Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 137 (2005) 29–41

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

On the Role of Metadata in Visual Language Reuse and Reverse Engineering – An Industrial Case

Mika Karaila[1]

Energy & Process Automation, Research & Technology Department
Metso Automation Inc.
P.O.Box 237 FIN-33101, Tampere, Finland

Tarja Systä[2]

Institute of Software Systems
Tampere University of Technology
P.O.Box 553, FIN-33101, Tampere, Finland

Abstract

Collecting metadata on a family of programs is useful not only for generating statistical data on the programs but also for future re-engineering and reuse purposes. In this paper we discuss an industrial case where a project library is used to store visual programs and a database to store the metadata on these programs. The visual language in question is a domain-specific language, Function Block Language (FBL) that is used in Metso Automation for writing automation control programs. For reuse, program analysis and re-engineering activities and various data and program analysis methods are applied to study the FBL programs. Metadata stored in a database is used to provide advanced program analysis support; from the large amount of programs, the metadata allows focusing the analysis to certain kinds of programs. In this paper, we discuss the role and usage of the metadata in program analysis techniques applied to FBL programs.

Keywords: Visual languages, domain-specific languages, metadata-driven program analysis, reuse, reverse engineering

[1] Email: mika.karaila@metso.com
[2] Email: tarja.systa@tut.fi

doi:10.1016/j.entcs.2005.07.003

1 Introduction

The largest portions of software life-cycle costs are due to maintenance activities [2,5,9]. Various reverse engineering techniques can be used to support software comprehension and visualization that, in turn, support maintenance. If available, metadata on the programs can be used to guide the reverse engineering and visualization activities. In this paper, we discuss metadata-driven program analysis techniques used for maintaining visual programs.

A visual language manipulates visual information, supports visual interaction, or allows programming with visual expressions [4]. Further, visual programming languages can be defined as languages for programming using visual expressions [3]. The visual programming language under study, called Function Block Language (FBL), is a domain-specific language used at Metso since 1988. FBL is used in paper machine controls, power plants, and marine automation and in other various continuous real-time control systems.

Along the years, the language itself has been modified and further developed. The programs written are stored in project library archives. In the storing process metadata is read from the programs and stored into the database. Metadata is used as search conditions, according to which desired programs can be downloaded from local project libraries. For instance, the engineer can look for existing solutions that could be reused for his current purposes. Stored programs and metadata together allow detailed analysis of programs themselves, relations of different programs, and statistical reports on the usage of FBL. Due to the analysis techniques developed and the syntax of the language, high reusability of programs and their parts have been achieved. The reverse engineering and visualization techniques are discussed in [8] in more detail. In this paper we focus on the metadata itself, describe the role of meta-models used for particular problem domains, discuss the iterative design process, and show statistical data collected on FBL usage along the years.

We first give a brief overview to FBL and discuss the language meta-models and their role in Section 2. We then present and discuss metadata-driven methods used for designing, maintaining, and reverse engineering FBL programs in Section 3. These benefits include, e.g., high reusability and improved program quality. The metrics used and types of metadata collected to support program analysis are introduced in Section 4. We also give some details of metadata statistics and discuss the benefits of using metadata to support program analysis activities. Finally, concluding remarks are given in Section 5.

2 Function Block Language (FBL)

2.1 An overview

With FBL, engineers can design visual programs that connect physical electrical measurement signals to program *parameters*. Those parameters are referenced by symbols containing other values needed as attributes. By connecting these symbols the engineer can create algorithms to control and run actuators, such as valves, motors, and pumps, in the process.

The visual notation of FBL consists of symbols and lines connecting symbols. In FBL, symbols represent advanced functions. The core symbols of FBL, *function blocks*, are subroutines running specific functions to control a process. A simple example could contain an input symbol to read a water level measurement. That input symbol could be connected to a function block symbol representing a subroutine for calculating and keeping the level. Then the function block symbol is connected to an output symbol that will modify a control valve position.

The parameters can be internal (private) or public. An internal parameter has its own local name that is not visible outside the program module. A public parameter can be an interface port with a local name or a direct access port with a globally unique name. The external data point is a reference to data that is located somewhere else. In distributed control systems, calculation is distributed for multiple processors. Therefore, if a parameter value is needed from another module, the engineer has to add an external data point symbol to the program. By using this symbol data is actually transferred (if needed) from another processor to local memory. From these elements, the engineer can, for instance, build visual programs that control some equipment in a factory that is running the process. These processes are continuous and controlled in real-time.

