8

Model Based Software Production Utilized by Visual Templates

Mika Karaila
Metso Automation Inc
Finland

1. Introduction

In the automation domain programs are written by engineers. Available programming languages are normally standard IEC 61131-3 or vendor specific visual language. Programming requires domain knowledge and programming skills. Reusing programs is often simple copy / clone a working solution. There are different kinds of solutions done to effective produce programs. In Metso Automation application programs are first modeled and second systematically reused. The principles are applicable to be used in other context.

2. Function block language

2.1 Introduction

The visual notation of FBL consists of symbols and lines connecting them. In FBL, symbols represent advanced functions. The core elements of FBL, function blocks, are sub-routines running specific functions to control a process. As an example, measuring the water level in a water tank could be implemented as a function block.

In addition to function blocks, FBL programs may contain port symbols (also called Publishers) for other programs to access function blocks and their values. The function block values are stored in parameters. As an analogy, the role of a function block in FBL is comparable to the role of an object in an object-oriented language. The parameters, which can be internal (private) or public, can, in turn, be compared to member variables. 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.

In addition to function blocks and ports, FBL programs may contain 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. An external data point is a reference to data that is located somewhere else. In distributed control systems, calculations are distributed to 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.

Source: Visual Servoing, Book edited by: Rong-Fong Fung,
ISBN 978-953-307-095-7, pp. 234, April 2010, INTECH, Croatia, downloaded from SCIYO.COM

From the FBL 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.
Visual languages have been extensively studied in the literature (Mohamed, 2000, Burnett 1995, Shu 1988, Pressman 1997). As mentioned earlier, computer programs are usually written using textual languages, but in more sophisticated or domain-specific environments, programming can be done in a visual way, as in LabVIEW (Rahman, 1995). LabVIEW is originated in 1986, while the roots of FBL go back to 1988 (Karaila, 1989). FBL is not a standardized language as IEC 61131-6 language.

2.2 Background

In the late 1980´s the first implementation was done for FBL. The first target was to replace a textual programming language because graphical documentation was already at that time one of the customer's requirements. FBL was successfully taken into use and there were only a few programs that were written in textual format.
One of the most important design goals was to design both the programming environment and FBL for extensibility. This means that developers could easily extend the visual language by adding new graphical symbols to it. Such new symbols, for example, may represent new types in this strongly typed language. In fact, in FBL, users can add new symbols to the language even without adding any new code in the programming environment. The reuse of visual code in an integrated programming environment is powerful and efficient. The same kinds of notifications are done (Debbie, 1995). Developers have implemented an engineering environment that allows extensions and integration of third party tools. Further, new symbol classes or categories can also be added to FBL. This, however, requires modifications to the programming environment. Usability is important to engineering efficiency. For cost effectiveness, using a commercial solution was a good way to share code maintenance costs. As a drawing editor Metso has used commercial CAD program, which can be AutoCAD® Copyright 2009 Autodesk or BricsCAD Copyright 2001-2009 Menhirs NV. Both can be used for that purpose. In this way, developers were able to focus our own work on the application domain instead of graphical editor issues.

2.3 Main design goals and principles

Developers had the following goals in the development of FBL and the programming environment:

- Basic product configuration and a tool for customer projects.
- Both FBL and the programming environment must be flexible and possible to extend because it was known from the beginning that new features are coming/needed every year.
- Maintaining the language should be feasible, and adding new types and functions should be easy.
- Easy to use, because typical users have minimal programming skills.
- Easy to reuse written applications, because customer projects are very similar.
- Third party tools and products should be easy to be integrated with the programming environment.

FBL can be used to program basic automation and advanced quality controls. Metso's engineers can implement different kind of applications with FBL. As the amount of different

sub domains are integrated into FBL, the use of FBL is growing. Our customers will maintain and modify those FBL programs. Customer's people are typically automation engineers. They will come to the FBL training. They are responsible for maintenance and process design. They usually do not have any programming experience. Most of the time goes into environment training and main principles of the automation system. The FBL language itself is not so much used, only a few programs are made during the training that is typically one week long. This is one way to evaluate the learning curve of FBL. There are other studies about advances in data flow programming languages (Johnston, 2004). These indicate the same findings as developers have experienced such as, 'The data flow semantics of visual programming languages are intuitive for non-programmers to understand and thus improve communication between the customer and the developer'.
Design principles of the language are briefly summarized next.

- In the visual drawing, symbols used should represent both data and functionality. There will be an artifact in the system that can be mapped into a symbol. So each symbol will have some meaningful concrete function or element in the system. There will be very direct mapping from the eq. IO card symbol to a program physical IO card that will run a real electrical connection.
- Symbols are for creating communication to transfer signal data. One symbol that contains an output and can be connected by line to another symbol input to represent data-flow. Data-flow will be in this way explicit.
- Layout should be organized so that inputs will be on the left and outputs on the right. There will be immediate visual feedback during testing program values can be visualized.
- All of the above will create a combination that merges algorithm and user interface to one functional entity.

These four strategies: concreteness, directness, explicitness and immediate visual feedback are listed in (Burnett, 1999).

2.4 Basic symbols

Function block language contains thousands of symbols. The following is a categorized list of basic symbols:

- Administration part symbol for defining purpose of the diagram,
- Function part symbol for defining CPU and execution parameters,
- External reference symbol for transferring data outside module,
- Local data symbol for allocating memory for temporary signal data,
- Port symbol for defining access name for external reference, and
- Function block symbol for making signal operation / handling / calculation.

Basic symbols are just for data (memory location) and function block symbols with numbers are functions that are executing algorithms. Language is not making a memory location or register references, instead that is actually done in the program loading phase into execution. Binding is done as late as possible.
Administration symbols contain metadata about the program like process area, short description of the program and customer logo. The program itself is drawn inside the frame of the administration symbol defines. There are different sizes available and the program can be extended to multiple pages. Signal connections between the pages can be created by reference symbols.
Functional administration symbol defines execution interval and logical location in the system. This symbol is used to define a new module.

A port can be either an interface or a direct access port. Interface port name is a suffix for the name of the module. Direct access port name is a global name that must be unique in one system (factory level). Port is an access point to a memory location with the name.
External reference is in our terminology an external data point. It contains a name and communication parameters. In the principle name is a reference to the port, which is a named memory location. According to the communication parameters, data is transferred from the port and updated to an external data point. In this way communication takes care of values.
Local data point is inside the module and is needed only to store values between function blocks. It can be needed for storing a value between calculation function blocks.
Function blocks in Figure 1 encapsulate actual subprograms. Encapsulation protects memory allocation and safe execution. Function block always uses the same amount of memory. Execution is controlled by execution order (number between 1--9999) that is given for each function block symbol. All function blocks are sorted and executed in given order. Function block contains inputs, outputs and parameters. Inputs are read before the execution and parameters are used for the calculation and after execution outputs are set. In this way, users can only use these building blocks to define their own program.

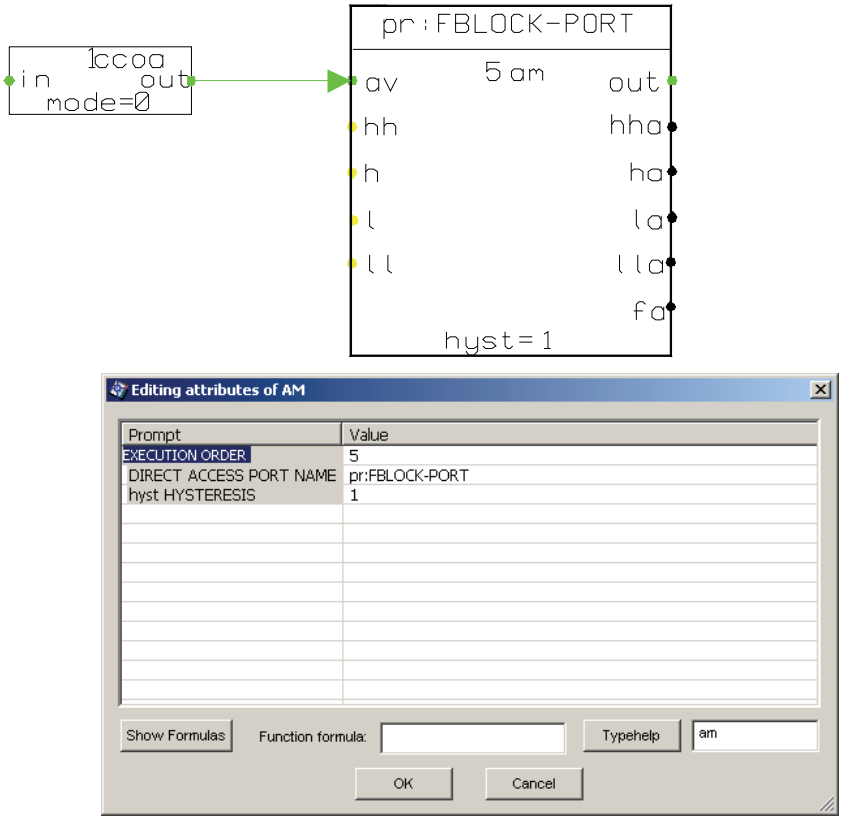

Fig. 1. Two function block symbols with the am symbol's parameter dialog.

Common function blocks are pid for controlling, logical and/or functions for boolean algorithm and calculations. Basic system function blocks are copy (ccox), select (disx), analog measurement (am, am2), binary measurement (bm) and device specific blocks like motor (mtr, mtre, mtr2) and valve (mgv, mgve, mgv2). More application specialized function blocks are for enthalpy calculation went and steam flow calculation (stfl).

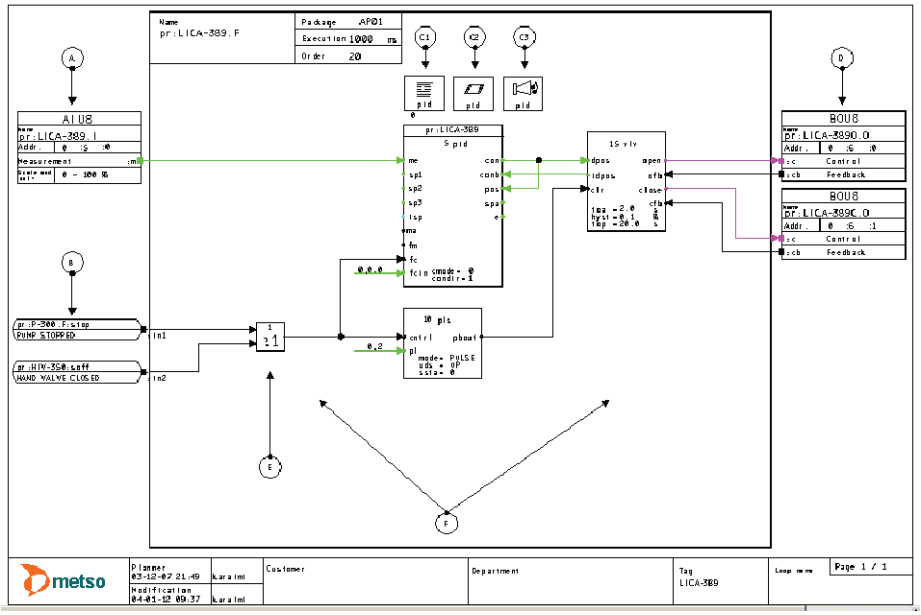

Fig. 2. FBL control loop program.

