International Journal

INFORMATION TECHNOLOGIES & KNOWLEDGE

2009 Volume 3 Number 1

International Journal
INFORMATION TECHNOLOGIES & KNOWLEDGE
Volume 3 / 2009, Number 1

IJ ITK is official publisher of the scientific papers of the members of the ITHEA International Scientific Society

IJ ITK rules for preparing the manuscripts are compulsory.
The **rules for the papers** for IJ ITK as well as the **subscription fees** are given on ***www.ithea.org***
The camera-ready copy of the paper should be received by e-mail: info@foibg.com.
Responsibility for papers published in IJ ITK belongs to authors.

General Sponsor of IJ ITK is the **Consortium FOI Bulgaria** (www.foibg.com).

International Journal "INFORMATION TECHNOLOGIES & KNOWLEDGE" Vol.3, Number 1, 2009

Edited by the **Institute of Information Theories and Applications FOI ITHEA®**, Bulgaria,
in collaboration with the **V.M.Glushkov Institute of Cybernetics of NAS, Ukraine**,
and the **Institute of Mathematics and Informatics, BAS, Bulgaria**.

Publisher: **ITHEA®**
Sofia, 1000, P.O.B. 775, Bulgaria. www.ithea.org, e-mail: info@foibg.com
Printed in Bulgaria

ISSN 1313-0455 (printed) **ISSN 1313-048X (online)** **ISSN 1313-0501 (CD/DVD)**

INFORMATION TRANSFORMATION SYSTEM BASED ON MAPPING OF GRAPH STRUCTURES

Margarita Knyazeva, Vadim Timchenko

***Abstract**: The paper develops and illustrates an approach to transformation of information represented as graph structures. It also provides the conceptual information transformation system based on mapping of graph structures. The system comprises three levels which makes it possible to adapt the transformer to information of different domains. Each level has its own models for describing information and specifying ways of its transformation. The transformer can operate with both structural and textual representations of information. The paper describes the general architecture of the transformer and its components. It offers models describing graph structures and mappings which are specifications for graph transformation. It also exemplifies descriptions of graph structures and syntax restrictions which are imposed on the former. The paper provides a fragment of description of one mapping of graph structure on the other. This approach is implemented in the prototype that ensures program translation from one procedural programming language into another. The prototype supports such languages as Pascal, C, languages of structure program models and is developed within the framework of the program transformation system. The prototype is implemented in the Java programming environment.*

***Keywords**: Transformation of information; rule-based transformation of graphs; structure mapping; structure editing.*

***ACM Classification Keywords**: I.2.5 Artificial intelligence: programming languages and software*

Introduction

Computer processing of information has proved to be one of the crucial activities in the majority of applied and theoretical domains. It comprises such tasks as acquisition, engineering and usage of various types of data and knowledge.

There is a constant information exchange among computer systems, their components, people and organizations. The types of information include artificial languages (programming languages, specification languages, data structure languages, etc.), knowledge, data and software represented in these languages. During their transmission, some aspects of information can change: representation format (changes in structure, model or language of representation), interpretation, level of detail.

This can be exemplified by information transformation at different stages of development of information systems. Conceptual data models independent from implementation are mapped on their relevant data models that are implementation oriented in a concrete operational environment. For instance, “entity-relationship” semantic model is mapped on a relational data model: UML language representation converts to XML language representation [Jean-Luc Hainaut, 2005].

Highly demanded task of translating programs from one representation language into another can be an additional illustrative example. It occurs in the following domains:

– development of systems meant for analysis, optimization and parallelization of programs, especially those working with several languages of source codes. The analyzed program in the source language is translated into

the internal representation that is later used for analysis and transformations [Partsch, H., Steinbrüggen, R., 1983; Shteinberg B.Ya., 2004];

– program reengineering that deals with the problem of program translation from outdated languages into new ones;

– development of domain-specific languages (DSLs). Usually, these are compact programming languages for solving specific tasks of a domain as opposed to the general-purpose languages designed to solve computational tasks in all domains [van Deursen, A. et al, 2000]. They are gaining ground due to the ever-increasing number of modern applied tasks.

Construction of translators is always determined as a hi-tech and time-intensive task which can be solved only by relevant specialists. This hampers mass development of translation tools.

Many tasks dealing with transformation of different types of information need further research. Various research groups have been developing approaches of different efficiency to their solving. Owing to their complexity, these tasks are mostly regarded to be independent and the methods of their solving are developed separately. At the research stage, checking these methods requires designing prototypes of program systems. Since the problem of program compatibility is not taken into account during the prototyping, designers concentrate on the methods of solving and tend to choose a specific representation of the information used. As a result, the derived computer systems are usually incompatible with each other; they are enclosed to their domains being unable to interact with each other to solve the information transformation tasks usually occurring at joints of domains. The maintenance often appears to be completely unprofitable in terms of time limits and labor input. Thus, they cannot be used to solve tasks of other domains and tasks at the domain joints, which lead to new software development.

Another problem of modeled program systems is that they are not usually developed to the level of distributed network applications (functioning in the Intranet/Internet environment). On the contrary, they are left as desktop personal versions and used only where they have been developed. This sharply limits accessibility, scope of practical application and consequently, demand for such tools that in turn decelerates their development and implementation pace.

Graph data structures are natural and visual tools of representation of complicated structures and processes. This makes it possible to use them widely in computer systems to solve different tasks. In many of these tasks, graphs are used to represent data of heterogeneous structures. Such tasks include representation of abstract syntax trees in translators of programming languages, description of structure and processing of object models of documents, processing of complicated data structures in domains in applied tasks solving, etc.

The majority of models that undergo transformations are represented as graph structures. The transformations are described in terms of graphs and operations and based on rules the performance of which results in a new graph on the basis of the specified source graph.

The aim is to develop models, methods and Internet tools of information transformation represented as graph structures.

This paper contains the description of the conceptual scheme and general architecture of information transformation system, the description of transformation model for graph structures.

The paper was written with financial support from FEB RAS (the Far Eastern Branch of the Russian Academy of Sciences) under Program 2 of the RAS Presidium “Intelligent Information Technologies, Mathematic Modeling, System Analysis and Automation", Project 09-I-П2-04 “Development of Control Systems for Knowledge Bases with Mutual Access".

Concept of Information Ttransformation Based on Mapping of Graph Structures

Let us examine conceptual schema of information transformation based on mapping of graph structures (Fig.1).

