochody, okręty itp. są składane z części pochodzących z różnych wytwórni. Czy jest nadzieja na spełnienie tych warunków w nieodległej przyszłości? Uważamy, że tak, że jest to możliwe. Stan wiedzy i umiejętności środowisk programistów i badaczy pracujących w informatyce pozwalają na opracowanie potrzebnych rozwiązań. Spośród różnych głosów i propozycji w tej sprawie chcemy przedstawić pogląd do którego doszliśmy w Instytucie Informatyki Uniwersytetu Warszawskiego podczas prac nad dwoma kolejnymi programami badawczymi AL - logika algorytmiczna i LOGLAN - uniwersalny język programowania. Wyniki osiągnięte w projekcie AL doprowadziły do wykrycia podstawowych praw wnioskowania o procesach obliczeniowych, praw niezależnych od wyboru komputera, języka programowania, danych itp. Wyniki te stanowić mogą podstawę pewnej metodologii programowania, zwłaszcza

w powiązaniu z nowymi narzędziami programowania jakie oferuje LOGLAN. Logika algorytmiczna może dostarczyć narzędzi niezbędnych do formułowania specyfikacji oprogramowania, może też służyć jako system dedukcyjny umożliwiający analizę modułów oprogramowania. Posługując się językiem logiki algorytmicznej jesteśmy w stanie dostarczyć pełnego aksjomatycznego opisu typów danych, zarówno pierwotnych typów danych z zadanego języka programowania jak i abstrakcyjnych typów danych, jakie wytwarzane są w procesie programowania. Okazuje się, że taka aksjomatyzacja czyni łatwiejszą analizę własności semantycznych, np. poprawności programów. Ponadto udało się znaleźć formalny odpowiednik pojęcia implementacji struktury danych, jest nim mianowicie pojęcie interpretacji jednej teorii algorytmicznej w drugiej. Projekt LOGLAN wniósł drugi czynnik do naszych rozważań o metodologii programowania: możliwość składania, rozszerzania i stosowania modułów oprogramowania, pochodzących od różnych autorów. Znana jest możliwość tworzenia bibliotek algorytmów. Tym czego potrzebuje nasze środowisko jest możliwość przechowywania, manipulowania, składania i aplikacji modułów, które implementują (pod)systemy. Pewne szanse działania w tym kierunku stwarzają pakiety w Adzie. Z żalem stwierdzamy, że zapoznane zostało narzędzie bardziej ogólne, a mianowicie prefiksowanie, wymyślone dwadzieścia lat temu przez twórców Simuli i rozwinięte w języku Loglan. Rozliczne zalety prefiksowania czynią programowanie w Loglanie wysoce efektywnym. Co prawda obiektowo zorientowany styl programowania w Simuli i Loglanie jest bardzo odmienny od stylu programowania w innych językach. Nie możemy tu nie wspomnieć o pewnych ograniczeniach nałożonych na programy w SIMULI, ograniczenia te zdają się zaprzeczać potencjalnym korzyściom jakie możnaby osiągnąć programując w tym języku. Ostatnio daje się zaobserwować rosnące zainteresowanie programowaniem obiektowo zorientowanym (języki Smalltalk, Loops, Paragon, ScratchPadII i wiele innych).

2. Potrzeby narzędziowe metodologii programowania.

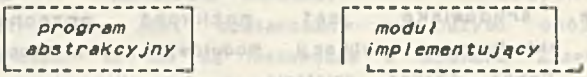
2.1 Abstrakcyjne typy danych

W większości przypadków nasze zadanie programistyczne polega na zaprogramowaniu algorytmu w którym używane są operacje spoza repertuaru operacji dostarczanych przez aktualne środowisko. (Na środowisko składają się komputer + system operacyjny + język programowania). Innymi słowami, nasz przyszły program ma być wykonany w nowej strukturze danych jaka powinna dopiero zostać zaimplementowana. W roku 1972 C.A.R.Hoare zauważył, że w takim przypadku zadanie powinno zostać *podzielone* na dwa podzadania:

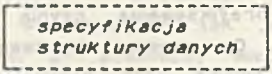
i) określenie i implementacja struktury danych,

ii) projekt, analiza i zastosowanie programu "abstrakcyjnego".

Zgodnie z tą radą powinniśmy wytworzyć dwa moduły:



Jedynym ogniwem pośredniczącym pomiędzy tymi dwoma modułami programistycznymi powinna być



Zgodnie z zasadą podziału (faktoryzacji) można powołać dwa zespoły programistów do pracy nad tymi dwoma modułami. Zespół opracowujący program abstrakcyjny powinien korzystać wyłącznie ze specyfikacji. Oznacza to, że semantyczne własności programu powinny być wyprowadzone (uzasadnione) tylko w oparciu o aksjomaty zawarte w specyfikacji. Zespół implementujący stosuje specyfikację jako kryterium poprawności implementacji. Zalety tej metody są wielorakie. Zasada podziału umożliwia wykonywanie programu abstrakcyjnego w towarzystwie różnych modułów implementujących. Poprawny program nie musi być przy tym aktualizowany, będzie to jeden program dla wielu modułów implementujących. Możemy zyskać bądź stracić na efektywności obliczeń w zależności od naszego wyboru modułu implementującego strukturę danych. Inna zaleta tej metody polega na możliwości wielokrotnego wykorzystania raz

opracowanego modułu implementującego. Moduł taki może być pojmowany jako język problemowo zorientowany.

W pracy nad oprogramowaniem dają się więc wyróżnić co najmniej trzy wyraźne etapy:

a) sformułowanie specyfikacji czyli aksjomatyzacji struktury danych,

b) projekt programu abstrakcyjnego i jego weryfikacja oparta na specyfikacji,

c) realizacja struktury danych i weryfikacja jej poprawności (w tym wypadku także opiearmy się na specyfikacji i sprawdzamy prawdziwość aksjomatów struktury danych w danej konkretnej implementacji).

2.2 SYSTEMY

Wielkie znaczenie ma możliwość manipulacji systemami. Obecnie mamy taką możliwość w odniesieniu do algorytmów, istnieją biblioteki procedur. Brakuje natomiast możliwości wykorzystywania modułów opisujących systemy w procesie tworzenia większych systemów. Na wzór procesów produkcji hardware'u powinna istnieć możliwość brania z półek podsystemów i składania ich w nowe systemy. Przede wszystkim więc potrzebujemy narzędzi do opisu i implementacji systemów. W tym kontekście pojawia się angielskie słowo "encapsulation", postuluje się by można było tworzyć "encapsulated systems" co w tłumaczeniu na polski może brzmieć systemy otorbione lub opakowane.

Co rozumiemy przez system? Każdą kolekcję składającą się ze zbioru elementów - zwanego *uniwersum*, ze zbioru operacji i ze zbioru relacji. Jest więc system strukturą algebraiczną, to pojęcie jest znane i podstawowe dla matematyki. Praktyka programowania jednak nakłada dodatkowe wymagania i spełnienie ich może być trudne lub wręcz niemożliwe do osiągnięcia w języku matematyki (przynajmniej takim jakim jest on dzisiaj). Poniżej wyliczymy niektóre trudności lub postulaty.

Po pierwsze uniwersum nie zawsze jest zbiorem podobnych obiektów. Często mamy do czynienia z przypadkiem, w którym uniwersum jest podzielone na rozłączne podzbiory zwane sortami. Na przykład, w systemie stosów rozważamy dwa sorty: E - elementów i S - stosów. Operacje mogą wymagać by argumenty były pewnych określonych sortów np. pierwszy argument ma być sortu E a drugi sortu S. Wynik operacji ma też pewien określony sort. Łatwo

zgadnąć, że sorty to po prostu typy w języku programowania, ale czy zawsze jesteśmy w stanie określić typ tak by odpowiadał on potrzebom implementowanego systemu? Czy narzędzia do tworzenia typów np. w Pascalu są to tablice i rekordy, są dostateczne?

Okazuje się ważnym by można było tworzyć systemy, w których obiekty zachowują pewną dynamikę, są zdolne do wykonywania swoich własnych akcji. Możemy tu wyróżnić trzy wzrastające poziomy takiego żądania:

A) W tym przypadku, obiekty są pasywne, mogą jednak na żądanie pewnego aktywnego obiektu wykonać pewne określone operacje. Dlaczego taka sytuacja jest pożądana? Po pierwsze, możemy w ten sposób tworzyć wyrażenia o znacznie większej czytelności. Inne uzasadnienie wypływa z obserwacji, że ten rodzaj pracy z obiektami systemu pozwala zaoszczędzić na czasie (i pamięci) potrzebnych na przekazywanie parametrów. Porównajmy dwa opisy systemu liczb zespolonych napisane w Pascalu i w Loglanie:

PRZYKŁAD

```
[PASCAL]                                                    [LOGLAN]

TYPE COMP = RECORD                UNIT COMP: CLASS(RE,IN:REAL);
   RE,IN : REAL
END RECORD;                                                  UNIT ADD:FUNCTION(Z:COMP):COMP;
                                                                      BEGIN
FUNCTION ADD(Z,T:COMP):COMP;                                RESULT:=NEW COMP(RE+Z.RE,IN+Z.IN)
   VAR ADDTEMP: COMP;                                        END ADD;
BEGIN
   ADDTEMP:= NEW COMP;                                       UNIT MULT:FUNCTION(Z:COMP):COMP;
   ADDTEMP.RE:= Z.RE+T.RE;                                   BEGIN
   ADDTEMP.IN:= Z.IN+T.IN;                                       RESULT:=NEW COMP(RE*Z.RE-IN*Z.IN,
   ADD:= ADDTEMP                                                 RE*Z.IN+IN*Z.RE)
END ADD;                                                          END MULT;

FUNCTION MULT(Z,T:COMP):COMP;          END COMP;
   VAR MULTEMP: COMP;
BEGIN
   MULTEMP:= NEW COMP;
   MULTEMP.RE:= Z.RE*T.RE-Z.IN*T.IN;
   MULTEMP.IN:= Z.IN*T.RE+Z.RE*T.IN;
   MULT:= MULTEMP
END MULT; /P
```

Podobieństwa są wyraźne, różnice wymagają słowa komentarza. Po lewej stronie widzimy trzy moduły które powinny współdziałać. Lecz co się zdarzy gdy nieuważny programista przesunie jeden z tych modułów w inne miejsce programu? Osierocone dwa pozostałe moduły nie mają już sensu o jaki nam chodzi. Powinniśmy pamiętać,

że tylko razem, w trójkę, te moduły tworzą sensowną całość. Po prawej stronie mamy opakowany moduł. Gdy zamierzamy go użyć, zawiera on wszystko co niezbędne dla działania na liczbach zespolonych. Zyski są wielostronne: w fazie programowania możemy skrócić tekst modułu COMP ponieważ Loglan dopuszcza parametry w obiektach klas, a Pascal nie stwarza takiej możliwości, w fazie wykonywania możemy zyskać na czasie dostępu do atrybutów RE i IM dla jednego z argumentów działania ADD i MULT. Operacje te są w naszym przykładzie loglanowskim zdefiniowane wewnątrz każdego obiektu klasy COMP, znowu inaczej niż w Pascalu, i mogą wobec tego wykorzystywać lokalne atrybuty w klasie COMP. Zyskujemy w ten sposób na czasie. Zauważmy jeszcze różnicę w sposobie posługiwania się tymi funkcjami. Załóżmy, że mamy deklarację:

```
VAR Z,T,U,V: COMP
```

wtedy wyrażenie (z+t)*(u+v) przyjmuje następujące formy

```
{PASCAL}                          {LOGLAN}
MULT(ADD(Z,T), ADD(U,V))          Z.ADD(T).MULT(U.ADD(V))
```

Warto też zwrócić uwagę na składnię. W przypadku pascalowym wyrażenie jest zapisane w notacji przedrostkowej i ma szesc parametrów aktualnych do przekazania. W przypadku loglanowskim nazwy działań znajdują się pomiędzy argumentami a parametrów przekazuje się o połowę mniej.

B) W programowaniu gier, w pakietach symulacyjnych często zamierzamy tworzyć obiekty, które mogą być aktywne w pewnych momentach a w innych są pasywne. Tak jak gracze w grze, pewne obiekty są wzywane do podejmowania swych czynności w miejscach gdzie te czynności były ostatnio przerwane. A gdy obiekt taki wykona akcje odpowiadające jednemu ruchowi w grze zawiesza swą aktywność, dopóki nie będzie ponownie obudzony. Można w tym miejscu dokonać rozróżnienia pomiędzy schematami postępowania w których obiekty "budzone" są wskazywane jawnie przez podanie ich nazwy (tzn. zmiennej wskazującej na takie obiekty) a schematami w których aktywność jest przekazywana anonimowo temu obiektowi który nas budząc sam zrzekł się praw do aktywności. W obu tych przypadkach zakładamy, że jest conajwyżej jeden obiekt aktywny. Pierwszy schemat to schemat współprogramów czyli coroutin. Drugi schemat to schemat semicoroutin czyli quasiwspółprogramów.

C) W trzecim przypadku mamy do czynienia z systemami w których obiekty wykonują swe akcje współbieżnie. Obiekty takie nazywane są procesami.

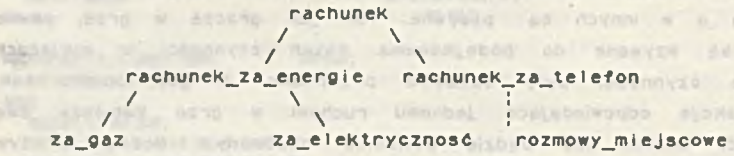
We wszystkich tych przypadkach obiekty mogą być manipulowane z zewnątrz, przez zdalny dostęp do ich atrybutów. Mogą one także być podmiotem działania samodzielnego, zgodnego z programem w nich zapisanym.

2.3 HIERARCHIE

Jest faktem dobrze znanym, że duże skomplikowane systemy mogą być projektowane, realizowane i pielęgnowane tylko pod warunkiem wprowadzenia odpowiedniej hierarchii. Hierarchia taka może odnosić się do różnych aspektów systemu. Czasem wystarczy rozważać hierarchię podzbiorów pewnego uniwersum obiektów. Rozpatrzmy dla przykładu ogólne pojęcie rachunku (wystawianego do zapłacenia). Każdy taki rachunek zawiera pewne wspólne rachunkom atrybuty np.

```
KWOTA_DO_ZAPLACENIA: ZLOTYCH
          ZAPLACONO: BOOLEAN
 ROK_MIESIAC_DZIEN:  DATA
```

a także inne atrybuty, odpowiadające specyficznym rachunkom. Można w szczególności mówić o rachunkach za energię, za telefon etc. Struktura podzbiorów może być dalej rozwinięta jako drzewo np.



Wydaje się istotnie ważnym posiadanie możliwość objęcia wspólnych cech rachunków przez wspólne (ogólne) algorytmy. Stąd postulat pewnego rozluźnienia zasady sprawdzającej zgodność typów. Chcemy mianowicie dopuścić by na zmienną typu rachunek można było podstawić obiekt typu rachunek_za_gaz. Ale nie odwrotnie, byłoby fatalnym błędem przypisanie zmiennej typu rachunek_za_gaz obiektu typu rachunek_za_telefon.

Omówiliśmy hierarchię podzbiorów zbioru = typu rachunek. W podobny sposób może wyłonić się naturalna potrzeba rozważania hierarchii podsystemów pewnego systemu.

2.4 PROTOKOŁY I AKSJOMATY

Wielkie znaczenie ma możliwość narzucania pewnych własności, aksjomatów lub protokołów zachowania się obiektów. Jako przykład chcielibyśmy przytoczyć procedury wejścia (ang. entry) w monitorach. Monitor jest strukturą danych przeznaczoną do wspólnego użytku wielu procesów. Procesy te mogą korzystać z zasobów monitora tylko kolejno, nigdy współbieżnie, gdyż grozi to naruszeniem struktury danych zapisanej w monitorze. Chcemy zapisać ogólną koncepcję monitora i procedur wejścia. Użytkownicy będą te koncepcje rozwijać i dopasowywać do swoich potrzeb. Zasada wzajemnego wykluczania procedur wejścia powinna zostać narzucona w każdym przypadku. Jak to osiągnąć?

Całkiem inne żądanie może się pojawić gdy zażądamy by wszystkie obiekty pewnego systemu spełniały specyficzne aksjomaty podczas swojego istnienia, np. by były "znormalizowane". Własność taka może być naruszona przez nieuważnego programistę, użytkownika. Na przykład, jak upewnić się, że podczas pracy z listami żadna z nich nie zostanie przekształcona w pierścień? Jak zapewnić spełnianie aksjomatów bazy danych? Jak zmusić obiekty reprezentujące graczy w grze by nie naruszały jej reguł?

We wszystkich tych przypadkach chcemy posiadać predefiniowane ramy zachowań, które mogą być rozwijane zgodnie z potrzebami ale bez naruszania pewnych, określonych niezmienników, aksjomatów.

2.5 SYTUACJE WYJĄTKOWE I ICH SYGNALIZACJA

Byłoby nierealnym założenie, że operacje w systemach realizowanych przez nas są zawsze określone, całkowite. W rzeczywistości mamy do czynienia z działaniami częściowymi. Jest to jedna z sytuacji "wyjątkowych". Wydaje się naturalnym i pożądanym posiadanie: 1° możliwości zasygnalizowania, że sytuacja taka zaistniała, 2° możliwości obsłużenia takiego sygnału.

Zdarza się często, że elementy pewnego systemu komunikują się ze sobą przez wysyłanie i przyjmowanie sygnałów. Sygnał przybywający może przerwać normalny tok obliczeń. Sygnały mogą być binarne - po prostu obecność lub nieobecność sygnału. Mogą być też sygnały przenoszące wiadomości o skomplikowanej strukturze. Warto też wspomnieć sytuacje kiedy sygnał jest generowany automatycznie przez firmware. W niektórych przypadkach sygnał np. nadmiaru, dzielenia przez zero itd. jest wysyłany przez hardware, w innych przez software np. index, za_wielka_tablica, referencja do obiektu pustego.

Chcemy mieć możliwość reagowania na sytuacje wyjątkowe. Oprócz tego potrzebne są nam narzędzia formalne do opisu i analizy systemów w których takie sytuacje "wyjątkowe" są "normalne", przewidziane.

2.6 WSPOLNE WZORCE

Jakże często stwierdzamy, że dwa nasze algorytmy mają wspólne części lub, że dzielą się wspólną strukturą nawet gdy są to algorytmy o całkowicie odmiennej naturze przetwarzanych przez nie obiektów. Jakże często chcielibyśmy "wyciągnąć przed nawias" część wspólną algorytmów. Przykładem takiej sytuacji są drzewa binarnych poszukiwań gdzie trzy operacje: INSERT, DELETE, MEMBER zawierają ten sam wspólny algorytm SEARCH poszukiwania elementu w drzewie. Jednym ze sposobów realizacji naszej uwagi jest wprowadzenie osobnej procedury SEARCH i wywoływanie jej w trzech wymienionych działaniach. Jeżeli postąpimy w ten sposób, to każda operacja na strukturze drzewa binarnego będzie obciążona kosztem wywołania dwu procedur (zamierzonej procedury np. INSERT i pomocniczej procedury SEARCH). Czy możliwe jest obniżenie tego kosztu przy jednoczesnym uniknięciu powtarzania kodu dla SEARCH?

Z drugiej strony stwierdzamy występowanie takich przypadków w których ta sama struktura algorytmu może prowadzić do całkowicie różnych wyników w zależności od interpretacji. Rozpatrzmy następujący przykład: zakładamy, że dana jest pewna struktura dla operacji Union i Find. Następujący krótki algorytm

```
WHILE NOT END
DO
   P := NEXT;
   X := FIND(P.I);
   Y := FIND(P.J);
   IF X<>Y
   THEN
      X := UNION(X,Y);
      CALL PRINTING
   FI
OD
```

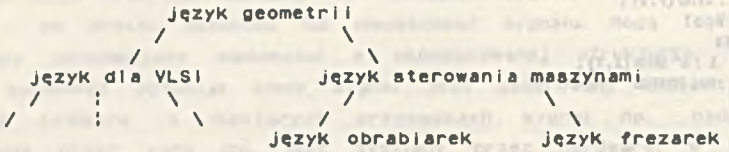
może albo obliczać najmniejszą relację równoważności generowaną przez daną listę par elementów, albo wyznaczać drzewo rozpinające dla danego grafu. Znaczenie algorytmu zależy od interpretacji działań: END, NEXT, PRINTING. Jakie narzędzia programistyczne powinny być dostępne byśmy mogli spożytkować tego rodzaju obserwacje?

2.7 JĘZYKI PROBLEMOWO ZORIENTOWANE

Obserwujemy stale rosnącą liczbę języków programowania. Przekracza ona 3000 i dalej rośnie. Można też zauważyć, że większość tych języków ma podobną strukturę operacji na algorytmach i strukturach danych. Różnice mogą zachodzić w zestawie operacji pierwotnych, ale reszta jest mniej lub więcej podobna. W każdym przypadku powstaje nowy kompilator, a to zwiększa koszty i opóźnia rozpowszechnianie nowego języka. Czy taka sytuacja jest normalna? Wierzymy, że znaleźć można takie rozwiązanie, które umożliwi tworzenie nowych języków problemowo zorientowanych na życzenie ich potencjalnych użytkowników, które nie pociąga za sobą konieczności budowania nowych kompilatorów. Jeden kompilator powinien wystarczyć. Rozwiązanie oparte jest o zasadę faktoryzacji. Co więcej przewidujemy możliwość rozszerzania języków problemowo-zorientowanych w różnych kierunkach i korzystania z modułów wcześniej zrealizowanych.

Rozpatrzmy następujący przypadek: mamy stworzyć język dla projektowania i wykonywania części mechanicznych wytwarzanych przez obrabiarki. Jest całkiem naturalne by rozpocząć od języka dla geometrii, jest zresztą pewna szansa, że język taki już wcześniej został zrealizowany. Później gdy ktoś dojdzie do wniosku, że system powinien zawierać także inne typy operacji , może on dodać nowe moduły. Języki problemowo-zorientowane mogą być

realizowane jako moduły implementujące struktury danych. Moduły te tworzą hierarchie np.



Przegląd ten nie wyczerpał wszystkich sytuacji jakie znane są w inżynierii oprogramowania i dla których obecne oprogramowanie: Pascal, Ada nie dostarcza zadowalających narzędzi.

3. Informacja o Loglanie

W tym rozdziale przedstawimy pokrótce język Loglan'82, jego cechy specyficzne i możliwości zastosowania Loglanu w sytuacjach wyliczonych powyżej. Dla krótkiego porównania Loglanu z Pascalem i Adą powinniśmy wymienić cztery rodzaje modułów jakie mogą wystąpić w programie loglanowskim: CLASS, COROUTINE, PROCESS i MANAGER. Najbardziej interesującą cechą Loglanu jest operacja składania modułów, znana także jako prefiksowanie lub dziedziczenie (ang. inheritance). Rozwiniemy tę kwestię za moment. Jak można się było zorientować z nazw przytoczonych powyżej, Loglan umożliwia obliczenia równoległe i prawie równoległe, obsługę sygnałów i sytuacji wyjątkowych w modułach (MANAGERS) obsługi sytuacji wyjątkowych. Język dopuszcza całkowicie dynamiczne tablice, ma niewiele ograniczeń na przekazywanie parametrów (np. można przekazać nazwę modułu jako parametr).

Język może być określony jako język obiektowo zorientowany. Rzeczywiście moduły klas, współprogramów(coroutin), procesów służą jako wzorce do konstrukcji obiektów zgodnie z opisami zawartymi w tych modułach(deklaracjach). Z drugiej strony język zachowuje wszystkie pożyteczne i sprawdzone mechanizmy tradycyjnego pro-

gramowania imperatywnego. Jesteśmy w posiadaniu argumentów wskazujących na to, że i inne style programowania np. programowanie w logice mogą być realizowane w Loglanie także.
Obiekty mają strukturę podobną do struktury rekordów aktywacji procedur. Różnią się jednak przewidzianym dla nich scenariuszem. Przypuśćmy, że dana jest deklaracja klasy:

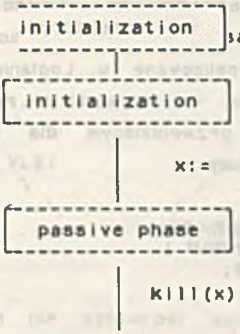
```
UNIT KLASA: CLASS (FORMAL_PARAMETERS);
    [LOCAL DECLARATIONS OF VARIABLES, CONSTANTS, PRO-
     CEDURES, FUNCTIONS, CLASSES AND OTHER MODULES ]
     UNIT PROC: PROCEDURE (FORMAL_PARAMS);
                  ...
     END PROC;
     UNIT FUNCT: FUNCTION (FORMAL_PARAMS);
                  ...
     END FUNCT;
BEGIN
          [ INSTRUCTIONS ]
END KLASA;
```

Wtedy wewnątrz zasięgu działania tej deklaracji można napisać a potem obliczyć wartość wyrażenia

NEW KLASA (PARAMETRY AKTUALNE)

co spowoduje stworzenie obiektu typu KLASA. Wewnętrzna struktura takiego obiektu odpowiada strukturze modułu. Tak więc każdy obiekt typu KLASA ma część przedstawiającą jego pamięć, część przedstawiającą jego stan sterowania i część (niedostępną dla użytkownika) przeznaczoną dla admistracji obiektami. Pierwsza część zawiera dane lokalne zgodne z lokalnymi deklaracjami modułu KLASA. Stan sterowania obiektu może być przedstawiany jako lista istrukcji obiektu które pozostają do wykonania. Początkowo lista ta jest nieco zmodyfikowaną listą instrukcji zawartych w module KLASA. Modyfikacje te są zgodne z znanymi regułami bloku i procedury (copy rule).

Poniżej przedstawiamy ogólny scenariusz dla obiektów klas.



Faza inicjalizacji jest podobna do wykonywania procedury i polega na wykonywaniu instrukcji przypisanych obiektowi przez moduł, według jakiego nasz obiekt został utworzony. Faza ta kończy się gdy wykonane zostaną wszystkie instrukcje obiektu lub równoważnie, gdy sterowanie osiągnie koniec treści obiektu (end KLASA). Osiągnięcie tego momentu oznacza, że obiekt został utworzony i wyposażony w swoją własną pamięć zainicjowaną zgodnie z programem zawartym w deklaracji modułu klasy. W fazie pasywnej obiekt zachowuje się mniej więcej tak jak rekord w Pascalu.

Zwracamy uwagę na istotną różnicę: obiekt może być wezwany do wykonania w swojej fazie pasywnej operacji lokalnych, zadeklarowanych w module klasy (por. funkcje ADD i MUL zadeklarowane w klasie COMP). Tak więc podczas fazy pasywnej można stosując mechanizm odległego dostępu nie tylko odczytać bądź zapisać wartość pewnego atrybutu rekordu, lecz także można wykonać instrukcję procedury

```
CALL    X.PROC(ACT_PAR)
```

lub obliczyć wartość wyrażenia

```
X.FUNCT(ACT_PAR)
```

Logian zezwala na tworzenie obiektów innych jeszcze modułów: współprogramów, procesów, modułów obsługi sygnałów. Scenariusze dla takich obiektów są bogatsze, obiekt może przejść przez więcej stanów w czasie swego istnienia.

Prefiksowanie jest dwuargumentową operacją na modułach programu. Prefiks powinien być klasą (współprogramem lub procesem).

Moduł prefiksowany może być dowolnego rodzaju: procedurą, klasą, współprogramem, funkcją, procesem lub blokiem. Wynik prefiksowania jest, w przybliżeniu, modułem otrzymanym przez złożenie (konkatenację) listy deklaracji obu modułów i przez złożenie list instrukcji zawartych w obu modułach. Za chwilę podamy dalsze szczegóły. Zwracamy uwagę na pewną trudność w przyswojeniu sobie koncepcji prefiksowania jaka wynika z tego, że rezultat prefiksowania nie pojawia się explicite w programie. W pewnym sensie operujemy w wolnej algebrze modułów z operacją prefiksowania tzn. moduł

<NAZWA> : <IDENTYFIKATOR PREFIKSU> <NAZWA PREFIKSOWANY>

reprezentuje wynik prefiksowania.

Wyobraźmy sobie, że dane są dwa moduły - klasy

```
UNIT C: CLASS                UNIT D: C CLASS
  <DECLARATIONS C>                    <DECLARATIONS D>
BEGIN                        BEGIN
    < INSTRUCTIONS C>                   < INSTRUCTIONS D >
END C                                   END D
```

Zgodnie z regułą prefiksowania-konkatenacji klasa D przedstawia sobą następujący moduł podany poniżej (uwaga nie oznacza to, że podajemy tu sposób implementacji prefiksowania):

```
UNIT D: CLASS
    <DECLARATIONS C>
    <DECLARATIONS D>
BEGIN
    < INSTRUCTIONS C >
    < INSTRUCTIONS D >
END D
```

Czytelnik nie powinien dać się zwieść pierwszemu wrażeniu: reguła konkatenacji jest wyjaśniana w terminach transformacji tekstowych, niemniej jednak realizacja nie powinna polegać na składaniu tekstów modułów. Byłoby to wielce nieefektywne. Przypomnijmy analogiczną sytuację z regułą kopiowania, wywoływania procedur i sposobem implementacji procedur.

