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.

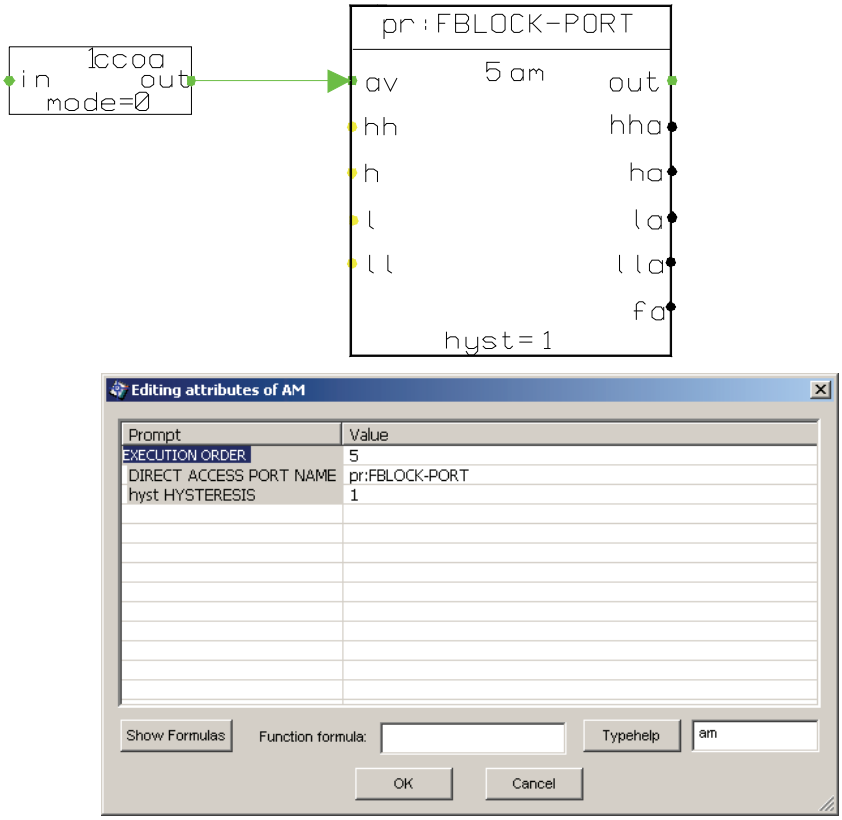

Fig. 1. Two function block symbols with the am symbol's parameter dialog.

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).

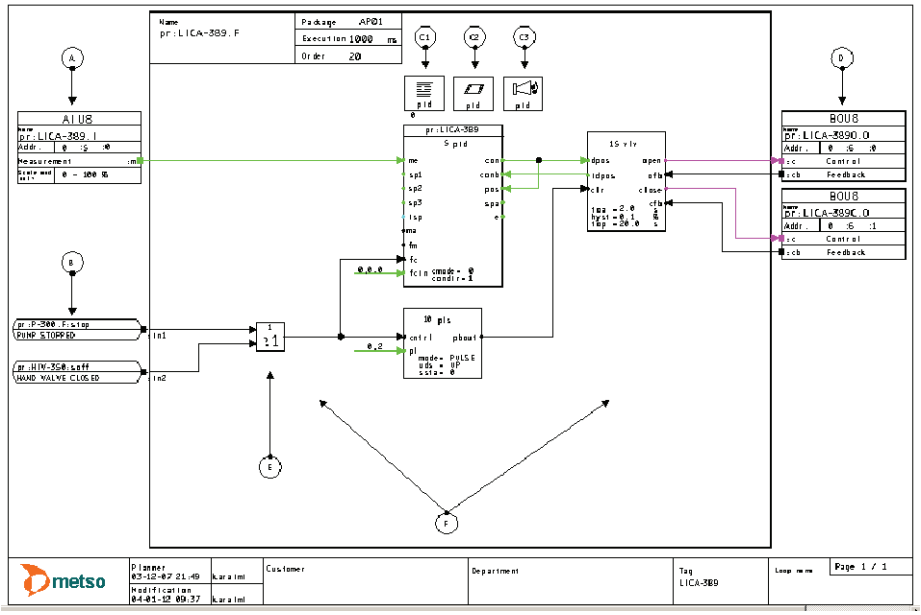

Fig. 2. FBL control loop program.

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.