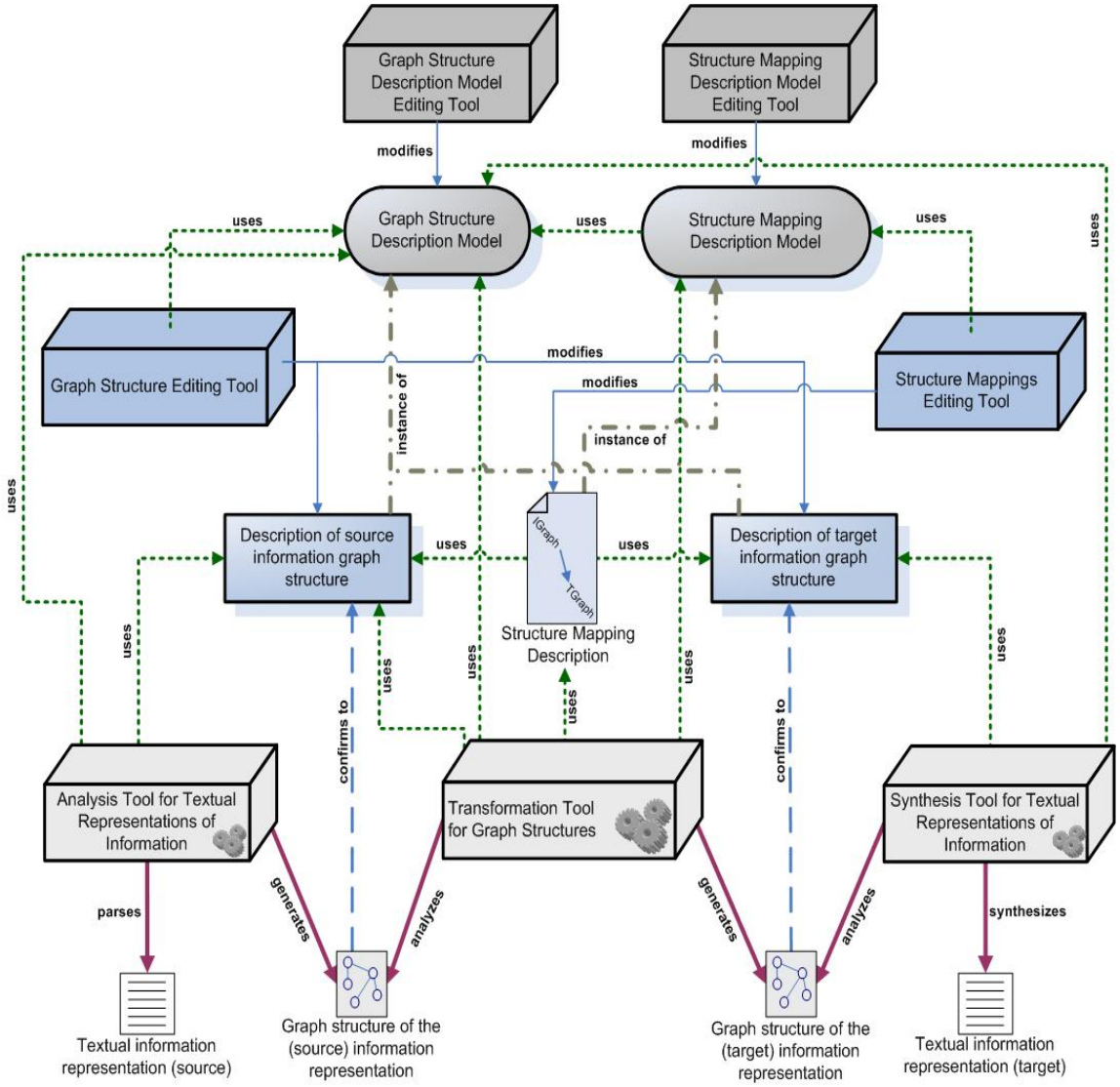

Fig.1. Conceptual schema of information transformation based on mapping of graph structures.

The basic idea of the approach is to represent source information and target information as a graph structure. The correspondent elements of the figure are as follows: *"Description of source information graph structure", "Description of target information graph structure".* This representation comprises:

– definition of a set of concepts of a described domain (graph vertices). Each concept (except those correspondent to the terminal vertices) is defined in terms of a set of some other concepts, and we say that concepts in terms of which some other concept is defined are included in the content of the latter;

– in a set of concepts there is a certain marked concept called an axiom, it does not comprise any arc. The same vertex can include several arcs;

– definition of a set of directed binary relations (with graph arcs corresponding to them) between the concepts. The concepts are linked with each other with the help of directed relations based on the following principle: the concept with an outgoing arc is defined in terms of the concept with an entering arc;

– additional information on a concept or a relation can be obtained from a set of attribute values connected with this concept or relation. A set of related attributes can be empty.

To be able to work with a textual information representation, it is necessary to describe the connection between the graph structure and elements of the syntax particular to the given structure.

This connection determines the content of the correct structures of the language of textual representation from the point of view of its syntax. The connection between the graph structure and its syntax contains such elements of language as punctuation, word order, etc.

This type of information limits the ability of the text to convey the sense given in the graph structure description, therefore this connection will be called syntax restrictions.

Graph structures are described in accordance with *Graph Structure Description Model*. The model regulates, in particular, the rules of specifying syntax restrictions and is used by *Graph Structure Editing Tool* to describe graph structures in terms of the given model. In its turn, *Graph Structure Description Model* is formed and edited with the help of *Graph Structure Description Model Editing Tool.*

Descriptions of the graph structure of information and its syntax restrictions are necessary to carry out syntax analysis of textual representation of information in the specified language and to form this information representation in the form of a graph structure. They are also necessary for the opposite procedure: to synthesize the textual representation of information according to its representation in the form of a graph structure. The procedures are executed respectively by *Analysis Tool for Textual Representations of Information* and *Synthesis Tool for Textual Representations of Information.*

Information Representation as Graph Structure (Target graph – Gt) is the *target* graph structure in regard to *Graph Structure Description* (Metagraph – Gm). This means that the Gt vertices that represent concepts constituting concept contents represented by vertices from Gm are the instances of these vertices from Gm graph structure, whereas the Gm graph structure vertices are prototypes for the Gt graph structure vertices. For example, the Gm graph contains a “name” vertex, while the Gt graph contains “Ivanov” and “Alexandrov” vertices. These latter vertices represent the concepts included into the concept content represented by the “name” vertex from Gt and are the instances of this vertex. Thus, it is possible to assert that Gt is described in terms of Gm.

For example, let there be specified a fragment of the graph structure description representing data on a patient.

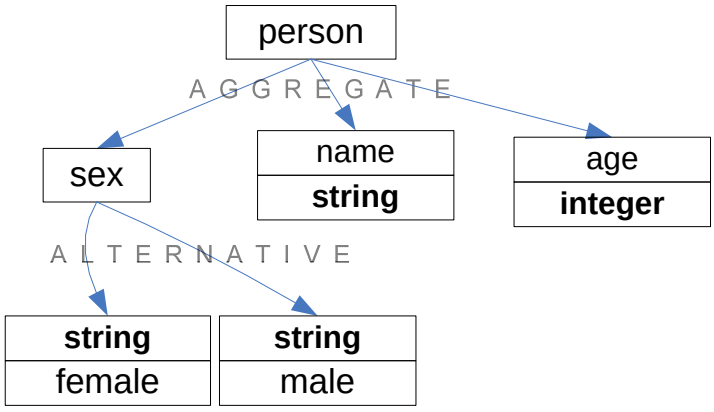

Fig. 2. Fragment of graph structure specifying patient's description.

Description of a particular “Patient 1” patient will be presented as it is shown on Fig.3.

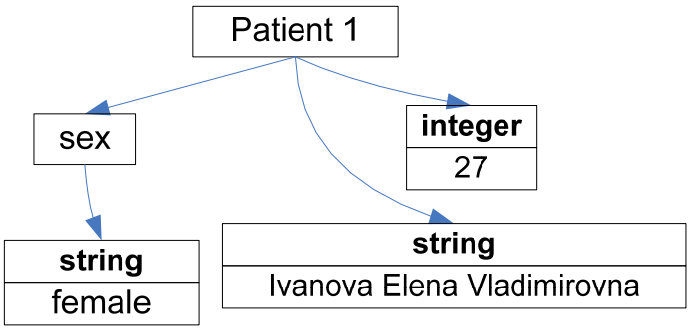

Fig.3. Fragment of graph structure specifying “Patient 1”.

By specifying mapping rules for this information in a particular textual form (graph structure syntax restrictions specifying patient's description) we can, for instance, receive the following representation of “Patient 1”: *Ivanova Elena Vladimirovna, fem., 27.*

The semantics of transformation is specified by *Structure Mapping Description* of the graph structure of the source information representation on the graph structure of the target representation. The mapping the fragments of source and target graph description registers a number of matches between structures. To describe the mappings, there is a language for description of structure mappings.

Structure mappings are described in accordance with *Structure Mapping Description Model*. This model is used by *Structure Mappings Editing Tool* to describe mappings in terms of this model. In its turn, *Structure Mappings Description Model* is formed and edited by *Structure Mapping Description Model Editing Tool.*

Transformation Tool for Graph Structures carries out transformation of information represented by graph structures on the basis of specified *Source Information Graph Description* mapping on *Target Information Graph Description*.

The proposed system served as the basis for the architecture of transformer for graph structures based on structure mapping description.

Let us consider the architecture of transformer for graph structures based on structure mapping description. (Fig.4).

The information component of the architecture comprises the graph structure description model, structure mapping description model, databases for storing graph structure descriptions and structure mapping descriptions respectively.

File system is generally considered to include any read-only memories for file storage available either on a local computer or on a remote one accessible via network interface.

Transformer comprises the following components:

– control subsystem for graph structure transformer;

– structure mapping editor;

– graph structure editor;

– text analysis and synthesis subsystem (TASS);

– graph structure generator.

Control subsystem for graph structure transformer connects components with each other and ensures information transmission and control within these components. It organizes internal interfaces with other system components and loads data from external data sources.