Figure 2 shows an example FBL program. Symbols A and D are standard input/output (I/O) symbols. Symbols C 1-3 contain texts and other operability and alarming parameter definitions (as priority and alarm group) for the control room functions. Operators in the control room look after the process status from the monitors. The process is constantly measured and run by the programs but people are still making decisions and performing actions (pushing buttons) to control the process. Symbol C3 is for alarm functions. Finally, F is the area for the actual control program. All other symbols representing function blocks and connections are in the same program as the other symbols are building their own individual programs. A function block is a basic subroutine running a specific function to control the process.

The graphical layout is to be read from left to right: inputs are on the left and outputs are on the right. Figure 2 represents a typical automation program in size and functionality. It gives a good overview for the user of one functional entity. The symbols inside one diagram are connected by lines, while connections outside one diagram are constructed using symbols that contain reference names, as shown in Figure 1 symbol B.

Figure 2 shows one Function Block Diagram that can be used to generate multiple textual files. Those files are from a one-page program to several pages long; each file is an individual program. In addition, variables that are connected by lines in a FBL program are

stored in each file. Program modules are distributed in different places in the system. The Process Control Server (PCS) runs I/Os and control programs. Operator Server (OPS) and Alarm Processor (ALP), in turn, run other configuration functions. For example, in the control room OPS is for Human-Machine-Interface (HMI); the operator can change displays and look at different parts of the process and manipulate control parameters from the monitor windows).

2.5 Module symbols

FBL module symbols are application programs that can be distributed in the system. As an example, the I/O- symbol generates a small application program that can be loaded to the field bus controller. It will load needed parameters into the I/O- card and transfer data from the I/O- card to the field bus controller that will communicate with the actual controlling CPU unit that runs function blocks. In the same way the gateway symbol that connects an external device to the system using communication protocol is loaded into the CPU unit that has a serial or an Ethernet connection.

Symbols for creating a connection can be divided into two major groups:

- I/O-symbol to connect a physical field device. I/O card makes analog/digital transformation to an electrical signal.
- Gateway-symbol to connect a software component to another system using communication protocol.

Different kind of I/O-symbols are available, they represent I/O-card. It contains parameters like I/O-address, filtering and other signal processing parameters. Gateway-symbol contains address for accessing data through software protocol. The physical connection can be Ethernet, RS-485 or RS-232. The address depends on used protocol. In MODBUS (MODBUS) protocol addressing is register-based (address format examples 'reg 1001' or 'dw 10'). Signal data-flow is coming in principle the same way as with I/O-connection. The interface module is executed by the driver and the actual data is connected with the external data point to transfer the data from the driver to the application program.

The wiring from I/O-card connections to the field device connects signal flow electrically. From the I/O-card the signal is processed digitally and field bus transfers data between the I/O-card and CPU unit. This is physical distribution and the signal route is illustrated in Figure 3.

Module symbols are usually for defining parameters for user interface and alarm handling, like texts, alarm priority and alarm area. These are loaded to all operator stations and alarm servers.

These module symbols are used for defining

- Text data for user interface,
- User interface panels,
- Alarm handling parameters,
- Long time history data collection parameters, and
- Feedback simulation (action response in virtual environment).

These application programs listed above are not connected by lines as function blocks are connected. The connections are fixed and the user can give one connection name that creates all other needed connections as external data points. This reduces the amount of lines in the diagram. They are usually located near the corresponding function block symbol they are referring. Reference is done by using the same names in the symbols.

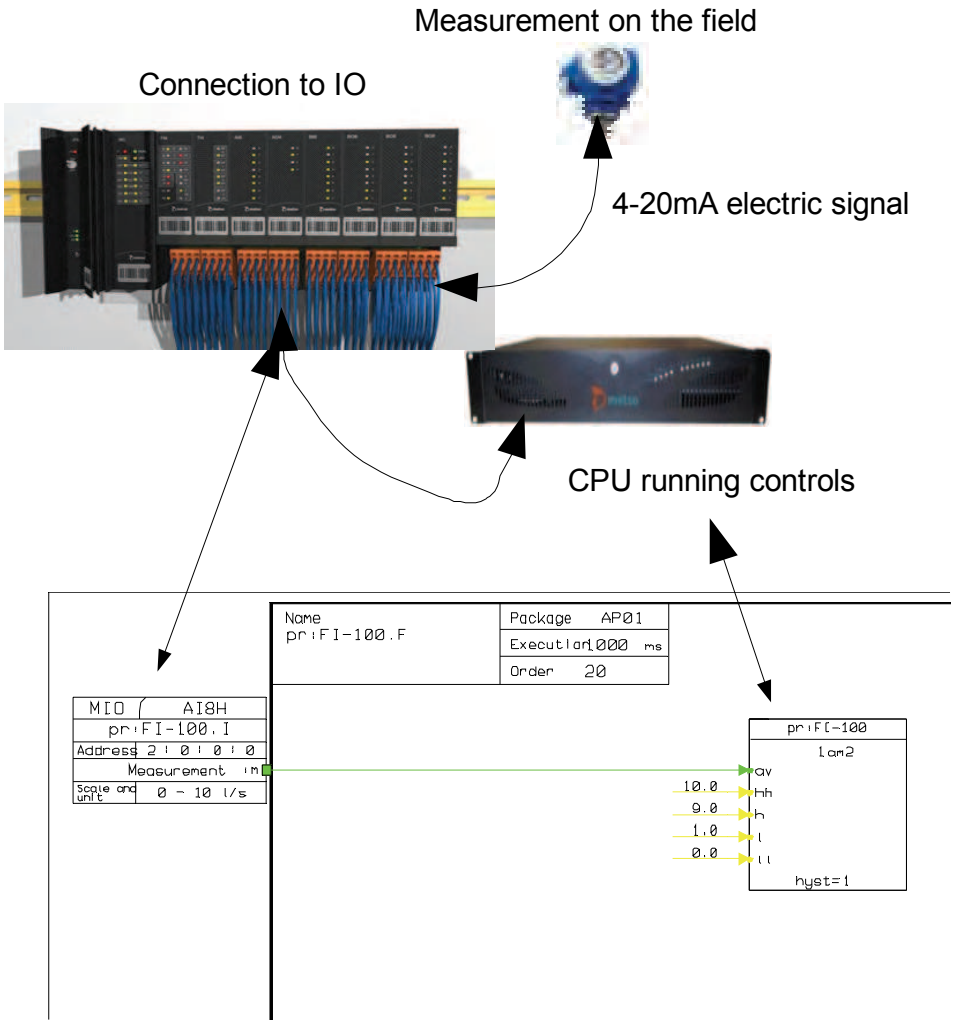

Fig. 3. I/O-signal data flow from the measuring device to the controlling CPU.

2.6 Connections and networks

The connection networks can be very simple point-to-point connections or very complex networks. The network structure solver will take all network connections together and find out the target connection. The target connection is the connection target for the rest of the network participants. In other words, the connection target is the named memory location that others will use.

Some examples of connection networks are shown in Figure 4:

- Point-to-point connection, where output is connected to input.
- Multiple connections, where lines can be connected together with a connection dot that will join underlying lines and creates a connection junction point.

- Connection references, where lines can be connected with symbols that contains reference name from other pages to the same logical connection network.

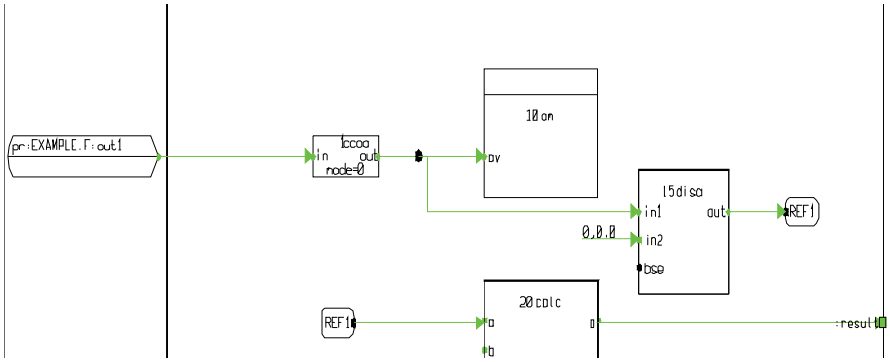

Fig. 4. Connection network examples.

Connection resolving must first always create the whole network from the sub-networks. After that it can run through the connection algorithm that finds the connection target. This is a very simplified explanation for the whole underlying system that contains a lot of specific rules for connection solving.

2.7 Strong typing

The system is strongly typed and simple basic types are represented by fixed colors. Only the basic and most common types are with color. Having too many colors would make it difficult for the user or programmer to distinguish the different types based on color (Whitley, 2001). Further, the benefits of using colors are diminished when printing the programs using a black and white printer; only some grey scales are available in that case or in some cases different line styles are used to indicate signal types (like dashed, dotted etc.). Colors are used in connection points and connection lines. Color defines the type of signal data. Basic types are with color in the following way:

- Green (ana): indicates two values, value (float) and fault bits (uns16)
- Black (bin): indicates a true/false bit (bit 0) and fault bits (bits 1-15)
- Brown (binev): indicates bin and time stamp
- Blue (intl): indicates long integer and fault bits
- Cyan (ints): indicates short integer and fault bits
- Magenta (bo): indicates bin and pulse time (time)
- Red (fails): indicates fault bits (uns16, bit 1-15)
- Yellow (float): indicates plain real number (float)
- Gray (any): all other types (less used misc. types)

Note that the above are scalar types / array \& other multi--dimensional types are drawn by a thicker line but with the same color as the element type of the vector / table.

The user can draw the connection line freely by routing the line and then the program creates the arrow-head automatically at the end of the line to represent data flow direction. Connection lines can cross and if they are connected there is a connection dot in crossing that will connect signals together. In addition, there are special data types for the communication. The function blocks are also based on types that are composed as structures.
At Metso we have developed our own meta-language for defining all the needed structures. Types and also more complex structures such as function blocks are defined with this metalanguage. This metainformation is available from the type database. This can be used to build function block symbols with default layout. Default layout is to place inputs on the left side of symbol, parameters in the middle and outputs on the right.

3. Template mechanism of Function Block Language

3.1 Introduction

Domain specific modeling is used in different levels in FBL. All the function blocks are small models that reflect real physical devices or some needed functionality. A motor, for instance, is modeled as a function block named mtr. The same model can be used for all basic motors and pumps. Similar way valve model is a function block named mgv (magnetic valve). In this way, function blocks are created to solve basic problems in the domain; the name of the block is the name of the focused object. Function Block can be parameterized and connected to other FBL elements. It will read inputs, run itself according to the parameters and write output values. FBL also contains elements that are for user interface and alarm handling. Modeling hides many complex operations.