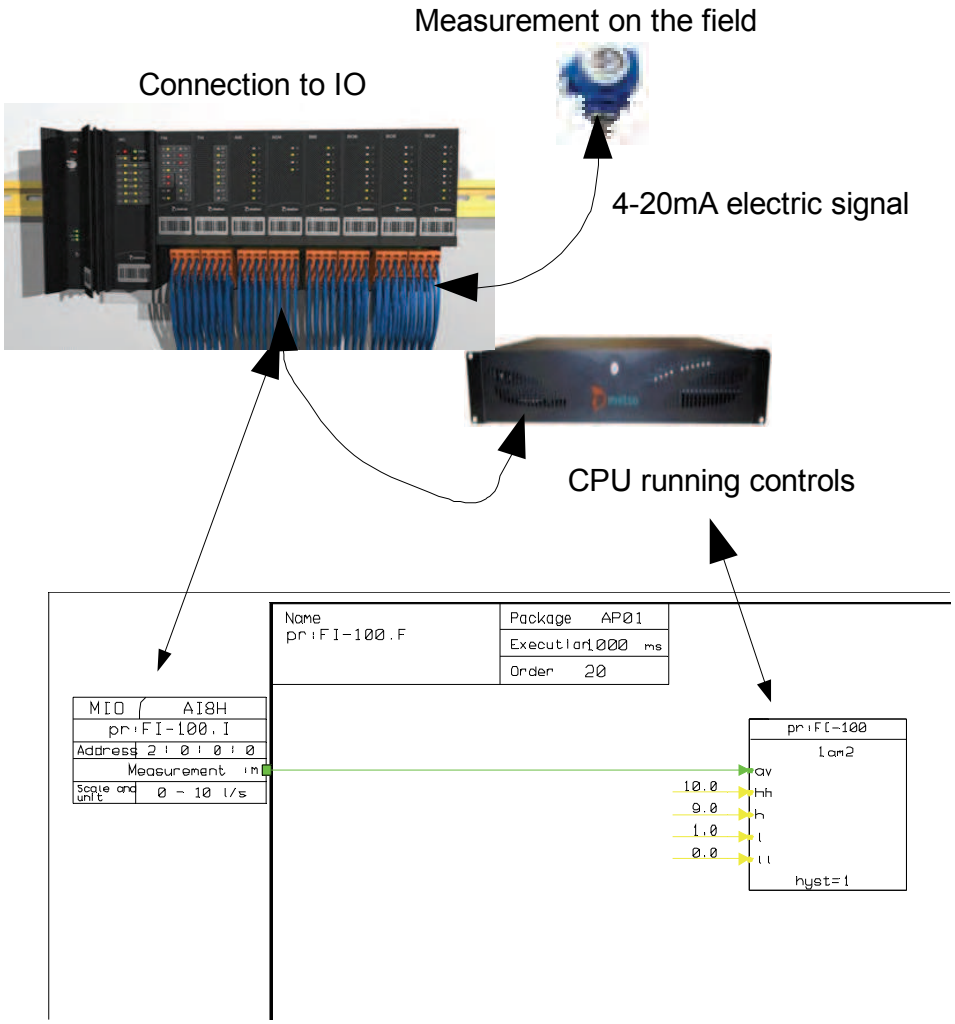

Fig. 3. I/O-signal data flow from the measuring device to the controlling CPU.

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.

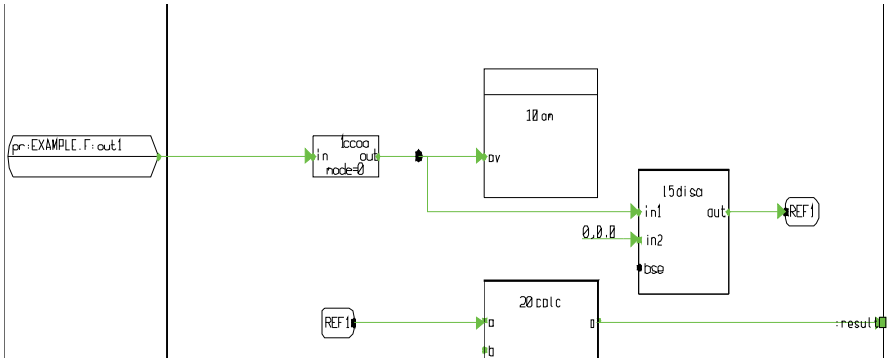

Fig. 4. Connection network examples.

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.