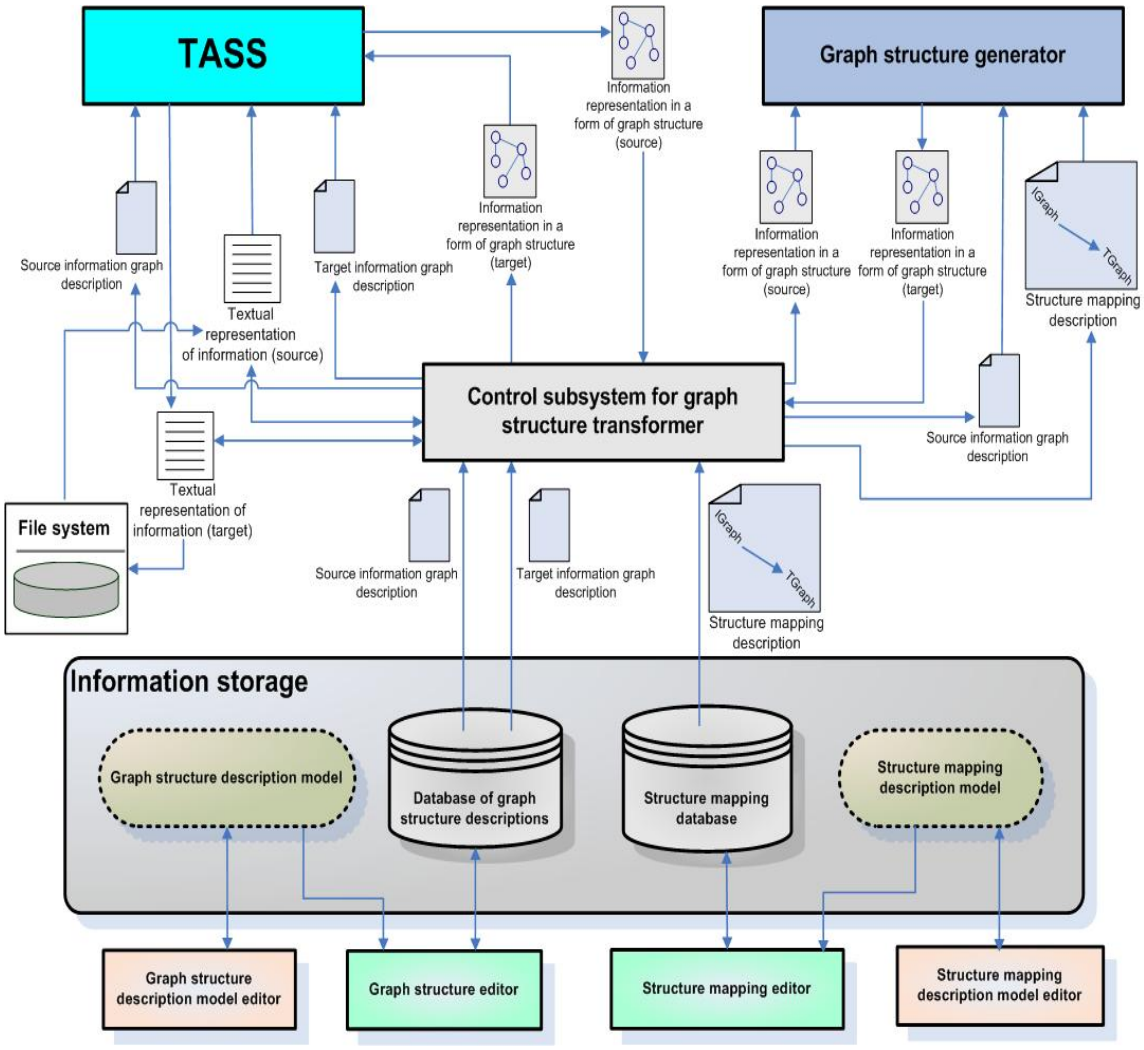

Fig.4. Architecture of transformer for graph structures based on structure mapping description.

Structure mapping editor provides filling and modification of structure mapping database. It is a structure editor the editing process in which is performed in terms of structure mapping description model. This means a certain support of the editing process by the editor itself. The model defines the restrictions on the type and structure of the edited information. This makes the user produce extension of a definition of information and specify values where they are necessary. At each stage, the editor provides the user with all necessary information to proceed to the next editing stage (for example, it offers acceptable variants of information editing where possible).

The structure mapping description model can be modified by structure mapping description model editor. All the modifications will be instantly reflected in the mapping editing (with no reboot of the structure mapping editor required).

Graph structure editor fills and modifies database of graph structure descriptions. This editor (as the structure mapping editor) is a structure editor that executes editing in terms of graph structure description model.

Graph structure description model can be modified with graph structure description model editor. All the modifications will be instantly reflected in the graph editing (with no reboot of the graph structure editor required).

Text analysis and synthesis subsystem is designed to solve the following tasks:

– textual information representation analysis;

– textual information representation synthesis.

Input information for textual information representation analysis consists of:

– description of graph structure of information, according to textual representation of which the graph structure representation must be constructed;

– textual information representation.

Information representation in a form of graph structure is output information.

Input information for textual information representation synthesis consists of:

– description of graph structure of information for which it is necessary to synthesize textual representation according to its representation in a form of graph structure;

– information representation in a form of graph structure.

Textual information representation is output information. As an alternative solution for synthesis task can also be underdefined textual representation, in which not all the words are part of a concrete syntax. Some concepts from graph structure description present in graph information representation can be underdefined. In this case, the target text will contain synthesized names of these underdefined concepts.

Graph structure generator is used to get the graph structure of target information on the basis of the graph structure of source information and structure mapping description.

Input information comprises:

– source information representation in a form of graph structure;

– mapping of source information graph description on target information graph description.

Target information representation in a form of graph structure is output information.

The tool solves the following tasks:

– source and target graph structures navigation;

– search for necessary match: the match for each concept of source graph structure is taken from mapping description; it describes a set of elements (possibly empty) belonging to source information graph structure description and corresponding to the given concept;

– match description interpretation and, as a result of interpretation, in case a set of elements of the target graph structure is not empty, synthesis of the construction described in the match and consisting of these elements.

Graph Structure Transformation Model Based on Mapping Description

Graph structure transformation model based on mapping description *GTM* is a pair (*GDM, MDM*), where *GDM* is a graph structure description model, *MDM* is structured mapping description model.

GDM graph structure description model = (*Concepts, Relations, Attributes, Axiom_Concept, Id_Concept, Const_Concept, Syntax_Restrictions*)

Concepts = $\{Concept_i\}_{i=1}^{conceptscount}$ – finite nonempty set of concepts. Each *Concept* is described by its unique name. Concept name is a nonempty sequence of characters and identifies a certain class of objects of a described domain.

Relations = $\{Relation_i\}_{i=0}^{relationscount}$ – finite, possibly empty set, of relations.

Each $Relation_i$ is a directed binary relation (arc) connecting two concepts which is described as: $Relation_i$ = (*Relation_Name*, *Begin_Concept*, *End_Concept*).

Relation_Name – name of relation that is nonempty sequence of characters.

Begin_Concept – name of concept with a outgoing arc – beginning concept of a relation. *Begin_Concept* $\in$ *Concepts*.

End_Concept – name of concept with an entering arc – ending concept of a relation. *End_Concept* $\in$ *Concepts*.

Attributes = $\{Attribute_i\}_{i=0}^{attributescount}$ – finite, possibly empty, set of attributes. Each $Attribute_i$ is a certain property of a concept and is described as: $Attribute_i$ = (*Attribute_Name*, *Attribute_Argument*, *Attribute_Value*).

Attribute_Name – name of attribute that is nonempty sequence of characters.

Attribute_Argument = (*Attribute_Argument_Type*, *Attribute_Argument_Value*) – attribute argument is described by its type – *Attribute_Argument_Type,* and value – *Attribute_Argument_Value.*

Attribute_Argument_Type = {"Concept", "Relation", "Identifier", "Constant"}.

Attribute_Argument_Value $\in$ *Concepts* $\cup$ *Relations* $\cup$ *Identifiers* $\cup$ *Constants*.

Identifiers – finite, possibly empty, set of all identifiers present in the information representation.

Constants – finite, possibly empty, set of all constants and constant values present in the information representation.

Attribute argument can be a concept (*Attribute_Argument_Type* = "Concept"), a relation (*Attribute_Argument_Type* = "Relation"), an identifier (*Attribute_Argument_Type* = "Identifier") or a constant (*Attribute_Argument_Type* = "Constant").