Prefiksowanie jest formą konstrukcji programów, która ma nieoczekiwanie wiele zastosowań. Nie możemy dziś stwierdzić, że znamy wszystkie sposoby jej wykorzystania.

4. Algorytmiczne teorie struktur danych

Rozpatrzmy następujący przykład. Przypuśćmy, że do firmy software'owej zgłosił się dyrektor banku ze zleceniem dokonania symulacji pracy oddziału banku jaki ma powstać w nowym osiedlu X. Idzie mu o to by w drodze eksperymentu symulacyjnego ustalić liczby okienek i personelu tak by nie tworzyły się kolejki klientów i by, z drugiej strony, nie zatrudnić zbyt wielkiego personelu.

W tym przypadku całkiem naturalny jest pomysł by stworzyć dwa moduły

```
BANK
definiuje
pojęcia i operacje
typ KlientBanku
    Urzędnik
    Okienko
proc Przychodzi
     Wpłaca
     ...
```

```
SYMULACJA BANKU
używa
pojęć i operacji
var F,G: Okienko,
   A,B:KlientBanku
   C,D,E: Urzędnik
    ...
call Przychodzi
call Wpłaca
```

Dalsza analiza prowadzi do wniosku, że BANK jest szczególnym przypadkiem ogólniejszej struktury BIURO. W strukturze BIURO występują pojęcia klient i obsługa, pojęcia klient banku, urzędnik, okienko stanowią rozwinięcia pojęć bardziej ogólnych. Podobnie ma się rzecz z operacjami: operacje klient przychodzi, czeka, wychodzi są wspólne wielu różnym biurom. Dla kierownika firmy software'owej jest więc całkiem naturalne by wyodrębnić i osobno zaimplementować moduł BIURO.

```
BIURO
definiuje
pojęcia i operacje
typ Klient
    Obsługa
proc Wchodzi
     Wychodzi
```

Może on być wykorzystany do innych prac np. do symulacji pracy firmy ubezpieczeniowej itp. Rozumowanie to można powtórzyć

kilkakrotnie i wyodrębnić po kolei: moduły SYMULACJA, KOLEJKA PRIORYTETOWA, KOLEJKA. W strukturze SYMULACJA mamy do czynienia z pojęciami: proces symulowany, zdarzenie (na zdarzenie składają się nazwa procesu i chwila zdarzenia), oś czasu na której umieszczone są zdarzenia zaplanowane do symulacji w przyszłości. Operacje struktury SYMULACJA to: zaplanuj, wykonaj, wznów, skasuj, bieżąca chwila, bieżący proces itp.

```
SYMULACJA
definiuje pojecia
typ Proces_Symulowany
    OCzasu
    Zdarzenie
i operacje
proc Zaplanuj
     Wstrzymaj
     Usun
     Bieżący_Proces
     Bieżący_Czas
```

Zdarzenia są elementami osi czasu, którą możemy pojmować jako kolejkę priorytetową(znaczenie tego terminu wyjaśnimy poniżej, por. też AHU[ ]).

```
KOLEJKA_PRIORYTETOWA
definiuje pojecia
typ Zbiór
    Element
i operacje
proc Wstaw (Insert)
     Usuń (Delete)
     Należy? (Member)
     Jakis_Element(amb)
     Min (minimalny)
     ≤ (porządek)
```

Procesy z kolei mogą być elementami kolejek (zwyczajnych).

```
KOLEJKA
definiuje pojecia
typ Kolejka
    Element_Kolejki
i operacje
proc Wstaw
     Pierwszy
     Usun
     Pusta_Kolejka?
     równość
```

Tak więc doszliśmy do wniosku, że dla rozwiązania zadania postawionego przez dyrektora banku należy zrealizować pięć modułów (rys.5.1). W dalszym ciągu rozdziału będziemy powracać do tego przykładu.

5. Kolejki

W tym paragrafie omówimy strukturę kolejek. Czym jest kolejka wie każdy. Zastanówmy się jednak w jaki sposób opisać pojęcie kolejki tak, by opis ten był wystarczająco precyzyjny dla poprawnej implementacji struktury kolejki i dla prawidłowego korzystania z niej.

Przez kolejkę chcemy rozumieć obiekt jaki powstaje w wyniku operacji typowych dla kolejek i który sam jest argumentem dla innych operacji na kolejkach. Należy więc opisać strukturę w której znajdują się obiekty: kolejki i ich elementy, a na tych obiektach dokonywane są operacje: *put* - wstaw element do kolejki, wynik tej operacji jest nową kolejką, *out* - usun z kolejki jej pierwszy element, wynik i argument są kolejkami, *first* - podaj pierwszy element z kolejki, argument jest kolejką, wynik jest elementem, *empty* - sprawdź czy kolejka jest pusta, wynik jest wartością logiczną, porównaj dwa elementy - wynik jest wartością logiczną.

Czy taki opis jest wystarczający? Łatwo zauważyć, że nie. Rozważmy bowiem jakąkolwiek strukturę danych w której występują: jedna operacja dwuargumentowa i dwie operacje jednoargumentowe, można wtedy uważać, że spełnione są wymagania opisane przez nas powyżej. A więc należy dodać dodatkowe szczegóły do opisu kolejek. Jakie własnosci mają mieć operacje na kolejkach?

aq1) (empty(q) => [q=$_Q$out(put(e,q))]]

aq2) [¬empty(q) => [put(e,out(q) =$_Q$ out(put(e,q))]]

aq3) (empty(q) => [e =$_E$ first(put(e,q))]]

aq4) [¬empty(q) => [first(put(e,q)) =$_E$ first(q)]]

aq5) ¬empty(put(e,q))

Czy możemy twierdzić, że każdy system składający się z dwu zbiorów(elementów i kolejek), w którym określone są dwa działania jednoargumentowe i jedno działanie dwuargumentowe i który spełnia wyliczone powyżej aksjomaty jest kolejką? Nie możemy uważać tych aksjomatów za definicję pojęcia kolejki ponieważ istnieją systemy o których nie chcemy mówić, że są kolejkami chociaż spełniają one aksjomaty. Rozpatrzmy dla przykładu system ciągów nieskończonych i skończonych. Nasuwa się naturalna interpretacja działań przy której trzeba określić tylko efekt operacji *put* wstaw element e do nieskończonej kolejki q. Określamy go jako równy kolejce q. Łatwo sprawdzić, że wszystkie aksjomaty są prawdziwe przy takiej interpretacji, a jednak tej struktury nie chcemy uważać za strukturę kolejek, ponieważ dopuszcza ona nieskończone ciągi. Wyjściem z tej sytuacji jest przyjęcie dodatkowego założenia: wszystkie kolejki są skończone. Własności tej nie da się zapisać w języku pierwszego rzędu. Ale można ją wyrazić jako własność algorytmiczną w następujący sposób:

aq6) while ¬empty(q) do q:=out(q) od true

Wspólnie z pozostałymi aksjomatami powyższa formuła stwierdza, o ile jest prawdziwa w danej strukturze, że dla każdej kolejki q można w skończonym czasie usunąć wszystkie jej elementy. Jak zauważymy za chwilę ta własność ma tę dodatkową zaletę, że można jej użyć podczas analizy programów, np. dla udowodnienia poprawności pewnego programu.

Przy okazji wprowadzenia formuły algorytmicznej jako aksjomatu zauważmy, że równość kolejek nie musi być przyjmowana jako pojęcie pierwotne. Można ją zdefiniować algorytmicznie /p

```
aq7)
q =Q q' ≡ begin q1 := q; q2 := q'; bool := true;
            while ¬empty(q1) ∧ ¬empty(q2) ∧ bool
            do
              if  first(q1)  ≠  first(q2)
              then
                 bool  :=  false
              fi;
              q1  :=  out(q1);
              q2  :=  out(q2)
            od
          end (bool ∧ empty(q1) ∧ empty(q2)) /P
```

Uwaga ta ma większy sens niż to się może zdawać na pierwszy rzut oka. Znane są systemy w których operacje są efektywne, a pomimo to

porównywanie elementów nie jest operacją efektywną(por. system rekurencyjnych liczb rzeczywistych Mazura). Razem, zbiór aksjomatów algorytmicznej teorii kolejek zawiera 7 aksjomatów. Wykażemy, że jest on przydatny w analizie algorytmów, że jest zupełną specyfikacją pojęcia kolejka. Pokażemy też w jaki sposób można użyć tych aksjomatów w procesie weryfikacji poprawności danej implementacji struktury kolejek.

TWIERDZENIE 5.1
Każdy model algorytmicznej teorii kolejek ATQ jest izomorficzny ze strukturą skończonych ciągów elementów ze zbioru E elementów danego modelu wraz z naturalnymi operacjami na ciągach:

put(e,s) dołącz element e do ciągu s na jego końcu,
first(s) pierwszy element ciągu s, o ile niepusty
out(s) ciąg otrzymany z s przez odrzucenie pierwszego elementu, o ile niepusty
empty(s) ≡ ciąg s jest pusty

□

TWIERDZENIE 5.2
Jeżeli system M jest modelem teorii ATQ, to system ilorazowy $M/(=_E,=_Q)$ jest modelem teorii ATQ właściwym dla identyczności.
Dowód. W systemie $M/(=_E,=_Q)$ elementami są klasy abstrakcji relacji równoważności $=_E$ i $=_Q$. Ponieważ relacje te mają następujące własności wymienione w lemacie to jak widać system ilorazowy też będzie modelem dla aksjomatów aq1 - aq7.

□

LEMAT 5.3
Dla dowolnych $q,q',q''\in Q$ i dla dowolnych $e,e'\in E$

(a) $q =_Q q$
(b) $q =_Q q' \Rightarrow q' =_Q q$
(c) $q =_Q q' \wedge q' =_Q q'' \Rightarrow q' =_Q q''$
(d) $e =_E e' \wedge q =_Q q' \equiv put(e,q) =_Q put(e',q')$
(e) $q =_Q q' \wedge \neg empty(q) \Rightarrow [(out(q) =_Q out(q') \wedge first(q)=_E first(q'))]$
(f) $q =_Q q' \Rightarrow empty(q) \equiv empty(q')$
(g) $empty(q) \wedge empty(q') \Rightarrow q =_Q q'$

Dowód twierdzenia o reprezentacji. Każdemu elemntowi $s \in S$ przyporządkowujemy skończony ciąg seq(s) elementów zbioru E. Odwzorowanie seq jest określone przy pomocy posiłkowych odwzorowań

dla każdej liczby naturalnej *i* określamy odwzorowanie częściowe

i-ty od góry: S -> E

i-ty od góry(s) = if ¬empty(s) then s := out(s) fi^i first(s)

oprócz tych odwzorowań rozpatrywać będziemy jeszcze funkcję

card: S -> N taką, że card(s)=najmniejsza liczba naturalna *i* taka, że empty(out^i(s))

Niech FSeq(E) oznacza rodzinę wszystkich ciągów skończonych o elementach ze zbioru E. Z każdym elementem $q \in Q$ stowarzyszymy skończony ciąg seq(s)

seq: S -> FSeq(E)

Dla elementu s_0 spełniającego warunek empty(s), zauważmy, że jest jeden tylko taki element, wartością $seq(s_0)$ jest ciąg pusty Λ. Dla pozostałych elementów $s \in S$ kładziemy

seq(s) = $[e_0, e_1, \ldots, e_{n-1}]$ gdzie n=card(s) i dla $0 \le i < n$ e_i=i-ty od góry(s).

Łatwo zauważyć, że dla każdego skończonego ciągu $e_1, \ldots, e_n$ zachodzi następująca równość

$seq(put(e_n, put(e_{n-1}, \ldots, put(e_1, s_0) \ldots))) = [e_1, \ldots, e_{n-1}, e_n]$

a więc odwzorowanie seq jest odwzorowaniem na zbiór FSeq(E).

Niech $q \neq q'$, a więc zachodzi także $q \neq_Q q'$. Z aksjomatu aq7 widzimy, że albo po wykonaniu programu jaki w tym aksjomacie występuje zachodzi (bool $\wedge$ ¬(empty(s)$\wedge$empty(s'))) i wtedy card(s) $\neq$ card(s') albo istnieje liczba naturalna *i* taka, że

i-ty od góry(s0 $\neq$ i-ty od góry(s') o ile po wykonaniu programu spełniona jest formuła ¬bool. W obu przypadkach seq(s) $\neq$ seq(s'), a więc odwzorowanie seq jest wzajemnie jednoznaczne. Nietrudno sprawdzić, że

seq(out(s)) = out(seq(s))

seq(put(e, s)) = put(e, seq(s))

first(s) = first(seq(s)).

Co kończy dowód twierdzenia o reprezentacji.

Twierdzenie to przekonuje nas, że udało się nam podać specyfikację struktury kolejek wolną od sprzeczności i zarazem w pełni opisującą rodzinę możliwych implementacji tego pojęcia.

6 Kolejki priorytetowe

Kolejka priorytetowa jest strukturą danych, której uniwersum składa się z dwu rozłącznych podzbiorów

E i S

zwanych E - typem elementów i S - typem zbiorów. Struktura ta ma następujące operacje:

insert: E X S --> S

delete: E X S --> S

min: S --> E

member: E X S --> B_0

empty: S --> B_0

$\leq$: E X E --> B_0

które spełniają następujące aksjomaty:

PQ1) zbiór E jest liniowo uporządkowany przez relację $\leq$ oznacza to, że prawdziwe są następujące własności

$(\forall x \in E)\ x \leq x$

$(\forall x,y \in E)$ jeśli $x \leq y$ i $y \leq x$ to $x = y$

$(\forall x,y,z \in E)$ jeśli $x \leq y$ i $y \leq z$ to $x \leq z$

$(\forall x,y \in E)\ [x \leq y$ lub $y \leq x]$

PQ2) $(\forall s \in S)$ **begin** $\neg empty(s)$ **do** $s := delete(min(s),s)$ **od true**

PQ3) $(\neg empty(s) \Rightarrow ((\forall e)\ member(e,s) \Rightarrow min(s) \leq e))$

PQ4) $member(e,insert(e,s))$

PQ5) $(e \neq e' \Rightarrow (member(e',s) \equiv member(e',insert(e,s))))$

PQ6) $\neg member(e,delete(e,s))$

PQ7) $(e \neq e' \Rightarrow (member(e',s) \equiv member(e',delete(e,s))))$

PQ8) $member(e,s) \equiv$ **begin** $s1 := s$; $bool := false$;
while $\neg empty(s) \wedge \neg bool$ **do**
$e1 := min(s1)$;
$bool := (e1 = e)$;
$s1 := delete(e1,s1)$
od
end $bool$

Ponadto przyjmujemy zwykłe aksjomaty identyczności.

Wykażemy, że powyższe aksjomaty stanowią definicję klasy kolejek priorytetowych ponieważ zachodzi następujące twierdzenie o reprezentacji:

TWIERDZENIE 6.1.

Każdy model aksjomatów pq1 - pq8 jest izomorficzny z modelem w którym zbiór S = Fin(E), jest rodziną wszystkich skończonych podzbiorów zbioru E.

Poniżej wprowadzamy definicję równości zbiorów. Wykażemy, że spełnione są zwykłe własności relacji identyczności (antysymetria, zwrotność i przechodniość).

DEFINICJA 6.1

Dla dowolnych s,s' ∈ S

```
eq(s,s') ≡ begin
             s1 := s; s2 := s';
             boo := true;
             while boo ∨ ¬empty(s1) ∨ ¬empty(s2)
             do
               e1 := min(s1);
               boo := boo ∨ member(e1,s2);
               if boo
               then
                 s1 := delete(e1,s1);
                 s2 := delete(e1,s2);
               fi
             od
           end (boo ∨ empty(s1) ∨ empty(s2))
```

LEMAT 6.2

Niech K oznacza program występujący w powyższej definicji. Program K nie zapętla się w żadnym modelu teorii ATPQ.

LEMAT 6.3

W każdym modelu teorii ATPQ zachodzi równoważność:

eq(s,s') ≡ [(∀e)(member(e,s) ≡ member(e,s'))]

LEMAT 6.4

Następujące formuły są prawdziwe w każdym modelu teorii ATPQ

(a) (∀s) eq(s,s)

(b) (∀s,s') [eq(s,s') => eq(s',s)]

(c) (∀s,s',s") [eq(s,s') ∧ eq(s',s") =>eq(s,s")]

Dowód (a) wynika natychmiast po podstawieniu s zamiast s' w definicji 6.1. Podobnie przebiega dowód własności (b). W przypadku (c) dowód jest nieco bardziej skomplikowany.

LEMAT 6.5

Dla dowolnych $e,e' \in E$ i dla dowolnych $s,s' \in S$ zachodzą w każdym modelu teorii ATPQ następujące własności:

$((e=e' \wedge eq(s,s')) \Rightarrow eq(insert(e,s),insert(e',s')))$

$(eq(s,s') \Rightarrow empty(s) \equiv empty(s'))$

$(e=e' \wedge eq(s,s') \Rightarrow eq(delete(e,s),delete(e',s')))$

$eq(insert(e,delete(e,s)),s).$

$empty(s) \Rightarrow min(insert(e,s)) =_E e$

$\neg\ empty(insert(e,s))$

$eq(delete(e,insert(e,s)),s)$

$(\neg\ eq(s,s') \wedge \neg\ member(e,s) \wedge \neg\ member(e,s') \Rightarrow \neg\ eq(insert(e,s),insert(e,s')))$

$(member(e,s) \Rightarrow eq(s,insert(e,s))).$

TWIERDZENIE 6.6

Każdy model M algorytmicznej teorii kolejek priorytetowych ATPQ jest izomorficzny z modelem standardowym tzn. z strukturą

$$< E \cup Fin(E),\ f_1,f_2,f_3,r_1,r_2,=,\in >$$

gdzie Fin(E) jest rodziną wszystkich skończonych podzbiorów zbioru E,

$f_1(e,s) = s \cup \{e\}$, $r_1(e,s) \equiv (e \in s)$

$f_2(e,s) = s - \{e\}$, $r_2(s) \equiv (s = \emptyset)$

$f_3(s)$ = najmniejszy element zbioru s □

Dowód.

Zacznijmy od uwagi, że każdy model M teorii ATPQ może być traktowany jako model właściwy dla pojęcia identyczności. Istotnie, Dla danego modelu M możemy rozważyć system ilorazowy w którym elementami są klasy równoważności elementów ze zbioru E i elementów ze zbioru S.

Każdemu elementowi $s \in S$ przyporządkowujemy zbiór h(s)

$h(s) = \{e \in E:\ member(e,s)\}$

Zbiór h(s) jest skończony na mocy aksjomatu PQ2.

Odwzorowanie

$h: S \to Fin(E)$

jest odwzorowaniem na, ponieważ dla dowolnie danego zbioru $\{e_1, \ldots, e_n\}$ możemy rozważyć następujący term

```
begin while ¬empty(s) do s := delete(min(s), s) od;
      s := insert (e1,s);
      ...
      s := insert(en,s)
end s
```

Odwzorowanie h jest odwzorowaniem wzajemnie jednoznacznym. Wynika to z lematu 6.3. Przypuśćmy że s ≠ s' tzn. ¬eq(s,s'), wtedy zachodzi (∃e) mb(e,s) ∧ ¬mb(e,s') lub symetrycznie (∃e) ¬mb(e,s) ∧ mb(e,s').

Łatwo sprawdzić, że

h(insert(e,s)) = h(s) ∨ {e} na mocy aksjomatów PQ4,5

h(delete(e,s)) = h(s) - {e} na mocy aksjomatów PQ6,7

member(e,s) ≡ e ∈ h(s) z definicji funkcji h

empty(s) ≡ h(s) = ∅ z definicji funkcji h

min(h(s)) = min(s) na mocy aksjomatu PQ3

Co kończy dowód twierdzenia o reprezentacji.

Wynika stąd uwaga odnosząca się do możliwych implementacji kolejek priorytetowych: jeżeli jakaś implementacja jest poprawna w tym sensie, że spełnia wszystkie aksjomaty teorii ATPQ to niezależnie od szczegółów wewnętrznych tej implementacji jej istota jest taka sama jak każdej innej poprawnej implementacji. W szczególności, jest ona izomorficzna z modelem standardowym, teorio-mnogościowym. Stwarza to możliwość wyboru postępowania. Dla sprawdzenia czy jakaś własność zachodzi w klasie wszystkich modeli teorii ATPQ możemy: 1) podjąć próbęx wyprowadzenia tej własności z aksjomatów, lub 2) sprawdzić, czy własność ta zachodzi w klasie modeli standardowych. Wybierzemy to co nam będzie wydawało się wygodniejsze i nie poniesiemy uszczerbku na ogólności ani poprawności rozważań.

7. Symulacja

W tym rozdziale przedstawimy zestaw aksjomatów opisujących strukturę danych niezbędną dla realizacji planowania symulacji procesów przez współprogramy. O własnościach współprogramów wspomnieliśmy krótko w rozdziale 3. Przypomnijmy, że współprogramy

są obiektami klas, które wyposażono dodatkowo w dwa działania attach(X) - operację powodującą przekazanie sterowania do obiektu coroutiny X z uprzednim zapamiętaniem gdzie obiekt Y wykonujący tę instrukcję ma wznowić swoje działanie jeżeli inny obiekt-współprogram wznowi go bądź przez instrukcję attach(Y) bądźz przez instrukcję detach. Ta druga instrukcja oddaje sterowanie temu współprogramowi który spowodował obudzenie aktualnie wykonywanego współprogramu. W żadnym momencie nie są wykonywane dwa współprogramy.

Przez system zarządzania procesami symulowanymi rozumiemy następujący system algebraiczny:

<SPuSQSuEVNuT; idle?, terminated, current, time, schedule, run, hold, passivate>

gdzie SP oznacza zbiór procesów symulowanych, SQS jest oznaczeniem "osi czasu", EVN oznacza zbiór zdarzeń, T oznacza czas (tu będzie to zbiór liczb rzeczywistych

System zawiera następujące działania i predykaty

current : SQS ->SP idle? : SP -> B_0

time : SQS -> T terminated? : SP -> B_0

schedule : SP$\times$T -> SQS run : SP- SQS

hold : T -> SQS passivate : SP -> SP

które spełniają aksjomaty wyliczone poniżej

s1) SQS jest kolejką priorytetwą

< SQSuEVN; ins, del, min, mb, ∈, = >

s2) EVN = SP $\times$ T

s3) time(sqs) = min(sqs).T current(sqs) = min(sqs).SP

s4) (∀ sqs) ¬idle(p) ≡ (∃t∈T) (mb((p,t),sqs)

s5) mb((p,t),sqs) mb((p,t'),sqs) => t = t'

s6) sqs = o ∧ idle(p) ∧ ¬terminated(p) =>

[sqs.schedule(p,t)]{¬idle(p) ∧ sqs = ins((p,t),o)]

s7) sqs = o ∧ ¬idle(p) ∧ ¬terminated(p) =>

[sqs.schedule(p,t)][¬idle(p) ∧ sqs = ins((p,t),del((p,time),o))]

s8) terminated(p) => [schedule(p,t)] {ERROR}

s9) hold(t) α ≡ [schedule(current,time+t)] α dla każdej formuły α

s10) current = p' ∧ ¬terminated(p) =>

[[run(p)]α ≡ schedule(p,time) α dla każdej formuły α

s11) sqs = o ∧ mb((p,t'),sqs) =>
[run(p)][current=p ∧ sqs = ins((p,time),del((p,t'),o))]

s12) p=current ∧ sqs=o =>
[passivate] [idle(p) ∧ sqs = del((p,time),o)]

s13) sqs = o => [cancel(p)][idle(p) ∧ sqs=del((p,t),o) ∧ terminated(p)]

Aksjomaty te mogą się wydawać skomplikowane, a jednak są znacznie prostsze do zrozumienia niż klasa SIMULATION, którą można obejrzeć lecz trudno zrozumieć jej sens bez tych 13 formuł.

W trakcie wykładu pokażemy program symulujący pracę oddziału banku.

BIBLIOGRAFIA

Mirkowska,G.,Salwicki,A., Algorithmic Logic, PWN Warszawa 1987
Raport języka Loglan, PWN 1983
Salwicki,A., por. Materiały Szkoły PTI w Rydzynie 1984

Jesienna Szkoła PTI
Mrągowo, grudzień 1987

KOMPUTERY W FIZYCE
I
FIZYKA W KOMPUTERACH*) **)

Łukasz A. Turski

Zakład Fizyki Teoretycznej
Polskiej Akademii Nauk
Al. Lotników 32/46
02-668 Warszawa

Prawdę powiedziawszy czuję się szalenie zażenowany występując przed Państwem z wykładami o zastosowaniu komputerów w fizyce i o pewnych problemach fizycznych związanych z przetwarzaniem informacji. Wydaje mi się bowiem, że zainteresowania przeważającej liczby informatyków są bardzo dalekie od tej tematyki. Nie zawsze tak było, zarówno Turing jak i von Neumann byli, o ile nie fizykami, to ludźmi bardzo fizyce bliskimi. Pierwsze komputery zostały zbudowane z myślą o zastosowaniach w fizyce ! Fizycy też, chociażby porzez wynalezienie tranzystora, umożliwili rozwój informatyki; komputery zbudowane z lamp elektronowych i bez obwodów scalonych nigdy by nie trafiły na półki supermarketów. Z biegiem lat zainteresowania informatyków odbiegły nieco od początkowych, mam nadzieję jednak, że nie jest to początek permanentnego rozwodu, a tylko czasowa separacja.

---

*) Opinia jednego fizyka

**) Praca wykonana w ramach programu CPBP 01.03 oraz CPBP 02.02

Fizycy ciągle stanowią dobry rynek dla producentów sprzętu komputerowego i ciągle jeszcze są jednymi z najpoważniejszych konsumentów profesjonalnego sprzętu komputerowego. Ażeby to uzasadnić przytoczę dane charakteryzujące dwa poważne ośrodki obliczeniowe pracujące dla potrzeb fizyki: Los Alamos National Laboratory i Kernforschungsanlage Jülich.

Los Alamos jest laboratorium, zarządzanym dla Departamentu Energii USA przez Uniwersytet Kalifornijski, prowadzącym zarówno badania związane z zastosowaniami wojskowymi jak i cywilnymi. Ośrodek Fizyki Jądrowej w Jülich jest federalnym centrum badawczym RFN, nota bene głównie fizyki niejądrowej, prowadzącym wyłącznie badania naukowe nie związane z wojskiem. Jülich zatrudnia ca 4000 ludzi, a Los Alamos 7000.

W Los Alamos[1] centralny system komputerowy oparty jest na "parku maszynowym" składającym się, w 1984 r., z sześciu komputerów firmy *Cray Research Inc.*: dwu CRAY X-MP i czterech CRAY 1; siedmiu komputerów firmy *Control Data Corporation*: czterech CDC 7600, jednego CDC CYBER 176 i trzech CDC CYBER 825; około siedemdziesięciu komputerów firmy *Digital Equipment Corporation*, w tym czterdzieści DEC VAX-11/780 i 25 PDP 11 obsługujących sieć abonencką wewnątrz Laboratorium. Na terenie Laboratorium działa ca 4000 terminali w tym 1200 TI 700 i TI 800, 1500 DEC VT 100, 500 Tektronix 4000 i 600 IBM PC różnych maści. Dane te ulegają b. szybkim zmianom np. w ostatnim roku zainstalowano w Los Alamos kilka maszyn tzw Clasy VII (CRAY 1 jest klasy VI , numeracja jest wg skali logarytmicznej o podstawie 10) i ,ponoć, maszynę klasy VIII. Dodatkowo należy wymienić jeszcze działającą w Los Alamos eksperymentalną maszynę *Denelcor Experimental Machine* HEP, pierwszą komercjalnie dostępną maszynę wykonującą równoległe przetwarzanie danych i strumieni danych.

Informacje o typach terminali są już dość stare. W zeszłym roku laboratoria narodowe Los Alamos i Livermore, to ostatnie będące o rząd wielkości większym centrum obliczeniowym niż Los Alamos, wymieniały część parku terminali zastępując je stacjami

---

[1] R. D. Ewald *An overview of Computing at Los Alamos.* w: Supercomputers in Theoretical and Experimental Science. J. T. Devreese i P. van Camp ed. Plenum Press ,New York (1985)

roboczymi SUN. Wg danych publikowanych przez Los Alamos ze sprzętu obliczeniowego w Los Alamos korzysta ca 6000 abonentów z czego mniej więcej dwie trzecie stanowią pracownicy Los Alamos, zarówno ci pracujący "na zewnątrz płotu" jak i ci z "wewnątrz"; reszta użytkowników to ludzie z zewnątrz współpracujący z Los Alamos przy realizacji programów badawczych lub kupujący czas dla innych potrzeb (ale tylko związanych z programami badawczymi finansowanymi przez DOE). Typowe stawki opłat to 1 godz CPU CRAYa w dzień - $ 594, w nocy-$297, w czasie weekendu - $148.50.