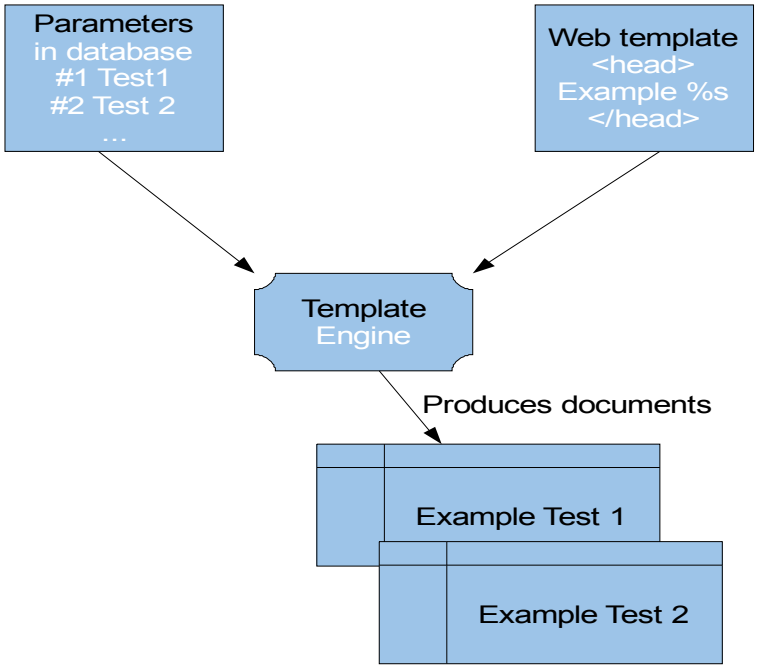

Fig. 5. Principle of web template engine.

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.

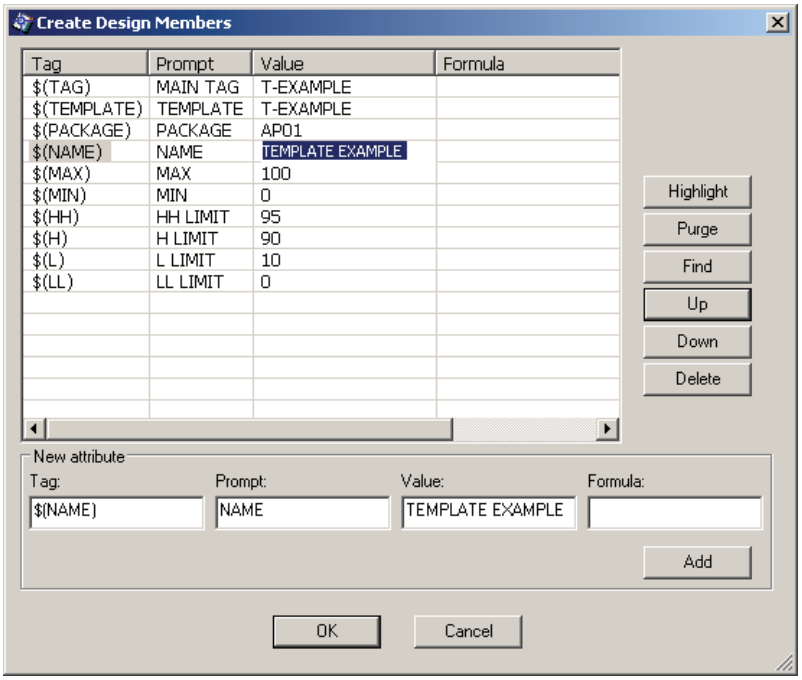

Fig. 6. Step 1: User defines first design parameters.

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)