3.2 Meta template mechanism

Our solution is to use visual templates for efficient programming (Karaila & Systä, 2007). A visual template can e.g. be used to implement motor control. The motor template will contain a set of parameters that are used to create an application program instance.
The engineering tools and database separate data and presentation, Application has a presentation role and actual parameter data is in the database. Transformation attaches template and the result is the implementation. This mechanism works in the same way as in the web applications. The Excel integration gives an effective way to modify existing data in the database. For version upgrades it is possible to export data into one's own XML file. These facts are behind the optimal combination of FBL and framework to maximize effective programming.
Templates are used for example in C++ programming language and in web applications. C++ templates are considered 'type-safe'. The FBL template engine differs from traditional template engines because the FBL template is evaluated immediately in design time. C++ templates are expanded at compile-time. FBL templates can be parameterized using database interface and this kind of principle is also used in web applications. Many languages that are used in web programming like Java or Python have own template engine. These kinds of web servers use primary data from the database and produce interface as shown in Figure 5. This makes effective separation between the business data and presentation. Data can be easily maintained and presentation can be modified. In this way they are loosely coupled.

In the same way FBL templates have parameters in the database and the FBL template contains transformation information. In traditional C++ programming, people use a Standard Template Library (STL). Web based templating testing needs to a run generator to check the end result. In the same way in using STL, compiling is needed to validate the template. In the FBL, template functions are evaluated immediately and transformation is made.

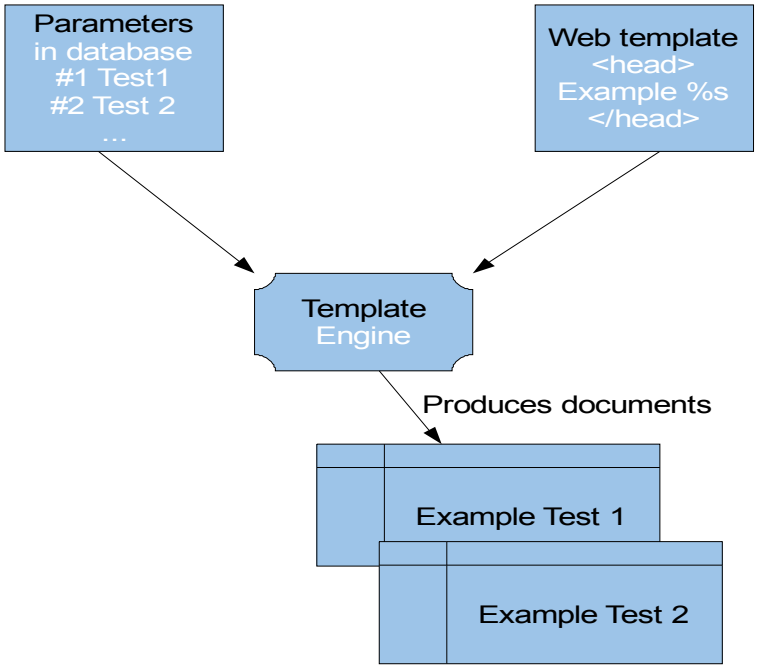

Fig. 5. Principle of web template engine.

Static metaprogramming (template metaprogramming) techniques in general are used to enable the customization of programs at compilation time. For instance, compilation of a program for different platforms can be made easier with such techniques like using generative programming (Czarnecki & Eisenecker, 2000). Static metaprogramming may, however, also be rather challenging. E.g. debugging is typically difficult due to the lack of proper tools. This, in turn, challenges the testing of static meta-programs. Processing and evaluation of template codes at compile-time causes an overhead, which, however, could and desirably does make the executable code more efficient. This overhead might have some significance in larger projects but is typically insignificant in smaller ones. In addition to efficiency, template meta-programming techniques support genericity and facilitate code minimization and maintenance. This is because the programmers can focus on designing and implementing general, perhaps architecture-level structures. FBL templates are used to define a common program structure for a family of application program instances. The templates are further used to create these instances which are called control loops in the terminology of the domain. One template can be used to create several program instances, up to 100 in practice. Each instance has its own identifier and parameter set. The program

structure which is derived from the template is the same in each program instance. In essence, FBL templates are programs that contain data structures and encapsulated functions. Templates are built by first defining parameters that can later be used as an interface to create an instance from the template. Templates further contain formulas, in which the parameters are used. Evaluation of the formulas is automatic. In some cases, the evaluation may modify the program structure, as in conditional compiling, as a result. Formulas are used in FBL templates for evaluating mathematical expressions and for concluding logical truth-values. Each formula is a mini-language statement. The mini-language used is a simple language without real programming capabilities. For practical reasons, e.g. for easy editing and understanding, the mini-language formulas and expressions are compact and fit in one line. FBL language is generative and each template is actually meta-programmed using the mini-language.
Larger models are for modeling more complex functions that need more connections and generic parameters. These connections are to other modules and ports in the system. Parameters are model specific and can be used in multiple elements.
Our engineering tools and FBL editor are main elements in a DSM environment. FBL editor is used for model building and testing. Engineering tools are for managing templates and instances.

3.3 Working with templates

A template is a key component for effective software production. As an example, a basic measurement is needed in every project. But the measurement can be a temperature, a pressure or a level measurement. There is some variation between the measurements like the measurement range is different as the unit depends on physical measurement. The program has input with an address and a range with a unit. The alarm limits of the measurement can be set in programming phase to some initial values. The basic analog measurement template is the model that solves this problem. A template contains the model that can be parameterized and the instance is varied by these parameters. One measurement template can be used in all these different measurements if there are no other requirements.
In practice, a visual template is built with an FBL editor. It contains commands for creating a template. The next step is to make first a program that will contain all other needed parts.
After that, templating can start by the following steps:

- Create design members, these are parameters for a visual template,
- Define needed formulas, these use parameters defined above,
- Save a template, and
- Create an instance and test it (modify parameter values).

First, the user defines all the parameters needed. This can be done using a specific dialog shown in Figure 6.
Parameters work like a placeholder and follow the same syntax rules as Python variables except that they are preceded by $ enclosed in {}. Parameter example: ${var}. Parameter identifiers are case sensitive.
After this, the user can define the formula like in Excel to a separated field that will store the formula as shown in Figure 7. In the evaluation phase the formula is evaluated and the result is placed in the actual value field. The engineer can already see the current value that is calculated from the design parameter value. Formula evaluation is automatic and it helps the engineer to always see evaluated values.

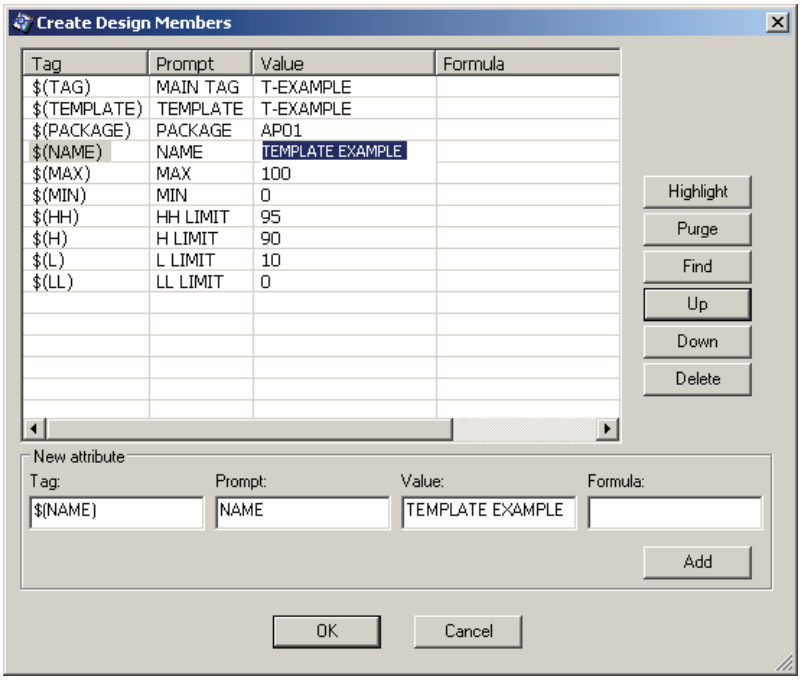

Fig. 6. Step 1: User defines first design parameters.

A complete parameter use example:

- Parameter identifier: \$(MYPARAMETER)
- Parameter value: Example text
- Usage: External datapoint, comment attribute
- Formula field: Test \$(MYPARAMETER)
- Comment field: Test Example text

After step three, template saving, the engineer can create a new FBL program instance from the template shown in Figure 8. Usually new instances are created by using Excel as a parameter entry interface. Template testing always needs multiple instances because otherwise there can be some non-formulated value or wrong formula that will create a non-unique identifier or overlapping address definition.

The FBL visual templating is implemented by mini-language that needs minimal programming. It can be extended when needed but the current functionality has been enough. Using these functions enables the user to meta-program FBL.

Template directives / functions are listed below. Some of them are domain specific.

- eval formula
- mathematical formulas
- strings and parameter value
- function-formula (conditional part, works like snippet)
- value reference (syntax for parameter, reference to outside)
- select formula
- prefix formula (special string handling with prefix)

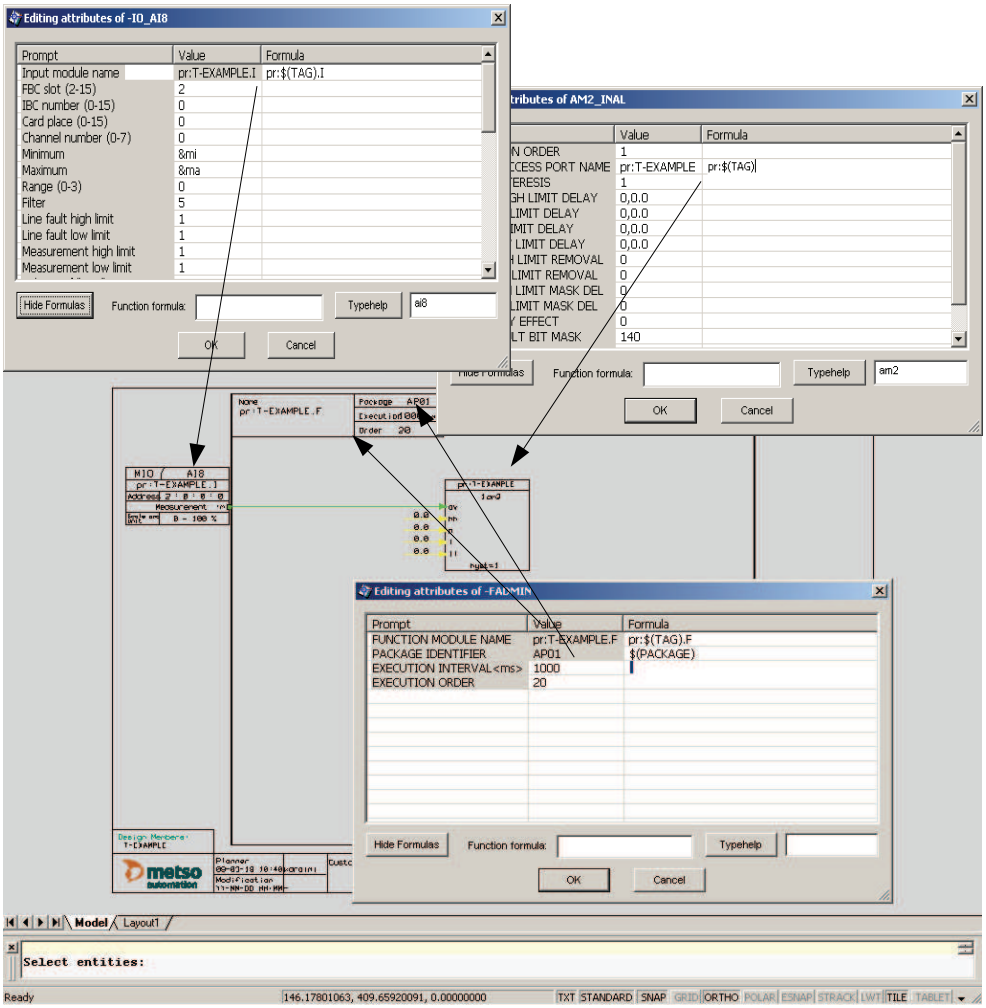

Fig. 7. Step 2: Formulas are defined in each needed location.

