Evolution of a Domain Specific Language and its engineering environment - Lehman's laws revisited

Mika Karaila
Metso Automation Inc.
Lentokentänkatu 11
33101 Tampere, FINLAND
+358407612563
mika.karaila@metso.com

ABSTRACT
Automation domain is under continuous change with new requirements. Metso Automation has been one of the first vendors of digital automation systems (1978 Damatic). The last 20 years of development and maintenance of system architecture and a dynamic, flexible engineering environment has enabled us to successfully live with the changes. In this paper, evolution of a domain specific visual system configuration language called Function Block Language (FBL) and a supporting environment is discussed. The evolution is reflected to Lehman's laws. Metso Automation's solutions for surviving with the implications of the laws are also discussed.

Categories and Subject Descriptors
D.3.2 [**Language Classifications**]: Specialized application languages – *domain specific language, visual language.*

General Terms
Design, Reliability, Experimentation, Languages.

Keywords
Visual Domain Specific Language, Evolution, Software Laws, Software Patterns.

1. INTRODUCTION
A distributed control system (DCS) refers to a control system usually of a manufacturing system, process or any kind of dynamic system, in which the controller elements are not central in location but are distributed throughout the system with each component sub-system controlled by one or more controllers. The entire system of controllers is connected by networks for communication and monitoring.

For building DCSs at Metso Automation, a multi-level architecture is used in MetsoDNA DCS. Controllers use hardware units such as input/output cards (I/O) to connect field devices into a system. Software is located in controllers, I/O cards and field devices. There are different kinds of firmware programs, bus protocol stacks, operating systems and different kinds of tools and databases. An automation system executes programs in real-time in a distributed environment. It can control a small process just some devices or a huge factory with several paper machines, stock preparation and own power plant. One key element is communication between the units. Communication must be deterministic, real-time, robust and scalable.

Function Block Language (FBL), developed at Metso Automation, is a visual programming language for writing real-time control programs for distributed environments. FBL programs are represented as diagrams that will implement application programs. Each diagram typically contains 5-10 smaller application programs, which are loaded into a distributed system. A typical paper manufacturing plant automation is built from 5000 to 10000 FBL programs. They control 15000 input/output connections (I/O). Total amount of small application programs is over 100000.

FBL is part of a bigger product family that has a long life cycle. We will show how in automation domain both FBL and supporting programming environments such as FBL editor and other tools will need effort for their controlled and successful maintenance.

Section 2 introduces FBL and its history in brief. Section 3 briefly introduces Lehman's laws of software evolution [12]. They characterize the ways large software systems tend to evolve. In each subsection, Lehman's laws are discussed with respect to the automation domain, FBL and its programming environment. This section also explains the methods that we use to survive with the evolution. It will explain Metso Automation's maintenance process and its benefits. The process has been developed to manage the challenges software evolving according to Lehman's laws creates. The key idea is to manage the main principles like working methods and product features and keep the system and domain specific language in balance during evolution. Section 4 will discuss and summarize items introduced

2. Domain Specific Languages
The term domain-specific language (DSL) [2] has become popular in recent years in software development to indicate a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique [11]. The concept is not new. Special-purpose programming languages and all kinds of modeling/specification languages have existed, but the term has become more popular due to the rise of domain-specific modeling (DSM) [10]. Domain-specific languages are 4GL programming languages. Examples include spreadsheet formulas and macros, YACC [5] grammars for creating parsers, regular expressions, Generic Eclipse Modeling System [21] for creating diagramming languages, advanced DSM tool MetaEdit+ [18], Csound [19], a language used to create audio files, and the input language of GraphViz [3], a software package used for graph layout. The opposite of a domain-specific language is a general-purpose programming language, such as C or Java, or a general-purpose modeling language such as UML.

There are advantages in using DSLs. Domain-specific languages allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, programs written using domain experts themselves can understand, validate, modify, and often even develop domain-specific languages. Also, the code is self-documenting in an optimal case. Furthermore, domain-specific languages enhance quality, productivity, reliability, maintainability, portability and reusability [10]. Finally, they allow validation at the domain level.
There are also disadvantages in using domain-specific languages. The cost of designing, implementing, and maintaining a domain-specific language can be very high. Also finding, setting, and maintaining proper scopes can be difficult. There can be a lack of processing capacity compared with hand-coded software. Finally, code can be hard or impossible to debug.

2.1 An overview of FBL

Metso Automation has created its own visual domain-specific language with a supporting FBL framework, i.e. a development environment that supports FBL and its usage.
FBL drawing is a visual program that can be compiled and downloaded into a real-time system. In real-time, the program will, for example, measure a tank's level and control a valve so that tank level will remain at the desired level. An FBL program is a signal flow diagram that contains multiple symbols that are connected by lines. Symbols represent variables or variable references and functions. An FBL diagram is self-documenting, because it is a graphical program that explains the code functionality visually. The generated textual code is multiple pages of text and connections are just text references in multiple places of text. An overview of connections is very hard to understand from the textual format. 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 (ports publish access names) for other programs to access function blocks and their values. 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 cannot be accessed outside the program. A public parameter can be an interface port with a local name or a direct access port with a globally unique name.
FBL programs may also 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 calculation units. 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 calculation unit to local memory. An example of an FBL program is depicted in Figure 1. This program is for detecting binary signal change; it will read the state 0 or 1 from the field using the I/O-input symbol BIU8 (A) on the left side. Then the signal is connected to a copy function block (B) and after it to a delay function block (C) that will filter out short time (under 5 second) changes. After the delay filtering, the signal is copied to a port (D) that can be connected to the user interface (E) to show the state in the actual real-time user interface. The state is also stored in the history database (F) that keeps all the state changes for a long time (months or years). The interface port (G) is for the interlocking usage. The signal can be used in other diagrams for interlocking. If the state is for example 1 it can prevent a motor from starting.