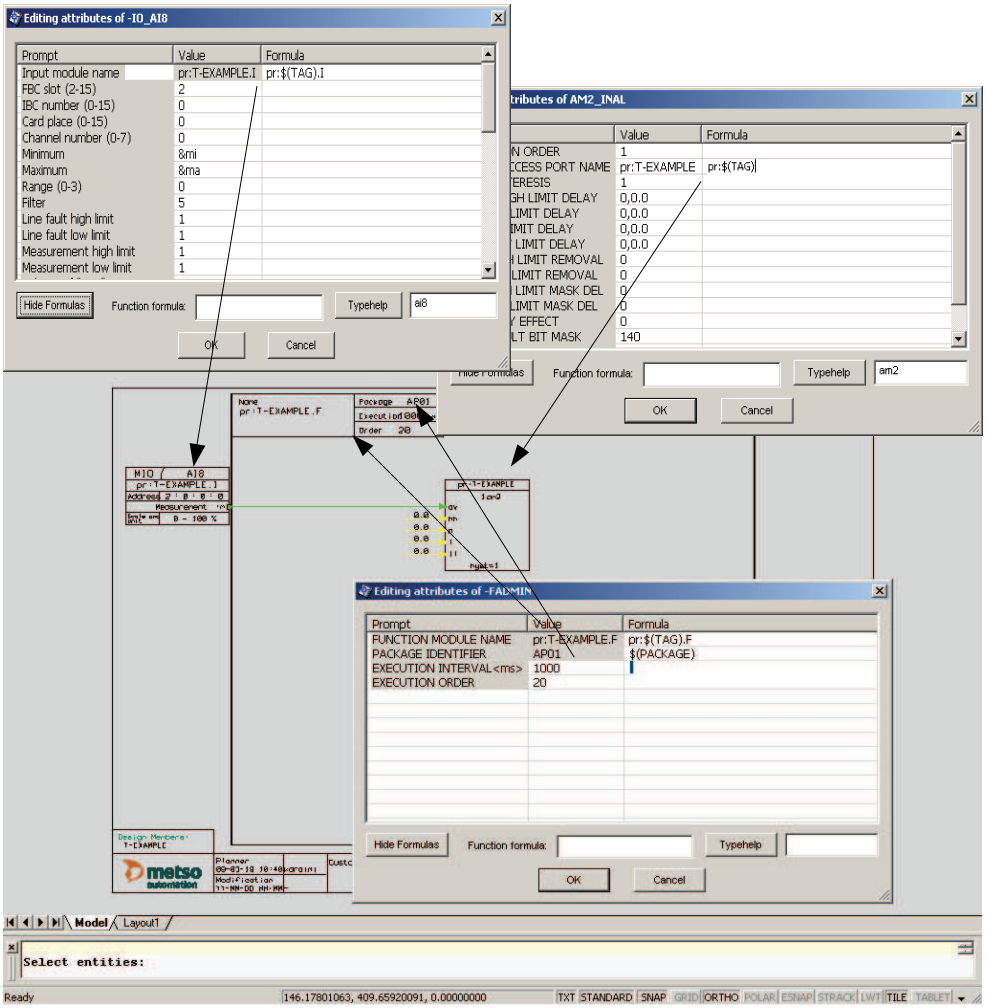

Fig. 7. Step 2: Formulas are defined in each needed location.