Eval is used in formulas to mark parts that will need mathematical evaluation. Otherwise all variables are evaluated as strings.

Mathematical formulas are evaluated according to standard evaluation order. Most of the basic calculations are implemented into the library.

Strings in the evaluation phase are replaced and formula evaluation result is in the value field. Value field is usually a symbol's attribute value but it can also be a comment text.

Function formula works like a snippet. Ordinarily, these are formally-defined operative units to incorporate into larger programming modules. In a visual template, function formula is always included into the template. The "code" amount is fixed but the connections and all parameters are evaluated inside elements belonging to the function formula. It can be turned on or off by a conditional statement. If the result is true, part of the

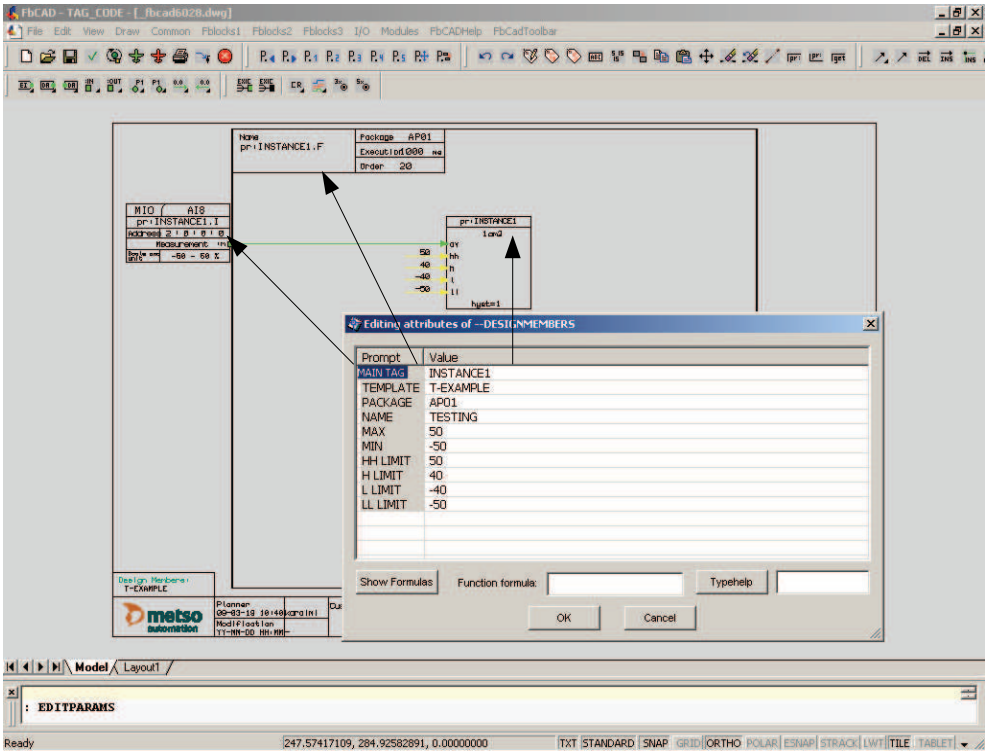

Fig. 8. Step 3: Testing template with new values. Modified design parameter values are evaluated and new values are visible in the diagram.

code is included, otherwise not. Function formula does not minimize the use of repeated code it is for selecting features. In FBL editor function formulas are usually marked with dashed blue boxes.

The following Figure 9 shows function-formula definition for selected elements and Figure 10 demonstrates action that hides a snippet.

Select formula can be used as 'switch...case' or 'if...then... else...' statement for selecting another value by given value. This is a kind of enumeration based transformation.

Prefix formula is used to minimize entering the full reference name. In automation domain, devices are named and in the programming phase it is easy to use a pure name without any prefix or suffix. This abstraction removes / hides programming details from the user.

In step one, shown in Figure 6, the user must first define design parameters that can be used as variables in formulas. Mandatory parameters are:

- TAG (instance identifier),
- PACKAGE (logical name for download target) and
- TEMPLATE (template identifier).

Usual parameters are MIN, MAX, UNIT, HH (high high alarm limit), H (high alarm limit), L (low limit), LL (lower low limit) and so on.

In step two, the user can look at properties of the symbol and add their own formula to calculate a new value.

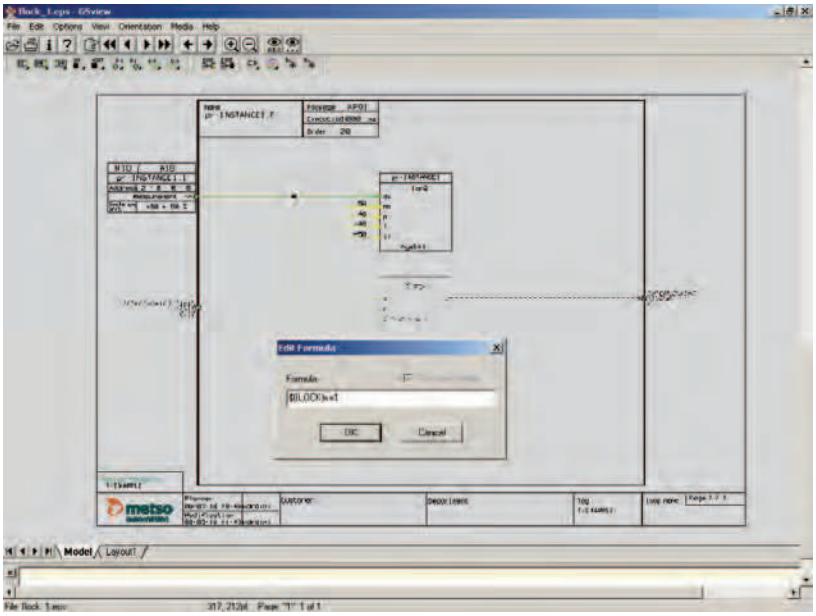

Fig. 9. Symbols are selected & active. Function formula defined for selected elements (lower function block and connections into it.

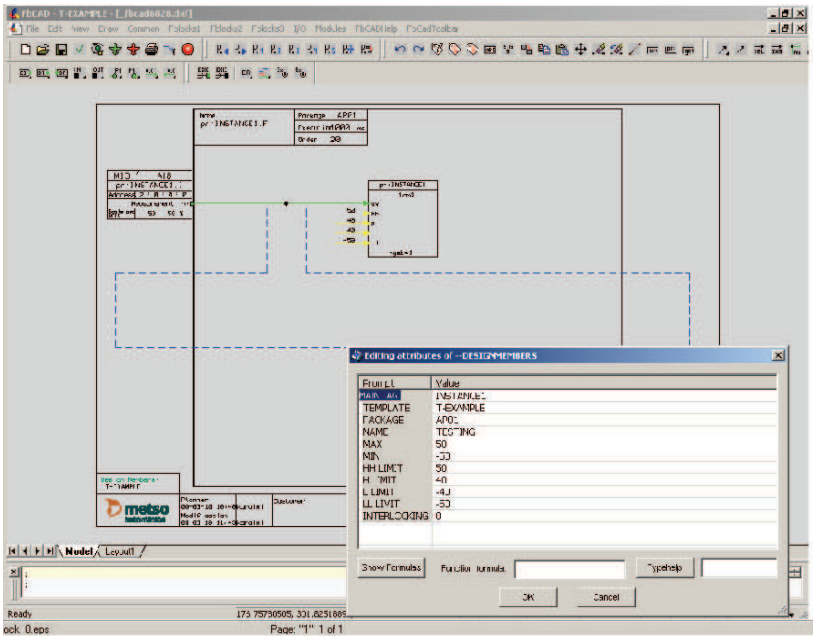

Fig. 10. Function formula 'hides' interlocking elements with the value 0. Elements can be activated with value 1 back to the diagram.

In the template creation process, the user has to save a diagram as a template into template storage.
In the last step, it is good to test the template so that it works correctly and all needed parameters are defined. The user has to create at least two instances to check that there are no overlapping identifiers (global names like module name or direct access name).
Testing is possible in a virtual environment. There are symbols for each actuator to create action feedback. The user can have a motor that will start from the start command and feedback will generate motor running status. In the same way a valve or a controller will get action feedback.
In this way, a higher level of abstraction is done to model larger functionality.
For this purpose Metso has implemented a visual template.

3.4 Experiences

Before Metso had visual templates, Metso's engineers were using typical for modeling FBL solutions. This first generation model is static and is based on more copying existing FBL diagram. The main principle was to replace tokens in the typical with real instance parameters.
When comparing visual template to other solutions, visual template is interactive and immediately evaluated. For instance, it is faster to modify and test. Before the final testing, the following actions are needed: specialized instance, compiling and loading into runtime environment.
Like in other 'Little Languages' (Deursen, 1998) visual templates contain small language, but gives an effective way to use metaprogramming.
The earlier way to create specialized instances was taking more time. An older template was named typical. A typical contained replacement tokens. Each parameterized value field actually contained a token. The user had to run replacement generation to get the specialized instance. This was always needed to test the typical. The replacement token was lost and it was possible to modify any value. The replacement did not support any transformation or calculation. Thus, it was limited to direct replacements.
A visual template can be parameterized and it will evaluate FBL immediately. It is more dynamic and faster to use than typical that is static and needs separate regeneration for updating FBL. One important difference to other template techniques is that the FBL instance contains all template functions and due to this fact it is still possible to parameterize again and again even though the FBL is edited to differ from the original template. Typical did not offer all the functionality that is implemented now with the domain specific formulas.
Mass production of FBL programs is the key productivity for templating. The new visual templating improves productivity by saving time and improving quality with standard project templates.
Productivity is measured in many places:

- Project department measurements (annual measurements existing, over 10 years).
- Value Added Reseller (VAR) partners, specific process area: 100 templates enough.
- In general, over 90 percent of programs made from a template (project library makes automatic calculation from each project).
- Excel or sheet as main parameter input method (data and implementation can be separated; engineering tools can separate data from implementation).