System w Los Alamos posiada funkcjonalnie wyodrębniony blok zajmuący się drukowaniem i graficznym opracowywaniem wyników PAGES (Print and Graphics Express Station) obsługiwany przez dwa VAX-11/780 zarządzający m.innymi trzema drukarkami laserowymi Xerox 9700 drukującym 10 000 wierszy/min. każda!. PAGES "przerabia" dziennie ca $2x10^9$ bajtów, co w skali miesięcznej daje około 3 milionów stron druku, 6 milionów slajdów i mikrofiszy oraz 250 tyś klatek 16 i 35 mm filmu.

Ośrodek w Jülich jest mniej więcej o połowę mniejszy od Los Alamos. Prowadzi tylko cywilne badania podstawowe i dlatego też jego centrum obliczeniowe jest skromniejsze. Dysponuje dwoma maszynami *Cray Research Inc.*: jednym CRAY X-MP i jednym CRAY 2. Trzema maszynami IBM: jedną IBM 370-168 i dwoma IBM 3030 oraz kilkudziesięciu maszynami DEC, głównie VAX-ami rozrzuconym po rozmaitych Instytutach badawczych wchodzących w skład Ośrodka. Sieć terminali ośrodka składa się z ~ 3000 gniazd głownie terminali Pericom i IBM PC. Wyposażenie Jülich w sprzęt drukujący jest podobne do tegoż w Los Alamos. Funkcje zarządzania całym kompleksem komputerowym pełni ZAM (Zentralinstitut für Angewandte Mathematik) . Za pośrednictwem sieci EARN (European Academic and Research Network) centrum obliczeniowe Jülich dostępne jest użytkownikom z de facto całego świata. EARN umożliwia też pracownikom Jülich korzystanie z znajdujących się w promieniu 80 km od Ośrodka superkomputerów CDC CYBER 205 na Uniwersytecie w Bohum i Fujitsu VP100 na Uniwersytecie w Moguncji.

Cały ten olbrzymi kompleks informatyczny, jakim są poważne ośrodki obliczeniowe laboratoriów, takich jak te, o których mówiłem przed chwilą, wymaga fachowej obsługi na codzień i fachowego kierownictwa zajmującego się utrzymaniem kontaktu z użytkownikami i dbającymi o to, aby cały ten szalenie kosztowny

sprzęt był sensownie wykorzystany do prac badawczych. Użytkownik takiego centrum może w każdej chwili uzyskać wsparcie fachowe w takich sytuacjach, kiedy jego problem badawczy wymaga *optymalnego* wykorzystania sprzętu. W Jülich np. można "wynająć" sobie pracownika z ZAM-u, który staje się członkiem zespołu roboczego wykonującego dane zadanie i po "spełnieniu swej roli" wraca do siebie. Dzięki temu fizycy mogą zajmować się fizyką, a informatycy informatyką, pracując nad jednym i tym samym problemem.

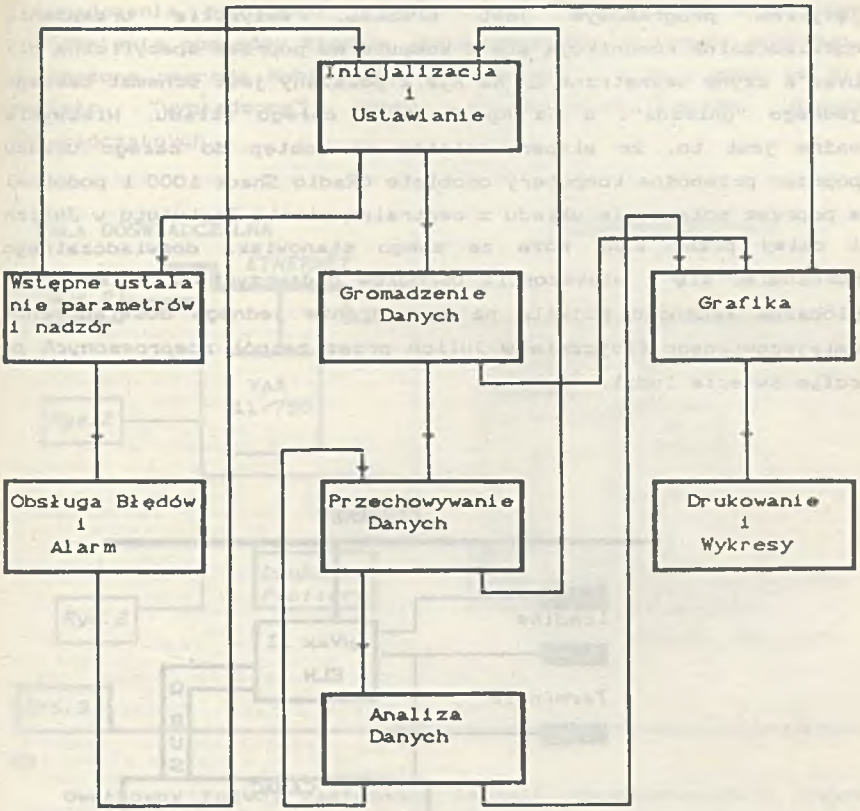
12

Nim przejdę do omówienia takich zastosowań w mojej własnej dyscyplinie, fizyce teoretycznej, chciałbym pokazać Państwu jak wygląda takie *informatyczne wsparcie* w fizyce doświadczalnej. Zrobię to na przykładzie układu zainstalowanego w Jülich i służącego do prowadzenia i opracowania doświadczeń z fizyki ciała stałego, głównie neutronograficznych, wykorzystujących eksperymentalny reaktor DIDO.[2]

Typowe doświadczenie z fizyki ciała stałego składa się z trzech zasadniczych części : (i) przygotowania próbek do badania, (ii) przygotowania odpowiedniego zewnętrznego zaburzenia, któremu poddane są wybrane próbki, (iii) zbadania i analizy zmian wywołanych w próbkach na skutek oddziaływania z zaburzeniem zewnętrznym. We wszystkich z nich elektroniczne przetwarzanie danych odgrywa wielką rolę. *Rys.1* ilustruje typowe funkcje wykonywane przez taki układ informatycznego wsparcia.

---

[2] G. Durcansky *Modern Aspects of Electronics and Data System Engineering in Solid State Research.* w: KFA Jülich IFF-Bulletin 30. (1987)

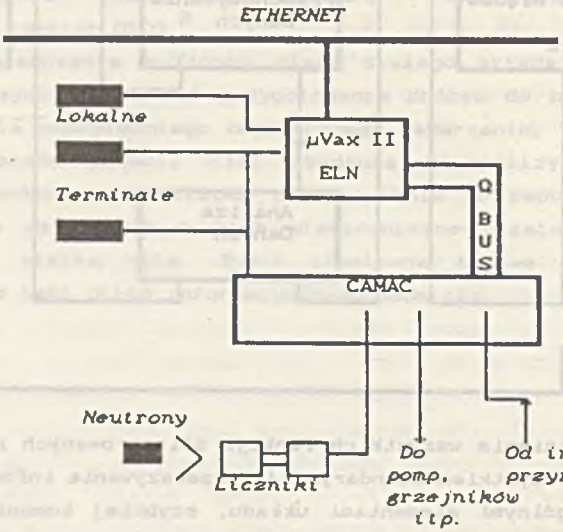


Rys. 1

Spełnienie wszystkich funkcji zilustrowanych na *Rys. 1* wymaga przede wszystkiem standaryzacji przekazywania informacji pomiędzy poszczególnymi elementami układu, szybkiej komunikacji pomiędzy nimi oraz zdolności układu do reakcji w tzw *czasie rzeczywistym*. (To ostatnie, w praktyce oznacza, że system operacyjny powinien być w stanie przerwać wykonywanie jednego zadania i zacząć "reagować" na nowe okoliczności z czasem reakcji rzędu milisekund).

Konkretny układ doświadczalny o którym mówię oparty jest na komputerach MicroVax II i wykorzystuje system operacyjny VAXELN,

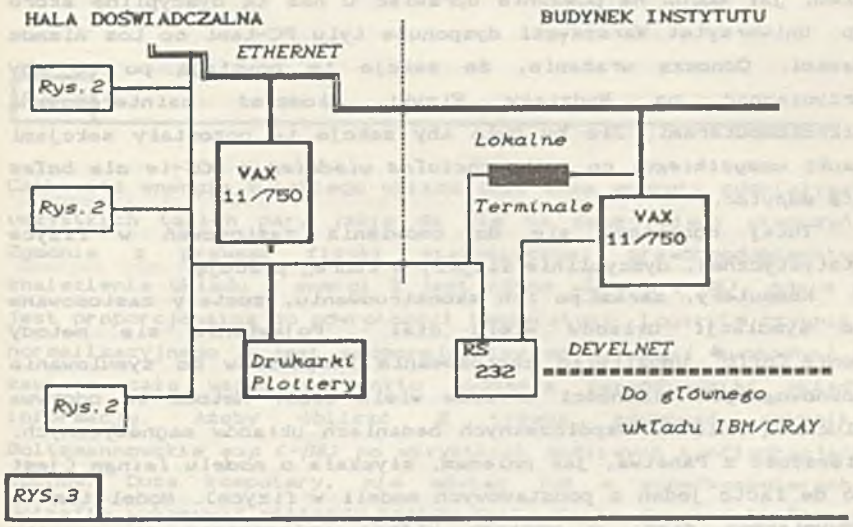
językiem programowym jest EPASCAL. Wszystkie urządzenia doświadczalne komunikują się z komputerem poprzez specyficzną dla μVax'a szynę wewnętrzną Q. Na *Rys.2* pokazany jest schemat takiego jednego "gniazda", a na *Rys.3* szkic całego układu. Niezwykle ważne jest to, że eksperymentator ma dostęp do całego układu poprzez przenośne komputery osobiste (Radio Shack 1000 i podobne) a poprzez połączenie układu z centralną siecią Instututu w Jülich i dalej przez EARN może ze swego stanowiska doświadczalnego porozumieć się z większością ośrodków badawczych na świecie. Ta globalna łączność pozwala na prowadzenie jednego doświadczenia umiejscowionego fizycznie w Jülich przez zespół *rozproszonych po całym świecie* ludzi.



Rys.2

Zastosowania komputerów w fizyce doświadczalnej powierzchownie li tylko mogą wydać się podobne do zastosowań w prowadzeniu działalności dużej fabryki. W istocie rzeczy warunki narzucone na sprzęt, a przede wszystkiem na oprogramowanie, są w

zastosowaniach fizycznych znacznie ostrzejsze. Przy prowadzeniu doświadczenia naukowego najważniejsze jest bowiem prawidłowe rozróżnienie pomiędzy błędami doświadczalnymi a *nowymi* efektami. Uwieńczone nagrodą Nobla odkrycie cząstek J i ψ o mało co nie zostało "wygładzone" przez automatyczną ocenę danych doświadczalnych.



RYS. 3

¶3

Gwałtowny rozwój zastosowań technik komputerowych w *fizyce doświadczalnej* jest zjawiskiem stosunkowo nowym. Przed 1970r. minikomputery były drogie i skomplikowane w użyciu, (głównie z powodu małej pamięci wewnętrznej: około 2 do 8 kB), tak że opłacało się je stosować prawie wyłącznie tam, gdzie duże koszta materiałowe i obsługi ludzkiej to uzasadniały. Typowym zastosowaniem były doświadczenia reaktorowe i akceleratorowe. Pojawienie się PDP-11 i systemu operacyjnego RSX-11M spowodowało stopniowe wypieranie elektroniki analogowej przez cyfrową.

Znacznie wcześniej, bo wraz z pojawieniem się pierwszych komputerów informatyka wkroczyła do fizyki teoretycznej. De facto pierwsze molochy obliczeniowe jak ENIAC powstały jako urządzenia pomocnicze dla programów z fizyki. Pierwsze ich zastosowania, do analizy dynamiki populacji neutronów w urządzeniach typu reaktora

lub bomby, były jednocześnie pierwszymi próbami tego co dziś stało się chlebem powszednim tzw *fizyki komputerowej*.

Pobieżne zcharakteryzowanie tego czym zajmuje się *fizyka komputerowa* zajęło by zbyt dużo czasu. Niekompletne omówienie zawarłem w pracy[3].

(Jak Państwo wiecie, w wielu uczelniach wyższych jak grzyby po deszczu, pojawiają się sekcje fizyki komputerowej. Nie bardzo wiem, jak można na poważnie uprawiać u nas tą dyscyplinę skoro np. Uniwersytet Warszawski dysponuje tylu PC-tami co Los Alamos Vaxami. Odnoszę wrażenie, że sekcje te powstają po to aby przyciągnąć na Wydziały Fizyki młodzież zainteresowaną mikrokomputerami. Źle by było aby sekcje te pozostały sekcjami nauki *wszystkiego, co tylko chciałeś wiedzieć o DOS-ie ale bałeś się zapytać*.)

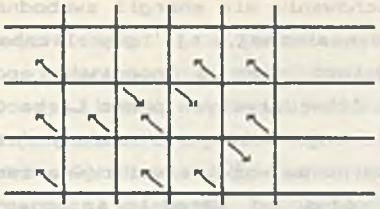
Tutaj ograniczę się do omówienia zastosowań w fizyce statystycznej, dyscyplinie fizyki, w której pracuję.

Komputery, zaraz po ich skonstruowaniu, zostały zastosowane do symulacji układów wielu ciał. Pojawienie się metody *Monte Carlo*[4] umożliwiło zastosowanie komputerów do symulowania *równowagowych* własności układów wielu ciał. Metoda ta odgrywa kluczową rolę we współczesnych badaniach układów magnetycznych. Większość z Państwa, jak mniemam, słyszała o *modelu Isinga* (jest to de facto jeden z podstawowych modeli w fizyce). Model ten to d-wymiarowa sieć, w węzłach której umieszczone są momenty magnetyczne-spiny. Każdy z tych spinów może przyjmować dwie możliwe wartości +1/2 i -1/2 albo po prostu 0,1 (*Rys. 4*). Dodatkowo owe momenty-spiny oddziałują ze sobą i w najprostszym możliwym przypadku oddziaływanie to ograniczone jest do najbliższych sąsiadów na sieci i proporcjonalne jest do iloczynu wartości dwu spinów.

---

[3] Ł. A. Turski *Computers in Physics and Physics in Computers*, w: Theory of Condensed Matter, volume dedicated to Professor Z. M. Galasiewicz. Wrocław (1986)

[4] N. Metropolis, i St. Ulam, *The Monte Carlo Method*. J. Am. Statist. Assoc. 44, 335, (1949)



Rys. 4

Całkowita energia $\mathcal{E}$ takiego układu jest sumą energii oddziaływań wszystkich takich par, jakie da się na danej sieci utworzyć. Zgodnie z prawami fizyki statystycznej prawdopodobieństwo znalezienia układu o energi $\mathcal{E}$ jest równe $\frac{1}{\mathcal{Z}} \exp(-\beta\mathcal{E})$, gdzie β jest proporcjonalna do odwrotności temperatury. Logarytm czynnika normalizacyjnego $\mathcal{Z}$ jest proporcjonalny do *eneregii swobodnej* i zawiera całą ważną z punktu widzenia *termodynamiki* układu informację. Ażeby obliczć $\mathcal{Z}$ trzeba zsumować czynniki Boltzmannowskie $\exp(-\beta\mathcal{E})$ po wszystkich możliwych konfiguracjach spinów. Duże komputery, nie mówiąc już o superkomputerach, potrafią wykonywać miliardy kroków Monte Carlo dla modeli Isinga z ilością spinów $N_s \sim 1000$.[5]

Metoda Monte Carlo pozwala na badanie własności modelu Isinga i jego pochodnych w dowolnej liczbie wymiarów d i dla dowolnych topologii sieci. W dodatku dla sieci dwuwymiarowych znamy ścisłe

---

[5] Podstawowe wiadomości o zastosowaniach metody Monte Carlo do badania układów spinowych można znaleźć w K. Binder Monte Carlo Methods in Statistical Physics. Springer Verlag, Berlin 1979, zastosowania metody Monte Carlo do badania układów cząstek omówione są np. w: J. P. Valleau, and S. G. Whittington *A Guide to Monte Carlo for Statistical Mechanics: 1. Highways*, J. P. Valleau, and G. M. Torrie *A Guide to Monte Carlo for Statistical Mechanics: . Byways*. w: Statistical Mechanics. Part A. B. J. Berne ed. Plenum Press New York 1977

wyniki dotyczące zachowania się energii swobodnej modelu Isinga w tzw granicy termodynamicznej, tj. gdy liczba spinów dąży do nieskończoności. Dzięki temu , naprawdę epokowemu wynikowi, uzyskanemu w latach czterdziestych przez Larsa Onsagera, wiemy że dwuwymiarowy model Isinga doznaje przemiany fazowej *paramagnetyk* (średnia wartość spinu na węzeł sieci równa zeru) *-ferromagnetyk* (ta sama średnia różna od zera) oraz znamy dokładnie tzw. wykładniki krytyczne opisujące wszystkie własności fizyczne modelu w pobliżu temperatury przemiany fazowej. W trzech wymiarach nie znane są ścisłe rozwiązania modelu Isinga, znane są natomiast, dzięki symulacjom komputerowym, wykładniki krytyczne i to z "dowolną" dokładnością.

Metoda Monte Carlo pozwala też na badanie własności dynamicznych modelu Isinga [6]. Prawdziwe bogactwo własności posiada uogólniony model Isinga , który nazywamy modelem *szkła spinowego*. W modelu tym współczynniki sprzężenia pomiędzy spinami są zmiennymi losowymi. Poza szeregiem własności interesujących fizyków (np. dwuwymiarowe modele szkieł spinowych posiadają szereg własności podobnych do ostatnio odkrytych *wysokotemperaturowch nadprzewodników*) modele szkieł spinowych mogą służyć jako modele układów uczących się i rozpoznających obrazy [7].

Model Isinga można sobie "symulować" także na komputerze typu PC-ta. Wyniki takich symulacji dla małego dwuwymiarowego modelu Isinga wykonane na komputerze MACINTOSH pokazane są na planszy *Pi*. Superkomputery wykorzystywane do prawdziwych symulacji modeli *a la* Ising są urządzeniami raczej drogimi. Dlatego też w latach 82-83 gwałtownie rozwinęła się technologia komputerów specjalnego zastosowania przeznaczonych do symulacji Monte Carlo układów spinowych. Jedną z pierwszych takich maszyn *Santa Barbara spin flipper* zbudowała grupa doktorantów i "postdoc-ów" Instytutu Fizyki Teoretycznej w Santa Barbara. Maszyna zbudowana została

---

[6] A.T.Ogielski *Dynamics of Fluctuations in the Ordered Phase of Kinetic Ising Models*. NSF-ITP (Santa Barbara)-87-82.

[7] J.J.Hopfield Proc. Natl.Acad.Sci.USA 79, 2554, (1982), E.Biennenstock, and C.von der Malsburg, *A Neural Network for Invariant Pattern Recognition*. w: Europhysics Letters 4,(1) 121,(1987)

całkowicie z gotowych części i zlutowana niemalże w pakamerze przez teoretyków o umiarkowanych zdolnościach manualnych. Spin flipper pobił rekordy ilości wykonywanych kroków Monte Carlo. Dziś takie maszyny posiada wiele centrów badawczych np. AT&T Bell Laboratories[8]. Andrew Ogielski w cytowanej powyżej pracy badał układy o 512x512 spinach, 64x64x64 i 16x16x16x16, odpowiednio w d=2,3,4 wymiarach. Wykonanie takich rozmiarów symulacji na superkomputerze ogólnego zastosowania byłoby finansowo rujnujące. Chciałbym podkreślić, że symulacja w d=4 jest szczególnie ważna. Tak się bowiem składa, że istnieje ścisły związek pomiędzy tego typu układami sieciowymi a uprawianymi z nadzieją na zrozumienie podstawowych praw mikroświata teoriami pola, np. chromodynamiką kwantową. Obecnie to właśnie specjaliści od teorii pola zużywają najwięcej czasu na symulacje sieciowe, na zarówno super- jak i specjalnego zastosowania komputerach.

Oczywiście nie wszystkie układy fizyczne można modelować przy pomocy modelu Isinga. Drugą bardzo ważną dziedziną zastosowań symulacji komputerowych jest tzw. *dynamika molekularna*, technika w której opisujemy zachowanie się układu cząstek, atomów lub molekuł, poruszających się w d=2,3 wymiarowj przestrzeni i oddziałujących ze sobą albo porzez zadane siły potencjalne albo jak zderzające się twarde kule. Tego typu symulacja wymaga rozwiązywania układu równań Newtona, tj, układu dxN_T równań różniczkowych zwyczajnych drugiego rzędu, gdzie N_T jest liczbą cząstek w badanym układzie.

Na *Rys.5* przedstawiłem schemat takiego modelowania cyfrowego. Zasadniczą rolę odgrywa tu wybór algorytmu rozwiązującego te równania [9]. Ponieważ N_T jest duże, rzędu kilkuset (dla np. CRAY'a XMP w d=3 N_T ~1000) to algorytmy wyższego rzędu, takie jak np. Runge-Kutta, które wielokrotnie obliczają prawe strony równań w ramach jednego kroku całkowania, są nie do przyjęcia. Z *Rys.5*

---

[8] patrz np. J.H.Condon i A.T.Ogielski, *Rev. Sci. Instrum.* 56,1961 (1985)

[9] patrz np. D.Greenspan Discrete Numerical Methods in Physics and Engineering. Academic Press, New York 1974. Doskonałe wprowadzenie do metod numerycznych dynamiki molekularnej zawarte jest w R.W.Hockney, J.W.Eastwood Computer Simulations Using Particles. McGraw-Hill, New York 1981

wynika, że ilość obliczeń prawych stron, tj. sił działających pomiędzy cząstkami, rośnie kwadratowo z ilością cząstek. Niemal cały czas obliczeń to obliczanie sił!.

Praktyka wykazała, że najlepszymi i najszybszymi są algorytmy takie jak w p.8 na *Rys.5*. Nazywam ten algorytm algorytmem szkolnym, ponieważ najważniejszym w nim jest druga linijka, związek pomiędzy pochodną czasową położenia a prędkością średnią. Związek ten jest dokładnie taki, jakim posługujemy się w szkole ucząc mechaniki punktu materialnego. Algorytm szkolny ma szereg innych ciekawych własności, np. zachowuje on energię (co nie ma miejsca w algorytmie Runge-Kutta), daje się prosto uogólnić na przypadek sił zależnych od prędkości (oddziaływania elektrodynamiczne mają tą "przykrą" własność) oraz dość prosto daje się przenieść na przypadek mechaniki relatywistycznej.

Dynamika molekularna jest dziś konwencjonalnym źródłem informacji o własnościach statystycznych układów. Zastosowanie jej opiera się na wykorzystaniu twierdzenia *a la* ergodycznego mówiącego o tym, że średnia statystyczna dowolnej wielkości fizycznej $A(\langle\vec{r}\rangle,\langle\vec{p}\rangle)$, będącej funkcją położeń i pędów cząstek jest równa średniej czasowej obliczonej dla tej samej wielkości po trajektorii dynamicznej układu:

$$\langle A(\langle\vec{r}\rangle,\langle\vec{p}\rangle)\rangle = \lim_{T\to\infty} \frac{1}{T}\int_0^T A(\langle\vec{r}(t)\rangle,\langle\vec{p}(t)\rangle)\,dt$$

Skąd jednak wiemy, że obliczona numerycznie przy pomocy algorytmu z *Rys.5* trajektoria układu, jest w istocie prawdziwą trajektorią dynamiczną układu wielu ciał? . Czy fakt skończonej dokładności obliczeń nie ma wpływu na nasze wyniki?. Na pytania te daje częściową odpowiedź uznawane powszechnie z obowiązujące, ale udowodnione li tylko w prostych przypadkach i przy całej masie dodatkowych założeń, twierdzenie o cieniach Anosova -Bowena[10].

---

[10] G. Benettin, M. Casartell, L. Galgani, A. Giorgilli, i J.-M. Strelcyn, Nuovo Cimento B49, 183, (1978). R. Bowen Lectures Notes in Mathematics 470, Springer Verlag Berlin 1975

```
1 Ile cząstek (N_T) i w ilu wymiarach (d)
2 Ustal typ oddziaływań F_ij = -∇_i V(r_ij)
  V(r_ij) = np. potencjał Lennarda-Jonesa
  ustal krok czasowy DT
3 wybierz początkową konfigurację Q_o(r_i)
4 wygeneruj początkowy rozkład prędkości P(v_i)
  np. metodą Monte Carlo tak aby
  0.5Σ m_i v_i^2 = 3N_T T k_B /2: T-temperatura
5 Wyzeruj wektory sił
  for i:= 1 to N_T do
    F_i := 0;
  end;
6 Zaczynamy iteracje  Kiter = 0
7 obliczanie sił
  for i:= 1 to N_T-1  do
   for j := i+1 to N_T do
     oblicz F_ij ;
     F_i := F_i + F_ij ;
     F_j := F_j - F_ij ;
   end;
  end;
8 całkowanie równań ruchu
  for i:= 1 to N_T  do
   v_i^nowe := v_i^stare + DT*F_i/m_i;
   x_i^nowe := x_i^stare + DT*(v_i^nowe + v_i^stare)/2;
  end;
9 Kiter := Kiter +1;
  if Kiter = Kwydruk then
    zapisz (x_i),(v_i)
  if Kiter =< Kmax goto 7
end.
```

Rys. 5

Niechaj $\mathcal{M}$ będzie zwartą rozmaitością metryczną i niechaj $d(a,b)$ oznacza odległość pomiędzy $a,b \in \mathcal{M}$. Niechaj $\mathcal{T}$ będzie diffeomorfizmem (klasy Anosova) $\mathcal{M}$ w siebie, a $\mathcal{T}^n$ niech oznacza n-tą iterację $\mathcal{T}$. Ciąg punktów $\mathcal{M} \ni x_i$ $i = 1,2,\ldots$ jest α-*pseudo-orbitą* w $\mathcal{M}$ gdy $\forall_i\ d(\mathcal{T}x_i, x_{i+1}) < \alpha$. Mówimy, że punkt $x \in \mathcal{M}$ jest β cieniem ciągu (x_i) gdy $\forall_i\ d(\mathcal{T}^i x, x_i) \leq \beta$. Twierdzenie Anosova-Bowena o cieniach można wypowiedzieć w następujący sposób:

Dla każdego $\beta>0$ *istnieje takie* $\alpha>0$, *że każda* α-*pseudo-orbita* (x_i) *z* $\mathcal{M}$ *posiada swój* β-*cień.*

Zastosowanie tego twierdzenia do dynamiki molekularnej jest dość oczywiste. $\mathcal{M}$ jest przestrzenią fazową układu, $\mathcal{T}$ jest jej przekształceniem kanonicznym odpowiadającym ruchowi (z dyskretnym czasem). Wybierając dostatecznie małe DT (*Rys.5*) spełniamy wymagania, aby generowany przez nas numerycznie zbiór wartości położeń i pędów cząstek stanowił α-pseudo-orbitę. Każda taka pseudo-orbita będzie miała swój "prawdziwy" newtonowski cień. Jedyną "trudnością" jaką musimy pokonać, to udowodnienie, że nasz układ dynamiczny, zdefiniowany przez zadanie oddziaływań międzycząstkowych, generuje diffeomorfizm Anosova. Z dowodami takiego typu są, jakże by inaczej, poważne kłopoty. Jeden z najważniejszych modeli w fizyce statystycznej -układ wielu twardych kul - nie jest układem spełniającym założenia twierdzenia Anosova -Bowena.

Zapewne fizycy nie bardzo przejmowali by się subtelnościami matematycznymi, gdyby nie obserwacja poczyniona przez Aldera i współpracowników[11] na podstawie symulacji komputerowych układów twardych kul, że zależne od czasu funkcje korelacji, np. funkcja korelacji prędkość-prędkość, zanikają dla długich czasów nie *wykładniczo*, jak wynikało to z konwencjonalnej analizy równań kinetycznych, ale *potęgowo* .

Powszechnie znana teoria oparta na równaniu Langevina przewiduje, że funkcja autokorelacji prędkości cząstek ma postać

---

[11] B. J. Alder i T. E. Wainwright *Phys. Rev. Lett.* 18,988, (1967). *Phys. Rev* A1,18, (1970) patrz także W. W. Wood w Fundamental Problems in Statistical Mechanics III. E. G. D. Cohen ed. North Holland Amsterdam 1975, J. P. Boon i S. Yip Molecular Hydrodynamics. McGraw-Hill, New York 1980

$\langle \vec{v}_1(t) \cdot \vec{v}_1(0) \rangle = \phi(t) = (k_B T/M) \exp(-t/\tau)$, gdzie $\tau = M/6\pi\eta R$. M jest masą cząstki, R- jej promieniem, T- temperaturą, η współczynnikiem lepkości cieczy, $\vec{v}_1(t)$ jest tu prędkością wybranej cząstki w cieczy. Analiza wyników komputerowych Aldera i współpracowników dała, ku wielkiemu zaskoczeniu wszystkich zainteresowanych, $\phi(t) \sim t^{-d/2}$ gdzie d jest wymiarem przestrzennym układu. Konsekwencje wyniku Aldera są dramatyczne. Całka z funkcji autokorelacji $\phi(t)$ jest równa współczynnikowi (samo)dyfuzji w cieczy.

$$\int_0^\infty \phi(t)dt = D$$

Potęgowy zanik funkcji $\phi(t)$ prowadzi do *nieistnienia* współczynnika D w $d \leq 3$! Prostą konsekwencją powyższego faktu jest nieistnienie równań hydrodynamiki dla mniej niż trzy wymiarowych układów. Oczywiście nadal ważne są *prawa zachowania* ilości cząstek, pędu i energii, ale *prawa konstytutywne* muszą być inne.