Attribute_Value = (*Attribute_Value_Type*, *Attribute_Value_Value*) – attribute value is described by its type – *Attribute_Value_Type* and value – *Attribute_Value_Value*.

Attribute_Value_Type = {"Concept", "Identifier", "Constant", "String", "Integer", "Real", "Boolean"}.

Attribute_Value_Value $\in$ *Concepts* $\cup$ *Relations* $\cup$ *Identifiers* $\cup$ *Constants* $\cup$ *String* $\cup$ *Integer* $\cup$ *Real* $\cup$ *Boolean*.

String – set of strings.

Integer – set of integers.

Real – set of real numbers.

Boolean – set {True, False}.

Attribute value can be a concept (*Attribute_Value_Type* = "Concept"), a relation (*Attribute_Argument_Type* = "Relation"), an identifier (*Attribute_Value_Type* = "Identifier"), a constant (*Attribute_Value_Type* = "Constant"), a sequence of characters interpreted as a string constant (*Attribute_Value_Type* = "String"), an integer from a set of integers (*Attribute_Value_Type* = "Integer"), a real from a set of real numbers (*Attribute_Value_Type* = "Real") or by an element of a set {True, False} (*Attribute_Value_Type* = "Boolean").

Axiom_Concept – a unique concept representing axiom in graph structure information description in terms of which no other concept can be defined, i.e. it has no entering arcs. *Axiom_Concept* ∈ *Concepts*.

Id_Concept – a unique concept representing identifier in graph structure information description that cannot be defined in terms of other concepts, i.e. it is a terminal concept. *Id_Concept* ∈ *Concepts*.

Const_Concept – a unique concept representing constant in graph structure information description that cannot be defined in terms of other concepts, i.e. it is a terminal concept. *Const_Concept* ∈ *Concepts*.

For example, let there be chosen the Milan language description is chosen as information. Graph structure that describes the Milan language can look like it is shown on Fig.5.

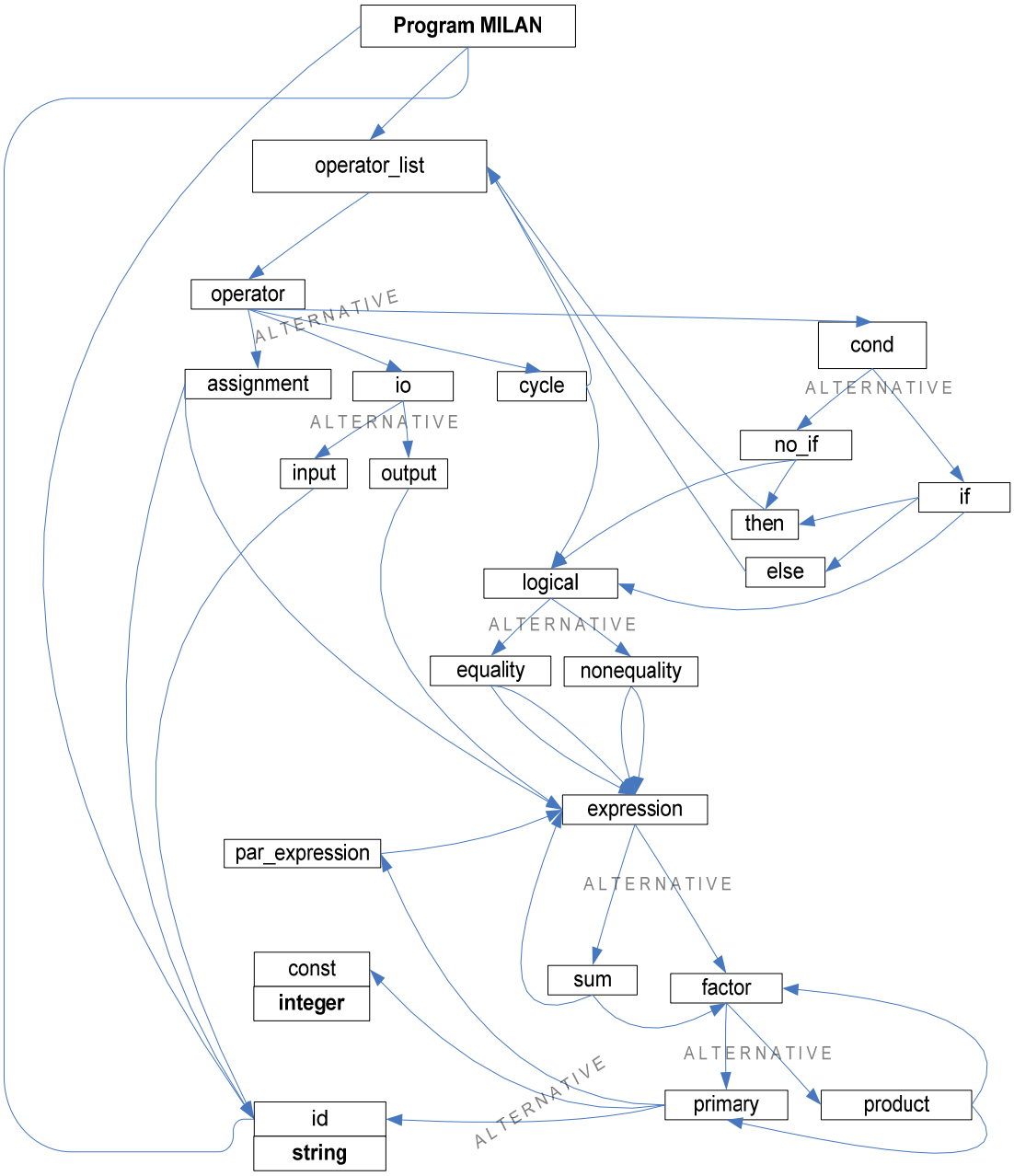

Fig.5. Example of graph structure for the Milan language description.

Program representation in the Milan language in a form of graph structure can look like it is shown on Fig. 6 below. Textual representation of the program is also shown on the figure.