Applicability to domain and product family principles is very good. Existing loop can be turned into a template by a few steps. Template programming adds variables and additional function into existing FBL diagram. Template programming is interactive and the user can immediately test functionality.
In other template based languages, a template is separated and needs rendering / generation that will create an instance from a template. This requires extra maintenance. In our domain, instances contain all template formulas. This is a benefit for us even it can be in some other domains a disadvantage. The framework allows template changes / updates so that it keeps all matching parameter values untouched. This flexibility gives the freedom to change an original template and update it afterwards for all needed instances.
The instance of the template can inherit values from another instance by a reference formula. This reduces the amount of parameters that the user must enter. Referenced template parameters are read--only values. A value change in parent instance is propagated into all children. The purpose of the feature is to reduce parameter amount and automate parameter value propagation. As an example, one design parameter contains text that is used in the primary loop, but the same text is also used in its own history collection definition loop. In this case it is easy to make reference from a history loop to a primary loop. An engineer can change text in the primary loop and it is automatically propagated into the history loop. And in the history loop, an engineer does not have to enter text anymore. An additional positive effect comes to maintenance. It is better to split functionality into its own features and bind needed parameters together by referencing. For us, our FBL and its metaprogramming support makes visual templating a practical reuse technology.
End customers are becoming more demanding.

- Easy and fast to create from specification to template and implementation. Specifications are coming later and later. Or in some cases the customer or process expert defines automation functionality at the factory in the start--up phase.
- Easy to make modifications and take those into use just by changing or updating the template.

Even through the template functionality has been in existence now for some years there is still work to do with usability and metaprogramming. There is the need to teach this technique. The conversion tool will need some tuning even it can transform an old typical to a template.
Time will show the life cycle of the templates. There have already been cases that the project is first done with templates and delivered without the formulas. This kind of downgrading is sometimes needed to support old installed systems.

4. Reuse mechanisms

4.1 Introduction

Support for software reuse can be hard to utilize. Systematic reuse will require process, analysis, feedback for continuous improvement and knowledge management.
Traditional software reuse can be implemented by components and libraries. In the similar way FBL contains build-in functions that are Function Blocks. These are documented in system manuals and are used to implement application programs.
For effective application programming, the solution is to reuse application programs. It is harder because they do not usually contain extra documentation or they are not categorized into any hierarchical structure like build-in Function Blocks are in the libraries. The system

level reuse also actually exists in product level because the automation system is based on a software product line (Ommering, 2005).
Another need to reuse already made projects is to estimate the effort needed to implement the same kind of project. A project can be a part of an earlier project like just one or two process areas are similar in a new project. 'Similar' means that the process area like "Fresh water treatment" is implemented with the same process equipments and can be used in the new project as a starting point. This kind of search and pre-study is needed and used in our sales. If there is existing the same kind of implementation, project engineers can start redesign using the existing implementation (Karaila & Leppäniemi, 2004).
In FBL, three types of reuse occur, in three abstraction levels:

- Level 1 Function Block (system level),
- level 2 Template (model reuse), parameter reuse between the template instances, and
- Level 3 Function Group (model group reuse, higher abstraction level).

The modeling is more demanding than the system level reuse. The user has to first select the template which is not always as clear as selecting a function block. The basic level function blocks are documented and always available. Templates are currently documented only in intranet level and loaded separately as their own library.
In a search for finding a possible template, there are parameters that can be used to narrow search results. This needs domain knowledge. The reuse library offers all parameters and allows the user to use those in search criteria.
Another reuse level is to reuse just parameter values. This can be done in the template level. The parent - child parameter referencing helps to maintain consistency between the same problem entities that is implemented with multiple instances. The main instance, core loop contains all common parameters like name and alarm area. Each child is referenced into those common parameters. In this way, a change in common parameter is propagated into each child instance.
Function Group level utilizes the next level in abstraction hierarchy. Function Group can handle a set of instances that are template based in one Function Group diagram. Function Group diagram visualizes connections between the application programs.

4.2 Reuse in practice

Project library application search dialog in Figure 11 is the starting point for reuse. The search interface allows users to search application solutions according to saved metadata and performed analysis. The search can be focused on certain process areas and projects. More detailed search criteria can include e.g. the main function of the program (function block like pid-controller or motor controller), the IO card type used and the application creator.
Application data is shown in Figure 12. The general part contains metainformation about the project and program itself. The entity count, primary function block, template generated information and user question count are created in the analysis phase. The IO data is also extracted in analysis. In the file information fields, data is needed to access file and template. The template match is in this case 100\ %. When no structural changes between the template and instance exist the match value equals 100. That is, only different parameter values may exist. Each structural chance diminishes match value by a certain amount. For example by deleting and adding one symbol the match value is decreased by two to 98.

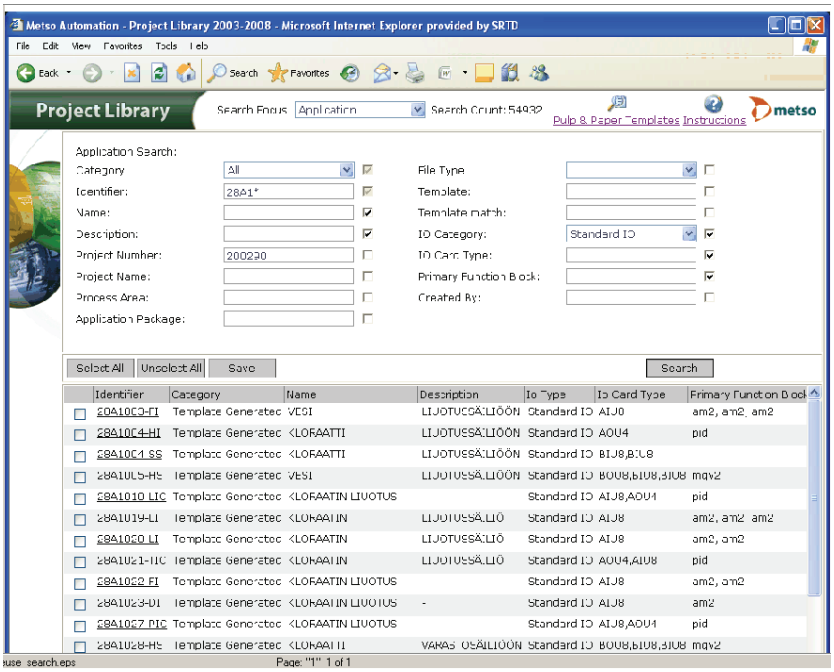

Fig. 11. Reuse library search dialog.

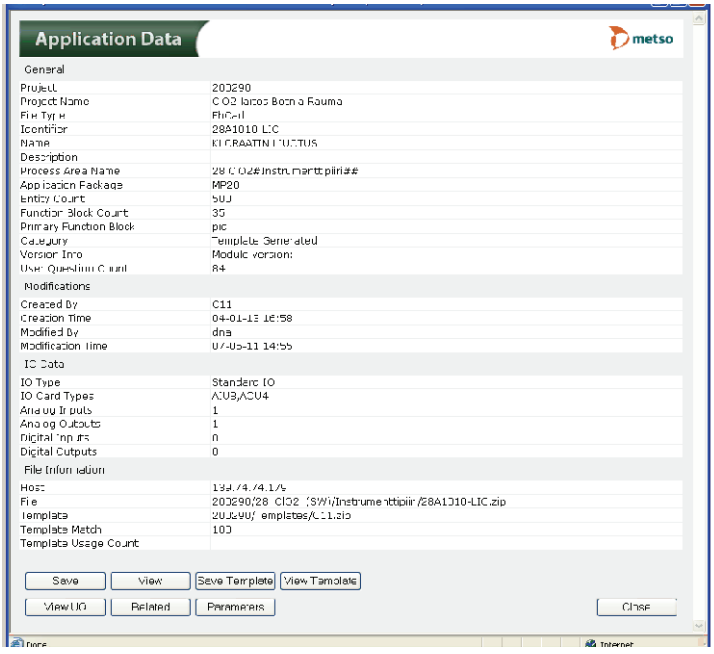

Fig. 12. Reuse library shows application data.

The search can be also focused on project, process area or template. The project data is shown in Figure 13. It contains major data from the delivery and for the practical reuse project team, main process and process supplier are needed.

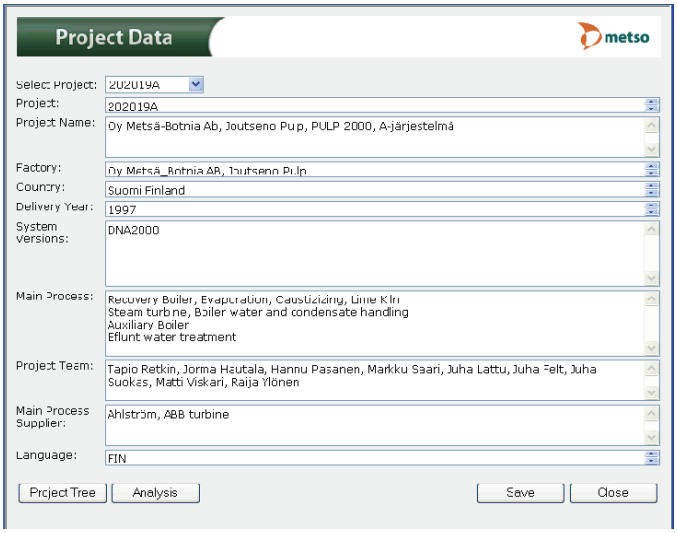

Fig. 13. Reuse library shows customer project specific data.

The user can search and navigate from the application to the template or to the related loops. User interface supports downloading multiple files together in one zip file.

4.3 Analysis

In reuse library, saving application will run FBL analyze that first creates a fingerprint from each application program. Fingerprint is a calculated value from the diagram entities. It is used to find similar diagrams faster. If the instance is template based, analysis will create a link to the template. In this way user can get the template easily. The project analyze will calculate summary information from the project. This information is used in estimating the project efficiency. Later the same information can be used to sell a new project. This makes better accuracy for estimating the cost of the new project.

The project library is for archiving projects, but it is actually a huge reuse library. It also contains the template library and its own special Quality Control library. This special library contains mainly handmade solutions that are needed for integrating some older actuator device into our system. The project library is integrated to the project delivery process. Each delivered project is archived into the project library for reuse.

4.4 Discussion

Traditional programming reuse analysis tries to find reusable patterns. Strategies for component analysis are well introduced in (Rothenberger et al., 2003). These practices are categorized to project similarity, reuse planning, measurement, process improvement, formalized process, management support, education, object technology and commonality of architecture.