Obserwacje Aldera zostały potwierdzone przez wielu innych badaczy, także dla układów z oddziaływaniami typu Lennarda-Jones'a. Obserwacja ta, poczyniona czysto komputerowo i bez żadnych przesłanek teoretycznych doprowadziła do kompletnej zmiany w naszym rozumieniu podstaw fizyki statystycznej i mechaniki ośrodków ciągłych.

Przeanalizujmy teraz proces uzyskiwania funkcji $\phi(t)$. W dynamice molekularnej generujemy pseudo-orbitę cząstki w przestrzeni fazowej $\mathcal{M}$. Trajektorie cząstek wykorzystywane do statystycznej analizy funkcji korelacji ϕ to trajektorie w "zwykłej" przestrzeni, trajektorie te *nie muszą* odpowiadać trajektoriom żadnych prawdziwych cząstek fizycznych. W istocie można się przekonać, że już po czasie rzędu kilkunastu czasów między zderzeniami odwrócona w czasie trajektoria cząstki nie będzie pokrywać się z trajektorią wygenerowaną "do przodu" w czasie. Zgodnie z tw. Anosowa-Bowena oznacza to, że po pewnym czasie cieniem orbity staje się trajektoria newtonowska, ale *innej* cząstki !. Oznacza to, że generując funkcję $\phi(t)$ dla dużych t naprawdę obliczamy

$$\langle \vec{v}_1(t)\cdot\vec{v}_1(0)\rangle \text{ dla } t\in(0,t_1)$$

$$\langle \vec{v}_2(t)\cdot\vec{v}_1(0)\rangle \text{ dla } t\in(t_2,t_3)$$

$$\langle \vec{v}_3(t)\cdot\vec{v}_1(0)\rangle \text{ dla } t\in(t_4,t_5)$$

$$\langle \vec{v}_n(t)\cdot\vec{v}_1(0)\rangle \text{ dla } t\in(t_{2n-2},t_{2n-1})$$

gdzie numeryczna trajektoria zgadza się z trajektorią Newtonowską n-tej cząstki w przedziale czasu (t_{2n-2},t_{2n-1}).

Obliczając funkcję $\phi(t)$, tak naprawdę badamy nie funkcje korelacji jednej cząstki, a funkcje korelacji między cząstkami. Takie wzajemne korelacje pojawiają się wtedy gdy mówimy nie o pojedynczych cząstkach, a o *płynach*. W istocie, rozważmy hydrodynamiczne pole prędkości $\vec{v}(\vec{r},t)$ układu wielu ciał zdefiniowane następująco :

$$\rho(\vec{r},t)\vec{v}(\vec{r},t) = \sum_{i=1}^{N_T}\vec{v}_i(t)\delta(\vec{r}-\vec{r}_i(t))$$

gdzie $\rho(\vec{r},t) = \sum_{i=1}^{N_T}\delta(\vec{r}-\vec{r}_i(t))$ jest gęstością cząstek w układzie. Obliczając teraz funkcję autokorelacji pola prędkości $\vec{v}$ otrzymamy N_T^2 członów zawierających korelacje pomiędzy różnymi cząstkami oraz N_T członów zawierających funkcje autokorelacji. Oczywiście pierwsze będą dawać zasadniczy wkład do obliczanej funkcji.

Z powyższego wynika, że potęgowy zanik funkcji $\phi(t)$ jest związany z własnościami kolektywnymi układu, a nie z dynamiką jednej cząstki w cieczy. Komputerowa analiza układu pozwoliła nam na dokonanie doświadczenia, w którym udało się wyodrębnić w dynamice to, co bardzo trudno byłoby zobaczyć w "prawdziwym" doświadczeniu[12]. Intensywne prace teoretyczne, zarówno w ramach teorii kinetycznej jak i korzystające z innych formalizmów, np. sprzężenia modów czy funkcji pamięci Mori-Zwanziga, dostarczyły przekonywujących argumentów za tym, że wynik Aldera jest

---

[12] Pomiarem bliskim takiego doświadczenia jest rozpraszanie spolaryzowanego światła na zawiesinie mikroskopowych cząstek. G. Paul i P. N. Pusey *J. Phys.* A14, 3301, (1981)

prawidłowy. Zechcą Państwo zwrócić uwagę na fakt, że mamy tu nie tylko do czynienia z nowym faktem odkrytym metodą symulacji komputerowej, ale również wynikiem numerycznym nie polegającym na produkcji tysiąca wykresów czy też tabel, lecz prowadzącym do prostego wyrażenia *analitycznego*. Oczywiście mając wygenerowane położenia i prędkości cząstek można się pokusić o graficzną ilustrację dynamiki wielu ciał. Na kilku planszach mogą Państwo obejrzeć np. trajektorie cząstek w komputerowym krysztale przed i po jego stopieniu.

Asymptotyczne zachowanie się funkcji $\phi(t)$ nosi nazwę ogona czasowego. Od momentu opublikowania wyników Aldera takie ogony zostały odkryte w wielu innych zagadnieniach fizyki statystycznej, we wszystkich są one związane z kolektywnymi, hydrodynamicznymi zachowaniami się układów wielu ciał.

14

W poprzednim paragrafie starałem się przedstawić Państwu w dużym skrucie jeden z najciekawszych wyników zastosowania symulacji komputerowych w fizyce statystycznej. Zakończyłem mówiąc, że zjawisko ogonów czasowych jest de facto zjawiskiem hydrodynamicznym. Zjawiska hydrodynamiczne symulowane były na komputerach od bardzo dawna. W latach 60-tych Edward Lorenz z MIT badał rozwiązania równań Naviera-Stokes'a opisujących powstawanie komórek konwekcyjnych w podgrzewanych od dołu warstawach cieczy. Analiza Lorenza doprowadziła do powstania działu fizyki, czy też matematyki stosowanej, *nauki o chaosie*[13].

Komputerowe symulacje w hydrodynamice, do mniej więcej końca 1985r, polegały na zastosowaniu metod dyskretyzacyjnych do rozwiązywania równań hydrodynamiki. Na *Rys.6* pokazałem przykład takiego układu równań służących do opisania zjawiska wypływu strumienia gazu z dyszy, do zbiornika zawierającego gaz pod zadanym ciśnieniem. Tego typu zagadnienia są powszechnie

---

[13] Michel Feigenbaum, jeden z twórców fizyki chaosu, odkrywca tzw. współczynników α i δ charakteryzujących przejście w chaos w odwzorowaniu logistycznym, twierdzi że pierwsze obserwacje *numeryczne* poczynił posługując się kalkulatorem HP-25. Rola jaką kalkulatory kieszonkowe odegrały we współczesnej fizyce wymagała by historycznego opracowania.

spotykane w wielu działach aero- i hydrodynamiki stosowanej, w fizyce plazmy, a także w zagadnieniach biologicznych np. w urologii.

Rys.7 przedstawia schemat takiego doświadczenia, a na planszy P3 pokazano wyniki uzyskane w *prawdziwym doświadczeniu laboratoryjnym* porównane, z wynikami symulacji. Zgodność tych wyników jest naprawdę fenomenalna. Porównanie kosztów symulacji z kosztami budowy, np. dużych tuneli hydrodynamicznych, wypada zdecydownie na korzyść pierwszych[14].

RÓWNANIA EULERA
DLA PRZEPŁYWÓW ŚCIŚLIWYCH

Zasada zachowania masy:

$$\partial_t \rho + \vec{\nabla} \cdot \rho \vec{v} = 0$$

Zasada zachowania pędu:

$$\partial_t \rho \vec{v} + \vec{\nabla} \cdot \rho \vec{v} \vec{v} + \vec{\nabla} P = \vec{\mathcal{D}}$$

Zasada zachowania energii:

$$\partial_t e + \vec{\nabla} \cdot \rho e \vec{v} + P \vec{\nabla} \cdot \vec{v} = \mathcal{D}$$

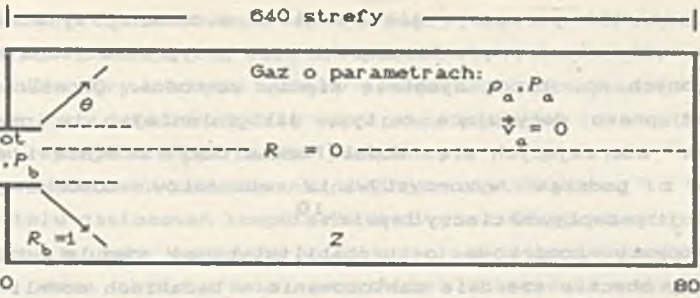
Równanie stanu gazu idealnego:

$$P = \rho c^2 / \gamma = (\gamma - 1) \rho e$$

ρ= gęstość; c=prędkość dźwięku
$\vec{v}$= prędkość; $\gamma = C_p / C_v$; P=ciśnienie
e= gęstość energii wewnętrznej
$\vec{\mathcal{D}}$ = "wektor lepkości"
$\mathcal{D}$="źródło ciepła"

Rys.6

[14] M.L.Norman i K-H.A.Winkler *Los Alamos Science* 12, (1985)



Rys. 7

W ciągu ostatnich kilku lat symulowanie hydrodynamiki na komputerach ulega gwałtownej zmianie. Zmiana ta związana jest z zastosowaniem do symulacji przepływów dwuwymiarowych *automatów komórkowych*[15].

Fizycy zainteresowali się automatami komórkowymi dopiero niedawno, znacznie wcześniej zaczęto je stosować w biologii matematycznej[16] i w badaniach nad sztuczną inteligencją[17].

Automaty komórkowe z deterministycznymi regułami określającymi stan automatu w *t+1* chwili czasu są intensywnie badane i zaczynają odgrywać taką samą rolę, jak równania różniczkowe cząstkowe, których są bliskimi "dyskretnymi"krewnymi. W 1973 r Pomeau[18] zauważył, że można zbudować sieciowy model, którego dynamika "przedrzeźnia" dynamikę cieczy lepkiej. W modelu

---

[15] J. von Neumann, Theory of Self-Reproducing Automata. Univ. of Illinois Press, 1966

[16] patrz np. Disordered Systems and Biological Organization. L. Bienenstock, F. Fogelman, i G. Weisbuch (Eds.) Les Houches (1985)

[17] patrz np. Pattern Recognition. B. Batchelor (Ed.) Plenum Press 1978

[18] J. Hardy,, O. de Pazzis, i Y. Pomeau, *J. Math. Phys.* 14,470,(1973); U. Frisch, B. Hasslacher, i Y. Pomeau, *Phys. Rev. Lett.* 56,1505,(1986)

Pomeau cząstki poruszają się po sieci o zadanej symetrii i w każdym węźle sieci prędkość cząstki może przyjmować jedną z kilku dozwolonych np. przez symetrię sieci, wartości. Określone jest również prawo decydujące o tym, jak zmieniają się prędkości cząstek zderzających się. Model Pomeau odżył obecnie i stanowi jedną z podstaw wykorzystywania automatów komórkowych do symulacji przepływów cieczy lepkich[19].

Automaty komórkowe o probabilistycznej regule działania znalazły obecnie szerokie zastosowanie w badaniach modeli fizyki statystycznej[20]. Model Isinga, o którym już wspominałem, może też być "przybliżony" przez probabilistyczny automat komórkowy.

Nic nie stoi na przeszkodzie aby badać automaty komórkowe i ich własności posługując się komputerami ogólnego zastosowania. Znacznie lepsze wyniki uzyskuje się jednak wykorzystując maszyny o specjalnej konstrukcji. Takie wyspecjalizowane maszyny pozwalają osiągnąć wyniki porównywalne lub *lepsze* od superkomputerowych przy znacznie mniejszym koszcie. Typowym przykładem takiego wyspecjalizowanego komputera jest CAM[21]. Pierwsze dostępne na rynku modele tegu komputera to karty, wstawiane do dowolnego komputera zgodnego z IBM AT plus wysokiej jakości monitor kolorowy pozwalający na obserwacje, w czasie rzeczywistym, ewolucji układu. Koszt CAM-u (bez monitora) jest rzędu $1500[22]. Nieco bardziej rozbudowany CAM, np. pozwalajacy na programowanie nie w języku FORTH, ale w normalnym FORTRAN-ie powoli zastępuje konwencjonalne komputery w obliczeniach aerodynamicznych. Np. dzięki zastosowaniu komputerów typu CAM obliczono porawki trzeciego rzędu do siły oporu Stokes'a, działającej na ciało poruszające się w cieczy lepkiej (dla kuli o promieniu R $F_S = 6\pi\eta R + \ldots$, gdzie η jest lepkością cieczy).

Na zakończenie tego fragmentu chciałbym podkreślić, że sukces zastosowania spin flipperów, czy też CAM, a także nie omówionych

---

[19] S. Wolfram, *Rev. Mod. Phys.* 55,601, (1983); *Cellular Automaton Fluid Dynamics* 1987 w druku; T. Toffoli, *Physica* D,117,(1984).

[20] P. Rujan, *J. Stat. Phys. (1987) w druku*

[21] T. Toffoli, *Physica* 10D,195,(1984)

[22] M. Droz, informacja prywatna tyle wynosił koszt zakupy CAM' u zainstalowanego w Instytucie Fizyki Teoretycznej Uniwersytetu Genewskiego.

przeze mnie prób zbudowania specjalnego komputera do dynamiki molekularnej wskazują na to, że w badaniach fizycznych coraz większą rolę odgrywać będą maszyny specjalnego zastosowani. Jest to jedna z tych dziedzin, gdzie współpraca pomiędzy informatykami i fizykami wydaje się być bardzo pożądana.

Jak już mówiłem, zastosowania w fizyce statystycznej to tylko jedno z wielu zastosowań komputerów w fizyce. Ciągle bardzo dużą rolę odgrywają duże komputery w obliczeniach *nie symulacyjnych*. Przykładem mogą być prawie wszystkie obliczenia własności kwantowych. (Symulacje kwantowe są dopiero na początku rozwoju. Pierwsze kwantowe symulacje dotyczyły prostych układów spinowych i pewnej wersji metody Monte Carlo). Jednym z zagadnień, z którego rozwiązaniem wiążemy duże nadzieje, jest problem *zimnej fuzji termojądrowej*. Nie wdając się w szczegóły jest to zupełnie nowe podejście do problemu wywoływania reakcji syntezy jąder wodoru, w której rolę katalizatora w pewnym sensie odgrywa miuon. W teorii tego zjawiska typowe obliczenia poziomów energetycznych i funkcji falowych układu trzech ciał, np. dwa deuterony i miuon ,sprowadzają się do rozwiązywania około 900 nieliniowych równań różniczkowych. Po to aby obliczyć energie własne z dokładnością 10^{-2} - 10^{-3} ev, grupa fizyków z Dubnej zużyła 1500 godzin CPU komputera CDC-6500 [23]. Typowy rozmiar zbioru danych tworzonego w tych obliczeniach waha się od 200 do 1000 Mbajtów.

¶5

Gdyby nie fenomenalny wręcz postęp w konstrukcji maszyn liczących większość zastosowań komputerów w fizyce, o których mówiłem uprzednio, pozostałaby w sferze fantazji. Postęp w konstrukcji komputerów mógł następować tak burzliwie dlatego, że fizyka ciała stałego "stanęła na wysokości zadania" i inżynierowie projektanci komputerów zawsze mogli liczyć na to, że pojawią się nowe materiały, ba, nawet nowe zjawiska fizyczne, umożliwiające zbudowanie szybszych i bardziej niezawodnych maszyn. Ale powoli zaczynamy zbliżać się do kresu tego, na co pozwala znana nam dzisiaj technologia.

---

[23] W.P.Gerdt, M.G.Meszczeriakov, i D.W.Szirkov, DUBNA preprint P2-86-848

Przez dłuższy czas w literaturze poświęconej fizycznym podstawom przekazywania i przetwarzania informacji "straszył" problem $k_B T \log 2$. Zgodnie z powszechną opinią przetwarzanie informacji jest zjawiskiem *nieodwracalnym*. W istocie rzeczy znając wynik prostego obliczania (n,m,: liczby całkowite) n+m =8 nie mogę powiedzieć ile wynosiła wartość n i m. Pewna ilość informacji została stracona w tym procesie obliczeniowym. Ponieważ wynik jest zawsze zawarty w danych wejściowych, to żaden proces przetwarzania informacji nie może prowadzić do produkcji informacji.

Komputery, takie jakie budujemy dzisiaj i zapewne budować będziemy w wyobrażalnej przyszłości, składają sie z gigantycznej ilości bramek elektronicznych, takich jak np. bramka AND, z których każda jest urządzeniem niszczącym informację. Pytanie, które powinniśmy postawić, to czy takie nieodwracalne bramki są istotnie potrzebne do wykonywania obliczeń?[24]. Jeżeli tak to komputery zawsze będą musiały tracić energię na wykonywanie swoich zadań. Można z grubsza oszacować, ile energii trzeba zużyć na wykonanie obliczeń analizując działanie przełącznika binarnego pracującego w temperaturze T. Z termodynamiki wynika, że średnio na jedno przełączenie takiego "klucza" potrzeba $k_B T \log(2)$ energii (k_B stała Boltzmanna $\cong 1.36\ 10^{-16}$ erg/stopień Kelvina). Na wykonanie N operacji przełączania potrzeba więc $Nk_B T \log(2)$ energii co dla dużych N zaczyna być niebagatelną ilością. Powstaje pytanie, czy rzeczywiście na wykonanie operacji zapisania albo zatarcia bitu potrzeba stracić te $k_B T \log(2)$ ergów i czy nieodwracalne tracenie bitów jest konieczne. Fredkin[25] zaproponował model odwracalnej bramki i pokazał, że można z szeregu takich bramek skonstruować bramkę NAND. Oczywiście sprawą otwartą pozostaje skonstruowanie nawet *przybliżonego* modelu takiej bramki. Najlepszymi kandydatami są tu układy typu pęcherzyków magnetycznych i elektoniczne układy wykorzystujące efekt Josephsona. (Kilka lat temu IBM zlikwidował grupę badawczą pracującą nad konstrukcją Josephsonowskiego tranzystora i układów

---

[24] R. Landauer IBM Journal of Research and Development 5,183 (1961), Inter. Journal of Theor. Phys. 21, 283, (1982)

[25] patrz np. E. Fredkin i T. Toffoli Int. Journal of Theor. Phys. 21,219, (19882)

liczących na tym tranzystorze opartych. Jak Państwo wiecie tegoroczną nagrodę Nobla w fizyce otrzymali K.Alex Müller i J.Georg Bednorz z laboratorium IBM w Ruschlikonie (pod Zürichem) za odkrycie wysoko temperaturowego nadprzewodnictwa - zjawiska otwierającego drogę do elektroniki Josephsonowskiej pracującej w temperaturach ciekłego azotu, a być może i pokojowej !).

Jednym z pomijanych w dyskusji odwracalnego przetwarzania informacji problemów jest *stabilność* takiego procesu. Głębsza analiza tego zagadnienia wykracza poza ramy mojego wykładu, chciałbym jednak nadmienić, że to co decyduje o tym, że urządzenia mechaniczne i elektroniczne działają w sposób niezawodny jest *dysypacja*. Zamek w drzwiach pozostaje zamknięty tylko dlatego, że na jego otwarcie, przy pomocy klucza lub inaczej,potrzeba stracić pewną ilość energii. Dysypacja jest też zjawiskiem, które w sposób naturalny implikuje logiczną kolejność zdarzeń. Proszę zwrócić uwagę na fakt, że puszczony "wstecz" film, lub program video wydaje się nienaturalny dlatego, że "wyskakujący" z wody aktor porusza się *co raz szybciej*!. Uwzględnienie dysypacji staje się bardzo złożonym, gdy zaczynamy dyskutować o układach rządzonych prawami mechaniki kwantowej. Efekty kwantowe są pomijane w analizie przetwarzania informacji (oczywiście samo działanie tranzystora jest efektem kwantowym), powoli jednak wraz z postępującą miniaturyzacją i integracją będziemy musieli się zastanowić nad tym jak działają kwantowe urządzenia liczące. Na razie jednak pomimo osiągnięcia praktycznie granic optycznej litografii w produkcji układów wielkiej integracji i pomimo pojawiania się pierwszych ultra szybkich układów zbudowanych z GaAs, technologiczne problemy tego rodzaju są jeszcze przed nami. Ciągle też CRAY zużywa ca 180kW !.

(Mieliśmy rzeczywiście sporo szczęścia, że w obu powszechnie stosowanych technikach wytwarzania "chipów": MOS i bipolarnej zmniejszenie liniowych wymiarów układu o współczynnik L prowadzi do zmniejszenia charakterystycznych napięć i natężeń prądu elektrycznego w poszczególnych elementach chipu również o współczynnik L. Moc pobierana przez element chipu maleje więc jak L^2. Ponieważ liczba elementów w chipie rośnie kwadratowo z L, to moc zużywana prze chip pozostaje w zasadzie ta sama. Dlatego też dzisiejsze chipy zawierające ca 500 000 tranzystorów pobierają zaledwie 5W!)

Informację generowaną w komputerach, radiostacjach itp trzeba w jakiś sposób przesyłać i odczytywać. Każdy przekaz informacji związany jest z przekazywaniem obiektów materialnych lub promieniowania i w związku z tym towarzyszy mu pewien przekaz energii. Czy można go zredukować do minimum poprzez jakieś inowacje technologiczne?. Pytanie to, zbliżone do zagadnienia $k_B T \log(2)$, o którym mówiłem poprzednio, było analizowane już przez Shannona[26]. Zawarta w jego pracach odpowiedź jest negatywna. Otrzymane przez Shannona ograniczenie na przekaz informacji przez kanał łączności z szumem zależy od stosunku sygnału do szumu i od zakresu widmowego kanału. Pytanie, które można sobie postawić to, czy ograniczenia sformułowane przez Shannona są ogólne, czy też nie wynikają one z podstawowych ograniczeń nałożonych na przbiegi zjawisk fizycznych przez prawa mechaniki kwantowej i teorii względności. Postaram się odpowiedzieć na to pytanie twierdząco. Ograniczenia na energetyczny koszt przekazu informacji, a także na maksymalną szybkość przetwarzania informacji wynikają z podstawowych praw przyrody i są niezależne od technologicznych inowacji.

Hans Bremmermann[27] był jednym z pierwszych fizyków, którzy zwrócili uwagę na to, że podstawowe prawa fizyki ograniczają w sposób istotny charakterystyki procesów przekazywania informacji. Oryginalne rozumowanie Bremermanna oparte na wykorzystaniu zasady nieoznaczoności Heisenberga jest nie prawidłowe. Podobne wyniki można jednak osiągnąć na innej drodze. Moje rozważania zawarte poniżej opieram na pracy Jacoba Bekensteina[28].

Steven Hawking[29] zauważył, przy okazji analizy zjawisk towarzyszących tworzeniu się czarnych dziur, że istnieje uogólnione sformułowanie II zasady termodynamiki, prowadzące do związku pomiędzy energia spoczynkową układu fizycznego i jego

---

[26] C. N. Shannon i W. Weaver The Mathematical Theory of Communication. Univ. of Illinois Press, Urbana 1949, d. Middleton An IOntroduction to Statistical Communication. McGraw-Hill, New York 1960.

[27] H. J. Bremermann *Limitations on data processing arising from quantum theory*. w Self-organizing Systems. M. C. Yovits, G. T. Jacobi, i G. D. Goldstein ed. Spartan Books, Washington D. C. (1962)

[28] J. D. Bekenstein *Phys. Rev. Lett* **46**, 623, (1981)

[29] S. W. Hawking Commun. Math. Phys. **43**, 199, (1975)

entropią. Zgodnie z tą zasadą stosunek entropii układu S do (średniej) energii wewnętrznej układu E_o, (S/E_o) nie może być większy niż $2\pi k_B R/\hbar c$, gdzie k_B jest stałą Boltzmanna, $\hbar$ stała Plancka i c prędkością światła.. Wielkość R, dla potrzeb naszych rozważań, można utożsamić z charakterystycznym wymiarem liniowym układu, np. z promieniem kuli opisanej na tym układzie. W stanie równowagi termodynamicznej entropia układu S osiąga wartość maksymalną; jeżeli wszytkie stany wewnętrzne układu wykorzystać w charakterze symboli to w układzie naszym dało by się zakodować $I = S/k_B \ln 2$ bitów. Tak więc w układzie o energii wewnętrznej E_o i promieniu R można zakodować nie więcej niż $I_{max} = 2\pi E_o R/\hbar c \ln 2$ bitów. Dla typowego kryształu o wymiarach liniowych 1cm i w temperaturze $T \sim 100K$ (energia wewnętrzna ciała stałego dla niezbyt wysokich temperetur jest proporcjonalna do T^4) $I_{max} \sim 10^{27}$ bitów. W porównaniu z tą liczbą, pojemności laserowych dysków tzw CD ROM' ów nie wyglądają już tak imponująco.

Z ograniczenia pojemności informatycznej I_{max} dowolnego układu fizycznego wynika ograniczenie na szybkość przekazu informacji $\mathcal{J}$, zdefiniowaną jako iloraz ilości informacji I przekazanej do układu przez czas τ potrzebny na jej zarejestrowanie. Wyobraźmy sobie, że wysłaliśmy paczkę informacji (paczkę fal elektromagnetycznych, impuls światła laserowego, wiązkę elektronów lub podobne), i żepaczka ta porusza się z prędkością nie większą niż c. Wyobraźmy sobie też obserwatora-odbiornik poruszającego się względem nas z prędkością v. Odbiornik nie może zarejestrować więcej bitów, niż wynosi jego pojemność informatyczna Tak więc I zarejestrowane nie może być większe niż $(2\pi ER/\hbar c \ln 2)/\gamma$, gdzie $\gamma = (1-v^2/c^2)^{-1/2}$ a E jest energią paczki mierzoną w układzie odbiornika. Szybkość przekazu informacji, $\mathcal{J}$, jest równa I/τ, przy czym τ jest czasem, jaki upływa pomiędzy dotarciem do odbiornika "początku" paczki informacyjnej, a zarejestrowaniem przez odbiornik ostatniego bitu sygnału. Czas $\tau \geq max\,(t_1, t_2)$ gdzie t_1 jest czasem potrzebnym na to, aby cała paczka została zarejestrowana "na raz", a $2t_2$ - to czas jaki potrzebuje światło na pokonanie najdłuższego wymiaru liniowego układu prostopadłego do prędkości ruchu. Z kinematyki szczególnej teorii względności wynika, że $\tau > 2R(v^2\gamma^2+8c^2)^{-1/2}$. Tak więc dla bardzo dużych lub bardzo małych prędkości v, czas $\tau > 2R/\gamma c$. Szybkość przkazu informacji $\mathcal{J}$ jest więc ograniczona od

góry : $\mathcal{I}_{max} = \pi E/\hbar \ln 2$. Oszacowanie to jest analogiczne do wyniku uzyskanego przez Bremermanna, ale jego stosowalność jest znacznie większa. Np. nasze oszacowanie stosuje się do hipotetycznego przekazywania informacji za pomocą pola neutrinowego.

Oszacowanie Bekensteina wielkości $\mathcal{I}_{max}$ wygodnie jest zapisać jako $E/I > \hbar \ln 2/\pi\tau$. Co wyraża koszt energetyczny na bit zarejestrowany w czasie τ.

Z istnienia tego ograniczenia wynika ważna konkluzja co do maksymalnej technicznie dostępnej ilości operacji na sekundę wykonywanych przez "fizycznie" realizowalny komputer.

Wyobraźmy sobie chip o charakterystycznym wymiarze liniowym L i niechaj ilość operacji na sekundę wykonywana przez ten chip wynosi μ. Oczywiście μ nie może być większe niż c/L. CRAY 2 wyposażony jest w nanosekundowy zegar, stąd wymiary liniowe komputera nie mogą być większe niż ca 2m). Oszacowanie to jest jednak nie przydatne, trudno bowiem powiedzieć, jakie są minimalne wymiary liniowe chipu. Ile atomów potrzeba na to, aby skonstruować chip ? $10^{5?}$.

Ażeby oszacować μ posłużę się wzorem Bekensteina na $\mathcal{I}_{max}$. Zakładam, że typowa operacja, jaką ma wykonać nasz chip to przesyłanie na odległość rzędu L dwu dziewięciocyfrowych liczb. Oznacza to, że typowy przekaz informacji w chipie wynosi 60μ [bitów/sec]. Zgodnie z oszacowaniem Bekensteina kosztować to musi $1.4 \times 10^{-26}\mu$ [ergów/operację], ponieważ czas τ musi być krótszy niż μ^{-1}, w przeciwnym wypadku, bowiem mielibyśmy kłopoty z przyczynowością. Tak więc moc tracona przez nasz procesor musi być nie mniejsza niż $M_d \sim 1.4 \times 10^{-26}\mu^2$[ergów/sec]. Dla procesora pracującego z $\mu \sim 10^{13}$ moc ta będzie porównywalna z mocą traconą w procesorze z "innych" powodów.