In addition to function blocks, FBL programs may contain the following elements: *port symbols* (also called *Publishers*) for other programs to access function blocks and their values, *external data point symbols* for subscribing data published by ports, *external module symbols* to represent external program modules, and *I/O module symbols* to represent physical input and output connections.

FBL is an extensible language that allows developers to extend the language by adding new graphical symbols to it, even without adding any new code in the programming environment. This and other features of FBL are discussed in more detail in [6,7].

Mika Karaila developed the first version of FBL in 1988 (Master of Science work). Since then, FBL has been further enhanced at Metso Automation. The metadata-driven program analysis methods presented in this paper have been developed to understand how the language has been used over 16 years and to give tools for further enhancing the language. One of the main advantages of FBL is high reusability [7]. This is crucial for reducing the costs of engineering work. As in traditional reverse engineering methods, higher-level models are constructed from existing programs. These models can be reused and new code can be generated from them. In fact, the high reusability and the support provided by the programming environment allow engineers to program at a high level of abstraction. The metadata-driven program analysis techniques applied are discussed in the sequel.

2.2 Meta-Models

FBL user interface domain is defined by those graphical objects that are needed to implement a visual language. The FBL itself is not a meta-language, but the symbols used contain enough meta-information to define new function block skeletons. This feature is especially useful for mapping the problem and solution domains. Namely, the real physical world with devices, e.g. pumps and valves, of the problem domain can be represented as appropriate graphical objects in the solution domain, i.e., in FBL programs. This is also useful when explaining the solutions to the customer: the solutions are easy to understand since they use familiar symbols.

A specific meta-language is used to define FBL structures and types based on a so-called *internal meta-model*. This allows FBL to be extended and customized. A *user-level meta-model*, in turn, allows customized graphical objects (appropriate for a specific problem domain) to be defined. FBL internal and user-level meta-models are shown in Figures 1 and 2, respectively. These meta-models will show that the balance between the models implementation and user domain exists. First, the graphical components of FBL programs (namely the user interface domain) are symbols, lines, and text. A symbol can be e.g. an input symbol or a function block symbol. Since FBL is extensible, also new user-defined symbols can be introduced. Lines, in turn, are used to connect symbols. Text elements may occur in various parts of the FBL programs (diagrams). Second, transformations are used to build object trees from the FBL diagrams. These object trees are then used to sort and categorize objects and connections for the code generation. Third, the code itself consists of code fragments and connections used as references to other code fragments. The code generation is a flexible, generic and generative compiler. It can compile new code directly from the new symbol. Thus, the transformation does not require any new code. The symbol contains enough meta-information to be used in transformation.

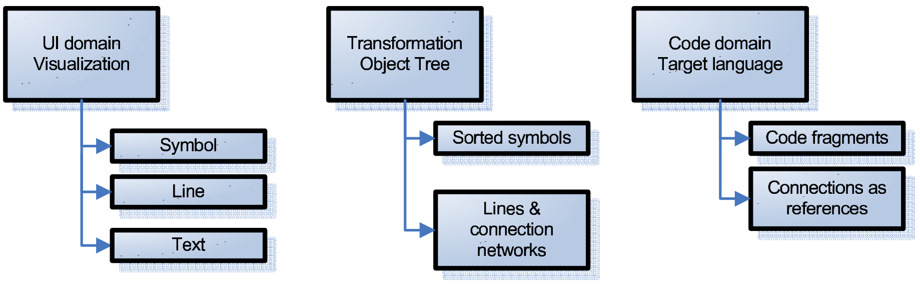

Fig. 1. Internal meta-models

Engineers are working with diagrams and real devices. The corresponding meta-models are shown in Figure 2. Function Diagrams contain symbols that are representing IO cards and function blocks. IO cards are connected by wires to devices. The function blocks relate with functional requirements.