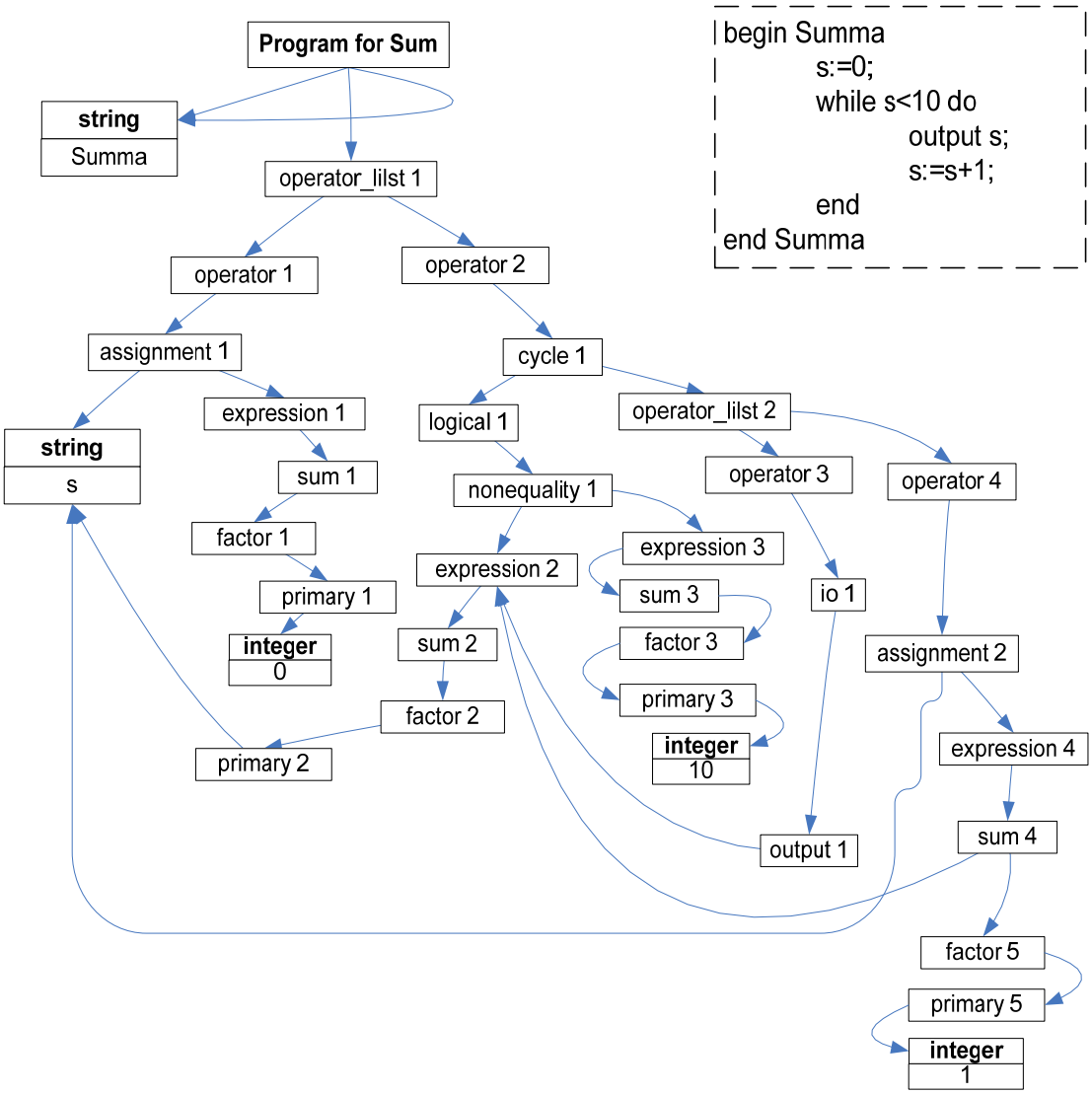

Fig. 6. Example of a program in the Milan language in the form of graph structure.

Syntax_Restrictions = (*Lexica*, *Syntax*) – description of syntax restrictions. It can be absent if it is not necessary to work with textual information representation.

Lexica = $\{<Lexem\_Type_i, Definition_i>\}_{i=1}^{5}$

Syntax = $\{<Concept_i, Definition_i>\}_{i=1}^{conceptscount-2}$

$Lexem\_Type_i \in$ {"Identifier", "Integer", "Real", "String constant", "String constant limiter"}.

$Concept_i \in Concepts \setminus \{ Id\_Concept, Const\_Concept\}$

$Definition_i$ – string of characters that includes metacharacters and represents lexical or syntax restriction for a concrete concept or a type of lexeme.

The syntax restriction model given above has its representation in terms of which the user can formally describe a language of textual information representation. We will further consider the general structure to be used for syntax restrictions representation in any context-free language (CF language). The model of CF language syntax restrictions is shown on Fig.7 below.

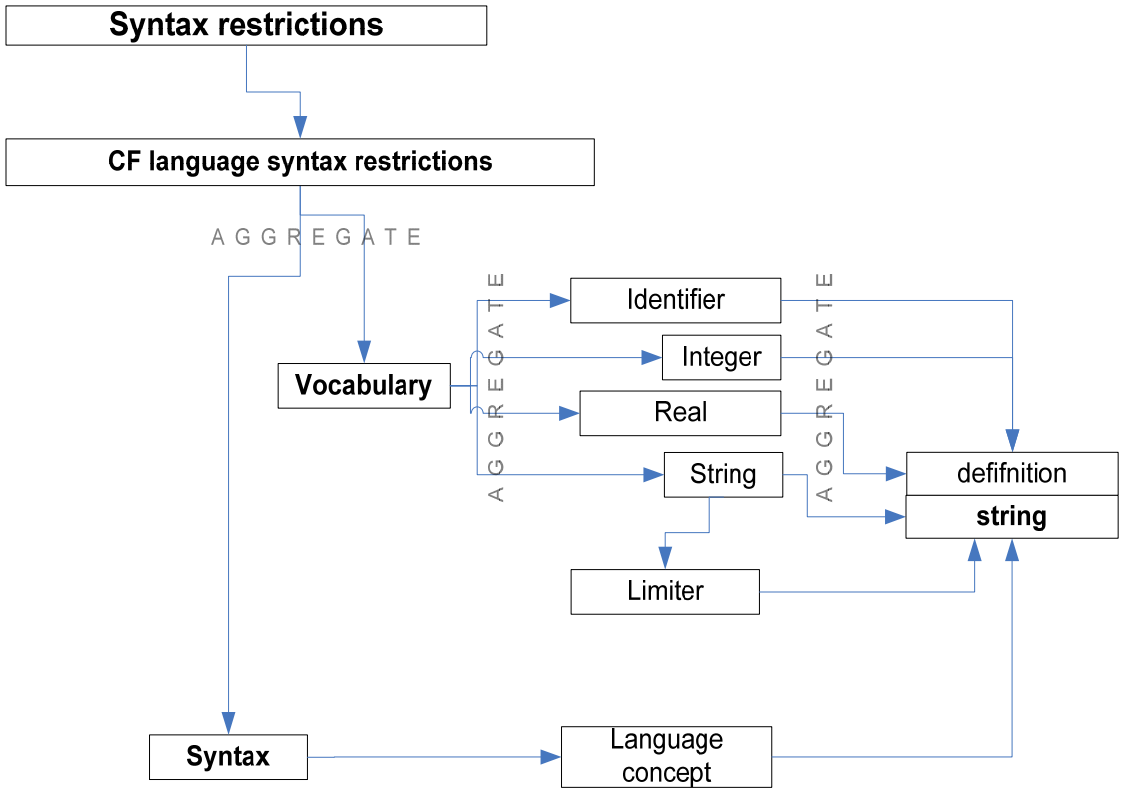

Fig. 7. Syntax restriction model.

Apart from syntax definitions, language syntax restrictions contain lexical definitions as well. For lexis and syntax, there can be a unique restriction specified for each element.

Syntax definition is a definition of a comprehensive concept of graph structure given in terms of comprehensive concepts and elements of a concrete syntax of a textual representation language.

Lexical dictionary of a language is defined by a fixed set of vertices corresponding to base data types – integer, real and string and, apart from it, by identifier type, or name.

Definition terminal vertex describes the *string* kind to which lexical and syntax restriction of a concept represented by a parent vertex corresponds. In concept network that describes syntax restrictions for a concrete textual representation language, the terminal vertex will be a vertex corresponding to the given one. The value of this terminal vertex specifies the concrete lexical and syntax restriction for a concrete language concept or a lexeme type.

All the restrictions on the type of CF language lexemes are specified with the help of regular expressions with the use of perl5 notation.

A string representing a syntax definition contains special metacharacters that carry the following meanings:

" " – space is a delimiter character used as a visual aid. Presence or absence of the space do not change the meaning of a writing.

"||" – a metacharacter of concatenation of syntax constructions. Concrete meaning of this metacharacter can be a parameter of synthesis subsystem. For example, it can be settled that syntax constructions in a concrete text would be delimited by the space character.

"?" – question mark. A character of optionality. It means nil or a single entering of an element it follows. This element can be either a concept from graph structure description or a concrete syntax element. Concrete meaning of this metacharacter can be a parameter of synthesis subsystem.

{;} – enumeration metacharacter. Squiggle brackets are used with a language syntax element that can randomly enter the parent element. The first character after squiggle bracket is considered as a delimiter character. Apart from the concatenation character, this character will delimit the iterative elements in a concrete text. If a delimiter character consists of several letters, this character has to be enclosed into double inverted commas. The same refers to a single letter delimiter, for example, {";"operator} instead of {;operator}. Enumeration restriction for iterative elements is accounted for the fact that only a vertex as opposed to a vertex structure can be in squiggle brackets. The string following the delimiter character up to the second bracket is entirely considered as a name of a certain vertex in the graph structure description.

(…|…|…) – option. It provides the possibility to choose one of many available textual synthesis or analysis variants. Optional variants are delimited by a vertical bar – "|". All these elements as it was with those of enumeration are considered as vertices' names from a graph structure description. The string containing an option character cannot contain any other construction outside the brackets referring to the specifying option.

«'» – single inverted commas. They are used to specify elements of semantic network of concepts. Text string enclosed into single inverted commas is considered as a vertex name in the graph structure information description.

«""» – double inverted commas. They are used to specify elements of a concrete language syntax that corresponds to a described syntax restriction.

"\" – backslash. It is used to specify elements of a concrete language syntax that are written is the same letters as service characters. For example, single inverted commas to specify string constants can be written the following way: "\'".

Metacharacters do not embrace the lexical restrictions.

Syntax restrictions for graph structure description in Milan language can look like it is shown below on the Fig.8.