Cała ta stracona energia, która jest niezależna od innych strat "technologicznych" występujących w naszym chipie i które można zwalczać na innej drodze, musi być w jakiś sposób z niego odprowadzona. Dla małych układów chłodzenie poprzez wymuszony obieg płynu *wewnątrz* chipu jest technicznie niemożliwe. Pozostają dwie metody chłodzenia zewnętrznego, przez wypromieniowanie i przez transport ciepła na zewnątrz. Chłodzenie przez wypromieniowanie jest najbardziej efektywnym w obszarze widzialnym widma elektromagnetycznego. Chłodzenie przez promieniowanie preferuje "kulistą" konstrukcję urządzenia.

Zakładając, że procesor o objętości L^3 ($\sim \mu^{-3}$) składa się z 10^{23} cząstek oraz przyjmując typowe wartości współczynników emisji Einsteina i energii kwantów dla światła widzialnego (odpowiednio $10^8 sec^{-1}$ i 10^{-12} ergów) stwierdzamy, że maksymalna moc "chłodzenia" przez promieniowanie M_c jest rzędu $10^{30}\mu^{-3}$ [erg/sec].

Chłodzenie przez przewodnictwo ciepła na zewnątrz układu faworyzuje geometrię cienkiej płytki. Oba możliwe mechanizmy przewodnictwa ciepła, tj. przewodnictwo przez drgania sieci krystalicznej (fonony) i przez nośniki ładunku elektrycznego (elektrony i dziury) dają podobne oszacowanie na maksymalną M_c rzędu $10^{35}\mu^{-2}$ [erg/sec]. Porównując M_d z M_c dostajemy, z dokładnością do współczynników rzędu jedności, takie samo oszacowanie na μ_{max}, a mianowicie $\mu_{max} \sim 10^{45}$ [1/sec].

Każdy procesor pracujący z $\mu \geq 10^{45}$ [1/sec] ulegnie zniszczeniu cieplnemu *niezależnie od usprawnień technologicznych i pomysłowości konstruktorów* !

Oczywiście jesteśmy daleko od konieczności przejmowania się istnieniem μ_{max}. Jest jednak rzeczą pouczającą, aby zdawać sobie sprawę z tego, że takie ograniczenie wogóle istnieje. Sądzę również, że fakt iż wynika ono z *ogólnej teorii względności* jest też interesujący.

§6.

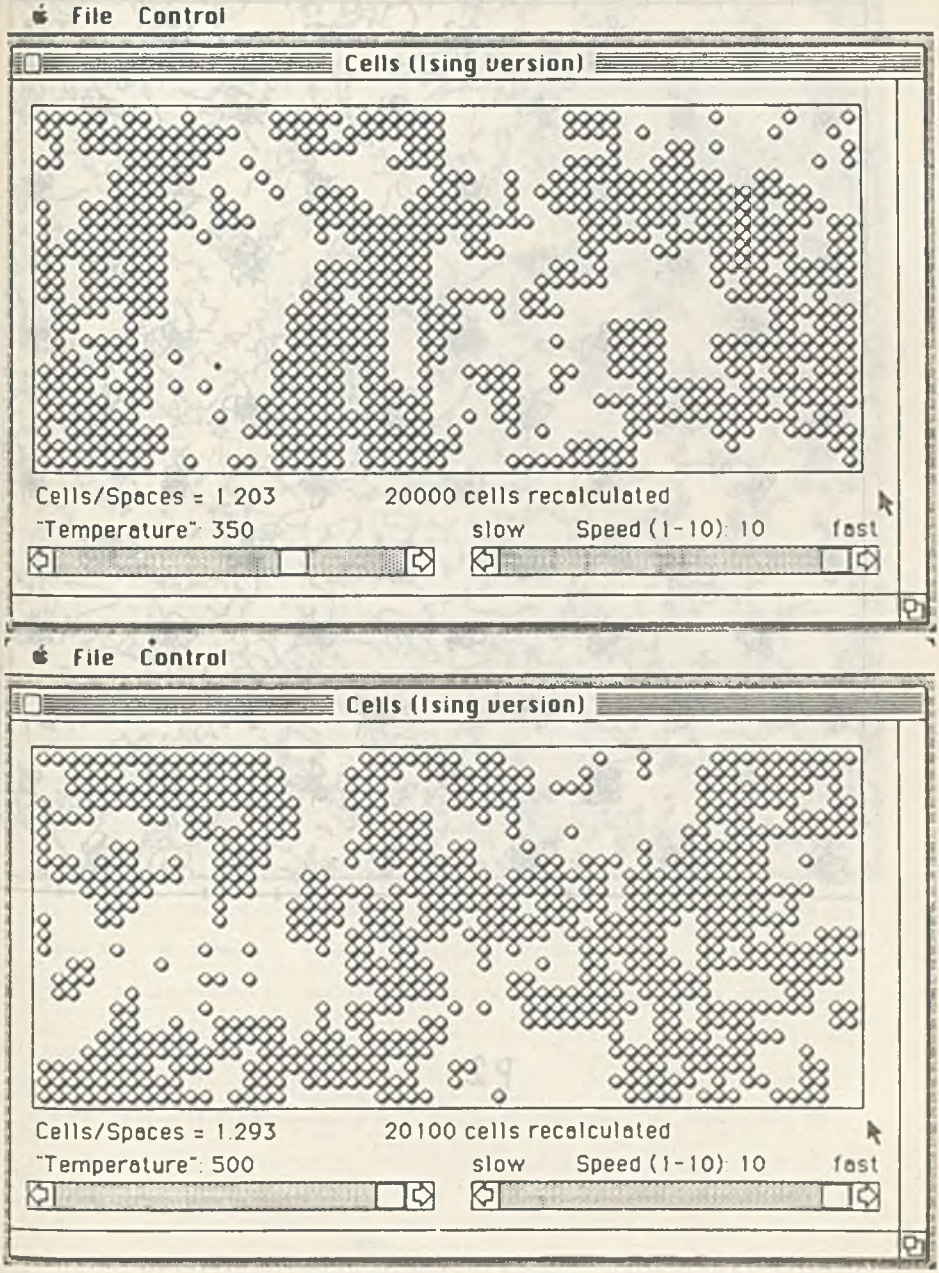
Tych z Państwa, którzy wytrzymali ze mną do końca tych wykładów powinienem przeprosić za to, że przedstawiłem tu wysoce osobiste spojrzenie na zastosowanie komputerów w fizyce i że moje uwagi o problemach fizycznych napotykanych w nauce o komputerach są raczej dalekie od dnia codziennego. Chciałbym zakończyć moje wykłady kilku uwagami dotyczącymi problemu edukacji informatycznej fizyków.

Muszą Państwo zdawać sobie sprawę z tego, że fizycy są szalenie niechętnie nastawieni do korzystania z "usług" innych nauk. Nasze stosunki z matematyką są tego przykładem. Fizycy wielokrotnie musieli tworzyć sobie całe działy matematyki i czekać długie lata na to nim, koledzy matematycy zdecydowali się na opracowanie tychże metod i ich ulepszenie. Typowym przykładem jest tu teoria dystrybucji. Dirac wymyślił "funkcję" delta w końcu lat dwudziestych. Gdy pojawiły się komputery, a szczególnie teraz, gdy nastąpiło lawinowe zastosowanie komputerów, fizycy

mają tendencje do robienia wszystkiego sami. Sądzę, że jest to niepotrzebna strata czasu. Moglibyśmy uzyskać znacznie szybciej wyniki, gdybyśmy mogli współpracować z kolegami informatykami czy też numerykami. Niestety, nie mogę całej winy złożyć na nasze głowy. Fizycy chętnie uczą się informatyki czy metod numerycznych. Z nauczaniem fizyki informatyków jest, oględnie mówiąc, gorzej. Fizycy nadal będą pracować nad nowymi materiałami i nad nowymi zjawiskami, które będą służyć ulepszaniu i radykalnemu zmienianiu się sprzętu komputerowego. Równie ważną, o ile nie ważniejszą sprawą jest jednak współpraca w dziedzinie *software'u*. Problemy wektoryzacji algorytmów numerycznych wykorzystywanych w symulacjach, zagadnienie równoległego przetwarzania danych i związane z tym możliwości ulepszenia algorytmów stosowanych w symulowaniu zagadnień hydrodynamicznych, a w końcu zagadnienia symulacji kwantowo-mechanicznych, to tylko przykłady problemów, w których współpraca pomiędzy specjalistami z obu dziedzin, tj. fizyki i informatyki, powinna przynieść rezultaty.

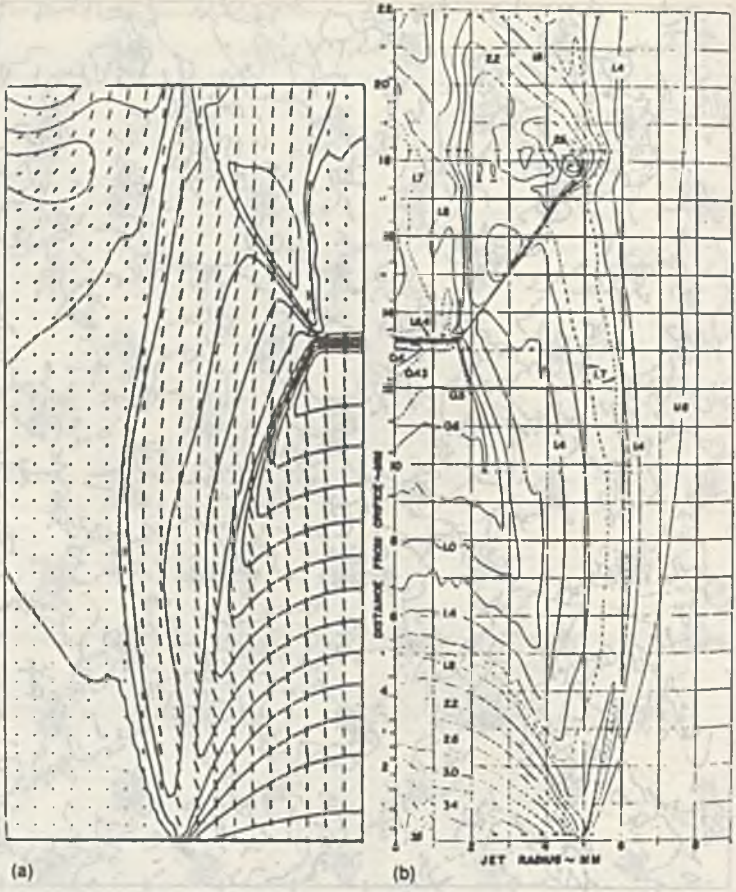
Moje wykłady na tej, wspaniale zorganizowanej przez Polskie Towarzystwo Informatyczne szkole, były próbą przekonania Państwa o konieczności takiej współpracy. Mam nadzieję, że przynajmniej nie odstraszyłem Państwa od niej.

Chciałbym podziękować Dr Danucie Śledziewskiej za przeczytanie tego tekstu i liczne, słuszne uwagi.


File Control
Cells (Ising version)
Cells/Spaces = 1 203
20000 cells recalculated
"Temperature": 350
slow
Speed (1-10): 10
fast
File Control
Cells (Ising version)
Cells/Spaces = 1.293
20100 cells recalculated
"Temperature": 500
slow
Speed (1-10): 10
fast

P I

P2



P3

Jesienna Szkoła PTI

Mragowo, grudzień 1987

KANONICZNY KROK PROCESU PROGRAMOWANIA

Władysław M. Turski

Instytut Informatyki

Uniwersytetu Warszawskiego

PKiN pok. 850, 00-901 Warszawa

W większości znanych metod budowy oprogramowania wyróżnia się kilka odrębnych faz jego projektowania i realizacji. Każda z wyróżnionych faz ma do spełnienia specyficzne zadanie (np. ustalenie wymagań funkcjonalnych czy określenie przepływu danych między modułami projektowanego oprogramowania). Z zadaniem realizowanym w danej fazie wiąże się wyspecjalizowane metody postępowania, języki opisu i narzędzia pracy, w tym - specyficzne dla danej fazy oprogramowanie narzędziowe. Po dostarczeniu klientowi gotowego oprogramowania użytkowego (i, ewentualnie, po uruchomieniu tego oprogramowania) rozpoczyna się faza pielęgnacji (maintenance), trwająca tym dłużej im bardziej udany jest dostarczony produkt programistyczny. Analiza kosztów poszczególnych faz cyklu życia takiego produktu (od wstępnego projektu do ostatecznego zaniechania użytkowania gotowego wyrobu) wskazuje, że ponad 3/4 wydatków pochłania faza pielęgnacji. W dużej mierze wynika to z całkowitej prawie nietechnologiczności tej fazy, dla której nie tworzy się matematycznie uzasadnionych metod postępowania, specjalnych języków opisu, ani narzędzi pracy.

Okazuje sie jednak, że wszystkie znane fazy projektowania, realizacji i pielegnacji oprogramowania można wyrazić jako proste kombinacje ciagów kroków kanonicznych i elementarnej operacji wycofywania się [1]. Przyjecie takiego modelu cyklu życia oprogramowania pozwala skupić wysiłek badawczy i konstrukcyjny na metodach postepowania, jezykach opisu i narzędziach pracy przydatnych w kroku kanonicznym; uzyskane wyniki (teoretyczne i praktyczne) znajduja bezpośrednie i natychmiastowe zastosowanie w *całym cyklu życia oprogramowania*. Cena stosowania metod opartych na koncepcji kroków kanonicznych jest wydatne zwiększenie ilości prac pomocniczych (np. dokumentacyjnych), która rośnie co najmniej liniowo z liczbą kroków, a przy stosowaniu metody kroków kanonicznych liczba wykonywanych kroków jednolitego procesu projektowania-realizacji-pielegnacji oprogramowania jest zazwyczaj bardzo wielka. Dlatego też stosowanie takich metod wymaga bardzo sprawnego wspomagania technicznego. W praktyce oznacza to konieczność stosowania tzw. zespolonych środowisk programistycznych (IPSE).

Przedstawienie metody kroków kanonicznych wymaga ustalenia poglądu na to, czym właściwie jest program i jego specyfikacja, oraz na czym polega proces tworzenia programu użytkowego i jego dalsza ewolucja w okresie stosowania. Niezbedne informacje wstępne przedstawimy z konieczności skrótowo, odsyłajac czytelników do pracy [2] i książki [3] po wiecej szczegółów.

Z czysto pragmatycznego punktu widzenia przyjmujemy, że program użytkowy powstaje po to, aby przy jego pomocy można było rozwiązywać pewne zadania odnoszace sie do jakiegoś aspektu realnego świata. Opis zadania musi więc nawiązywać do realiów innych niż komputerowe (bywaja oczywiście wyjatki, ale zdarzaja sie one stosunkowo rzadko: np. wtedy gdy zadanie dotyczy organizacji obliczeń lub prezentacji wyników). Zbudowany już program komputerowy jest tworem czysto formalnym, którego sens wynika z przyjętej interpretacji składajacych się nań symboli i ich kombinacji (to, w jaki sposób przypisuje się ów sens programowi, zależy od wyboru semantyki: np. operacyjnej czy denotacyjnej, nie ma to jednak wpływu na generalny charakter związku programu z jego znaczeniem).

Mamy więc do czynienia z taką sytuacją: z jednej strony realia wybranych aspektów pewnej dziedziny świata rzeczywistego, odzwierciedlone w opisie zadania, z drugiej zaś - komputerowy program i jego znaczenie, twory z natury rzeczy formalne. (Nie wdając się w niepotrzebną tu dyskusję o wyższości tych czy innych rodzajów semantyki, warto zuważyć, że wyniki wykonania programu na sprawnym komputerze wynikają ściśle kalkulacyjnie - a więc w sposób formalny! - ze znaczenia programu.)

Informatyczne rozwiązanie problemu prowadzi przeto od dziedzin realnych do formalnych; w praktycznym postępowaniu styk tych dziedzin jest nieunikniony. Z wielu względów wygodnie jest przyjąć, że styk ten następuje w specyfikacji, która traktujemy jako teorię o dwu modelach: w świecie rzeczywistym i w świecie programów. Różnice między nimi mogą być pokaźne. Jako teoria świata realnego, specyfikacja podlega rygorom nauk przyrodniczych, m.in. jej słuszność podlega sprawdzeniu doświadczalnemu. Jako teoria programu, specyfikacja podlega rygorom matematycznym; w szczególności, związek poprawności programu względem specyfikacji podlega dowodowi. Jako teoria świata rzeczywistego, specyfikacja jest jego abstrakcją. Jako teoria programu, specyfikacja jest przepisem charakteryzującym jego zasadnicze cechy. Wynika stąd dualny, deskryptywny i preskryptywny (normatywny), charakter specyfikacji. (W pracy zbiorowej [4] można znaleźć wiele przykładów ilustrujących bądź jeden, bądź drugi aspekt specyfikacji; łączne traktowanie obydwu pochodzi z pracy [5], por. także łatwiej dostępny raport [6].)

Jak dotąd, związek specyfikacji ze światem rzeczywistym nie zyskał powszechnego uznania jako pełnoprawny przedmiot zainteresowania informatyki. Mówi się często, że praca informatyka zaczyna się *po ustaleniu formalnej specyfikacji zadania*. Doświadczenia poważnych firm żyjących z wytwórstwa oprogramowania użytkowego, zwłaszcza zaś oprogramowania dla jednostkowych klientów, wskazują jednak na fundamentalne znaczenie jakości tego związku. Truizmem jest bowiem stwierdzenie, że koszt korygowania specyfikacji funkcjonującego oprogramowania wyraźnie przekracza koszty poprawiania wszystkich innych błędów w nim dostrzeżonych.

Jeśli więc ostatecznym celem przedsięwzięcia programistycznego jest zaspokojenie potrzeb klienta, co można osiągnąć tylko dzięki właściwemu funkcjonowaniu systemu informatycznego w realnym środowisku, efektywność pracy informatyka zależy od umiejętności sprawdzenia (i ewentualnego skorygowania) specyfikacji *zanim zostanie zbudowane poprawne względem niej oprogramowanie.*

Czysto formalne (kalkulacyjne) metody sprawdzania specyfikacji, aczkolwiek wielce przydatne np. przy badaniu jej niesprzeczności [7] - są z natury rzeczy bezsilne wobec pytania o jej słuszność względem realiów, w których będzie ostatecznie funkcjonować zrealizowany system. W ostatnich latach rozwijają się dwa kierunki badawcze, mające na celu utechnologicznienie procesu sprawdzania specyfikacji; określa się je hasłami:

1. wykonywalnych specyfikacji (*executable specification*),
2. szybkiego makietowania (*rapid prototyping*).

Ponieważ ta (skądinąd pasjonująca) tematyka nie mieści się w ramach zakreślonych dla niniejszego referatu, ograniczymy się do szkicowego streszczenia głównych idei tych dwu kierunków, nie zawsze zresztą ściśle rozgraniczanych [8].

Idea wykonywalnych specyfikacji polega na tym, że pisze się je w specjalnym języku, dla którego istnieje algorytm interpretacyjny, czyli dla którego można skonstruować program generujący następstwa (w obowiązującym dla danego języka systemie dedukcyjnym) z napisanej specyfikacji. Mając do dyspozycji taki program, możemy postępować na dwa sposoby: generować ciąg następstw (twierdzeń wypływających ze specyfikacji) i sprawdzać, czy zachodzą one w rzeczywistości, albo formułować twierdzenia opisujące interesujące nas zjawiska zachodzące w rzeczywistości i próbować udowodnić te twierdzenia w teorii wyznaczonej przez specyfikację. Zależnie od wybranego sposobu postępowania powinniśmy nieco zmieniać algorytm interpretacyjny; w jednym przypadku mamy bowiem do czynienia z *generatorem*, a w drugim - z *weryfikatorem* twierdzeń. I w jednym, i w drugim przypadku czysto kalkulacyjne algorytmy są zapewne zbyt mało sprawne

(nieskończoność zbioru następstw i jego nieciekawa morfologia stanowią główne przeszkody czysto mechanicznego postępowania); prawie na pewno nie da się tu uniknąć stosowania reguł heurystycznych. Ponadto, bezpośrednia postać konsekwencji specyfikacji może być mało czytelna dla klienta, co zmusza do ich przekładu na inny język, np. graficzny, bardziej przystępny dla użytkownika, dla którego projektujemy system i który *wie* czy dane zjawisko występuje, czy nie, w interesującej go dziedzinie.

Idea szybkiego makietowania polega na tym, żeby ze specyfikacji możliwie szybko wyprowadzić poprawny choć być może niezbyt sprawny program. Eksperymentalna eksploatacja tego programu pozwala sprawdzić słuszność specyfikacji. Aby w pełni docenić sens takiego postępowania, należy zauważyć, że różnica kosztów między makietą (tj. naprędce skonstruowanym programem poprawnym) a ostateczną, *handlową* wersją programu może być bardzo pokaźna. Tworząc makietę można posługiwać się różnymi protezami, modułami wziętymi z bibliotek gotowych, uniwersalnych (a przeto nieoptymalnych) rozwiązań. Przeznaczając makietę do próbnej eksploatacji można osłabić wymagania odnośnie do dokumentacji użytkowej i konstrukcyjnej, a przede wszystkim - nie liczyć się z koniecznością jakiejkolwiek modyfikacji, gdyż makietę, która pomyślnie przeszła próby sprawdzające *specyfikację* i tak zaraz się wyrzuca, a w przypadku wykrycia błędów (specyfikacji!) - tworzy się następną.

Niezależnie od tego, czy i jakimi metodami sprawdzenia słuszności specyfikacji będziemy się posługiwać, przyjmujemy, że

1. specyfikacja jest teorią żądanego rozwiązania informatycznego,
2. sprawdzenie słuszności specyfikacji jest zabiegiem, którego wykonanie dopuszcza inne niż czysto formalne (kalkulacyjne) metody postępowania i kryteria oceny.

Ponadto, zgodnie z ustalonym już poglądem nowoczesnej inżynierii oprogramowania, przyjmujemy, że jedynym kryterium oceny *poprawności* programu względem specyfika-

cji jest matematycznie ścisły dowód spełnienia ustalonego związku formalnego. (Nie oznacza to oczywiście, że nalegamy na aposterioryczne dowodzenie poprawności oprogramowania; przeciwnie, postępowanie takie uważamy za ostatnią deskę ratunku. Prawidłowy przebieg procesu budowy oprogramowania powinien przebiegać tak, aby poprawność oprogramowania można było zagwarantować *na mocy konstrukcji*.)

Niech S będzie specyfikacją programu. Aby S można było zaprezentować, musimy dysponować pewnym językiem J0 i logiką L0.

Reguły J0 określają składnię zdań, którymi wyrażamy specyfikację. Pomijamy tu całą masę szczegółów technicznych, dotyczących składniowego wyróżnienia poszczególnych kategorii językowych, alfabetu, nazewnictwa zmiennych, funktorów, sygnatury, a także ewentualnej wielotypowości języka. W każdym konkretnym przypadku języka użytego do pisania specyfikacji szczegóły tego rodzaju muszą być oczywiście ustalone, ale nie wydaje się rzeczą rozsądną preferowanie jakiegoś jednego rozwiązania, tak samo jak nie jest rzeczą rozsądną forsowanie jakiegoś jednego języka programowania.

Reguły L0 składają się z dwu części: *reguł wnioskowania* i *aksjomatów logicznych*. Reguły wnioskowania mają postać

jeśli zdania Z0, Z1, ..., Zn są twierdzeniami to zdanie Z też jest twierdzeniem

Dobrze znanym przykładem reguły wnioskowania jest tzw. reguła odrywania (*modus ponens*)

jeśli A i A => B są twierdzeniami to B też jest twierdzeniem

(Reguły wnioskowania zapisuje się często w postaci ułamka, którego licznik stanowią przesłanki, a mianownik - wniosek, np. $\frac{A,\ A \Rightarrow B}{B}$)

Aksjomaty logiczne są zbiorem zdań, których prawdziwość postulujemy dla wszystkich dziedzin stosowania wybranej logiki. Inaczej powiedziawszy, wybór aksjomatów logicznych ogranicza (na ogół bardzo liberalnie!) zbiór dziedzin, w których możemy bezpiecznie posługiwać się LO. Typowym przykładem aksjomatu logicznego jest np. zasada wyłączonego środka (*tertium non datur*), głosząca

z dwu zdań Z i nonZ dokładnie jedno jest prawdziwe.

Aksjomaty logiczne i reguły wnioskowania wybiera się tak, by posługując się językiem JO można było opisywać problemy powstające w interesujących nas dziedzinach. Wybór LO charakteryzuje nasz pogląd na to, jakiego rodzaju wnioskowania są typowe (przydatne) w pewnej klasie problemów.

Ze szkoły średniej pamiętamy logikę Arystotelesowską, nie jest ona jednak jedyną logiką używaną we współczesnej informatyce. Tytułem przykładu wymienimy kilka innych logik, jakimi posługują się informatycy: logika klauzul Horna (podstawa Prologu) jest dość istotnym ograniczeniem Arystotelesowskiej; różne logiki modalne, zwłaszcza zaś tzw. logiki temporalne, służą do opisu zjawisk zachodzących w obliczeniach współbieżnych; logiki niemonotoniczne są stosowane w komputerowych modelach wypowiedzi w języku etnicznym.

Opis konkretnego problemu - specyfikację - wyrażamy formułując *aksjomaty specyficzne (pozalogiczne)*. Podobnie do aksjomatów logicznych, są one zdaniami, których prawdziwość jest postulowana; w przeciwieństwie do aksjomatów logicznych, aksjomaty specyficzne wyodrębniają stosunkowo wąską klasę dziedzin, w których są spełnione. Weźmy przykład:

dla dowolnych dwu różnych elementów, x i y, istnieje element z, taki że

$x < z < y$

Jak łatwo widzieć, aksjomat ten jest spełniony w dziedzinie liczb rzeczywistych, ale nie w dziedzinie liczb całkowitych (przyjmując naturalną interpretację symbolu <).

Zgodnie z naszym poglądem na nature specyfikacji, przyjmujemy, że jest ona teoria, czyli zbiorem wszystkich następstw aksjomatów pozalogicznych, tj. wszystkich syntaktycznie poprawnych zdań języka J0, które w logice L0 można wyprowadzić z tych aksjomatów . Zbiór następstw jest zazwyczaj nieskończony, dlatego też skupiamy naszą uwagę na samych aksjomatach pozalogicznych, charakteryzujących teorię w znacznie zwięźlejszy sposób. Zauważmy, że pytania o niesprzeczność i zupełność tak pojmowanej specyfikacji nabierają dobrze określonego znaczenia.

Specyfikacja jest niesprzeczna jeśli w języku J0 nie ma takiego zdania Z, że zarówno Z, jak i nonZ jest następstwem ustalonych aksjomatów pozalogicznych. Sprzeczne specyfikacje nie mają, naturalnie, żadnego sensu praktycznego. Specyfikacja jest zupełna, jeśli każde poprawne zdanie języka J0, Z, lub jego zaprzeczenie nonZ należy do teorii przyjętej za specyfikacje. Przypomnijmy, że każda sprzeczna teoria jest w tym sensie zupełna; to elementarne spostrzeżenie powinno być przestrogą wobec przywiązywania nadmiernej wagi do zupełności specyfikacji. Nieco dalej wskażemy inne, bardzo praktyczne obiekcje wobec dążenia do formułowania zupełnych specyfikacji.

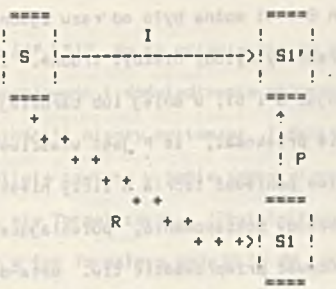
Zadanie programisty (zespołu programistów) polega na zrealizowaniu specyfikacji S w języku J1 z logika L1. W przyjętej konwencji oznacza to, że poszukujemy takiego poprawnie skonstruowanego *rozszerzenia* P systemu S1 = (J1,L1) na system S1', żeby istniała dobrze określona interpretacja I: S --> P spełniająca następujący warunek:

jeśli Z jest następstwem S w systemie (J0,L0),
to I(Z) jest następstwem P w systemie (J1,L1). (W)

W powyższym sformułowaniu warunku poprawności P względem S uczyniliśmy kilka uproszczeń technicznych; pełne sformułowanie można znaleźć w książce [3], do której odsyłamy czytelników zainteresowanych szczegółami (stanowiącymi o matematycznej poprawności przyjętych konwencji).