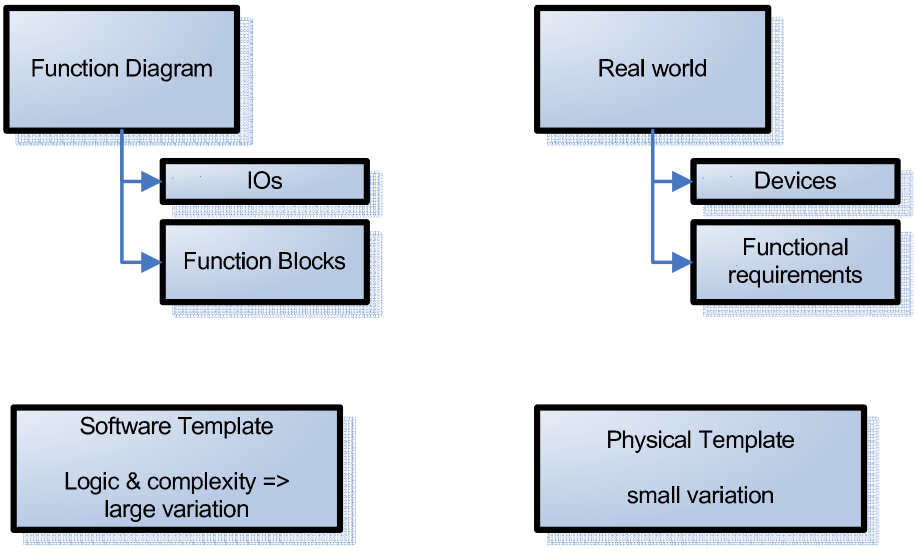

Fig. 2. User-level meta-models

Multiple meta-models are used for FBL programming. Information from one meta-model is transformed to another. Because the transformations use meta-model information, the requirement of dynamic and extendable systems is fulfilled. The small meta-models are simple and easy to understand and transformations can be used in a generic way.

Function groups are constructed for different processes to compare control structures and patterns used. As an example, some diagrams are built in pairs and have "fixed" connections between them. This is a one found pattern in our domain. Another example is the real world devices that are modeled as own templates. These "physical" templates are more constant and the variation between them is smaller than variation between “software” templates. The “software” templates contain more logic and both the variation and the number of these templates is bigger. As an example indicating this, there can be two almost similar devices with same connections but the internal functionality can be totally different. In this case the “physical” template can be same for both devices, but the “software” template is different.

The division to “physical” and “software” template isolates the variation to the “software” templates. It is better to have modifications only in that template, because the real world is not changing as often as features implemented by “software” template. The “physical” template contains always those electrical connections to devices that are “fixed”. The connections are not modified or they are not changing after the device is constructed and assembled.

3 Metadata-driven reverse engineering and design

Reverse engineering and program analysis activities aim at constructing representations and models of the subject software systems in another form or at a higher level of abstraction [1]. The new representations are constructed after identifying the system s components and their interrelations.

Clustering in traditional reverse engineering methods can be constructed, for instance, by taking advantage of the syntax of the programming language used, using software product metrics to identify highly cohesive clusters, or using existing software architecture models and mapping them with the lower level details. Software product metrics used for identifying subsystems typically measure inter couplings and intra cohesion of the sets of software elements. In the approach presented in the paper, the syntax of the language is used to construct high-level models for the programs. This together with the usage of metadata stored and metrics values calculated form the basis for the program analysis techniques to be discussed next.

3.1 Design process

The design process involves multiple domains. The problem domain is the real physical world with devices, e.g. pumps, valves, temperature / flow measurements etc., which are modeled in a solution domain with FBL language. The FBL diagrams (FBL programs) designed and constructed by the engineer can be further connected to each other. The diagrams that belong together in a solution space are also close to each other in the real world as devices. So, the solution directly reflects the real world situation.

An iterative program analysis process is applied for analyzing FBL programs. The meta-model defines the language used in a particular problem domain. The customers of Metso Automation have their own equipment and factory control problems. We provide solutions to these problems, implemented using FBL. In FBL, symbols used are equal to terms used by the customers. Various program analysis techniques can then be used to get higher-level views to these programs (solutions). These high-level views can then be compared with the user domain (problems).

One of the key things in the program analysis process is that problems and their solutions can be analyzed and compared at the same high level of abstraction. First, for a certain problem there is an existing solution, or in fact, multiple solutions in practice. Second, we can produce other high-level overviews to the solutions for comparing these solutions. These views can be again compared with existing problems and reused efficiently: they can be used directly as such without going into a detailed solution. This will challenge the users to find several possible solutions. In practice, the program analysis process is iterative. The engineer selects one or more possible solutions and uses them. After a while, more specific and detailed information and requirements are available. The solution template is then modified and tuned according to the new requirements. In the testing phase some changes are most probably still made.