Mapping description model (*MDM*) is a generating model that specifies a range of calculations. Every generating model [Uspensky V.A., Semenov A.L., 1987; Kleshchev A.S., 1986] consists of a language formal specification of calculations for this model (language for writing calculation rules) and universal formulation of this model. The latter is defined by the structure of states of generating process, the method of initial state formation, the method of generating process construction based on the calculation rules and initial state, as well as by the generating process stopping rule.

Any calculation specified in the generating model language defines a set of generating processes of graph structures. Thus, a set of generating processes received as a result of universal formulation performance of the generating model for each concrete calculation is considered as a model of every possible generating process of graph structures.

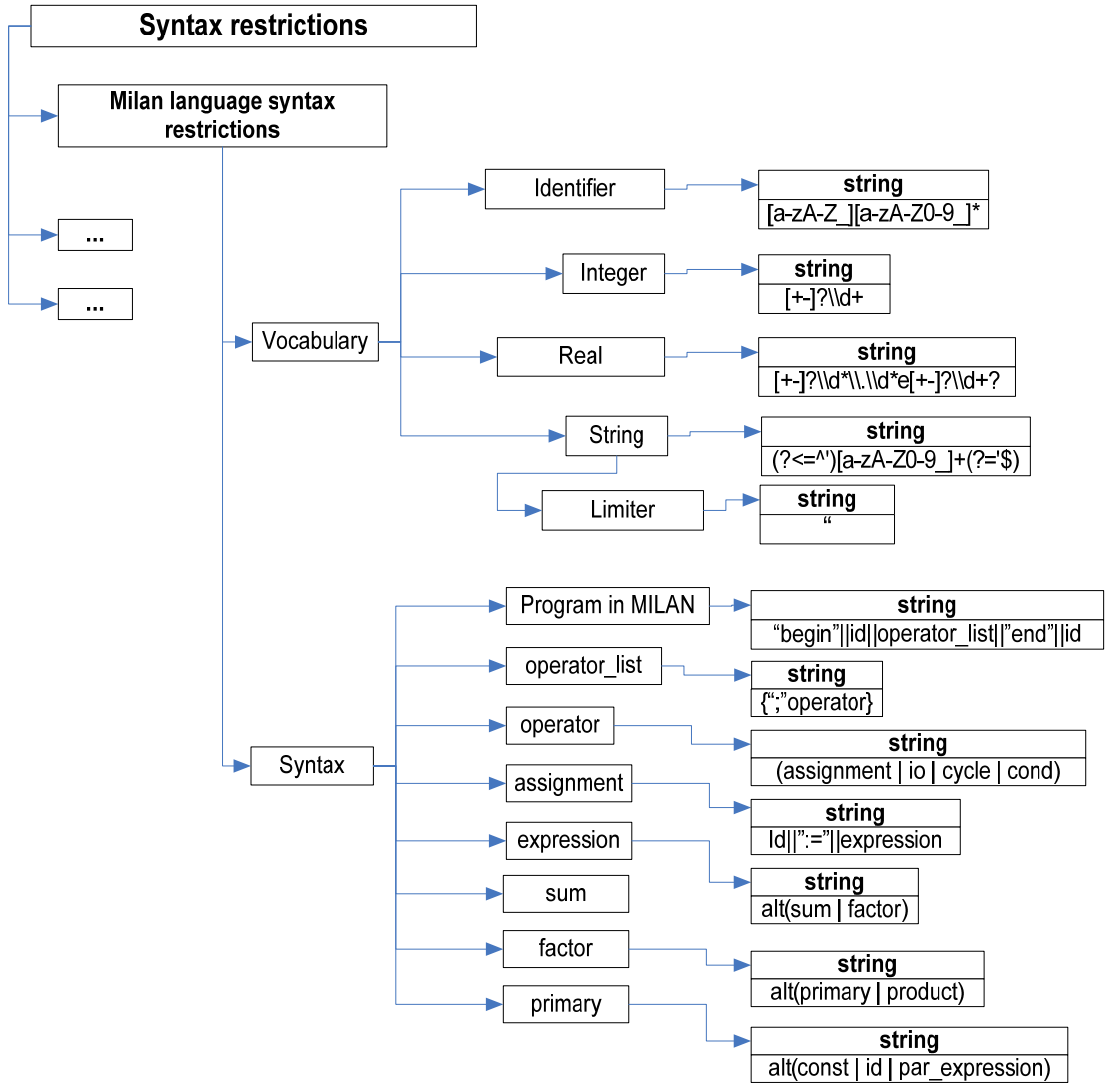

Fig. 8. Example of description of syntax restrictions for the Milan language.

Let us define the language of formal specification of calculation rules for the generating model of graph structures as a group of seven:

(*Concepts*, *Relations*, *Attributes*, *Computable*, *Storable*, *Loadable*, $\{\%\$\}_{i=0}^{n\geq 0}$), where

Concepts – set of concepts,

Relations – set of relations,

Attributes – set of attributes,

Computable = {True, False},

Storable = <*boolean_value*, *alias_name*>,

Loadable = <*boolean_value*, *alias_name*>,

boolean_value ∈ {True, False},

alias_name – charter string that is a pseudonym for the attribute value under which this value should be saved to the hash-table or by which this value should be received from the hash-table.

Each calculation of the generating model is specified by a finite nonempty set of written in this language generating rules of translation of the source graph structure into the target one: $\pi = \{Rule_i\}_{i=1}^{rulescount}$. Each rule $Rule_i$ is of the form of $\alpha \rightarrow \beta$, where $\alpha \in Concepts \cup Relations \cup Attributes$, $\beta \subseteq Concepts \cup Relations \cup (Attributes \times Computable \times Storable \times Loadable)$. $Concepts \cup Relations \cup Attributes \neq \varnothing$ and for any $\alpha_1, \alpha_2 \in Concepts \cup Relations \cup Attributes$, if the rules $\alpha_1 \rightarrow \beta_1$, $\alpha_2 \rightarrow \beta_2$ are included into π, then $\alpha_1 \neq \alpha_2$.

To formulate the rules of the generating model language, there is a structure mapping description language developed. The user can formally describe graph stucture mappings in this language. Each match specifies the composition of concepts, relations and attributes of the target graph structure juxtaposed with the element of the source graph structure.

Specificaiton of abstract syntax of structure mapping description language is given below. The following specification language designations are used [Yershov A.P., Grushetsky V.V., 1977]: **unit** – uniting; [a] – a is not necessary (can be absent); = – concept definition; **:** – position opening, * – possibly indefinite attribute; **ser** – serial component.

Mapping description language = (**ser** mapping description **:** Mapping description)

Description: mapping description language defines a set of mapping descriptions.

Mapping description = (name of graph structure of source information**:** STRING, name of graph structure of target information**:** STRING, matches **:** Matches)

Description: name of graph structure of source information, name of graph structure of target information and matches that describe the connection between them are defined in the description.

Matches = (**ser** match **:** Match description)

Description: This term defines a set of descriptions of matches between the elements of source and target descriptions of graph structures.

Match description = (element of source graph structure**:** STRING, target graph structure construction **:** Target graph structure elements)

Description: In accordance with that, the construction of target graph structure is defined. It has to be juxtaposed to source graph structure element, i.e. a specifically organized set of elements of target graph structure corresponds to a source graph structure element.

Semantics: To interpret the description of the structure of the target graph construction represented in the given match for the specified source graph structure element.

Target graph structure elements = ([concepts **:** Set of concepts], [relations **:** Set of relations], [attributes **:** Set of attributes])

Description: A construction composed of target graph structure elements comprises concepts, relations and attributes specifically connected with each other. Each of these sets can be empty.

Semantics: Make a construction with specified structure of the target graph elements.

Set of concepts = (**ser** concept **:** Concept)

Concept = (name **:** STRING)

Description: *Set of concepts* is a set of target graph structure concepts.

Each made concept is added to the set that contains only those elements that were made by the interpretation of current match description. Before the interpretation of the following match of this set, all of the set elements mush be deleted so that the set becomes empty.

Then each made concept is added to a history of matches between the source graph structure element and a set of target graph structure elements. During the source graph analysis, the history stores the information on made target graph elements, the information on to which source graph element they were juxtaposed. The history provides step-by-step monitoring of the target graph structure generation.