Przez *następstwo* rozumiemy tutaj *wynikanie zgodne z regułami wnioskowania*; oznacza to, że jeśli logiki L0 i L1 różnią się nie tylko składniowo (np. pod względem użytych symboli zmiennych i funktorów), lecz również co do reguł wnioskowania, relacje następstwa w obu rozważanych systemach mogą być nader różne. W codziennej praktyce programowania różnice między logikami L0 i L1 są zazwyczaj czysto techniczne, wynikają z konieczności uwzględnienia odmiennych symboli zmiennych i funktorów oraz z tego, że rozszerzenie P wprowadza dodatkowe symbole. Czytelnicy, którzy chcą sobie uzmysłowić praktyczny sens rozszerzenia P mogą - bez obawy popełnienia grubej pomyłki - wyobrazić sobie P jako program w języku J1, czyli syntaktycznie poprawny zbiór deklaracji typów, zmiennych i procedur.

Schematycznie, zadanie programistyczne możemy przedstawić tak:



Realizacja R specyfikacji S w S1 = (J1,L1) jest wynikiem rozszerzenia P i interpretacji I.

Jest rzeczą oczywistą, że przy zadanej specyfikacji S i systemie lingwistycznym (J1,L1) możliwe jest wiele różnych rozszerzeń P realizujących S w S1.

Jeśli np. specyfikacja opisuje problem sortowania ciągów liczbowych, a S1 jest poziomem lingwistycznym, powiedzmy, Pascala, to zarówno program sortowania

naiwnego, P1, jak i program sortowania dystrybucyjnego, P2, dopełniaja realizacji S w S1. Programów takich, różniacych sie miedzy soba pod istotnymi względami (np. zajetości pamięci i czasu wykonywania) może być bardzo wiele. Wszystkie one sa jednak w tej samej mierze podyktowane specyfikacja S i wyborem S1. Ponieważ system S1' powstający z rozszerzenia S1 przez P można traktować jako model teorii S, dochodzimy do wniosku, że w jednej i tej samej dziedzinie S1 teoria S może mieć wiele nie zupełnie podobnych modeli, czego nie da sie pogodzić z kategorycznością (silna forma zupełności) teorii S.

W tym, co powiedzieliśmy dotychczas, dominuja elementy opisowe: używajac troche dziwnej terminologii przedstawiliśmy dobrze znana sytuacje. Czas przejść do konstruktywnego wykorzystania wprowadzonych pojeć i związków.

Gdyby przy zadanych S i S1 można było od razu zgadnać odpowiednie rozszerzenie P, zadanie programisty byłoby niezbyt trudne. Załóżmy, że tak jest w istocie, tj. że dysponujac S i S1, w mniej lub bardziej magiczny sposób możemy otrzymać P. Jak można sie przekonać, że P jest właściwe? Dosłowne sprawdzenie warunku (W) jest niemożliwe ponieważ teoria S liczy nieskończenie wiele zdań. Na szczeście istnieja dwie metody postepowania, pozwalające opanować ten problem. Po pierwsze, możemy spróbować przeprowadzić tzw. meta-dowód, tj. dowód pewnej wypowiedzi o wszystkich twierdzeniach teoryj S i S1'; na ogół nie jest to jednak łatwe. Po drugie zaś, jeśli zadbamy by interpretacja I spełniała pewne dość naturalne warunki techniczne (szczegóły opisano w książce [3]) to możemy sie posłużyć ogólnym wynikiem, głoszacym:

> jeśli aksjomaty pozalogiczne teorii S sa w interpretacji I twierdzeniami teorii S1' to wszystie zdania tej teorii w interpretacji I też sa twierdzeniami S1'.

Wystarczy więc przeprowadzić *skończona* liczbe dowodów by przekonać sie o spełnieniu warunku (W), a tym samym o poprawności wskazanej realizacji specyfikacji S na poziomie jezykowym (J1,L1).

Nie jest to jeszcze wszystko, co można uzyskać z przyjętego wzorca postępowania programistycznego, ale nie jest to zły początek: badanie poprawności realizacji sprowadziliśmy do dobrze określonego zadania o czysto kalkulacyjnym charakterze! (Podkreślmy całkowitą zgodność z ideologią wyłożoną na początku: badanie słuszności specyfikacji może wymagać postępowania przyrodniczego, badanie poprawności jej realizacji jest procesem czysto formalnym!)

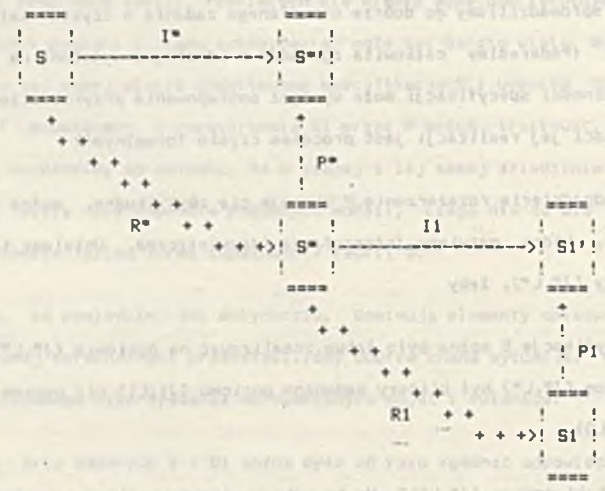
Jeśli odgadnięcie rozszerzenia P okazuje się zbyt trudne, można zastosować postępowanie, które nazwiemy *interpolacją lingwistyczną*. Ustalamy taki poziom lingwistyczny (J^*,L^*), żeby

1. specyfikację S można było łatwo zrealizować na poziomie (J^*,L^*).
2. poziom (J^*,L^*) był bliższy zadanego poziomu (Ji,Li) niż poziom startowy $(J0,L0)$.

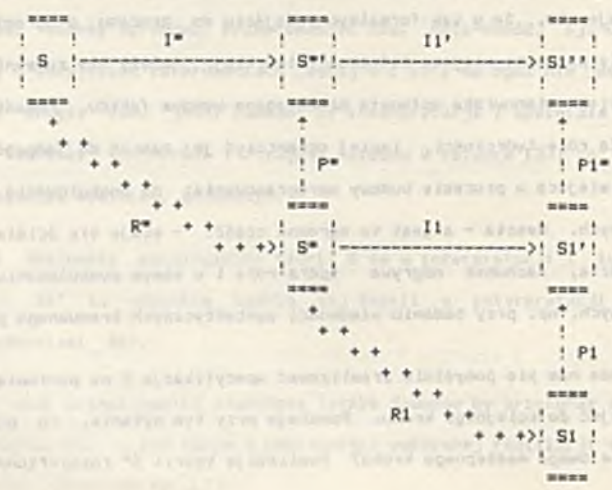
Jak znaleźć taki poziom (J^*,L^*)? Na to pytanie nie ma ogólnej odpowiedzi! Wybór tego poziomu dyktuje pomysłowość i doświadczenie programisty. One też są jedynymi kryteriami „odległości'' między poziomami lingwistycznymi. Konstruowanie oprogramowania niewątpliwie zawiera w sobie spory element twórczości ludzkiej, która nie prędko podda się formalizacji. (Czytelnikowi, którego martwi albo zgoła szokuje to, że w tak formalnym podejściu do procesu programistycznego przyznaje się poczesne miejsce twórczości ludzkiej, należy się zapewnienie, że na wybór takiego stanowiska wpłynęła nieubłagana wymowa faktów. Ponadto, uznawszy doniosłą rolę twórczości, lepiej ograniczyć jej zakres do jednego, dobrze określonego miejsca w procesie budowy oprogramowania: do postulowania poziomów lingwistycznych. Reszta - a jest to ogromna część! - staje się ściśle kalkulacyjna. Zresztą, rachunek odgrywa spora rolę i w samym postulowaniu poziomów lingwistycznych, np. przy badaniu własności syntaktycznych kreowanego poziomu.)

Jeśli uda nam się pomyślnie zrealizować specyfikację S na poziomie (J^*,L^*), możemy przejść do kolejnego kroku. Powstaje przy tym pytanie, co potraktować jako zadanie owego następnego kroku? Realizację teorii S^* rozszerzonej o pro-

gram P*, czy realizacje samej S*? Zanim odpowiemy na to pytanie, przyjrzyjmy się schematowi



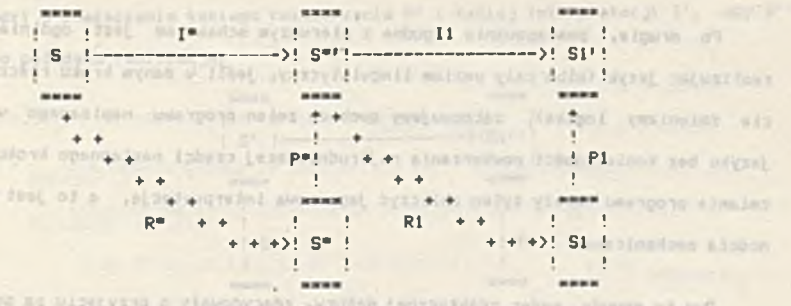
którego dolny „trójkąt'' jest dokładnym powtórzeniem górnego (z pominięciem różnic oznaczeń). Jeśli wariant realizacji specyfikacji S na poziomie (J1,L1) za pomocą tego schematu ma być w ogóle brany pod uwagę, musimy dysponować stosunkowo prostą metodą wypełnienia prawego górnego rogu:



Okazuje sie (niestety, po szczegóły znowu musimy Czytelnika odesłać do [3]), że można sformułować warunki wystarczajace na to, by rozszerzenie P1* było po prostu I1(P*), interpretacja zaś I1' była określona jak następuje: wyrażenia należace do języka J* interpretuje sie w I1' tak samo jak w I1, a wyrażenia należace do roszerzenia P* pozostawia bez zmiany. Realizacje specyfikacji S w systemie S1 wyznacza rozszerzenie P = P1‡P1' i interpretacja I = I*‡I1', gdzie przez ‡ oznaczono złożenie przekształceń. Z praktycznego punktu widzenia, po zrealizowaniu S w S* i S* w S1, realizacje S w S1 otrzymuje sie czysto mechanicznie (algorytmicznie).

Zreferowany powyżej wynik można rozciągnąć na dowolna skończona liczbe kroków: S0 --> S1 --> ... --> Sn, gdzie S0 jest poziomem lingwistycznym początkowej specyfikacji, a Sn - poziomem ostatecznej implementacji programu, przy czym *wszystkie kroki maja identyczna strukture*!

Do dość podobnego wyniku doszlibyśmy stosujac inny schemat:



pozornie prostszy nawet od poprzedniego, odpowiadajacy realizacji S*' (czyli S* poszerzonej przez P*) w S1. Schemat ten wydaje sie niezwykle atrakcyjny: w kolejnym kroku realizujemy nie *język* kroku poprzedniego lecz uzyskany w owym poprzednim kroku *program*, a więc praktycznie nie mamy już nic więcej do zrobienia; nawet czysto mechaniczne przekształcenia nie sa już potrzebne. Wnikliwy Czytelnik zapewne zauważył, że rozważany obecnie schemat dokładnie odpowiada metodzie programowania zstępujacego (analitycznego, *top-down*). Tego też schematu

używano w pracy [10] i, w znacznym stopniu, w pracach [1] i [5]. Podobnie jak rozważany uprzednio, i ten schemat (realizacji pośredniego poziomu językowego rozszerzonego o program) można rozciągnąć na dowolna skończona liczbę kroków. W stosunku do poprzedniego ma on jednak dwie poważne wady.

Po pierwsze, ponieważ dla każdego kroku musimy sprawdzić spełnienie warunku (W), im mniej jest aksjomatów, których interpretacje należy udowodnić na nowym poziomie językowym, tym lepiej. W drugim schemacie postępowania musimy udowodnić nie tylko prawdziwość interpretacji aksjomatów pozalogicznych języka, lecz także wszystkich aksjomatów dodanych przez rozszerzenie (program). Niekiedy nie jest to bardzo poważne zwiększenie zakresu pracy (np. gdy roszerzenie ma charakter czysto definicyjny - wprowadza nowe nazwy czy symbole, całkowicie równoważne kombinacjom wyrażeń z języka podstawowego); jeśli jednak rozszerzenie wprowadza istotnie nowe jakości semantyczne (do czego prędzej czy później musi dojść w procesie konstruowania każdego niebanalnego programu!), to ilość pracy wyraźnie się zwiększa.

Po drugie, postępowanie zgodne z pierwszym schematem jest ogólniejsze: realizując język (albo cały poziom lingwistyczny, jeśli w danym kroku rzeczywiście zmieniamy logikę), zachowujemy swobodę zmian programu napisanego w tym języku bez konieczności powtarzania najtrudniejszej części następnego kroku. Po zmianie programu należy tylko obliczyć jego nową interpretację, a to jest czynnością mechaniczną.

Dwa te powody, nader praktycznej natury, zdecydowały o przyjęciu za postawę metodyki opisanej w [3] schematu pierwszego. Następujący przykład ilustruje zalety tego wyboru.

Przypuśćmy, że rozwiązaliśmy zadanie programistyczne tj. przy pomocy programu P zrealizowaliśmy specyfikację S na poziomie językowym Si = (Ji,Li). Jak to w życiu zwykle bywa, nasz klient zdecydował się jednak rozszerzyć początkowe zadanie, czego wyrazem jest poszerzenie specyfikacji S o dodatkowe wymagania D.

Schematycznie, możemy to przedstawić tak:

Nowe zadanie polega na zrealizowaniu *poszerzonej* specyfikacji S' na poziomie językowym S1. Bez trudu można jednak zauważyć, że jest to zadanie identycznej postaci co rozważane uprzednio zadanie ,,wypełnienia prawego górnego rogu''! Chodzi o znalezienie takiego rozszerzenia P' i takiej interpretacji I', aby S'' było pożądaną realizacją:

Jeśli D spełnia pewne dość naturalne warunki (szczegóły, jak zwykle, w książce [3]), wówczas P' okazuje się po prostu I(D) i nowe zadanie ma banalnie proste rozwiązanie, włączające bez zmiany wyniki uprzednio wykonanej pracy. (Oczywiście, nie wszystkie zmiany S dają się równie łatwo zrealizować!)

Dotychczasowe rozważania były dość abstrakcyjne, zilustrujemy je teraz w miarę konkretnym przykładem.

PRZYKŁAD

Zadanie polega na zrealizowaniu relacji równoważności (w przykładzie wykorzystano wyprowadzenie algorytmu Fishera-Gallera opisane przez Corella [11]).

Na pewnym skończonym zbiorze Z elementów ponumerowanych od 1 do n określono relację równoważności, equiv. E jest zbiorem kolekcji zawierających pary elementów, o których wiadomo, że należą do relacji equiv.

Specyfikacja zadania obejmuje język specyfikacji:

INIT: --> E

ENTER: E X Z X Z --> E

EQUIV: E X Z X Z --> (true, false)

i,j,k,l - zmienne typu Z (ich wartościami są numery elementów)

e - zmienna typu E (wartościami e są *kolekcje* par równoważnych)

i aksjomaty:

EQUIV(INIT,i,j) <=> (i = j) (A1)

EQUIV(ENTER(e,i,j),k,l)
<=> (EQUIV(e,k,l)
v (EQUIV(e,i,k) ^ EQUIV(e,j,l)) (A2)
v (EQUIV(e,i,l) ^ EQUIV(e,j,k)))

Aksjomaty pozalogiczne (A1) i (A2) łacznie z rachunkiem predykatów pierwszego rzędu wyznaczają teorię relacji equiv. Posługując się tymi aksjomatami, można np. udowodnić, że relacja ta jest symetryczna EQUIV(i,j) <=> EQUIV(j,i), zwrotna EQUIV(i,i) = true i przechodnia EQUIV(i,j) ^ EQUIV(j,k) => EQUIV(i,k). Zauważmy, że jeśli explicite nie uznaliśmy żadnych par za równoważne (tj. nie wykonaliśmy operacji ENTER), to jedynymi parami spełniającymi EQUIV(i,j) są pary i = j. Jeśli natomiast konkretna kolekcja zawiera np. pary (1,2) i (2,7) to spełnione są warunki:

EQUIV(2,1) (na mocy symetrii)

EQUIV(1,7) (na mocy przechodniości)

EQUIV(1,2) (na mocy umieszczenia w E)

EQUIV(2,7) (na mocy umieszczenia w E)

Tyle o teorii (specyfikacji) S.

Za nowy język J1 weźmiemy język zawierający INIT i ENTER z J0 oraz nową funkcję:

REP: E X Z --> Z

a za aksjomaty pozalogiczne przyjmiemy:

REP(INIT,i) = i (A11)

REP(ENTER(e,i,j),k)

= if REP(e,j) = REP(e,k) then REP(e,i) (A12)

else REP(e,k)

(Użycie w zapisie aksjomatu (A12) notacji if...then...else nie zmienia charakteru logiki, jest to czysto stenograficzny skrót notacyjny.)

Dla ułatwienia interpretacji S w S1 rozszerzamy język J1 o predykat EQUIV i dodajemy definicję:

EQUIV(e,i,j) <=> (REP(e,i) = REP(e,j)

Interpretacje ustalamy jako tożsamościowe przekształcenie symboli J0 na symbole rozszerzonego J1, tj. I(INIT) = INIT, I(ENTER) = ENTER, I(EQUIV) = EQUIV. Należy teraz sprawdzić spełnienie warunku (W), tj. udowodnić, że (A1) i (A2) - po zinterpretowaniu! - wynikają z (A11) i (A12). Dowód (A1) jest banalnie prosty:

(A1): EQUIV(INIT,i,j) <=> (i = j)

(interpretując): (REP(INIT,i) = REP(INIT,j)) <=> (i = j)

(na mocy (A11)): (i = j) <=> (i = j) QED

Dowód (A2) jest sporo dłuższy, ale i on nie przedstawia żadnych istotnych trudności.

Następny krok polega na przejściu od języka zbiorów (którym posługiwaliśmy się dotąd) do języka drzew, pozwalającego nadać zbiorowi par równoważnych strukturę przydatną do algorytmicznego przetwarzania. Składnię tego języka definiujemy tak:

LINK: E X Z X Z --> E

FATHER: E X Z --> Z

ROOT: E X Z --> (true, false)

INIT: --> E

pozalogiczne aksjomaty mają zaś postać:

ROOT(INIT,i) = true (A21)

ROOT(LINK(e,i,j),j) = false (A22)

ROOT(LINK(e,i,j),k) ^ (j ≠ k) ==> ROOT(e,k) (A23)

FATHER(LINK(e,i,j),k) = if j = k then i else FATHER(e,k) (A24)

Aby wygodnie zinterpretować symbole języka J1 rozszerzamy J2 o definicje:

ENTER(e,i,j) <=> if REP(e,i) = REP(e,j) then e

else LINK(e,REP(e,i),REP(e,j))

```
REP(e,i) <=> [t := i;
              while ROOT(e,t) = false
                 do t:= FATHER(e,t) od;
              result t]
```

W drugiej definicji użyliśmy aparatu wykraczającego poza rachunek predykatów pierwszego rzędu; logika L2 musi być na tyle silna by można było w niej wnioskować o własnościach wyrażeń zawierających konstrukcje programowe. Korzystając z takiej właśnie logiki musimy udowodnić, że w podanej interpretacji języka J1 aksjomaty (A11) i (A12) wynikają z (A21) - (A22). Dowody te można znaleźć w cytowanej pracy [11]. (Na pełna realizacje algorytmu Fishera-Gallera składa się jeszcze jeden krok, w którym drzewa zastępuje się tablicami; nie wnosi on jednak nic nowego do ilustracji metody postępowania.)

(KONIEC PRZYKŁADU)

Wyidealizowany przebieg procesu konstrukcji oprogramowania zgodnie z wyłożoną (w zarysach) metodą można przeto przedstawić tak

```
S0 --> S1 --> ... --> Si --> S(i+1) --> ... --> Sn
       !              !       !                 !
       !P1            !Pi     !P(i+1)           !Pn
       ¥              ¥       ¥                 ¥
       S1'            Si'     S(i+1)'           Sn'
```

gdzie Si, i = 0,1,...,n, są kolejnymi teoriami (specyfikacjami) poziomów językowych, a Pi, i = 1,2,...,n - złożeniami rozszerzeń realizujących S(i-1)' na poziomie Si (S0' = S0).

Warto zwrócić uwagę na dwie sprawy o pierwszorzędnym znaczeniu pragmatycznym: (1) ponieważ wykonanie każdego kroku wymaga przeprowadzenia dowodów spełnienia warunków (W), kroki są tym łatwiejsze, im mniej się zmienia z kroku na krok; (2) natura wszystkich kroków jest taka sama. Można się więc spodziewać, że postępowanie zgodnie z opisaną metodą będzie liczyć bardzo wiele kroków jednakowych co do schematu postępowania i natury zobowiązań dowodowych.

Na tym właśnie polega główna różnica między opisywaną i wszystkimi poprzednio znanymi metodami. Zamiast kilku wyodrębnionych etapów, różniących się co do swego charakteru, występuje tu ciąg wielu jednorodnych kroków. Praktyczny zysk ze stosowania nowej metody polega przede wszystkim na tym, że instrumentarium narzędzi pomocniczych nie musi być zróżnicowane (inne dla każdego etapu). Przyjmując, że jakość i kompletność instrumentarium są proporcjonalne do poniesionych na nie nakładów, bez trudu dostrzegamy wielką szansę lepszego wyposażenia programistów pracujących wedle opisanej metody: przy takich samych nakładach instrumentarium wspomagające wykonywanie kroków kanonicznych będzie tyle razy lepsze ile różnych etapów zostało wyeliminowane. (Lub odwrotnie, podobnej jakości instrumentarium pomocnicze można osiągnąć tylekroć taniej!)

Na wstępie wspomnieliśmy, że wszystkie znane problemy budowy oprogramowania można przedstawić jako kombinacje kroków kanonicznych i operacji wycofywania się. Przedstawiony powyżej schemat postępowania nazwaliśmy wyidealizowanym dlatego, że nie występuje w nim operacja wycofywania. Skąd bierze się potrzeba wycofywania? Ma ona dwa źródła:

(1) ukończywszy pewien krok dochodzimy do wniosku, że nie możemy ustalić rozsądnego następnego poziomu językowego (np. na żadnym poziomie, który można osiągnąć z bieżącego, nie potrafimy zrealizować bieżącej specyfikacji); nasze postępowanie konstrukcyjne zaszło w ślepy zaułek.

(2) klient zgłasza modyfikację specyfikacji, której to modyfikacji nie potrafimy wrazić jako rozszerzenia bieżącej specyfikacji (możliwość takiego dezyderatu wynika stąd, że w porządnie kierowanym procesie programistycznym sprawdzamy słuszność każdej lub przynajmniej *prawie każdej* specyfikacji pośredniej).

W obydwu przypadkach istnieje radykalne lekarstwo: powrót do specyfikacji pierwotnej i rozpoczęcie całego postępowania od początku. Aczkolwiek na pewno skuteczne, jest to jednak lekarstwo bardzo drogie, gdyż powoduje zmarnowanie całej dotychczas wykonanej pracy. Dlatego też oszczędniej będzie wycofywać się krok po kroku, aż do napotkania pierwszego takiego poziomu językowego, na którym

można będzie zaradzić powstałemu kłopotowi. Samo wycofywanie się krok po kroku nie przedstawia żadnych trudności, naturalnie jeśli zawczasu zadbaliśmy o stworzenie odpowiednich po temu warunków (np. zapisując całą historię danego projektu). Ocena, czy wycofaliśmy się dostatecznie daleko, zależy od natury problemu. W pierwszym z wymienionych przypadków ocena ta jest kwestią umiejętności programisty (podobnie jak wszystkie decyzje co do wyboru poziomów językowych!). W drugim, możemy polegać na tym, czy żądana modyfikację można wyrazić na tym poziomie, do którego już się wycofaliśmy (ewentualnie, czy dany poziom da się niesprzecznie odpowiednio poszerzyć).

Po nawrocie i ustaleniu poziomu językowego, od którego zaczynamy nową próbę dotarcia do zadanego poziomu ostatecznego, kolejne kroki mogą (ale nie muszą) powtarzać decyzje co do wyboru poziomów językowych. Ponieważ należy się liczyć z tym, że konieczność wycofania się wystąpi wielokrotnie, rzeczywisty proces konstruowania oprogramowania odbiega od wyidealizowanego, liniowego schematu przedstawionego powyżej. Faktycznie mamy do czynienia z dwoma ciągami kroków kanonicznych: z historycznym następstwem kroków rzeczywiście wykonanych (jeśli rezygnujemy z reprezentowania nawrotów ciąg ten przypomina drzewo o gałęziach nierównej długości - jeden liść jest ostateczną realizacją, pozostałe odpowiadają krańcom ślepych zaułków), oraz z czysto liniową sekwencją kroków, które doprowadziły do ostatecznej realizacji (jest to, oczywiście, jedna z gałęzi drzewa).

Do różnych celów praktycznych potrzebne są oba ślady procesu konstrukcji oprogramowania. Dokumentacja użytkowa powinna np. zawierać opis ciągu kroków prowadzących do ostatecznej realizacji, natomiast dokumentacja techniczna, przeznaczona dla dalszej pielęgnacji oprogramowania - całe drzewo, gdyż nigdy nie wiadomo, do jakiego poziomu trzeba będzie wrócić aby zaspokoić przyszłe żądania klienta.

Ponieważ - podkreślmy to raz jeszcze - metoda kroków kanonicznych wymaga wykonania bardzo wielu drobnych kroków, a stosowanie operacji wycofywania się

(Mimo wszystkie zalety standardyzacji i powielanego oprogramowania użytkowego, zapotrzebowanie na jednostkowe oprogramowanie stanowi - i zapewne będzie stanowić - dominujący aspekt rynku oprogramowania: każdy nowy port lotniczy, każda duża korporacja, każdy bank i każdy rodzaj usług informacyjnych wymagają własnego, unikatowego oprogramowania, a są to systemy, których ceny kontraktowe wynoszą dziesiątki milionów dolarów za sztukę.)

Koncepcja budowy oprogramowania metodą kroków kanonicznych legła u podstaw dwu poważnych przedsięwzięć programistycznych realizowanych przez londyńską firmę software'ową IST (Imperial Software Technology). Są to projekty ISTAR i GENESIS.

Projekt GENESIS [12], częściowo finansowany przez EWGowski program badawczy Esprit i prowadzony wspólnie z laboratoriami Philipsa w Holandii, dotyczy mechanizmów jednego kroku: przekształceń formalnych między dwoma poziomami lingwistycznymi. Obejmuje on pokaźny zestaw narzędzi programistycznych ułatwiających prezentowanie specyfikacji i ich rozszerzeń (różnego rodzaju edytory i weryfikatory syntaktyczne, a także programy wpomagające badanie niesprzeczności), dowodzenie poprawności realizacji itp. Cechą charakterystyczną projektu GENESIS jest brak uprzywilejowanej logiki: narzędzia programistyczne są tak zaprojektowane, by mogły być stsosowane z różnymi logikami (konkretną logikę zadaje się jak ,,parametr''), nawet wtedy, gdy logika poziomu językowego specyfikacji jest inna niż logika systemu językowego realizacji. Prototypowa wersja GENESIS została oddana do użytku latem 1987r.

Projekt ISTAR [13] został zrealizowany na zamówienie BT - niegdyś państwowego, dziś zdenacjonalizowanego brytyjskiego koncernu telekomunikacyjnego. Pierwsza wersja została uruchomiona w centrum programistycznym BT na przełomie lat 1985/86, po czym firma IST podjęła sprzedaż ISTARów ,,na wolnym rynku'', klientom tak różnym jak: jeden z największych światowych producentów układów scalonych, wielkie przedsiębiorstwo poszukiwań geologicznych, koncern lotniczy i ministerstwo spraw zagranicznych pewnego niewielkiego, ale bardzo aktywnego pań-

bardzo komplikuje strukture nastepstwa kroków, wydaje się zupełnie niemożliwe opanowanie całości procesu bez odpowiednio silnych środków wspomagajacych, których przeznaczeniem jest archiwizowanie poszczególnych kroków. Nie może to być byle jakie archiwum, powinno ono maksymalnie ułatwiać rekonstrukcję dowolnego minionego stanu procesu konstruowania oprogramowania.

Tak wiec widzimy, że stosowanie niesłychanie prostej (ideowo!) metody kroków kanonicznych wymaga bardzo silnego środowiska programistycznego, na które składaja sie dwa, prawie niezależne elementy: instrumentarium wspomagajace wykonywanie jednego kroku (przeniesienia specyfikacji z jednego poziomu językowego na drugi) i baza danych (archiwum wykonanych kroków wraz z odpowiednio dobranymi mechanizmami ,,nawigacji'', pozwalajacymi rekonstruować minione stany procesu konstrukcji). Obydwie składowe środowiska sa całkowicie uniwersalne, nie zależa od semantyki rozwiazywanego zadania, obydwie też, niestety, sa bardzo skomplikowane i wymagaja bardzo silnych środków obliczeniowych. Nie należy się łudzić, że środowiska tego rodzaju beda dostepne w sprzedaży wysyłkowej, ani że beda mogły funkcjonować na komputerach z waska szyna albo wolna pamiecia niesekwencyjna o pojemności kilkunastu megabajtów.

Powstaje wobec tego pytanie, czy cała metoda ma sens? Odpowiedź zależy od tego, co jest przedmiotem naszych zainteresowań.