The design process applied is depicted in Figure 3. The design typically starts by searching existing solutions. The solutions are stored in a metadata repository. To increase program reuse, so called *templates* can be searched. A template provides an applicable solution when the structure of the program is concerned, but requires some changes in parameter values. Templates are further discussed in Section 3.2. Next the template found is reused. In many cases the template found needs some modifications and re-engineering to function and match to the real world requirements better. With aid of reverse engineering activities, the real world situation is compared withto already built solutions. The reverse engineered documentation also helps the engineer to understand the actual FBL program. After the modifications are applied a program instance is composed, which in turn is stored in a program library. Before storing the new program, an abstraction operation is applied to it. This may yield to construction of a new program template that can then be reused in the future. To find out if such template already exists, it is first compared with the existing templates. This is not yet an automated process; rather the user needs to decide if the new template is really useful and good enough to be available in own template folder. The design process itself thus contains a feedback mechanism, which keeps the project archives and metadata repositories up to date.

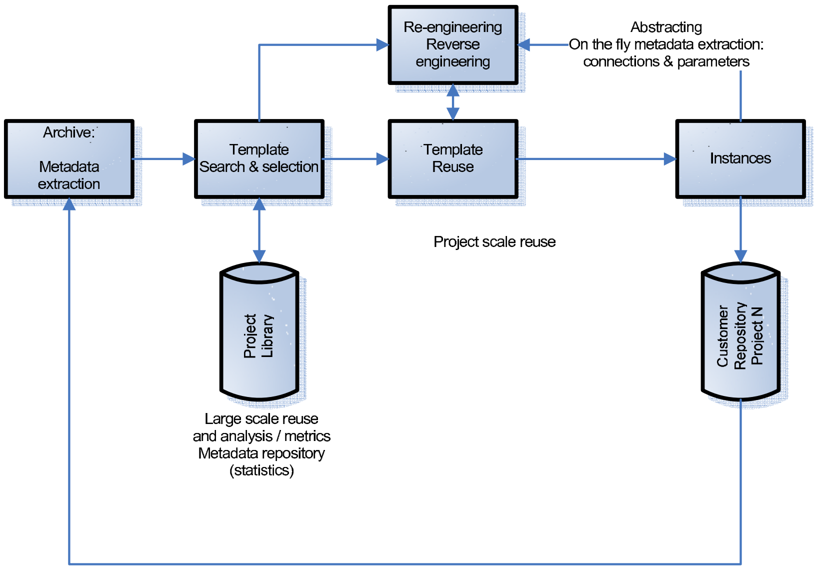
Re-engineering
Reverse engineering
Abstracting
On the fly metadata extraction: connections & parameters
Archive:
Metadata extraction
Template
Search & selection
Template
Reuse
Instances
Project scale reuse
Project Library
Customer Repository Project N
Large scale reuse and analysis / metrics Metadata repository (statistics)

Fig. 3. The design process.

3.2 *Managing large FBL programs*

For an overview, the details inside individual FBL elements (e.g., function blocks or I/O modules) are seldom interesting. The relevant components and relations, e.g. for reuse purposes, are function blocks, interfaces, and global parameters.

For generating an abstract view from Function Block Diagram, the details of the program are filtered out and only the input and output symbols are preserved. The abstracted program is called *Function Group*, indicating that one symbol contains several functions (Function Blocks and IOs). A program visualization function creates new symbols on the fly for each abstracted component. To compare existing solutions, the engineers need few function groups to get the main idea of the principles used in solution.

The levels of reuse are illustrated in Figure 4. The basic components, function blocks, are typically reused multiple times in every function block diagram. One diagram can contain from few to several hundreds of function blocks. In each project, there are some thousands to tens of thousands of diagrams. Thus, the function block instance count is huge compared to the function block diagram count. To handle and understand these function block diagrams we therefore need abstract views, namely function groups. A typical project contains from few to couple of hundreds of function groups.

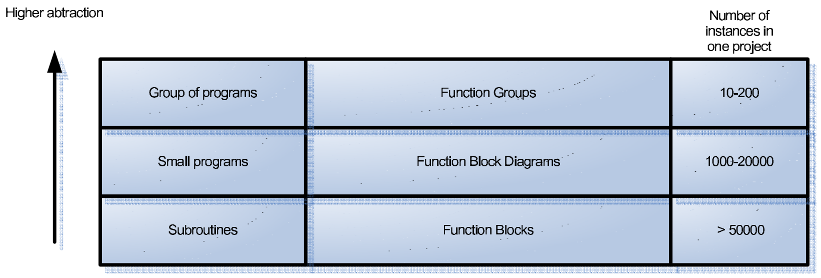

Fig. 4. Reuse levels and object counts.