Our project library and reuse model covers project similarity very well because it is one starting point in finding reusabable FBL programs. The reuse process is planned. The analysis measures template usage and the feedback system with template library targets for improved templates. Because every project is archived into the project library in the same way, the process is formal and repeatable. The analysis also gives good numbers for the management. Knowledge management is not so visible in our process but the reuse and template based design are part of the project delivery process. The knowledge needed to successfully use the templates takes some time. The automation domain is based on product family and the basic architecture has remained solid. The technology is based on different solutions and the object technology is used in various places.
Evolution during last four years has not affected reuse. There are new IO cards and new function blocks. Domain specific language reuse in dynamic domain is discussed in (Korhonen, 2002). This focuses more on code generation and language principles than reusing actual applications. The project library internally uses XML in many places and it has worked as a good transformation base. This was originated partly from the first agent-based implementation. This solution offered easier maintenance for the whole reuse library because it allowed transformations and extensions.
The publication implemented agent-based software is currently a simpler java application. It no longer uses agents anymore. The search engine user interface was enhanced in 2008 and new features were added by user requests. One important feature is to search special applications, only 1-2 applications per project. These applications contain rare I/O-cards and can be found using the card type in the search criteria. In the same way, some special Function Blocks can be searched.
The project library for reuse is in active use. The current search request amount is still almost one thousand searches monthly. The main page contains the amount of searches. It shows the current value 54932 (end of 2008). This makes the last four years of use an average of 1000 searches per month. In the initial phase in 2004, the amount of metadata was less than 2 Gb. The current (measured in the end of 2008) amount of metadata in the library is over 3.5 Gb and there are millions of application programs stored in the file system.
The actual metadata in the reuse database is growing and there has now been added more data about process such as machinery supplier and project people. If the salesman compares similar kinds of processes they have to check the supplier to validate reuse possibility. For tacit information and other not formalized information about the project, people are listed in the database. This makes it possible that people can be contacted and a short discussion can solve other unclear things.
The metadata makes searches more exact and implements actually feature based reuse library as is discussed in (Park & Palmer, 1995). The key factor is to select features as adding primary function block and IO card type among other metainformation. But instead of reusing components as stated in the article, Metso reuses application programs and templates. This kind of reuse affects to both productivity and quality much better.

5. Maintenance and round-trip engineering

5.1 Introduction

The biggest parts of software life-cycle costs are shown to be due to maintenance activities (Sneed, 1996), (Jones, 1998) (Erlikh, 2000). The systems that have long life cycles and require high maintainability, a key for lower maintenance costs is quality. Maintenance can be

supported by various reverse engineering techniques like comprehension and visualization. Software visualization techniques applied to software written in traditional, textual programming languages can be problematic to be linked with reengineering activities afterwards, especially if standard notations, such as UML (UML, 2009), are not used: if the reverse engineering tool uses a different notation than the one used in software design, mappings between the different notations are needed. Since the models and views constructed from the existing program are presented with the same language used for development, the reverse engineering activities can be conveniently mapped with re-engineering activities, therefore enabling full round-trip support.
FBL application programs are located at the customer's own factories. Those programs are modified when there are some changes needed. These are frequent changes that must be done quickly. Even though FBL evolves and a version is upgraded, old programs can be used without any major work. This is part of the maintenance work that requires compatibility.
The following goals have been set for FBL maintenance:

- application level implementation remains the same even when symbols are updated,
- better performance: faster open and save, switch to testing faster,
- better usability and
- modern outlook: style is according to operating system and CAD platform.

5.2 Reverse and forward engineering

Reverse engineering activities aim at constructing representations and models of the subject software systems in another form or at a higher level of abstraction (Chikofsky & Cross, 1990). 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, by using software product metrics to identify highly cohesive clusters, or by using existing software architecture models and mapping them with the lower level details. In Java, for instance, package hierarchies can be used to structure classes and interfaces of the system. These hierarchies can be extracted by automated means. However, there are no guarantees that the packages contain sets of classes that conceptually form subsystems or components. Software product metrics used for identifying subsystems typically measure inter couplings and intra cohesion of the sets of software elements. These methods can only give educated guesses for clustering. Architectural models used in top-down reverse engineering approaches provide a good way to form a clustering. However, such high-level models do rarely exist and the construction of mappings with lower level software elements is typically difficult. In Metso's case, program uses the syntax of the language to construct high-level models for the FBL programs (Karaila & Systä, 2005).
In FBL, abstraction can be done by creating a new symbol from the existing application program. In Figure 14, a low-level FBL program is shown. For generating an abstract view to this program, the details of the program are filtered out and only the input and output symbols are preserved. An abstracted view is shown in the lower part of the same Figure 14 as one symbol. The abstracted program is called Function Group, indicating that one symbol contains several functions (function blocks and IOs). The symbol has two input points on the left: HLIM1 and LLIM1. These inputs limit values to form interlock interfaces H, H1 and

L. On the right there are five outputs HH, LL, H, L and H1. The outputs, in turn, are for interlocking and for different limit thresholds. If the measurement is over H value then the function group generates a high interlocking. If the value is even bigger and goes over HH value, then the function block generates a higher high limit. Correspondingly, the function group will generate low and lower low limits as signal value goes below a given limit. Parameters are captured inside the symbol. Program visualization creates new symbols on the fly for each abstracted component.

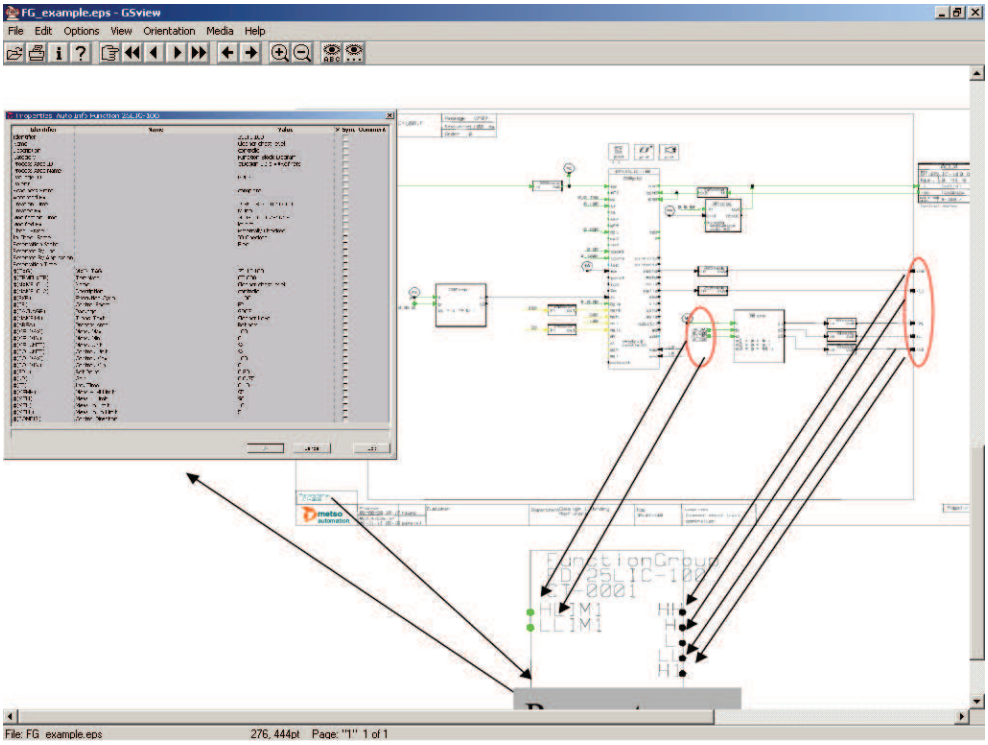

Fig. 14. Function Group example: parameters, implementation and symbol.

When compared to traditional reverse engineering techniques, a function group can be considered to correspond to a subsystem. Unlike in traditional approaches where various heuristics or metrics are used to help clustering program elements to subsystems, FBL syntax and information stored in the database are used to extract high-level views. This difference is significant: when reverse engineering FBL programs, the abstractions are always "correct", not educated guesses: the abstractions can be used for forward engineering activities as such. The differences between high-level views can only be due to different information filtering actions, not caused by different clustering. This makes reverse engineering of FBL programs significantly easier than reverse engineering programs written in traditional programming languages. On the other hand, this also means that the reverse engineering activities can be conveniently integrated with forward engineering activities, providing full round-trip support.

After constructing the higher-level function groups, they can be connected to each other. In FBL, internal communication connections are drawn inside modules by lines, while for external connections the engineer has to give a name. These external connections are stored in the database. To visualize external connections, database information is used to connect symbols as shown in Figure 15.

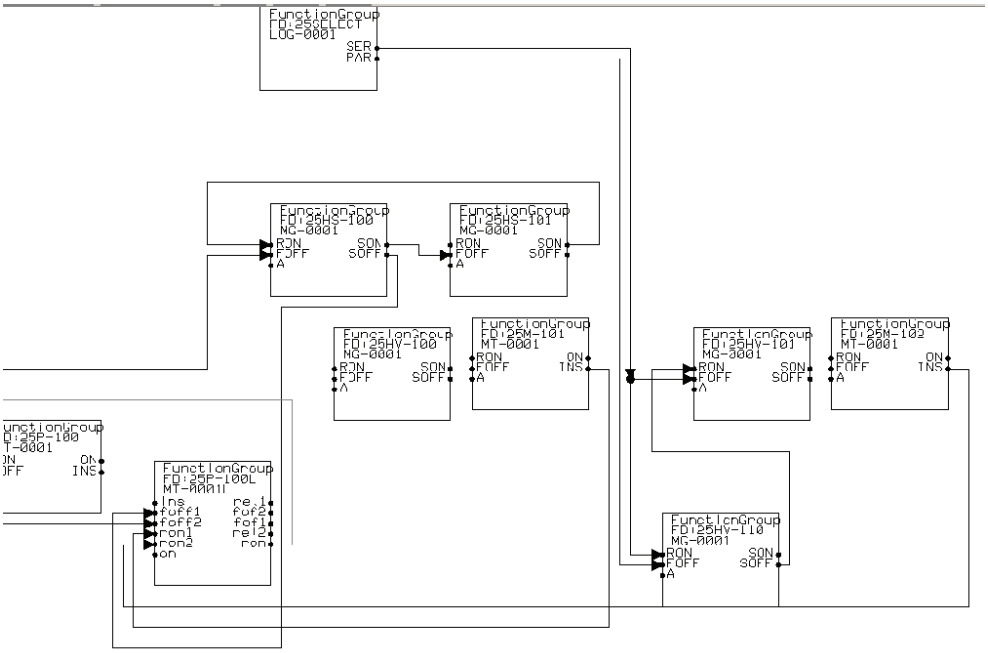

Fig. 15. Function group abstraction from FBL refiner programs.

To limit the size of the group of function group symbols, the engineer can select only a part of information stored in the whole database. This selection can be based on the metadata stored as well. In the domain FBL has been used, reasonable many of a large group of modules are from the same process area. In Figure 15, for instance, 10 symbols depicted are from the Refiner process area. Each function group symbol has a function that will need a user interface. Each device motor or valve has its own instance in both. Controller and selection logic are represented but the only one that is pure software is the interlocking logic. It is instantiated in the function group, but not in the normal user interface. The interlocking is in own display that the operator can open on demand.