Po pierwsze, opisana metoda ma pewne znaczenie poznawcze, gdyż redukuje całość procesu konstrukcji oprogramowania, od wstepnej specyfikacji do ostatecznej implementacji i nastepujacej po niej pielegnacji funkcjonujacego systemu, do kombinacji jednorodnych kroków. Pozwala to lepiej poznać nature atomów działalności programistycznej.

Po drugie, aczkolwiek mało użyteczna dla amatorów i nawet profesjonalistów korzystajacych z ,,gotowców'', opisana metoda ma sporo zalet dla konstruktorów dużych jednostkowych systemów budowanych od podstaw.

stwa. Rozrzut tematyczny instytucji kupujących ISTARy świadczy o uniwersalności systemu, inaczej powiedziawszy, dowodzi jego całkowitej neutralności względem semantyki konkretnego oprogramowania. Trzeba jeszcze dodać, że ISTAR jest dostarczany zarówno w wersjach indywidualnych (jako część wyposażenia osobnych stanowisk pracy), jak też jako system wielodostępny, działający na kilku, a nawet kilkudziesięciu stanowiskach pracy (do pięćdziesięciu). (Dla ciekawych: wersja wielodostępna kosztuje 0.5 miliona funtów + 50 tys. za każde obsługiwane stanowisko, oczywiście bez kosztu sprzętu; istnieje też wielodostępna wersja edukacyjna - za 300 tys. funtów, bez względu na liczbę stanowisk.)

ISTAR jest zespolonym środowiskiem programistycznym przeznaczonym do wspomagania pracy programistów realizujących duże zadania z dziedziny konstrukcji oprogramowania. Jednocześnie ISTAR wspomaga kierowników zespołów programistycznych w zakresie planowania pracy, bieżącego nadzoru i kontroli, kosztorysowania, fakturowania i sprawozdawczości. Umiejętne korzystanie z ISTARa pozwala też znacznie uprościć prace pomocnicze, w tym - właściwe dokumentowanie przedsięwzięć programistycznych. Całą menadżerską i dokumentacyjną stronę ISTARa zmuszeni jesteśmy tutaj pominąć, aczkolwiek pomysłowe wykorzystanie odnośnych możliwości (stworzonych przez podstawowy model krokowy) niewątpliwie przyczyniło się do handlowego sukcesu tego środowiska.

Podstawową jednostką działalności programisty korzystającego z ISTARa jest krok. Na wykonanie każdego kroku zawierany jest kontrakt pomiędzy zleceniodawcą i zleceniobiorcą (por. prace [14]). Kontrakt zawiera opis zadania kroku i formułuje kryteria jego odbioru, ustalając jednocześnie, kto ma dokonać odbioru (rolę tę zazwyczaj odgrywa sam zleceniodawca, ale może on wskazać dowolnego innego ,,agenta''). Kontrakt wskazuje też przewidywaną datę ukończenia kroku, tj. spełnienia kryteriów odbioru wykonanego zadania (kontrakt zawiera też inne informacje wykorzystywane przez menadżerskie programy ISTARa).

Zleceniobiorca może przystąpić do wykonywania kroku, albo rozpisać zadanie na dwa lub więcej kroków. W tym przypadku staje się on zleceniodawcą, zawierają-

cym kontrakty na wykonanie tych kroków. W trakcie wykonywania kroku zleceniobiorca zawsze może sformułować ,,podzadanie'', które zleci - jako odrębny krok kontraktowy - do wykonania innemu zleceniobiorcy. (Nic zresztą nie stoi na przeszkodzie by zawierać kontrakty na podzadania z sobą samym: ISTAR wymaga tylko by każdy krok miał ściśle określonego wykonawcę, by na jego realizację był zawarty legalny kontrakt i by przewidywane terminy realizacji spełniały naturalne zależności ścieżek krytycznych.)

Przystępując do wykonywania kroku, zleceniobiorca pobiera świeży egzemplarz jednego z wielu warsztatów ISTARa. Warsztatem (*workbench*) nazywa się zespół powiązanych ze sobą środków programistycznych. Warsztat pascalowy np. składa się z edytora syntaktycznego, kompilatora, programów wspomagających analizę wykonywania programów w Pascalu, optymalizatorów kodu wynikowego, edytora programów (*pretty printing*) itd. Obecna wersja ISTARa zawiera warsztaty Pascala, Chilla, Ady, Prologu, SDLu, COBOLu, kilku baz danych oraz VDMu; liczba warsztatów jest stale zwiększana. Kontrakt na wykonanie kroku może ograniczyć swobodę wyboru warsztatu; również wybór warsztatu może nałożyć odpowiednie ograniczenia na kontrakty zawierane na ewentualne podzadania.

Informacja o wybranym warsztacie jest na trwałe związana z krokiem; każdy nawrót do tego samego kroku automatycznie otwiera ten warsztat i to w stanie, w jakim był, kiedy ostatni raz pracowano nad danym krokiem.

Praca nad realizacją kroku może zająć jedną lub wiele sesji; przerywając pracę, tj. kończąc sesję, programista zamyka warsztat, wznawiając pracę - otwiera go, a ISTAR dba o to, by warsztat był w takim samym stanie jak na końcu poprzedniej sesji.

Całość informacji generowanej w trakcie kroku jest gromadzona przez archiwum ISTARa, wykonawca kroku może zażądać zniszczenia części (a nawet całości) tej informacji, ale kontrakt może tę swobodę silnie ograniczyć. W żadnym zaś przypadku nie można zniszczyć skrupulatnie odnotowywanej informacji o zlece-

niach niszczenia informacji. Kontrakt może też przyznać zleceniodawcy prawo niszczenia informacji wygenerowanej w czasie pracy zleceniobiorcy. Podobne reguły dotyczą praw inspekcji informacji powstającej w trakcie kroku.

Po ukończeniu kroku, jego wyniki są poddawane odbiorowi. Po pomyślym odbiorze, wyniki stają się własnością zleceniobiorcy, ale archiwum ISTARa nadal przechowuje informacje o zakończonym kroku, pozwalająca na jego pełna rekonstrukcję.

Powyższe, bardzo pobieżne wyliczenie niektórych głównych funkcji ISTARa (pominięto np. nader rozbudowane środki obrazowania globalnego postępu prac i wszelkiego rodzaju transformacje międzywarsztatowe) pozwala zorientować się co do zakresu usług środowiska. Warto dodać, że ISTAR istnieje w postaci metasystemu, co pozwala generować ISTARy na różne komputery i ich konfiguracje.

WYKAZ CYTOWANEJ LITERATURY

[1] LEHMAN M M, STENNING V, TURSKI W M: Another Look at Software Design Methodology. *ACM Software Engineering Notes* 9(2), April 1984, pp. 38-53.

[2] MAIBAUM T S E and TURSKI W M: On What Exactly Goes On When Software is Developed Step-by-Step. *Proc. 7 ICSE (Orlando, Fla.)*, 1984, pp.528-533.

[3] TURSKI W M and MAIBAUM T S: *The Specification of Computer Programs*. Addison-Wesley, 1987

[4] GEHANI N and McGETTRICK A D (Eds.): *Software Specification Techniques*. Addison-Wesley, 1986.

[5] TURSKI W M: *Specification as a Theory with Models in the Computer World and in the Real World*. INFOTECH State of Art Report,IX (6), 1981, pp. 363-377.

[6] TURSKI W M: *O informatycznym sposobie rozwiązywania zadań nie tylko matematycznych*. Referat plenarny na Zjeździe PTM w Kielcach, wrzesień 1985. Sprawozdania IInf UW Nr. 148.

[7] GUTTAG J V, HORNING J J and WING J: Some Notes on Putting Formal Specifications to Productive Use. *Science of Computer Programming*, 2 (1), 1982.

[8] GOGUEN J A and MESEGUER J: *Rapid Prototyping in the OBJ Executable Specification Language*. SRI Technical Report CSL-137, 1982.

[9] COHEN B, HARWOOD W T and JACKSON M I: *The Specification of Complex Systems*. Addison-Wesley, 1986.

[10] TURSKI W M: *Design of Large Programs*. w: Software engineering - Entwurf und Spezifikation (C. Floyd und H. Kopetz, Hrsg.), Teubner Verlag, 1981, pp. 129-160.

[11] CORELL C H: Proving Programs Correct Through Refinement. *Acta Informatica* 9 (1978), pp. 121 - 132.

[12] BUSTANY A, HARWOOD W, HELMINK L, JORDAN L and WRAY M: Overall Requirements for GENESIS (project 1041). IST, March 1986.

[13] ISTAR: Integrated Project Support Environment: Technical Overview. IST, August 1985.

[14] TURSKI W M: Software Quality Control: Techniques and Users' Criteria. w: *Constructing Quality Software* (P.G. Hibbard and S.A. Schuman, Eds.), North Holland (1978), pp. 37 - 43.

Jesienna Szkoła FTI
Mrągowo, grudzień 1987

OCHRONA PROGRAMÓW KOMPUTEROWYCH NA GRUNCIE PRAWA AUTORSKIEGO

doc. Bogusław Gawlik
Katedra Prawa Cywilnego
Uniwersytet Jagielloński
ul. Wenecja 2
31-145 Kraków, tel. 341335 w.280

1. Definicje oprogramowania komputerowego jako przedmiotu ochrony prawnej

Dotychczas podejmowane /zwłaszcza w pracach międzynarodowych organizacji ochrony własności intelektualnej/ próby określenia oprogramowania komputerowego jako przedmiotu ochrony prawnej nie doprowadziły do ustalenia powszechnie akceptowanych, w pełni zadawalających definicji. Trudności osiągnięcia porozumienia w tym zakresie są zrozumiałe, jeżeli się zważy, że ustalenie per definitionem przedmiotu ochrony prawnej łączy się zawsze z niebezpieczeństwem niezamierzonego i nieusprawiedliwionego ograniczenia zakresu tej ochrony.

W piśmiennictwie prawniczym powołuje się najczęściej propozycje definicyjne przedstawione w ustawie wzorcowej o ochronie oprogramowania komputerowego, opracowanej przez międzynarodowy zespół ekspertów Światowej Organizacji Ochrony Własności Intelektualnej /WIPO/. W projekcie tej ustawy zdefiniowano dla celów ochrony prawnej pojęcia "programu komputerowego", "opisu programu",

"materiałów pomocniczych" oraz "software".

"Program komputerowy" oznacza "zbiór instrukcji, zapisanych w sposób umożliwiający maszynie ich odczytanie i zdolnych do spowodowania, aby maszyna wyposażona w urządzenie do przetwarzania danych wskazała, wykonała lub osiągnęła określoną funkcję, zadanie lub rezultat". Wymaganie zapisu instrukcji w sposób czytelny dla maszyny spełniają "zapisy na taśmie magnetycznej lub taśmie perforowanej, a także zapisy w języku symbolicznym, które mogą być za pomocą odpowiednich urządzeń przetłumaczone na zapisy umożliwiające maszynie ich odczytanie".

"Opis programu" oznacza "proceduralne, kompletne przedstawienie programu w formie werbalnej, schematycznej lub innej w sposób dostatecznie szczegółowy dla oznaczenia zbioru instrukcji tworzących program komputerowy".

"Materiały pomocnicze" oznaczają "wszelkie materiały - inne niż program komputerowy lub opis programu - opracowane w celu ułatwienia zrozumienia i zastosowania programu". Należą do nich m.in. opisy problemu oraz instrukcje dla użytkowników wyjaśniające sposób stosowania programu.

"Software" oznacza bądź jeden z określonych wyżej elementów oprogramowania /program komputerowy, opis programu lub materiały pomocnicze/, bądź też wszystkie te elementy łącznie.

Tak określone elementy oprogramowania komputerowego należą do klasy produktów intelektualnych, których zdatność do ochrony na gruncie prawa autorskiego może być niejednokrotnie sporna. Ze względu na możliwość uzyskania takiej ochrony tworzą one silnie zróżnicowaną grupę: co do niektórych elementów oprogramowania komputerowego należy wykluczyć możliwość uznania ich za przedmiot prawa autorskiego, co do innych elementów należących do software możliwość taka zachodzi, ale w żadnym przypadku nie jest bezsporna. Objaśnienia wymaga przede wszystkim status prawny tego rodzaju produktów intelektualnych. W szczególności należy rozważyć pytanie, które spośród elementów oprogramowania komputerowego mogą spełniać wymagania, jakie

przepisy prawa autorskiego łączą z kwalifikacją produktu intelektualnego jako utworu podlegającego ochronie.

2. Indywidualność /oryginalność/ utworu podlegającego ochronie w prawie autorskim

Przepisy prawa autorskiego z 1952 roku nie określają w sposób syntetyczny wymagań, jakie powinien spełniać produkt intelektualny, aby można go było kwalifikować jako utwór podlegający ochronie prawa autorskiego. Przepis art. 1 § 1 prawa autorskiego stanowi jedynie, że "każdy utwór literacki, naukowy i artystyczny, ustalony w jakiejkolwiek postaci, jest przedmiotem prawa autorskiego". W przepisie tym nie należy jednak dostrzegać normatywnej rezygnacji z wymagania innych - poza "ustaleniem" - cech konstytuujących produkt intelektualny jako "utwór" w rozumieniu prawa autorskiego. Użycie bowiem w art. 1 § 1 prawa autorskiego zwrotu: "utwór literacki, naukowy i artystyczny" oznacza odesłanie do wykładni sądowej i doktrynalnej, na której spoczywa zadanie zrekonstruowania znaczenia, jakie należy przypisywać terminowi "utwór", a tym samym sformułowania przesłanek, od których spełnienia zależy udzielenie ochrony prawa autorskiego.

Przy określeniu tych przesłanek orzecznictwo posługuje się różnokształtnymi sformułowaniami cechy wyróżniającej "utwór". Określa się tę cechę jako "oryginalną treść i postać nowego dzieła", "samodzielność i oryginalność opracowania", "osobistą twórczość", "twórczy rezultat pracy", "twórczą inwencję", "wytwór indywidualnej myśli" itp. Określenia te używane są zamiennie, a wspólna dla nich denotacja wskazuje na wymaganie "oryginalności" /"indywidualności"/ utworu. Jednakże w wypowiedziach doktrynalnych interpretacja tego wymagania nie jest wolna od wątpliwości i niejasności. Należy tu wymienić dwa ujęcia "oryginalności" utworu: ujęcie subiektywne, które odwołuje się wyłącznie do cech procesu powstawania utworu oraz ujęcie obiektywne, w którym bierze się pod uwagę cechy utworu w zestawieniu z wytworzonymi uprzednio, już znanymi produktami intelektualnymi.

W ujęciu subiektywnym przypisuje się wymaganiu "oryginalności" utworu znaczenie "słabe", tzn. żąda się jedynie tego, by produkt intelektualny kwalifikowany jako "utwór" został samodzielnie wytworzony przez autora, był wynikiem jego osobistej pracy, a nie polegał na prostym naśladownictwie wytworzonych uprzednio produktów intelektualnych. Spełnienie tak określonego wymagania ma stanowić dostatecznie rzetelny wskaźnik twórczego charakteru pracy autora, a zarazem także "oryginalności" rezultatu tej pracy.

Wobec takiej interpretacji "oryginalności" utworu można wysunąć dwa podstawowe zarzuty. Po pierwsze, cechy procesu powstawania produktu intelektualnego /w szczególności zaś samodzielność pracy jego wytwórcy/ nie są wystarczające do wyróżnienia tego produktu spośród innych rezultatów pracy intelektualnej, nie wskazują bowiem na jego odmienność, różnice, zindywidualizowaną postać itp. w stosunku do znanych, uprzednio wytworzonych produktów intelektualnych. Na gruncie zaś ocen, które uzasadniają udzielenie ochrony w prawie autorskim nie wszystkie samodzielnie wytworzone produkty intelektualne korzystają z takiej ochrony, lecz tylko takie spośród tych produktów, które wykazują dostatecznie doniosłe różnice w porównaniu z uprzednimi wytworami intelektualnymi. Po drugie, o twórczym charakterze pracy wytwórcy produktu intelektualnego można orzekać przede wszystkim na podstawie oceny właściwości, które przysługują temu produktowi w porównaniu z innymi rezultatami intelektualnymi; natomiast odwrócona inferencja, tzn. orzekanie o twórczym charakterze produktu intelektualnego na podstawie swoistych cech powstawania tego produktu opiera się na kryteriach intersubiektywnie niesprawdzalnych i wskutek tego nieprzydatnych w ocenach prawnych.

Potrzeba obiektywizacji ocen prawnych wymaga zmiany w wartościowaniu okoliczności, które powinny być doniosłe dla ustalenia "oryginalności" utworu: w ocenach tych nie należy przypisywać rozstrzygającego znaczenia do relacji: autor - wytworzony przez niego produkt intelektualny, a należy przede wszystkim uwzględniać relację: badany produkt intelektualny - inne uprzednio wytworzone produkty intelektualne.

Rozstrzygnięcie zatem pytania, czy dany produkt intelektualny jest zdatny do uzyskania ochrony na gruncie prawa autorskiego jest nieuchronnie uwikłane w wartościowanie różnicy między porównywanymi

produktami intelektualnymi. Sąd wartościujący jest skierowany w tym przypadku na ocenę doniosłości nowych elementów produktu intelektualnego, które wyróżniają ten produkt w relacji do zestawionych z nim, uprzednio wytworzonych produktów, przy czym doniosła dla uzyskania ochrony "nowość" produktu intelektualnego oznacza indywidualną jego postać. W wielu przypadkach sformułowanie takiego sądu wartościującego jest intuicyjnie oczywiste. Dotyczy to sytuacji, w których twórca w procesie kształtowania utworu korzysta ze znacznej swobody w wyborze i uporządkowaniu jego elementów. Sytuacje takie aktualizują się zwłaszcza w dziedzinie utworów artystycznych, a więc tam, gdzie wielość możliwości wyboru i uporządkowania elementów składowych utworu wyklucza przewidywalność jego postaci końcowej. W takich właśnie przypadkach ustalenie "oryginalności" subiektywnej jest dostatecznie rzetelną podstawą domniemania, że badany produkt intelektualny spełnia zarazem wymaganie "oryginalności" obiektywnej, tzn. wykazuje cechy indywidualne doniosłe dla udzielenia ochrony w prawie autorskim.

Inaczej rzecz się przedstawia w sytuacjach, w których postać produktu intelektualnego jest mniej lub bardziej dokładnie określona przez uprzednio oznaczone metody i reguły jego kształtowania i/lub rodzaj jego przeznaczenia, co w konsekwencji powoduje, że zakres swobody twórcy w wyborze i uporządkowaniu elementów produktu intelektualnego zostaje znacznie ograniczony. W takich przypadkach presumpcja "oryginalności" obiektywnej jest zabiegiem nieuprawnionym; aby przypisać badanemu wytworowi intelektualnemu cechę "oryginalności" obiektywnej należy ustalić, czy i jakie nowe elementy w nim występują, a następnie rozstrzygnąć, czy różni się on dostatecznie od porównywanych z nim wytworów intelektualnych i wykazuje względem nich indywidualną postać.

Potrzeba wartościowania "oryginalności" produktu intelektualnego pojawia się właśnie w rozważanej tu dziedzinie, tj. w odniesieniu do rezultatów pracy programistów. Zakres bowiem swobody decyzji programisty w wyborze i uporządkowaniu elementów wytwarzanego przez niego produktu intelektualnego /program dla maszyny cyfrowej/ jest w znacznym stopniu ograniczony przez 1/ strukturalne i funkcjonalne powiązania między programem dla maszyny a zaprojektowanym uprzednio algorytmem rozwiązania problemu oraz opisem pro-

gramu zawierającym plan realizacji algorytmu, 2/ sformalizowane reguły dotyczące sposobu sporządzenia programu oraz 3/ przeznaczenie programu dla oznaczonego użytkownika lub oznaczonego typu użytkowników. Wskazane wyżej okoliczności powodują, że proces kształtowania programu komputerowego podlega ograniczeniom, które zwężają obszar możliwości dokonywania wyborów rozstrzygających o indywidualności produktu intelektualnego.

Ograniczenie w procesie kształtowania programów komputerowych możliwości wyboru i uporządkowania ich elementów bez wątpienia zmniejsza, ale wcale nie wyklucza prawdopodobieństwa pojawiania się dostatecznie indywidualnych rezultatów pracy programistów. Nie da się zatem odrzucić a limine możliwości powstawania w wyniku pracy programisty produktów intelektualnych zdatnych do ochrony na gruncie prawa autorskiego. Takie rozstrzygnięcie nie dostarcza jeszcze praktycznie użytecznej odpowiedzi na pytanie o dyrektywy, jakimi należy posługiwać się w ocenie "indywidualności" rezultatów pracy programisty. Idzie tu w szczególności o wskazanie elementów treści i formy wytworów programowania, co do których może aktualizować się swoboda wyborów twórczych oraz sformułowanie kryteriów wartościowania decyzji, które w ramach tej swobody zostają przez programistę zrealizowane.

3. Indywidualność rezultatów pracy programisty - oceny in abstracto

3.1. Algorytm rozwiązania problemu

Opracowanie algorytmu rozwiązania problemu łączy się, bez wątpienia, ze znacznym obszarem możliwości dokonywania wyborów twórczych, a w konsekwencji rezultat pracy analityka niejednokrotnie ujawnia cechę "oryginalności". Znikome bowiem jest prawdopodobieństwo występowania w praktyce takich problemów czy zadań, które mogą być poprawnie rozwiązane przez zastosowanie tylko jednej metody postępowania. Ten sam problem, przedstawiony do opracowania niezależnie kilku analitykom zostanie zazwyczaj rozwiązany w różny, ale ekwiwalentny sposób, tzn. zaprojektowane rozwiązania będą realizowały różnymi drogami ten sam cel.

Indywidualność algorytmu przejawia się w jego t r e ś c i, tj. koncepcji rozwiązania postawionego problemu, natomiast f o r m a przedstawienia algorytmu w postaci zapisu wzoru matematycznego lub sekwencji operacji logicznych jest w pełni zdeterminowana przez jego treść, co wyklucza swobodę wyboru - indywidualnego ze względu na formę - sposobu przedstawienia algorytmu.

3.2. Opis programu

3.2.1. Ogólny schemat blokowy

Funkcja takiego opisu polega na przedstawieniu w postaci schematycznej /werbalnej i graficznej/ zaprojektowanego przez analityka sposobu rozwiązania problemu. Idzie tu zatem o takie przedstawienie algorytmu, które ułatwiałoby jego zrozumienie i przyczyniałoby się do poprawnego sporządzenia programu. Treść ogólnego schematu blokowego jest wprawdzie nieco bogatsza od treści algorytmu, ale nie zawiera w stosunku do niego takich nowych elementów, które stanowiłyby zindywidualizowane, twórcze ukształtowanie zaprojektowanego sposobu postępowania. Treść ogólnego opisu programu może być zatem indywidualna tylko w takim zakresie i w takim stopniu, w jakim cecha "indywidualności" przysługuje treści odwzorowanego w opisie algorytmu.

W zakresie kształtowania formy ogólnego opisu programu programista nie dysponuje swobodą wyboru decyzji, które mogłyby uzasadniać uznanie tej formy za "indywidualną". Strukturalna i funkcjonalna zależność ogólnego opisu programu od treści zaprojektowanego algorytmu oraz standaryzacja środków przedstawienia treści w takim opisie wykluczają możliwość indywidualnego ukształtowania jego formy. Dotyczy to zarówno graficznych, jak i werbalnych środków przedstawienia algorytmu.

Swoboda wyboru co do kształtu, wielkości i przestrzennego rozmieszczenia graficznych elementów schematu blokowego jest tak nikła, że należy odrzucić a limine możliwość realizacji ekwiwalentnych, ale dostatecznie zindywidualizowanych ze względu na formę graficznych reprezentacji tego samego algorytmu. Graficzne schematy algorytmu, wykonane niezależnie przez kilku programistów będą

wprawdzie zawsze wykazywać różnice w formie jego przedstawienia, ale stopień doniosłości tego zróżnicowania nie jest wystarczający do ustalenia między porównywanymi schematami różnicy kwalifikowanej ze względu na udział w nich elementów nowych, oryginalnych.

Także werbalny opis schematu graficznego nie może być uznany za rezultat indywidualnych, twórczych wyborów programisty. Dobór słów wpisywanych w elementy graficznej reprezentacji algorytmu jest wyznaczony przez cechy przedmiotowe rozwiązanego problemu: zjawisku i relacje występujące w analizie tego problemu. Z tego też względu wyłączona jest możliwość posługiwania się w opisie schematu graficznego zindywidualizowanymi środkami wyrażenia treści.

3.2.2. Szczegółowy schemat blokowy

Analogicznie przedstawia się ocena "indywidualności" formy szczegółowego schematu blokowego. Podobnie jak w przypadku ogólnego opisu programu nie występuje tu swoboda wyboru decyzji programisty, która uzasadniałaby przypisanie rezultatom jego wyborów cechy "oryginalności". W obydwu przypadkach graficzne i werbalne środki opisu programu są najczęściej identyczne; toteż wskazane uprzednio powody uchylające "indywidualność" formy pozostają aktualne także w odniesieniu do szczegółowego schematu blokowego. Ocena ta nie ulega zmianie również wtedy, gdy programista posługuje się w opisie takiego schematu językiem symbolicznym stosowanym w programowaniu. Stopień bowiem formulizacji tych języków wyklucza możliwość zindywidualizowanego doboru wyrażeń i ich uporządkowania w struktury syntaktyczne.

Natomiast znaczny obszar możliwości, a zarazem konieczności dokonywania przez programistę zindywidualizowanych wyborów pojawia się przy kształtowaniu treści szczegółowego opisu programu. Wprawdzie treść takiego opisu obejmuje elementy uprzednio zaprojektowanego algorytmu, ale recypowana warstwa treści zostaje przekształcona i wzbogacona przez opracowanie szczegółowego planu realizacji algorytmu. Przy budowie tego planu zachodzi konieczność uwzględniania takich czynników jak zużycie czasu pracy maszyny, niezbędnego do wykonania programu, wykorzystanie określonej pojem-

ności pamięci maszyny, czytelność i przejrzystość programu ze względu na potrzeby użytkowników. Poszukiwanie rozwiązań spełniających w sposób udany te wymagania otwiera przed programistą znaczny obszar możliwości zindywidualizowanych wyborów i wymaga od niego odpowiednich umiejętności matematycznych, doświadczenia praktycznego oraz pomysłowości. Umiejętne skonstruowanie szczegółowego opisu programu zapewnia - sporządzonemu na jego podstawie - programowi dla maszyny doniosłe wartości użytkowe /oszczędność czasu pracy maszyny lub miejsc w jej pamięci/ oraz "estetyczne" /przejrzystość programu/. W konsekwencji opracowanie planu realizacji algorytmu może ujawniać cechy indywidualne, wynikające z twórczych decyzji podejmowanych przez programistę. Na możliwość taką wskazuje już dostatecznie okoliczność, iż ten sam algorytm może stanowić podstawę do opracowania różnych szczegółowych opisów programu.

3.3. Program dla maszyny

Na podstawie tego samego szczegółowego planu realizacji algorytmu można sporządzić różne zbiory instrukcji dla maszyny, zapisane w języku umożliwiającym maszynie ich odczytanie. Możliwość taka świadczy o tym, że ograniczenia w procesie kodowania programów dla maszyny nie wykluczają swobody wyboru i uporządkowania ich elementów, a w konsekwencji nie uchylają możliwości sporządzania dostatecznie zindywidualizowanych programów.

Ze względu na strukturalną i funkcjonalną zależność programu dla maszyny od treści szczegółowego schematu blokowego obszar swobody wyborów rozstrzygających o indywidualności programu zależy od stopnia szczegółowości opracowania schematu blokowego. Im większy jest stopień szczegółowości planu realizacji algorytmu, tym mniejszy pozostaje obszar swobody wyboru i uporządkowania elementów programu dla maszyny. W przypadkach skrajnych schemat blokowy może być opracowany z takim stopniem szczegółowości, iż proces kodowania programu sprowadza się do prostego przełożenia treści schematu na język symboliczny czytelny dla maszyny. W takich przypadkach indywidualne cechy działalności programisty wyczerpują się w opracowaniu opisu programu, zaś sporządzenie programu dla maszyny stanowi reprodukcję tego opracowania w języku czytelnym dla maszyny. Możliwa jest również realizacja przypadków przeciwstawnych. Schemat blokowy może być

wprawdzie zawsze wykazywać różnice w formie jego przedstawienia, ale stopień doniosłości tego zróżnicowania nie jest wystarczający do ustalenia między porównywanymi schematami różnicy kwalifikowanej ze względu na udział w nich elementów nowych, oryginalnych.

Także werbalny opis schematu graficznego nie może być uznany za rezultat indywidualnych, twórczych wyborów programisty. Dobór słów wpisywanych w elementy graficznej reprezentacji algorytmu jest wyznaczony przez cechy przedmiotowe rozwiązanego problemu: zjawiska i relacje występujące w analizie tego problemu. Z tego też względu wyłączona jest możliwość posługiwania się w opisie schematu graficznego zindywidualizowanymi środkami wyrażenia treści.