We will next briefly describe the reverse engineering and program analysis techniques supported by our environment, especially focusing on features that take advantage of metadata. Details of the overall reverse engineering, reuse and maintenance support is discussed in [8].

3.3 Re-engineering and reuse

When re-engineering programs, existing program instances can be changed by extending or modifying them. For instance, new function blocks can be added, parameter values of existing programs can be changed, and connections between function groups can be changed. The engineer can thus create new programs that were first extracted from the database using reverse engineering techniques.

For increasing the degree of reuse and thus decreasing the development times, reusing existing function groups instead of modifying individual programs is preferred. This assumes that the existing function groups are general enough to be usable in various programs. In many cases, the structure of the program itself is reusable but the differences occur in parameter values. For enabling reuse in such cases, a concept of a *template* has been introduced to FBL. Templates are ready-made programs without instance values. They can be used by external module symbols. In other words, templates describe individual parts of process control software, without project specific definitions. Actual application instances are created when project specific data is combined with a template. For a template, the environment generates a program according to given parameter values.

4 Using metrics and metadata for analyzing FBL programs

The project library archives contain all implemented application solutions. Application instances and templates used are stored as DXF-files (Data eXchange Format) in a directory structure corresponding to the project s process hierarchy. These archives are accessible for project engineers by a web interface. These detached project libraries are then bound under a single content management entity. The centralized content management solution stores only the essential application metadata to a content management server and allows the archived files to remain in local project libraries. The stored metadata includes also links to actual application solution files.

4.1 Metadata

Project library archives and a database used to store FBL programs together with metadata allow the extraction of various kinds of statistical data of the programs. When extracting and storing metadata from a particular FBL program, a special program is used. The metadata stored is used as search conditions, according to which desired programs can be downloaded from local project libraries. For instance, the user can look for exact solutions using a specific tool by defining criteria to match a description or a primary function block. Navigating in a process hierarchy is also a way to search FBL programs. This is a common way to find similar problems and existing solutions. All the information that is used in a search / navigation is based on metadata. Metadata includes: a description of the FBL program, a process hierarchy it belongs to, parameters used, and primary function block and I/O types.

Other, more detailed metadata is also available for analysis. That metadata contains more detailed information from each FBL program: function block count, primary function blocks, entity count, I/O count, creation time, modification time.

A rough estimate for the project complexity metric is calculated as an average: number of function blocks divided by the number of I/O connections. When analyzing traditional programming languages, a metric Lines of Code (LOC) is often used. A corresponding metrics used when analyzing FBL programs is Number Of Entities (NOE), where an entity is a symbol, text, a line, or any other graphical object in the diagram.

Even though a lot of metadata is stored on each FBL program, its amount is relatively small compared to the amount of programs (diagrams). As an example, information that is archived is currently roughly from 200 projects. There are about 40 Gb of compressed diagrams, while the amount of metadata is only about 850 Mb. Typically, the access rate is about 2000 searches in a month.

4.2 Metadata-driven analysis

As an example scenario, the engineer can give conditions like description and process area to find certain type of solutions. As a result, he might get a list of a couple of hundreds of diagrams. By using the information provided in this list and other more detailed metadata on the programs, the user can further focus the search to limit matching results. The user can also quickly view the actual visual programs.

Special tools implemented by Karaila and his engineering group support another, less user-dependent way to use metadata to find programs that could be reused. These tools analyze implemented FBL programs. The tools look for basic program structures that are used multiple times in different FBL programs and use the following metadata in its matching algorithm: entity count of a FBL program, primary function block count, and function block count. This is used as a fingerprint to make first level matching faster and easier. The second level is more detailed and makes the actual comparison between the FBL programs.

Calculations are based on metadata stored in the database. Summary groups are sums of loop types and primary function blocks. Project analysis window shows an average number of different IO types from diagrams and the total amount of each IO type. This information can be used to estimate the scope of similar projects. Entity and function block counts and averages give an estimate of the capacity needed (memory and load). Finally, complexity is used later to analyze the project working hours. When selling a new project with similar process & devices, the complexity value can be checked to ensure we should allocate the right amount of hardware (devices) in that kind of project.