In the Refiners process wood is mechanically cut / bladed to fibers. This mixture of paper fibers and water is pulp. Paper machines make paper from the pulp. The Refiner process is controlled by human operators from the display like the one shown in Figure 16.

Reverse engineering and data analysis techniques are used to get an overview of FBL programs. The environment can be used to generate high-level visual programs automatically.

A typical problem in this step is the layout. As indicated in studies, e.g. by (Storey et al.1997), the quality of layouts may have a significant impact on program understanding.

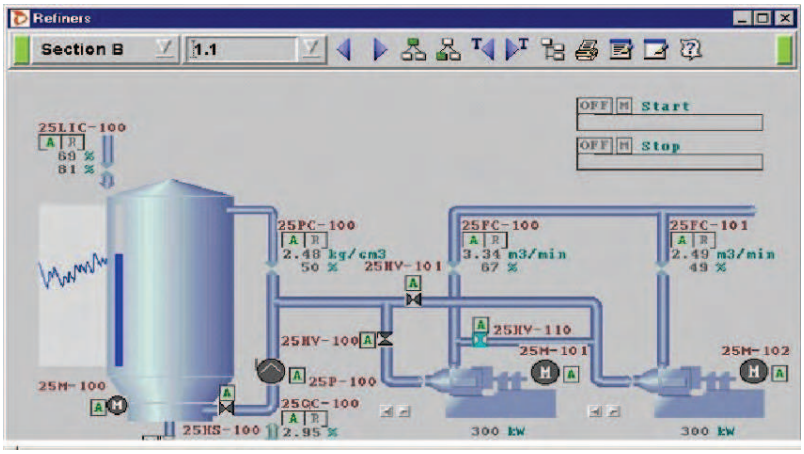

Fig. 16 Refiner user interface, operator display for controlling process.

According to our experiences, this also applies to visual programs. A commonly used solution for placing symbols is to use some automatic spatial spacing and auto-routing methods. The layouts of FBL programs have some fixed properties. The FBL programs are always read from left to right: inputs are on the left and outputs are on the right. The layout problem thus mainly concerns the rest of the FBL program. The solution selected for lay outing FBL programs is semiautomatic. The engineer needs to show a place for each symbol which is created automatically on the fly. Even though this approach requires manual intervention, it also has its advantages. The same tool environment is used for viewing and reverse engineering on the one hand and for programming on the other. Namely, the processes of forward and reverse engineering are not separated. In fact, the engineer is typically programming at the same time as analyzing a reusable (reverse engineered) solution. To be able to reuse the existing program, one has to learn the program structure first. After inserting all symbols needed, the engineer can activate a function that completes drawing with auto-routed connection lines. This feature is really powerful because in a normal case the engineer has to write each external data point / port connection manually in each FBL program. Now he can modify symbols and connections and in this way re-design the solution, e.g., to be more common and easier to understand.

5.3 Template maintenance

Trends in our template variation will focus on isolating IO from basic templates. This will reduce maintenance work that is needed. If a template contains some additional features like IO (standard IO, ACN IO, and LIS IO) and a new connection is implemented like FF IO, then all templates should be updated in case the IO is included inside the template. This is one fact that suggests separating IO from the core template. An example of separation is shown in Figure 17 that contains core templates in the middle and IO templates in the lower part. Other auxiliary features are placed in the upper part in own templates, like start and restart.

Figure 18 explains IO template in more detail. The tag application contains IO template and CORE. Communication is in its own part. This allows changes in application both in design time and in runtime easier. The flexibility is better because the new IO templates can be

used without changes in the CORE templates. This will help in the future as new IO cards are designed and taken into use with IO templates.

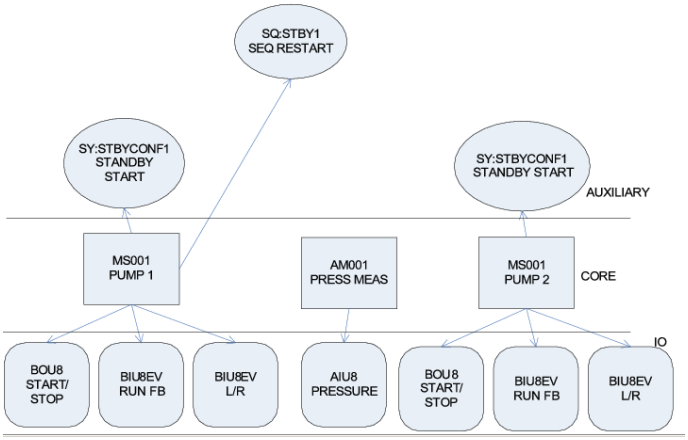

Fig. 17. Template separation levels: IO, core, auxiliary.

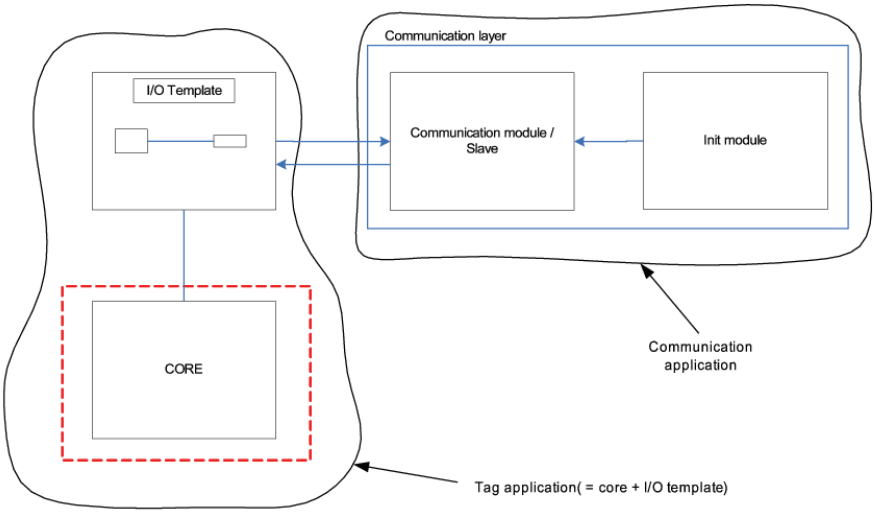

Fig. 18. Template modularization aims for managed variation and easier maintenance.

5.4 Discussion

According to the experiences on FBL and its programming environment at Metso Automation, in a combined reverse and forward engineering environment for visual programming, the role of layouts becomes quite important. Since the program analysis activities are often followed by forward engineering activities, the layouts constructed when analyzing programs should be "correct" and usable from the point of view of forward engineering activities. Also, since the engineer needs to understand the programs before

being able to re-engineer or reuse them, semi-automated approaches for constructing layouts have shown to be quite feasible.
Re-engineering existing program instances means that they can be changed by extending or modifying them. For instance, new function blocks can be added, parameter values of existing programs can be changed, or 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: he first creates a group of modules which are then visualized with the aid of reverse engineering techniques and finally re-engineered and/or reused.
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. The function group can use a template as a symbol to instantiate it. In this way, function groups are built from specialized templates.
The architecture layering and template mechanism gives us good tools for managing maintenance. At the template level, the model gives new maintenance needs as variation points but it needs more metainformation from the context (Cuccuru et al., 2007). There are sub-domain specific features in the templates such as power plant automation needs more accurate time stamps and chemical process automation requires more statistical data. The measurement template needs its own variation to fit from paper machine temperature measurement to oil refining temperature measurement. The oil refining measurement is more demanding and needs parallel measurements and statistical validation to insure reliability and robustness. This kind of knowledge management is needed in the future.
The long history can be used to reflect and analyze different maintenance activities. Normal maintenance activities focus on updating existing symbols and templates. From time to time people find bugs, which also call for maintenance. Sometimes cosmetic changes are also needed, like new better looking symbols or new layout that will make a program easier to read.
One practical issue is to support application maintenance. In the system level framework, tools can help a lot in this work. But designers have also had some bad experiences like making a modification in existing function block structure will make a big maintenance effort. After this designers have kept old function block structures untouched. It is better to create a new function block. A new function block can replace an old symbol if the connection points are matching. The framework can run a script that will automate the work. In exactly the same way, templates are versioned. A base template will be left untouched and a new template will be extended. An instantiated template can be easily upgraded to the new version. This is the normal method in customer projects. The project engineer can make a better template and changes / updates will keep all existing parameters. This is an efficient working method that improves quality.

6. Summary

FBL is a visual domain specific language that heavily relies on the usage of templates and meta-programming. FBL has been developed for writing automation control programs. Based on several years of practical use, it has proved to be easy to learn and adapted by its users.

Despite their undeniable benefits, template meta-programming techniques also have some drawbacks. Many compilers historically have quite poor support for templates. The use of templates can, in fact, make code somewhat less portable. Further, when errors are detected in template codes, most of the compilers produce confusing, unhelpful error messages. This can make templates difficult to develop. Debuggers also often have difficulties in working with templates.

A large group of methods and tool support for visual domain-specific programming is available. For example, (TRACE MODE, 2009), supports several IEC 6-1131/3 standard languages that can also be used to program control systems and business applications. One of the languages, namely Function Block Diagram (FBD), resembles FBL. Another toolkit is the Generic Modeling Environment (GME, 2009) that supports creating domain-specific modeling and program synthesis environments. In (Fröhlich et al., 2002), propose a meta-modeling based approach to provide and enforce modeling rules relevant for specific types of conceptual models used in automation domain, e.g. industrial plants or control systems. MetaEdit+ (Luoma et al., 2005 & MetaCase, 2006), in turn, supports meta-modeling for defining new domain-specific modeling languages and provides CASE-tool support for their use. While these approaches are partly related to ours, in this paper we have discussed yet new ideas, aspects, and working methods that are novel in using visual domain-specific languages.

In reference (Czarnecki, 2000), points out the following goals of generative programming: (i) decreasing the conceptual gap between program code and domain concepts, (ii) high reusability and adaptability, (iii) simplified managements of many variations of a component, and (iv) increased efficiency. In our case, where a visual domain-specific language FBL is used, all these generative programming goals can be achieved.

First, FBL as a visual language is intuitive. Moreover, custom symbols and icons can be used when programming certain types of applications. This provides a nice and customer-friendly way to map domain concepts with program elements. Second, templates have a significant role in FBL programs. A typical programming scenario includes selection of an appropriate template and its customization to a real program. A specific template library has been constructed and is constantly updated to better support programmers. In practice, the degree of reuse is very high. In new projects that are utilizing templates to a full extent, almost 100% of application programs are implemented by using templates. On the other hand, there are still projects that do not use any templates. New templates can, however, be easily constructed by comparing similarities of existing programs. i.e., new families of programs can be identified. This also supports the management of the programs belonging to this family. Finally, having ready-made templates can increase efficiency.

Reuse library developed has enabled an efficient way for users to archive and share implemented solutions and knowledge. The current java-based application solution filing process together with search tool has proven to be an efficient and practical solution.