Semantics: To consecutively make concepts a set of names of which is defined by the *Concept* component. For each made concept:

− to add the concept to the set of elements made during the interpretation of current match description;

− to add the source graph element and the concept to the history of matches between the source graph element and a set of elements of the target graph.

Set of relations = (**ser** relation **:** Relation)

Description: *Set of relations* is a set of relations.

Relation = (name **:** STRING, beginning concept of relation **:** Concept, ending concept of relation **:** Concept)

Description: *Relation* is characterized by its name and connects two concepts: one defined by beginning concept of relation selector and the other defined by ending concept of relation selector.

Semantics: If the concept defined by *ending concept of relation* selector has been made by the moment of interpretation of current match description, then

To make a relation with a name defined by *name* selector, specify concept defined by *the beginning concept of relation* selector as the initial concept of the relation, specify the concept defined by *the ending concept of relation* selector as the end concept of the relation.

If the concept defined by *the ending concept of relation* selector has not been made by the moment of interpretation of current match description, then

To add the information on the given relation to the set of not constructed elements. This set stores the history about the elements, structure descriptions of which have been received but they cannot be made because not all of the required structure components of this element have been made.

If during the interpretation of another match the concept defined by *the ending concept of relation* selector appears in a set of elements made as a result of this match interpretation, then this concept is specified as the end concept of this relation after which the relation construction begins. After the relation is ready, this concept is deleted from the set of nonconstructed elements.

Set of attributes = (**ser** attribute **:** Attribute)

Description: *Set of attributes* is a set of attributes.

Attribute = (name **:** STRING, computable **:** LOG, caching value **:** Caching value, value cached **:** Value cached, argument **:** Attribute argument, value **:** Attribute value)

Description: *Attribute* is characterized by its name, argument, on which it is defined, and value it can get. It can be computable and noncomputable. Computable property determines the source of the attribute value: it is either the value of terminal element of the analyzed source graph structure or the value specified during the mapping description. Moreover, attribute value can be saved to the hash-table under a specified pseudonym which is defined by state of the *Caching value* component, and attribute value can be received in the hash-table by entering a specified pseudonym which is defined by the state of the *Value cached* component.

Then, on the basis of the specified information about the attribute value type, the following attributions can be implemented:

If the value type is an identifier or a constant, then attribute value is to be searched among the elements of identifier and constant tables in the source graph structure. If the element with determined value is found, it is assigned to attribute, if it is not, then a constant or an identifier are made, the information on the element is put into a correspondent table and then it is assigned to attribute as its value;

If the value type is a concept or a relation, then attribute value is received as a result of the search among already known (generated) elements of a target graph structure fragment;

If the value type is a string, integer, real or a Boolean value, then the attribute value is assigned during the mapping description, or attribute can get a value of the first found terminal source graph structure element extracted during the search among its elements.

Semantics: To make an attribute with a name defined by *name* selector the argument of which is defined by *argument* selector. To assign *Attribute value* component defined by *value* selector to the attribute as its value.

If the attribute value defined by *computable* selector is *true,* then if the value of *Value cached* component of the attribute defined by *cached* selector is *true,* then take the attribute value from the hash-table by the pseudonym the value of which is defined by *pseudonym* selector.

Otherwise attribute value defined by *value* selector is to be taken from the source graph structure, as the first found value of terminal element of program, therefore at the stage of mapping description it is not necessary to specify this value. If attribute value will be specified during the mapping description and it will be indicated that attribute is computable then the specified value will be disregarded.

Otherwise, if the value defined by *computable – false*, then the attribute receives the value specified during the mapping description. If the attribute value is not specified during the mapping description and it is indicated that the attribute is noncomputable, then its value is considered *empty*.

If the value of *Caching value* component of the given attribute defined by *caching* selector is *true*, then save attribute value to the hash-table under the pseudonym the value of which is defined by *pseudonym* selector.

Caching value = (caching**:** LOG, pseudonym **:** STRING)

Description: *Caching value* is a pair the first element of which is boolean *true* or *false* value indicating whether there is a need to save attribute value in the hash-table. The second element is a string of characters that determines pseudonym under which a value must be saved. If the value of the first element is *false,* then the second is disregarded.

Value cached = (cached **:** LOG, pseudonym **:** STRING)

Description: *Value cached* is a pair the first element of which is boolean *true* or *false* value indicating whether there is a need to save attribute value in the hash-table. The second element is a string of characters that determines pseudonym under which a value must be received. If the value of the first element is *false,* then the second is disregarded.

Attribute argument = (type **:** Argument type, value **:** STRING)

Description: *Attribute argument* is a pair (type, value). The type of attribute argument defined by the *type* selector indicates the way of interpreting attribute argument proper, which is defined by *value* selector and determines the acceptable domain for the attribute argument.

Argument type = **unit** (*Concept*, *Relation*, *Identifier*, *Constant*)

Description: *Argument type* can be one of the mentioned.

Argument value = (type **:** Value type, [value **:** STRING])

Description: *Argument Value* is a pair (type, value). The type of attribute value defined by the *type* selector indicates the way of interpreting attribute value, which is defined by *value* selector, and determines the acceptable domain for the attribute.

Value type = **unit** (*String*, *Integer*, *Real*, *Boolean value*, *Concept, Relation*, *Identifier, Constant*)

Description: *Value type* can be one of the mentioned.

Let us fix the source and target information and provide a fragment of a concrete mapping description. On fig. 9, 10 and 11 there is the fragment of the Milan language (source information) mapping description on the Language of structure program models (target information) [Artemieva I.L. et al, 2002; Artemieva I.L. et al, 2003]. The fragment is presented in terms of the mapping description language. On the figures there are matches between "expression", "cycle", "sum" concepts of the Milan programming language and their correspondent constructions in the structure program model language.

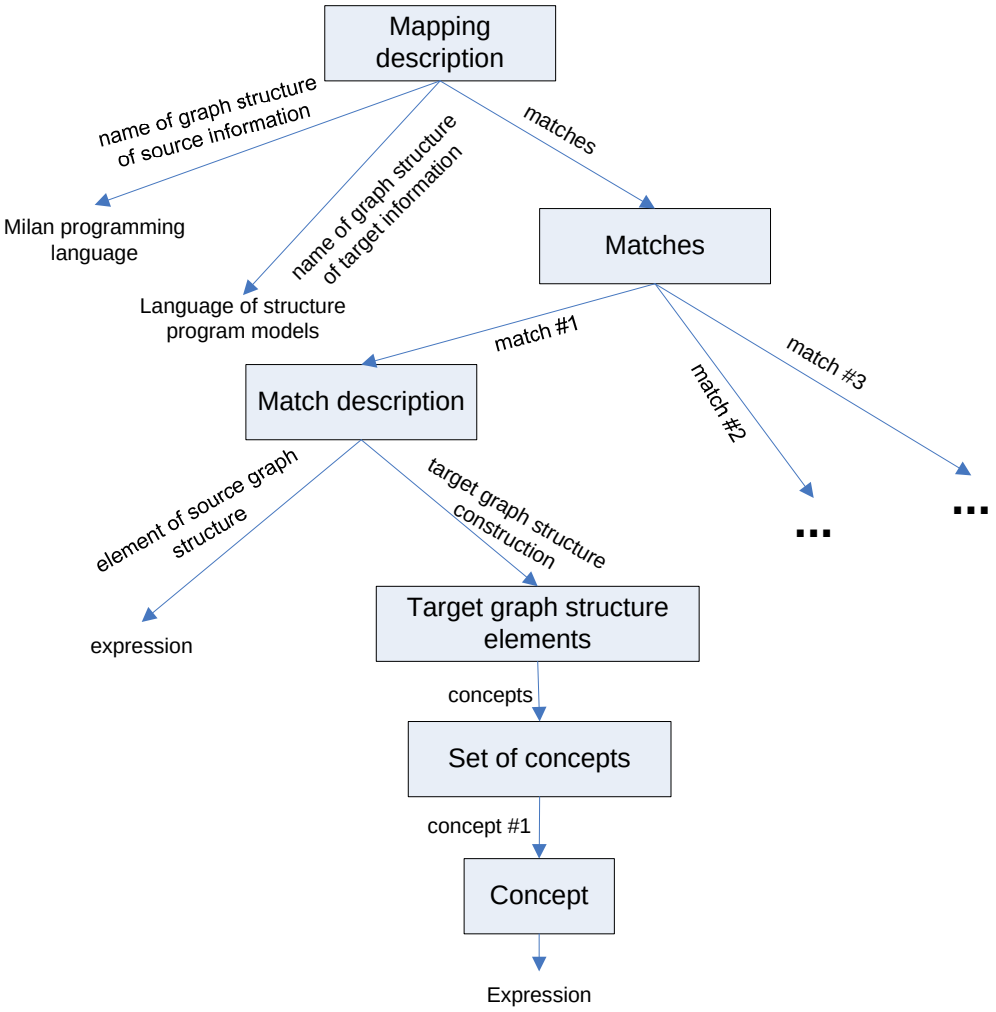

Fig. 9. Fragment of Milan language mapping description on structure program model language. Correspondence description for the *expression* concept