4.3 Templates

Using a database for storing FBL programs provides a good source for evaluating and comparing individual FBL programs and their parts. Similarities among several programs that do not reuse same core elements indicate that unnecessary rewriting has been carried out and that reusable components may be needed. By composing such common structures as reusable components (templates), the existing programs become more maintainable. Also, such reusable components are likely to be used when writing new programs. In fact, the analysis tools can be used for developing an optimal set of reusable templates: a compact set of highly reused templates. If the set of templates is too large, their benefits become less obvious and suitable templates are more difficult to be found. Metadata on the usage of templates can be collected and used to conclude such an optimal set of parameters used inside templates to limit parameter variation. In practice, the variation mainly occurs in certain templates only. This is further discussed in Section 5. Limiting parameter variation also limits the total amount of templates, because the most common reason to have a "new" template is to add another parameter into it. The new parameter is needed mainly for a new feature. Also, statistics calculated of the usage / frequency of a template tunes most common templates to be practical and understandable. We can then focus on them to improve quality. This will also decrease the time in design process shown in Figure 3.

In small scale projects templates are used heavily but the same templates appear only few times. Namely, there are a lot of templates that are used only few times. In bigger projects, however, same templates are reused more often. The statistical values show that in 2000 diagram instances there are over 400 templates and a bit over 10% hand-made instances where templates have not been used. Most of the hand-made instances are not FBL-based diagrams. Instead, they are other needed instances, for example user interface modules and diagnostic modules that are not implemented with FBL. Thus, the templates are heavily used in practice.

5 Summary

In this paper we have discussed metadata-driven program analysis techniques used to support re-engineering and reuse of visual FBL programs. FBL is a domain-specific language used in automation industry. FBL is used e.g. in paper machine controls, power plants, and marine automation and in other various continuous real-time control systems. We have especially discussed the role and usage of metadata, characterized the design process, and discussed some statistical values collected on FBL usage along the years.

FBL programs are stored in project library archives. The analysis of these programs can be focused to certain kinds of programs with the aid of metadata stored on these programs.

The core of FBL was implemented in the late 1980s. The software quality and usability has been improved based on internal measurements carried out at Metso and based on feedback from satisfied customers. In the programming environment, there has been a steady evolution and a desire to improve it. Lessons learnt from the long history of using FBL have guided the development of the program analysis techniques and especially helped the FBL environment developers to conclude what kind of metadata is needed and useful for program analysis, reuse, and re-engineering purposes.

The actual experiences of the environment are still being studied. However, the experiences gained so far have shown that FBL and the engineering environment used is a flexible, practical, and well suited for the domain it is designed for, namely, automation industry.

As a part of our future work we aim at further studying the role of FBL meta-models, especially the domain models. The metadata-driven program analysis techniques could be improved if there was a semantic relation between the metadata and the meta-models, namely, if metadata with semantic meaning could be used. This means that information from the real world would be tied with existing metadata. This semantic information could help us e.g. in the problem of recognizing a template for a device automatically.

References

[1] Chikofsky, E. and J. Cross, *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software, **7**, 1, 1990, 13-17.

[2] Erlikh L., Leveraging legacy system dollars for E-business, IEEE IT Pro, 2000, 17-23.

[3] Golin E.J., *Parsing Visual Languages with Picture Layout Diagrams*, Journal of Visual Languages and Computing, 2,4, 1991, 371-393.

[4] Golin E.J. and S. P. Reiss, “The Specification of Visual Language Syntax”, In The *Proc. of IEEE Workshop on Visual Languages*, 1989, 105-110.

[5] Jones T.C., *Estimating Software Costs*, McGraw Hill, 1998.

[6] Karaila M., “Designing a Flexible Visual Language in the Automation Industry”, an unpublished manuscript.

[7] Karaila M. and A. Leppäniemi, “Multi-Agent Based Framework for Large Scale Visual Program Reuse”, In The *Proc. of BASYS*, 2004.

[8] Karaila M. and T. Systä, “Maintenance and Analysis of Visual Programs – An Industrial Case, a manuscript”, In The *Proc. of CSMR 2005*, to appear.

[9] Sneed H., “Encapsulating Legacy Software for Use in Client/Server Systems”, In The *Proc. of WCRE 1996*, 104-119.