The current content management database size exceeded 3.5 Giga bytes (2008). Database contains over hundreds of projects and links together over 62 Giga bytes of compressed files (1.2 million files). The usage of search tool has become a part of application engineers working manners. Approximately 1000 searches are performed monthly.

The analyses and template-matching processes implemented have allowed Metso to study more the real problem of finding a higher abstraction level for mass customization. Reuse helps sales and pre-design is started usually from the reuse library.

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. User group feedback has been collected to make further improvements, in a similar way to works presented in (Costagliola et al., 2002, Cox et al., 1997, Smedley & Cox, 1997).
The same environment that is used for development is also used for reverse engineering and maintaining FBL programs, thus providing a full round-trip support. The implemented environment together with the information on existing FBL programs gives engineers better understanding on the large existing group of FBL modules and their connections. The same kind of presentation of control diagrams and applications for interlocking are actually presented in German energy sector. This association of power and heat generating utilities is named VGB (German appreviation from Vereinigung der Großkraftwerksbetreiber; VGB, 2009). The documentation of the whole factory and its processes (water system or power generation) are normally written according to association guidance that is quite close to Metso's function group. Similar standard is System Control Diagram that is specified in Norway (SCD, 2009). The symbols and principles are almost the same as in programming with Function Groups.
To a great extent, the future design of controls could be carried out using function groups. Engineers who design advanced controls are seldom interested in details, but would rather like to program at higher level of abstraction, namely using function groups. The engineering environment indeed allows that. The actual experiences of the environment are still under study. Function groups are constructed for different processes to compare control structures and patterns that are used. From these existing solutions we will find out most common building blocks by statistical analysis using metadata stored in a reuse library. During last five years more entities and diagrams have been used in projects than before. The complexity of the programs has been almost at the same level. The conclusion of this five years trend is that the automation level is increasing steadily. Therefore, there is more implementation work in each project. The experiences gained so far indicate that similar physical processes with the same kind of machinery are easier to understand and reuse as high-level models, namely as packages with function groups in our case. Similar experiences have been presented (Wilkening et al., 1995). This also supports understanding on how to combine hardware and software as complete products (Holz, 2003). The experiences gained 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. We further believe that many of the features and advantages of the proposed FBL environment can be useful in traditional reverse engineering environments. In fact, features and benefits of an engineering framework corresponding to one discussed have been presented (Tilley, 1998). One of the most valuable parts of the proposed work is a possibility to reuse and re-engineer existing solutions. Unlike what is often used in traditional reverse engineering environments, semi-automated methods for constructing layouts have shown to be quite useful and feasible in the FBL environment. The semi-automated layout encourages the engineer to gradually learn the program, which is in any case required before he is able to re-engineer or reuse it. In addition, the usage of metadata has shown to be quite useful for querying the program database and to support program comprehension and analysis, especially concerning the evolution of the programs. Similar advantages could also be gained in traditional reverse engineering and program analysis tool support. We believe that

traditional reverse engineering environments could provide more advanced support for using metadata than what is currently available.

To summarize, the development of the template meta-programming support for FBL proceeded as follows. After the first release, fast feedback from the users had to be utilized in order to increase usability. Metso development team focused development on mini-language functionality in order to match our domain requirements. After that, the tools were modified to support different kinds of maintenance activities. The most important factor was always efficiency. Development team has learnt that getting feedback continuously from the users is crucial for successful maintenance and further development of FBL and its programming environment. These maintenance and development activities should and will continue as long as FBL is in use.

Future research and development will focus on further enhancing support for template meta-programming, e.g. by extending the template mini-language and by providing the additional means to raise the abstraction level of programming. Modern techniques and programming principles can be applied to the automation domain. Visual programming requires own specialized support that can be tuned to fit into the language and domain.

7. References

Burnett M., A. G. & Lewis, T. G. (1995) Visual Object-Oriented Programming Manning Publications Co. Greenwich, 280.

Burnett M. M., Webster, J. G. (ed.) (1999) Visual Programming *In Encyclopedia of Electrical and Electronics Engineering*, John Wiley & Sons Inc., New York.

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

Costagliola G., Francese R., Risi M., Scanniello G. (2002), A Component-Based Visual Environment Development Process, *In The Proc. of Software Engineering and Knowledge Engineering (SEKE'02)*, pp.327-334.

Cox P.T., Smedley T.J., Garden J., and McManus M. (1997), Experiences with Visual Programming in a Specific Domain – Visual Language Challenge, *In The Proc. of IEEE 1997 Symposium on Visual Languages (VL '97)*.

Cuccuru, A.; Mraidha, C.; Terrier, F. & Gérard, S. (2007) Templatable Metamodels for Semantic Variation Points Model Driven Architecture- Foundations and Applications, *Model Driven Architecture - Foundations and Applications*, Springer, 68-82.

Czarnecki, K. & Eisenecker, U. (2000) Generative Programming: Methods, Tools, and Applications Addison-Wesley Professional.

Deursen, A. V. (1998) Little Languages: Little maintenance?

Debbie K. Carter, Albert D. Baker, W. B. A. (1995) I-I-Con: A Visual communications paradigm to integrate industrial control system engineering, *ISA Transactions, Elsevier Science Ltd.*, 34 (2), 153-163.

Erlikh L., (2000) Leveraging legacy system dollars for E-business, *IEEE IT Pro*, pp. 17-23.

Fröhlich P., Hu Z., and Schoelzke M. (2002), Imposing Modeling Rules on Industrial Applications through Meta-modeling, ER 2001 Workshops, HUMACS, DASWIS, ECOMO, and DAMA, LNCS 2465, pp. 166-182.

GME (Last visited September 200), Institute for Software Integrated Systems, The Generic Modeling Environment, http://www.isis.vanderbilt.edu/projects/gme/.

Hotz, L, Krebs, T. Günter, A.(2003) A Knowledge-based Product Derivation Process and some Ideas how to Integrate Product Development (position paper), *Workshop on Software Variability Management, Groningen, The Netherlands)*, February 13-14, 2003.

Johnston, W. M.; Hanna, J. R. P. & Millar, R. J. (2004). Advances in dataflow programming languages, *ACM Comput. Surv. 36, pp 1-34.*

Jones T.C., (1998) Estimating Software Costs, McGraw Hill.

Karaila, M. and Leppäniemi, A. (2004), Multi-Agent Based Framework for Large Scale Visual Program Reuse*, IFIP, Volume 159*/2005, 91-98.

Karaila M., Systä T. (2005), On the Role of Metadata in Visual Language Reuse and Reverse Engineering – An Industrial Case *Electronic Notes in Theoretical Computer Science, 2005, Volume 137, Issue 3*, 29-41.

Karaila, M. and Systä, T. (2007), Applying Template Meta-Programming Techniques for a Domain-Specific Visual Language - An Industrial Experience Report, *ICSE 2007.*

Korhonen, K. (2002), A case study on reusability of a DSL in a dynamic domain 2nd OOPSLA Workshop on Domain Specific Visual Languages.

Luoma J., Kelly S., Tolvanen J.P., (2005) Defining Domain-Specific Modeling Languages: Collected Experiences, *In Proc. of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM'04)*, LNCS 3714, Springer, pp. 198-209.

MetaCase, (2006) Domain-Specific Modeling with MetaEdit+, http://www.metacase.com/.

Mohamed E. Fayad, R. E. J. (ed.) (2000) Domain-Specific Application Frameworks. *Frameworks Experience by Industry* Wiley, 681

MODBUS, http://www.modbus.org/ Last visited September 2008.

Pressman, R. S. (1997) Software Engineering a Practioner's Approach McGraw-Hill.

Ommering, R. (2005) Software Reuse in Product Populations Software Engineering, *IEEE Transactions on*, 31, 537-550.

Park, S. & Palmer, J. D. (1995) A feature based reuse library Springer Berlin / Heidelberg, 1995, Volume 945/1995, 493-500.

Rahman Jamal, L. W. (1995). The Applicability of the Visual Programming Language LabVIEW to Large Real-World Applications, *In the Proc. of the 11th International IEEE Symposium on Visual Languages, IEEE Computer Society Washington, DC, US.*

Rothenberger, M. A.; Dooley, K. J.; Kulkarni, U. R. & Nada, N. (2003) Strategies for Software Reuse: A Principal Component Analysis of Reuse Practices *IEEE Transactions on Software Engineering, IEEE Computer Society*, 2003, 29, 825-837.

SCD, The Standardization Organizations in Norway, I-005 System Control Diagrams (Last visited September 2009), http://www.standard.no/.

Shu, N. C. Visual Programming Book Van Nostrand Reinhold Company. New York, 1988

Smedley T.J. and Cox P.T. (1997), Visual Languages for the Design and Development of Structured Objects, *Journal of Visual Languages and Computing, 8*, pp. 57-84.

Sneed maintenance costs, H. Sneed, (1996) Encapsulating Legacy Software for Use in Client/Server Systems, *In The Proc. of WCRE 1996*, pp. 104-119.

Storey M.-A.D. , K. Wong, F.D. Fracchia and H. A. Müller , (1997) On Integrating Visualization Techniques for Effective Software Exploration, *In Proc. of IEEE*

Symposium on Information Visualization (InfoVis'97), Phoenix, Arizona, U.S.A., 1997, pp. 38-45.

Tilley S (1998), A Reverse-Engineering Environment Framework. *Technical Report CMU/SEI-98-TR-005*, April 1998, 44 pages.

TRACE MODE, AdAstrA Research Group, (Last visited September 2009) TRACE MODE (IEC6-1131/3, http://www.tracemode.com/products/overview/IEC61131/,

UML, (Last visited September 2009), http://www.uml.org/.

VGB, Association of power and heat generating utilities. (Last visited September 2009), http://www.vgb.org/.

Whitley K.N., A. F. B. (2001) Visual Programming in the Wild: A Survey of LabVIEW Programmers *Journal of Visual Languages and Computing*, 2001, 12, 435-472.

Wilkening D.E., Loyall J. P., Pitarys M. J. and Littlejohn K. (1995), A Reuse Approach for Reengineering. *Journal of Systems Software 30*, pp. 117-125.

Visual Servoing

Edited by Rong-Fong Fung

ISBN 978-953-307-095-7
Hard cover, 234 pages
Publisher InTech
Published online 01, April, 2010
Published in print edition April, 2010

The goal of this book is to introduce the visional application by excellent researchers in the world currently and offer the knowledge that can also be applied to another field widely. This book collects the main studies about machine vision currently in the world, and has a powerful persuasion in the applications employed in the machine vision. The contents, which demonstrate that the machine vision theory, are realized in different field. For the beginner, it is easy to understand the development in the vision servoing. For engineer, professor and researcher, they can study and learn the chapters, and then employ another application method.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Mika Karaila (2010). Model Based Software Production Utilized by Visual Templates, Visual Servoing, Rong-Fong Fung (Ed.), ISBN: 978-953-307-095-7, InTech, Available from: http://www.intechopen.com/books/visual-servoing/model-based-software-production-utilized-by-visual-templates

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821