3.2.2. Szczegółowy schemat blokowy

Analogicznie przedstawia się ocena "indywidualności" formy szczegółowego schematu blokowego. Podobnie jak w przypadku ogólnego opisu programu nie występuje tu swoboda wyboru decyzji programisty, która uzasadniałaby przypisanie rezultatom jego wyborów cechy "oryginalności". W obydwu przypadkach graficzne i werbalne środki opisu programu są najczęściej identyczne; toteż wskazane uprzednio powody uchylające "indywidualność" formy pozostają aktualne także w odniesieniu do szczegółowego schematu blokowego. Ocena ta nie ulega zmianie również wtedy, gdy programista posługuje się w opisie takiego schematu językiem symbolicznym stosowanym w programowaniu. Stopień bowiem formalizacji tych języków wyklucza możliwość zindywidualizowanego doboru wyrażeń i ich uporządkowania w struktury syntaktyczne.

Natomiast znaczny obszar możliwości, a zarazem konieczności dokonywania przez programistę zindywidualizowanych wyborów pojawia się przy kształtowaniu treści szczegółowego opisu programu. Wprawdzie treść takiego opisu obejmuje elementy uprzednio zaprojektowanego algorytmu, ale recypowana warstwa treści zostaje przekształcona i wzbogacona przez opracowanie szczegółowego planu realizacji algorytmu. Przy budowie tego planu zachodzi konieczność uwzględniania takich czynników jak zużycie czasu pracy maszyny, niezbędnego do wykonania programu, wykorzystanie określonej pojem-

ności pamięci maszyny, czytelność i przejrzystość programu ze względu na potrzeby użytkowników. Poszukiwanie rozwiązań spełniających w sposób udany te wymagania otwiera przed programistą znaczny obszar możliwości zindywidualizowanych wyborów i wymaga od niego odpowiednich umiejętności matematycznych, doświadczenia praktycznego oraz pomysłowości. Umiejętne skonstruowanie szczegółowego opisu programu zapewnia - sporządzonemu na jego podstawie - programowi dla maszyny doniosłe wartości użytkowe /oszczędność czasu pracy maszyny lub miejsc w jej pamięci/ oraz "estetyczne" /przejrzystość programu/. W konsekwencji opracowanie planu realizacji algorytmu może ujawniać cechy indywidualne, wynikające z twórczych decyzji podejmowanych przez programistę. Na możliwość taką wskazuje już dostatecznie okoliczność, iż ten sam algorytm może stanowić podstawę do opracowania różnych szczegółowych opisów programu.

3.3. Program dla maszyny

Na podstawie tego samego szczegółowego planu realizacji algorytmu można sporządzić różne zbiory instrukcji dla maszyny, zapisane w języku umożliwiającym maszynie ich odczytanie. Możliwość taka świadczy o tym, że ograniczenia w procesie kodowania programów dla maszyny nie wykluczają swobody wyboru i uporządkowania ich elementów, a w konsekwencji nie uchylają możliwości sporządzania dostatecznie zindywidualizowanych programów.

Ze względu na strukturalną i funkcjonalną zależność programu dla maszyny od treści szczegółowego schematu blokowego obszar swobody wyborów rozstrzygających o indywidualności programu zależy od stopnia szczegółowości opracowania schematu blokowego. Im większy jest stopień szczegółowości planu realizacji algorytmu, tym mniejszy pozostaje obszar swobody wyboru i uporządkowania elementów programu dla maszyny. W przypadkach skrajnych schemat blokowy może być opracowany z takim stopniem szczegółowości, iż proces kodowania programu sprowadza się do prostego przełożenia treści schematu na język symboliczny czytelny dla maszyny. W takich przypadkach indywidualne cechy działalności programisty wyczerpują się w opracowaniu opisu programu, zaś sporządzenie programu dla maszyny stanowi reprodukcję tego opracowania w języku czytelnym dla maszyny. Możliwa jest również realizacja przypadków przeciwstawnych. Schemat blokowy może być

pozbawiony cech indywidualnych i pozostawiać twórcy programu dla maszyny szeroki zakres wyboru i uporządkowania elementów składających się na treść instrukcji przeznaczonych dla maszyny. Ścisła zależność między schematem blokowym a programem dla maszyny powoduje, że zakresy swobody wyboru decyzji programisty, występujące w tych etapach procesu programowania nie dają się wyraźnie rozgraniczyć.

Forma zewnętrzna programu, tj. dobór i uporządkowanie wyrażeń języka symbolicznego stosowanego w kodowaniu programu pozbawiona jest cech indywidualizujących rezultat pracy programisty. Forma ta spełnia w opracowaniu programu rolę drugorzędną ze względu na jej ściśle instrumentalne funkcje wobec treści programu. Znaczny stopień formalizacji języków programowania wyklucza możliwość indywidualnego, oryginalnego kształtowania formy programu dla maszyny.

Indywidualność programu może natomiast przejawiać się - podobnie jak w przypadku opracowania szczegółowego opisu programu - w wyborze i uporządkowaniu elementów treści związanych z realizacją algorytmu. W procesie kodowania programu dla maszyny występuje znaczny obszar swobody dla zindywidualizowanych decyzji programisty w zakresie budowy programu i organizacji procesu przetwarzania danych, tj. wyboru, zestawienia, uszeregowania lub układu poszczególnych elementów algorytmu. Decyzje programisty /np. dotyczące wyboru instrukcji dla maszyny spośród zbioru ekwiwalentnych instrukcji, wyboru liniowych lub cyklicznych metod realizacji algorytmu i łączenia ich elementów/ rozpatrywane d y s t r y b u t y w n i e mogą przedstawiać się jako pozbawione cech "oryginalności", a w każdym razie mogą wskazywać jedynie na minimalny stopień ich indywidualności. Jeśli jednak zmienimy punkt odniesienia oceny "indywidualności" i będziemy rozpatrywać decyzje programisty k o l e k t y w n i e w ich łącznym zestroju, wówczas rezultat jego czynności będzie niejednokrotnie wykazywał intuicyjnie oczywistą cechę "indywidualności". Łączny efekt decyzji podejmowanych w procesie kodowania tworzy "tkaninę" programu, jego strukturę złożoną z indywidualnie uporządkowanych elementów realizacji algorytmu. Swoboda kształtowania tej struktury jest zazwyczaj tak znaczna, że jej konkretna

realizacja - wynikająca z decyzji danego programisty - będzie różniła się od innych możliwych realizacji programu w sposób wystarczający do przypisania jej cech indywidualnych.

4. Program komputerowy jako przedmiot prawa autorskiego

Przedstawiona analiza wskazuje, że cecha "indywidualności" rezultatów pracy analityka i programisty /algorytm, opis programu, program dla maszyny/ może przejawiać się w warstwie ich t r e ś c i, natomiast nie występuje w warstwie ich f o r m y, tj. w sposobie wyrażenia treści. Rozstrzygnięcie zatem zagadnienia, czy produktom intelektualnym wytwarzanym w działalności analityka i programisty może przysługiwać ochrona na gruncie przepisów prawa autorskiego należy przede wszystkim od ustalenia odpowiedzi na pytanie, czy ocena "indywidualności" /"oryginalności"/ utworów chronionych w prawie autorskim może opierać się w y ł ą c z n i e na analizie ich formy, czy t a k ż e na analizie ich treści. Pytania tego przepisy prawa autorskiego wyraźnie nie rozstrzygają. Pogląd tradycyjnie przyjmowany w doktrynie prawniczej /i nadal w niej zdecydowanie przeważający/ głosi, że jedynie forma utworu podlega ochronie w prawie autorskim, natomiast jego treść nie jest warstwą zdatną do takiej ochrony. Pogląd ten wywołuje jednak poważne zastrzeżenia zwłaszcza w odniesieniu do utworów naukowych, do których można zaliczyć również rezultaty czynności analityka i programisty /zwrotowi "utwór naukowy" nadaje się w doktrynie prawa autorskiego szerokie, pojemne znaczenie, obejmujące także utwory o przeznaczeniu głównie praktycznym, a oparte na zastosowaniach wiedzy naukowej/.

Aprobata poglądu, wedle którego treść utworu naukowego jest pozbawiona wszelkiej doniosłości dla oceny jego indywidualności jako koniecznej przesłanki udzielenia ochrony w prawie autorskim prowadziłaby do generalnego zakwestionowania zdatności do takiej ochrony utworów z wielu dziedzin nauki. We wszystkich tych dziedzinach nauki, w których wykształciły się sformalizowane języki w celu jednoznacznego, dokładnego i intersubiektywnie sprawdzalnego formułowania i przekazywania wiedzy /matematyka, logika, fizyka teoretyczna itd./ forma zewnętrzna wypowiedzi /tj.wybór znaków słownych, sym-

bolicznych itp. i ich uporządkowanie w struktury syntaktyczne/ spełnia zawsze drugorzędną rolę ze względu na jej ściśle instrumentalny związek z treścią wypowiedzi, a w konsekwencji - nie może ujawniać cech indywidualnych. Wprawdzie referowany pogląd doktryny prawniczej dopuszcza ocenę indywidualności utworów naukowych również na podstawie ich tzw. formy wewnętrznej, ale takie rozszerzenie podstawy oceny tej przesłanki nie uchyla w pełni wątpliwości co do zdatności niektórych utworów naukowych do uzyskania ochrony w prawie autorskim.

Forma wewnętrzna utworu naukowego może być zrekonstruowana w drodze abstrahowania od bezpośredniej percepcji tekstu utworu. Idzie tu o porządek przedstawienia i analizy zagadnień, tok rozumowania, sekwencje operacji logicznych itp. Sięgnięcie do analizy formy wewnętrznej utworu nie zawsze jednak "ratuje" ochronę dzieł naukowych, ponieważ w wielu przypadkach nie da się sensownie odgraniczyć formy wewnętrznej utworu od jego treści, zaś ta ostatnia warstwa dzieła naukowego - stosownie do przeważającego poglądu doktryny - nie podlega ochronie w prawie autorskim. I tak np. artykuł z dziedziny matematyki lub logiki, napisany w języku sformalizowanym i przedstawiający dowód jakiegoś twierdzenia należałoby uznać za niezdatny do uzyskania ochrony w prawie autorskim, gdyż jego forma zewnętrzna jest pozbawiona cech indywidualnych /wybór i uporządkowanie znaków wyznaczone są przez ściśle sformalizowane reguły zastosowanego języka/, zaś jego forma wewnętrzna jest tożsama z jego treścią, która - w świetle referowanego poglądu - nie ma doniosłości dla udzielenia ochrony w prawie autorskim. Jeśli bowiem forma wewnętrzna utworu naukowego obejmuje takie elementy jak tok rozumowania, następstwo operacji logicznych, sekwencje kolejnych kroków dowodzenia itp., to nie ulega wątpliwości, że w rozważanym przypadku właśnie te elementy stanowią treść artykułu. Gdybyśmy uznali, że wymienione elementy nie należą do treści artykułu, lecz stanowią składniki jego formy wewnętrznej, to należałoby konsekwentnie przyjąć, iż artykuł ten jest w ogóle pozbawiony treści. Ta swoista redukcja ad absurdum wskazuje na potrzebę rewizji tradycyjnego poglądu, uznającego treść utworów naukowych za czynnik zupełnie nierelewantny dla oceny ich indywidualności, a w konsekwencji również dla udzielenia im ochrony w prawie autorskim. Nikt bowiem nie twierdzi, że utwory naukowe z dziedziny mate-

matyki, logiki lub innych dyscyplin posługujących się sformalizowanymi językami są a limine niezdatne do uzyskania ochrony na gruncie prawa autorskiego.

Jako punkt wyjścia należy przyjąć - reprezentowany przez część doktryny prawniczej - pogląd, wedle którego relewatne dla udzielenia ochrony w prawie autorskim jest nie rozgraniczenie między treścią i formą utworu /przeprowadzane zresztą w sposób zróżnicowany i nieostry/, lecz rozróżnienie między indywidualnymi cechami utworu a takimi jego właściwościami, które należą do domeny publicznej, stanowią składniki dobra powszechnego. Utworowi przysługuje ochrona w prawie autorskim zarówno wtedy, gdy cechy indywidualne wykazuje jego forma, jak i wtedy, gdy cechy te przejawiają się t a k ż e lub w y ł ą c z n i e w jego treści. Przesłanka indywidualności utworu spełniona jest natomiast wtedy, gdy elementy jego formy i/lub treści nie są w p e ł n i wyznaczone przez uprzednio dane okoliczności należące do domeny powszechnej. Innymi słowy oznacza to, że przy kształtowaniu formy i/lub treści utworu jego twórca wykorzystał obszar swobody w wyborze i uporządkowaniu elementów utworu.

Cecha "indywidualności" utworu może wykazywać różny stopień natężenia w zależności od udziału w utworze elementów nowych w stosunku do elementów wyznaczonych przez okoliczności należące do powszechnego dorobku intelektualnego. Uzależnienie ochrony od występowania w badanym produkcie intelektualnym cechy "oryginalności" nie oznacza, aby cecha ta miała się przejawiać w jakimś określonym stopniu jej natężenia. Także w razie minimalnego stopnia "oryginalności" dopuszczalne jest kwalifikowanie ujawniającego tę cechę produktu intelektualnego jako przedmiotu prawa autorskiego. Kwalifikacja taka opiera się na ocenach wartościujących. Toteż nie da się generalnie oznaczyć minimum "oryginalności", które stanowiłoby wartość progową dla uzyskania ochrony w prawie autorskim i pozwalałoby w sposób dostatecznie bezpieczny rozróżniać produkty intelektualne zdatne i niezdatne do takiej ochrony. W każdym przypadku budzącym wątpliwości, tj. wtedy, gdy "oryginalność" badanego produktu intelektualnego nie jest intuicyjnie oczywista zachodzi konieczność odwołania się do ocen wartościujących. W ocenach tych należy kierować się dyrektywą, nakazującą uwzględniać aksjologiczne uzasadnie-

nie norm prawa autorskiego, jak i właściwości przedmiotów intelektualnych poddanych ocenom wartościującym.

Możliwość aktualizacji twórczych wyborów w procesie opracowywania programów komputerowych uzasadnia generalne stwierdzenie, że programy takie mogą spełniać przesłanki ochrony na gruncie prawa autorskiego. Należy jednak zastrzec, że ustalenie in concreto tych przesłanek wymaga odwołania się do ocen wartościujących, dokonywanych na podstawie sformułowanych uprzednio kryteriów. W ocenach tych nie jest doniosły kontekst powstania programu komputerowego, lecz jego właściwości przejawiające się w indywidualnym ukształtowaniu i konkretyzacji algorytmu.

W szczególności nie są dostatecznie rzetelnym wskaźnikiem "oryginalności" programu takie okoliczności, jak nakład pracy niezbędny do sporządzenia programu oraz rodzaj kwalifikacji i umiejętności programisty. Poprawne opracowanie programu komputerowego wymaga od jego autora odpowiednio wysokich kwalifikacji fachowych i znacznego nakładu pracy. Rodzaj czynności wykonywanych przez programistę trafnie przyrównuje się do metod pracy naukowej, a rezultat tych czynności w postaci szczegółowego opisu programu lub programu dla maszyny równie zasadnie uznaje się za produkt naukowy. Ponieważ w obiegowych intuicjach uznaje się niemal każdy rezultat pracy naukowej za przedmiot zdatny do ochrony w prawie autorskim, jest w pełni zrozumiałe, że - w oparciu o taką presumpcję, a bez ustalenia in concreto przesłanki "oryginalności" - można łatwo uprobować zdatność rezultatów pracy programisty do uzyskania ochrony na gruncie prawa autorskiego. Jednakże posługiwanie się taką presumpcją nie jest dostatecznie rzetelną podstawą oceny "oryginalności" produktu intelektualnego. Ani wielkość nakładu pracy autora, ani też jej naukowy charakter nie są same w sobie wystarczające do ustalenia, że produkt tej pracy wykazuje cechy "oryginalne".

Dla możliwości indywidualnego ukształtowania produktu intelektualnego rozstrzygające jest to, czy w pracy autora /niezależnie od wielkości jej nakładu i od jej rodzaju/ skierowanej na wytworzenie tego produktu aktualizują się możliwości wyboru elementów treści i formy. Właśnie stwierdzenie możliwości takich wyborów w pracy programisty rozstrzyga o zdatności in abstracto rezultatów

tej pracy do uzyskania ochrony w prawie autorskim. Nie uchyla to jednak potrzeby dokonywania in concreto ocen wartościujących w celu rozstrzygnięcia pytania, czy zrealizowane przez programistę wybory prowadzą do "oryginalnego" ukształtowania programu komputerowego.

Przeciwko tezie dopuszczającej ocenę "oryginalności" utworów naukowych na podstawie analizy ich treści wysuwa się najczęściej zarzut, że udzielenie ochrony prowadziłoby do ustanowienia - na rzecz twórców takich utworów - monopolu korzystania z rezultatów badań naukowych. Taka zaś konsekwencja pozostawałaby w sprzeczności z doniosłymi interesami ogólnymi, skierowanymi na zapewnienie swobodnego rozwoju nauki. Zaspokojenie tych interesów wymaga umożliwienia każdej zainteresowanej osobie - wolnego od przeszkód prawnych - korzystania z cudzego dorobku intelektualnego zarówno w dyskusji i krytyce naukowej, jak i w zastosowaniu do własnych badań. Ani doniosłość tych interesów, ani też potrzeba ich zaspokojenia nie może być, rzecz jasna, kwestionowana. Ale nie ma również wątpliwości, że interesy twórców dzieł naukowych są pragmatycznie niezgodne z postulatem zupełnego wyłączenia ochrony utworów naukowych ze względu na indywidualne cechy ich treści.

Rozstrzygnięcie przedstawionego konfliktu interesów nie musi jednak polegać - tak jak się to zwykle przyjmuje - na zupełnej rezygnacji z ochrony interesów twórców na rzecz realizacji interesu ogólnego. Zadawalające rozwiązanie można osiągnąć wtedy, gdy zagadnienie oceny indywidualności utworu naukowego odgraniczy się od zagadnienia realizacji interesów ogólnych związanych ze swobodnym rozwojem wiedzy. Utwór naukowy spełnia wymaganie "indywidualności", jeśli jego elementy nie są w pełni wyznaczone przez uprzednio dane składniki dorobku intelektualnego. Dla oceny tej nie jest relewantne ani rozróżnienie treści i formy utworu, ani też występowanie interesów ogólnych skierowanych na swobodny rozwój nauki. Dopiero w razie ustalenia, że dany utwór wykazuje cechy indywidualne, a zatem zdatny jest do uzyskania ochrony w prawie autorskim pojawia się potrzeba uwzględnienia interesów zorientowanych na zapewnienie swobodnego dostępu do rezultatów badań naukowych. Właśnie wskutek uznania doniosłości tych interesów uprawnienia autorskie, które powstają względem utworu naukowego zostają o g r a n i c z o n e: od chwi-

li opublikowania utworu naukowego każdemu przysługuje wolność korzystania z jego treści. Twórca nie jest uprawniony do wyłącznego rozporządzania utworem, korzysta natomiast z ochrony autorskich dóbr osobistych, w szczególności przysługuje mu uprawnienie do autorstwa utworu.

W dziedzinie rozważanych tu produktów intelektualnych podstawy do wspomnianego wyżej ograniczenia uprawnień autorskich zachodzą niewątpliwie w odniesieniu do algorytmu rozwiązania problemu. Wprawdzie w zestawieniu z innymi etapami pracy nad oprogramowaniem właśnie w etapie projektowania algorytmu występuje najszerszy obszar możliwości dokonywania twórczych wyborów, a w konsekwencji jest wysoce prawdopodobne, że rezultaty pracy analityka będą zazwyczaj spełniały wymaganie "indywidualności" utworu w rozumieniu prawa autorskiego, ale interesy ogólne związane z rozwojem wiedzy i zapewnieniem swobodnego dostępu do rezultatów badawczych uzasadniają uchylenie prawnej wyłączności rozporządzania takimi utworami, które zawierają algorytm rozwiązania określonego zadania. W takim rozstrzygnięciu przejawia się swoistość ochrony wszelkich utworów naukowych, co do których występuje społecznie uzasadniona potrzeba wyłączenia monopolu korzystania z rezultatów badawczych.

Potrzebę taką dostrzega się niekiedy również w odniesieniu do szczegółowych opisów programów oraz programów dla maszyny. Postualty ograniczenia ochrony tych elementów oprogramowania pozostają w bezpośrednim związku z teoretyczną koncepcją tzw. programów optymalnych, tj. programów, które zapewniają osiągnięcie pożądanego wyniku przy najmniejszych kosztach oraz maksymalnej oszczędności czasu pracy maszyny i miejsc jej pamięci. Wykazanie praktycznej możliwości realizacji takich programów uzasadniałoby rozważenie zagadnienia, czy interesy ogólne nie przemawiają w sposób dostateczny za swobodnym korzystaniem z optymalnych konkretyzacji algorytmów.

Jednakże programy optymalne są raczej teoretyczną konstrukcją niż aktualizowanymi w praktyce rezultatami programowania. Przede wszystkim pojęcie programu optymalnego jest nieostre. W zależności od tego, jakie przyjmuje się kryterium optymalizacji programu /np. czas pracy maszyny, ogólne koszty realizacji programu, przejrzystość programu itd./ nie jeden, lecz różne programy odpowiadają konstrukcji programu optymalnego. Tylko w razie ustalenia wszyst-

kich doniosłych w danym przypadku kryteriów optymalizacji można byłoby teoretycznie wyróżnić tylko jedną realizację programu jako konstrukcję optymalną. W praktyce mamy jednak do czynienia z "luźnym" oznaczeniem takich kryteriów, wyznaczającym dla programisty jedynie ogólne ramy opracowania programu. W takich przypadkach konstrukcja programu optymalnego może być rozumiana wyłącznie jako dyrektywa eliminowania w budowie programu operacji nieproduktywnych i zbędnych.

Dla konkretyzacji algorytmuw programie komputerowym pozostaje zatem zawsze tak obszerny zakres możliwości zindywidualizowanych wyborów, że ukształtowanie przez programistę planu realizacji algorytmu i sporządzenie na jego podstawie programu komputerowego nie pozbawia innych programistów możliwości konkretyzacji tego samego algorytmu w inny zindywidualizowany sposób. W konsekwencji nie zachodzi dostatecznie uzasadniona potrzeba, by w interesie ogólnym zapewnienia rozwoju wiedzy wyłączyć spod ochrony prawa autorskiego - zindywidualizowane ze względu na ich treść - plany realizacji algorytmu i sporządzone na ich podstawie programy dla maszyny.

5. Treść prawa autorskiego do programu komputerowego

Stosownie do postanowienia art. 15 prawa autorskiego należy wyróżnić dwie kategorie uprawnień przysługujących twórcy programu komputerowego: uprawnienia osobiste i uprawnienia majątkowe. Zakres poszczególnych uprawnień nie jest w ustawie dostatecznie wyraźnie sprecyzowany, ale podstawę do ich wyróżnienia można dostrzegać w dwu różnych funkcjach prawa autorskiego: z jednej strony idzie o ochronę osobistego stosunku twórcy do stworzonego przez niego dzieła, z drugiej zaś strony o ochronę ekonomicznych interesów twórcy w związku z gospodarczą eksploatacją jego dzieła.

Autorskie uprawnienia osobiste są nierozerwalnie związane z osobą twórcy, wobec czego nie mogą być przenisione na inne osoby i nie ulegają zrzeczeniu się. Natomiast majątkowe uprawnienia autorskie mogą przysługiwać osobom trzecim na podstawie umowy zawartej z twórcą. Sytuacja taka realizuje się m.in. w odniesieniu do programów komputerowych, które powstają jako rezultat świadczenia programisty w ramach stosunku pracy lub umowy o dzieło.

5.1. Uprawnienia osobiste

Podstawowym uprawnieniem osobistym twórcy programu komputerowego jest uprawnienie do autorstwa, które od strony pozytywnej oznacza możność domagania się ujawnienia pochodzenia programu od osoby jego twórcy, od strony zaś negatywnej - możność żądania, by nikt nie przywłaszczał sobie autorstwa programu.

Z uprawnieniem do autorstwa pozostaje w ścisłym związku możność decydowania przez twórcę o tym, czy i w jaki sposób powinno być oznaczone jego dzieło. Wybór takiego sposobu nie jest jednak pozostawiony dowolnej decyzji twórcy, lecz podlega ograniczeniom wynikającym z przyjętych w praktyce zwyczajów oraz technicznych możliwości zrealizowania wskazanego przez twórcę sposobu oznaczenia utworu. Mając na względzie właściwości materialnych nośników informacji, w których inkorporowane są programy komputerowe należy uznać za nieuzasadnione żądanie autora programu, by jego nazwisko zamieszczono na wszystkich materialnych nośnikach jego utworu.

5.2. Uprawnienia majątkowe

Prawo do wyłącznego rozporządzania utworem służy ochronie interesów majątkowych twórcy lub jego następcy prawnego bezpośrednio związanych z gospodarczą eksploatacją utworu. Obejmuje ono wszelkie postacie rozpowszechniania utworu i wykorzystywania jego wartości majątkowej.

Wśród doniosłych dla twórcy lub jego następcy prawnego /w szczególności jednostki organizacyjnej, z którą twórca pozostaje w stosunku pracy/ uprawnień do wykorzystania wartości majątkowej programu komputerowego należy wymienić przede wszystkim uprawnienie do zwielokrotniania programu oraz uprawnienie do jego rozpowszechniania.

Pierwsze z wymienionych uprawnień polega na prawie sporządzania przedmiotów fizycznych, w których program dla maszyny zostaje utrwalony i dzięki którym może być przez osoby trzecie percypowany i eksploatowany. Nośnik materialny, w którym zostaje utrwalony program komputerowy jest dla oceny jego zwielokrotnienia zupełnie obo-

jętny. W szczególności nie ma znaczenia ani rodzaj przedmiotów fizycznych, które spełniają funkcję nośnika programu, ani też rodzaj środków technicznych, przy użyciu których następuje utrwalenie programu. Bez znaczenia jest również liczba sporządzonych utrwaleń; także sporządzenie jednego egzemplarza należy kwalifikować jako "zwielokrotnienie" w znaczeniu prawnym.

Uprawnienie do zwielokrotniania programu komputerowego umożliwia osobie uprawnionej wykonywanie w szerokim zakresie kontroli nad korzystaniem przez osoby trzecie z programu komputerowego i zapewnia jej ochronę przed najistotniejszymi naruszeniami. Zwielokrotnienie programu dla maszyny naszępuje bowiem m.in. w razie reprodukcji programu źródłowego lub wynikowego w innych materialnych nośnikach informacji niż nośnik wytworzony przez osobę uprawnioną, maszynowego przekształcenia programu źródłowego w program wynikowy, wprowadzenia programu do urządzenia pamięci maszyny lub utrwalenia programu w obwodach scalonych.

Zagadnieniem spornym jest kwalifikacja jako "zwielokrotnienia"/w znaczeniu prawnym/ użycia programu w celu wykonania przez maszynę zgodnych z jego treścią funkcji. Kwestia ta nie ma znacznej doniosłości praktycznej w ewentualnym sporze z osobą nieuprawnioną do korzystania z programu komputerowego. W takim bowiem przypadku przed użyciem programu w maszynie osoba nieuprawniona musiała uzyskać materialne nośniki informacji, w których inkorporowany jest chroniony program albo sporządzić z nich kopie. Uprawniony może domagać się wydania lub zniszczenia bezprawnie uzyskanych lub wytworzonych materialnych nośników programu i w ten sposób zapobiec dalszemu używaniu programu przez osobę nieuprawnioną. Wydaje się jednak, że zachodzą dostatecznie uzasadnione podstawy do uznania zastosowania programu w maszynie za jego "zwielokrotnienie". Z użyciem bowiem programu w maszynie zazwyczaj łączy się - choćby przejściowa - reprodukcja w jej urządzeniach kolejnych instrukcji programu.

Uprawnienie do rozpowszechniania programu komputerowego obejmuje udostępnienie zainteresowanym odbiorcom programu jako dobra niematerialnego. Może to nastąpić przez oferowanie sprzedaży lub licencji programu, zawieranie umów sprzedaży lub umów licencyjnych,

a także przez wprowadzanie do obrotu materialnych nośników programu /sprzedaż, najem/.

Ochrona programu komputerowego wynikająca z przepisów prawa autorskiego zapewnia twórcy lub jego następcy prawnemu wyłączność gospodarczej eksploatacji programu. Przedmiotowy zakres udzielonej ochrony dotyczy tych elementów programu, które wykazują cechy indywidualne. Naruszenie uprawnień autorskich następuje nie tylko w razie prostego kopiowania programu /co stanowi zresztą w praktyce najczęstszą postać naruszenia interesów dysponenta programu/, ale także w razie przekształcenia programu przez wprowadzenie do niego nieistotnych zmian, tj. z zachowaniem indywidualnych cech zmienionego programu. Niedozwolone jest również korzystanie bez zgody osoby uprawnionej z programu przełożonego na inny język programowania.

Korzystanie z programu lub jego opisu w celu opracowania takiego samego lub istotnie podobnego programu lub opisu prowadzi do niedozwolonej reprodukcji. Dozwolone jest natomiast rekonstruowanie na podstawie analizy programu zastosowanego w nim algorytmu i opracowanie w oparciu o ten algorytm nowego, istotnie różniącego się programu.