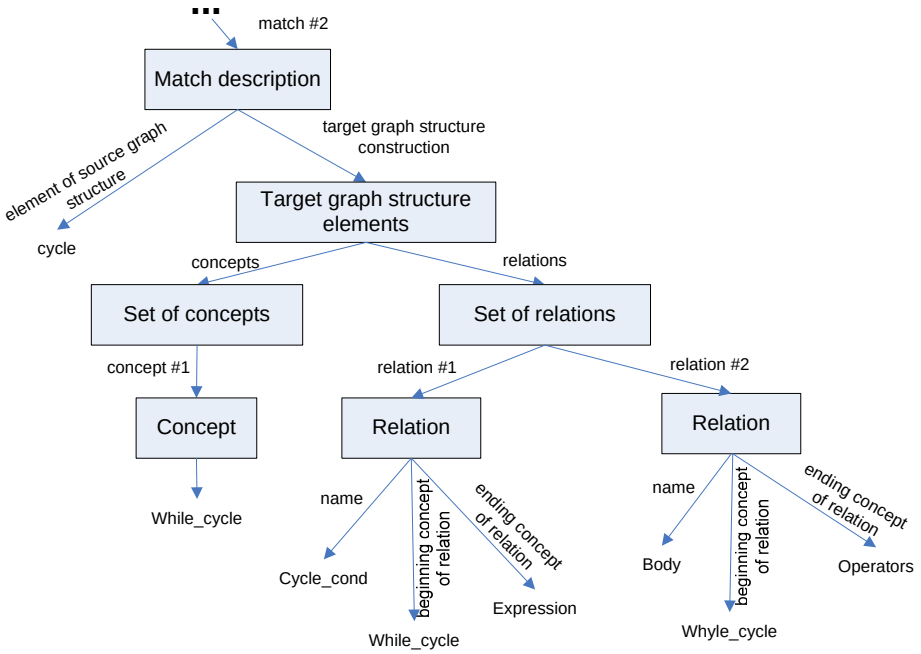

Fig. 10. Fragment of Milan language mapping description on structure program model language (continuation). Correspondence description for the *cycle* concept

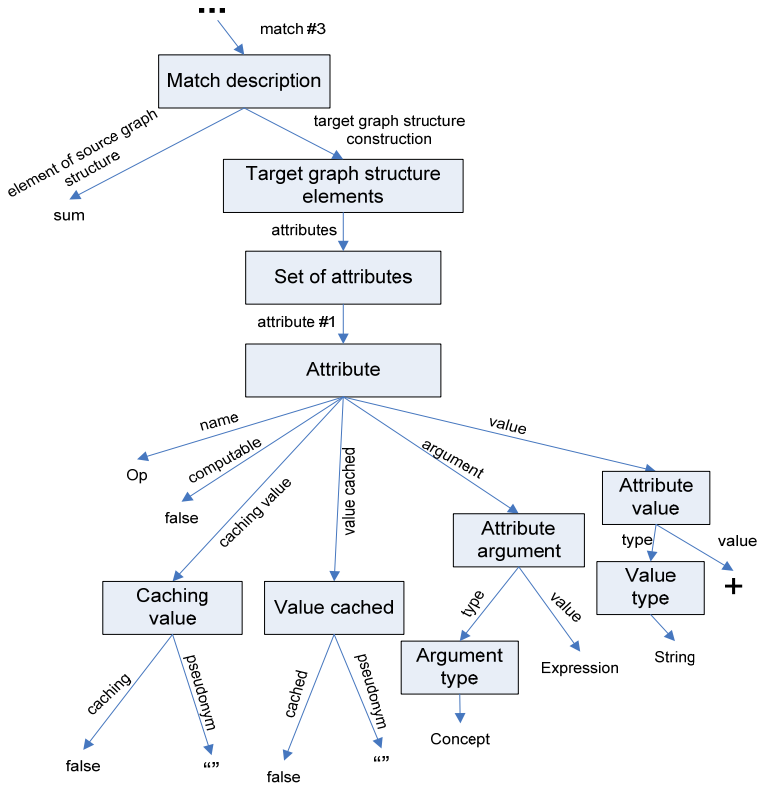

Fig. 11. Fragment of Milan language mapping description on structure program model language (continuation). Correspondence description for the *sum* concept

Conclusion

The paper develops and illustrates an approach to transformation of information represented as graph structures. It also provides the conceptual scheme and general architecture of information transformation system. It offers models describing graph structures and mappings which are specifications for graph transformation.

This approach is implemented in the prototype that ensures program translation from one procedural programming language into another. The prototype supports such languages as Pascal, C, and languages of structure program models developed within the framework of the program transformation system in the specialized bank of knowledge about program transformation. The prototype is implemented in the Java programming environment in the IntelliJ Idea application development environment using resources of the multipurpose knowledge bank (http://mpkbank2.dvo.ru/mpkbank/).

Bibliography

[Artemieva I.L. et al, 2002] Artemieva I.L., Knyazeva M.A., Kupnevich O.A. A model of a domain ontology for "Optimization of sequential computer programs". Terms for optimization process description. In 3 parts. // Scientific & Technical Information. Part 1: 2002. №12: 23-28. (In Russian).

[Artemieva I.L. et al, 2003] Artemieva I.L., Knyazeva M.A., Kupnevich O.A. A model of a domain ontology for "Optimization of sequential computer programs". Terms for optimization process description. In 3 parts. // Scientific & Technical Information. Part 2: 2003. №1: 22-29. (In Russian).

[Jean-Luc Hainaut, 2005] Jean-Luc Hainaut, Transformation-Based Database Engineering, Transformation of knowledge, information and data: Theory and Applications / Patrick van Bommel, Information Science Publishing (2005), pp. 1–28.

[Kleshchev A.S., 1986] Kleshchev A.S. Semantic generating models. The general point of view on frames and productions in expert systems: Preprint of IACP FESC AS USSR: Vladivostok, 1986. – 39 p.

[Partsch, H., Steinbrüggen, R., 1983] Partsch, H., Steinbrüggen, R. Program transformation systems. Computing Surveys, 15(3), 1983.

[Shteinberg B.Ya., 2004] Shteinberg B.Ya. Open parallelizing system // Mathematical methods of parallelizing of recurrent loops for upercomputers with parallel memory - Rostov-on-Don: Rostov University, 2004.- P. 166-182. (In Russian).

[Uspensky V.A., Semenov A.L., 1987] Uspensky V.A., Semenov A.L. Theory of algorithms: main ideas and applications. – M.: Nauka. Main office of physico-mathematical literature,1987. – (Programmer's library). – 288 p. (In Russian).

[van Deursen, A. et al, 2000] van Deursen, A., P. Klint and J. Visser, Domain-specific languages: an annotated bibliography, SIGPLAN Not. 35 (2000), pp. 26–36.

[Yershov A.P., Grushetsky V.V., 1977] Yershov A.P., Grushetsky V.V. Realization-oriented method of description of algorithmic languages. Preprint, Computational Center of Siberian Branch of Academy of Sciences of the USSR, Novosibirsk, 1977. (In Russian).

Authors' Information

Margarita Knyazeva – senior researcher, Institute of Automation & Control Processes, Far Eastern Branch of the Russian Academy of Sciences, 5 Radio Street, Vladivostok, 690041, Russia; e-mail: maknyazeva@mail.ru
Major Fields of Scientific Research: Knowledge based problem oriented systems, Software technologies

***Vadim Timchenko** – engineer-programmer, Institute of Automation & Control Processes, Far Eastern Branch of the Russian Academy of Sciences, 5 Radio Street, Vladivostok, 690041, Russia; e-mail: rakot2k@mail.ru*
Major Fields of Scientific Research: Software technologies, Information systems and data bases