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

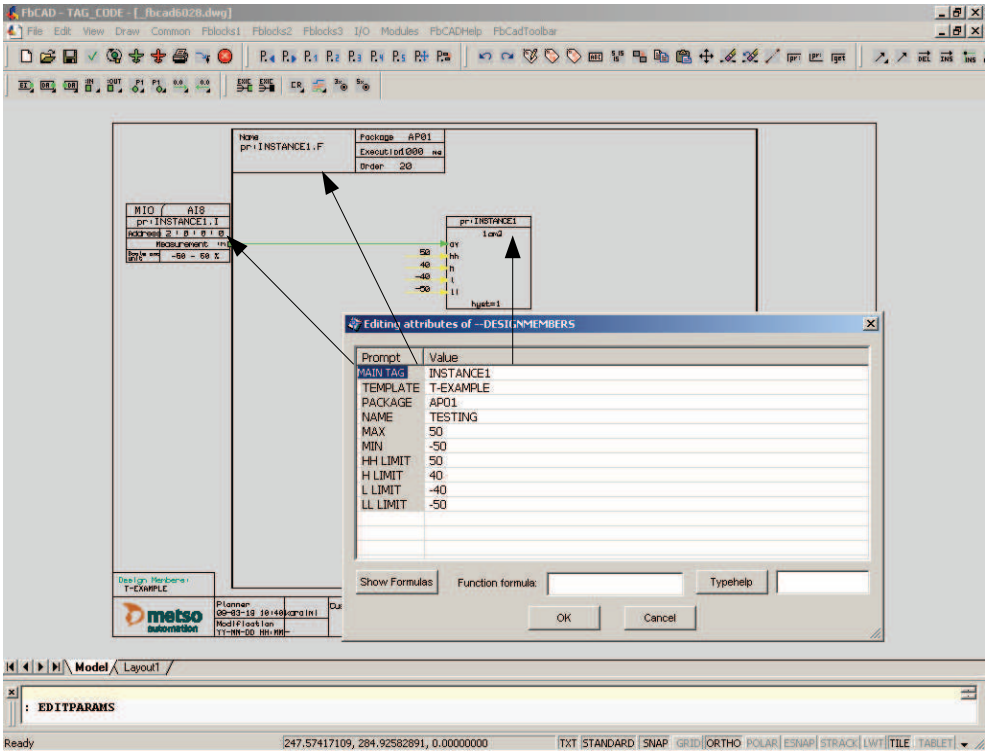

Fig. 8. Step 3: Testing template with new values. Modified design parameter values are evaluated and new values are visible in the diagram.

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.

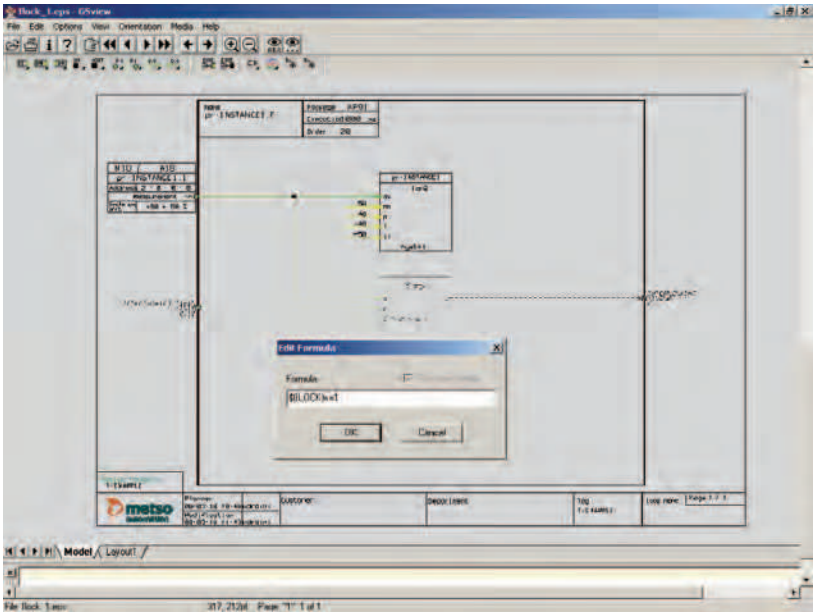

Fig. 9. Symbols are selected & active. Function formula defined for selected elements (lower function block and connections into it.

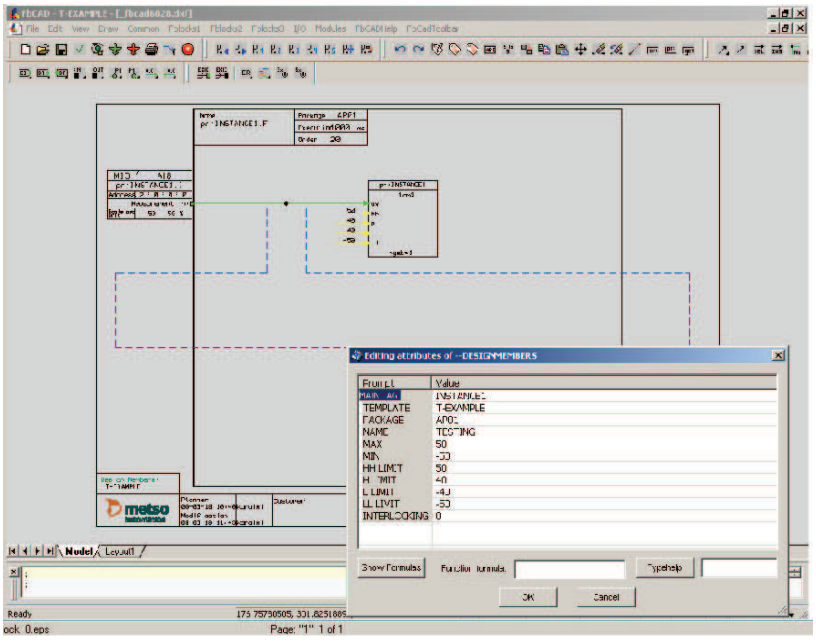

Fig. 10. Function formula 'hides' interlocking elements with the value 0. Elements can be activated with value 1 back to the diagram.

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