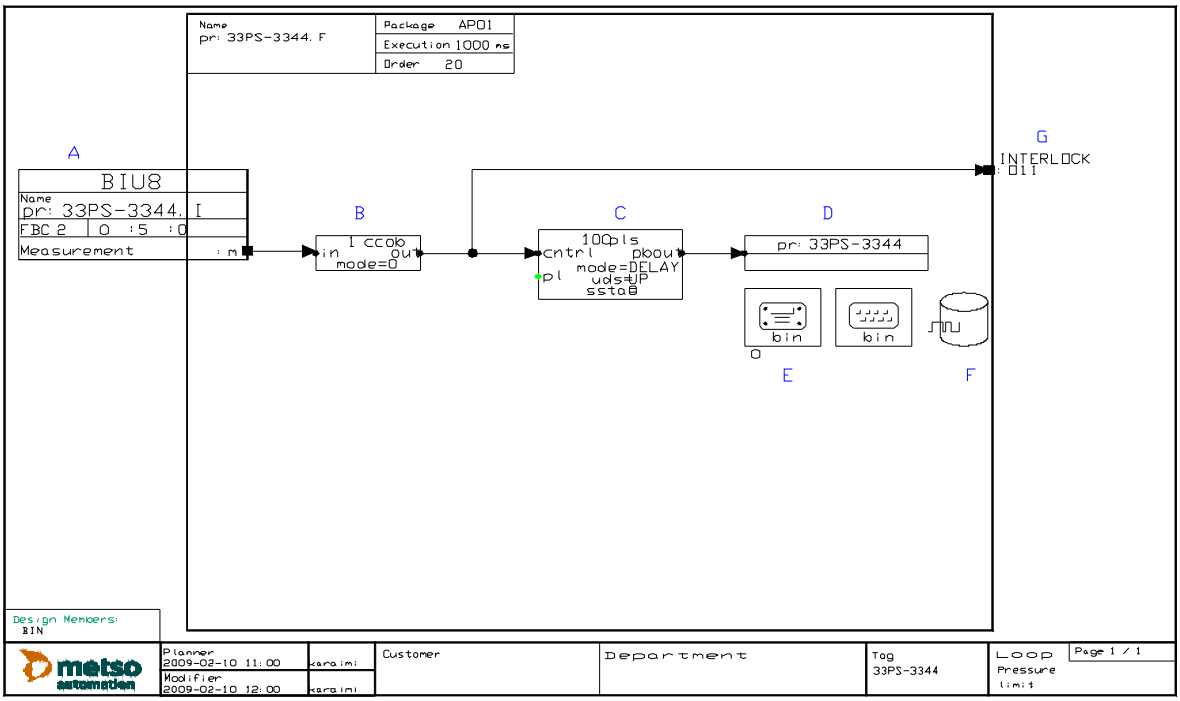

Figure 1 FBL overview.

2.2 Development history of FBL

In 1988, the first release of FBL was introduced to customers. It included all the basic elements: ports, externals, function blocks and I/O modules of the language itself. The code generator was hand written and the implementation contained a pattern based text generator [9] that used output templates. Currently, there are framework tools available for that purpose [17].

The Automation Language is a simple domain specific textual language. The FBL code generator produces Automation Language. FBL has been used from the year 1988. The FBL programming IDE, editor is FbCAD (product name for the Function Block CAD). The Function Test tool was introduced in 1990. It allows the user to debug run-time values in the FbCAD (visual debugging mode). This is not always trivial with DSL because generated code may need its own IDE that can be hard to implement. Function Explorer is a tool for multiple users in a client-server environment. It allows concurrent programming and is an easy way to change parameters for the FBL programs saved in the database. The last major development step was support for "Templates" in the year 2003. Visual templates are domain specific models that can be used to create new FBL program instances efficiently [8]. Another way to support reuse relies on patterns. In the last years, we have detected patterns from visual programs. They are solutions to small problems that can be solved with few symbols.

The development of FBL has had various goals. One of them is to aim at a visual language to make programs understandable and the programming environment easy to use and learn. The user can always add both textual and graphical comments to the program.

The initial setup for developing FBL was the fact that the automation language was too hard to understand and the connection network was impossible to figure from the textual code. The basic architecture separates user interface to its own part and the code generator to its own. In the beginning, the basic user interface was static and very simple. It was then extended, and more dynamics were added, like the visual testing tool. Now the focus of user interface development is on usability issues. Also a lot of new dynamics have been added to fulfill new requirements. Programming effectiveness is one major target. We always try to make programming faster. There are small improvements in the user interface level and in the language level, but the biggest improvement has been the introduction of templates [8]. Templates are reusable domain models that can be instantiated. An instance will need only a set of parameters to work. It has significantly reduced engineering work and made large modifications easy to make. Adding new elements to it have also extended the FBL language itself. Finally, improving the code generation in FBL has raised the quality of the programs. It is more accurate and detects more errors and also gives warnings to users.

If we compare FBL programs created in 1988 and now, the major difference is that they are now bigger and more complex. This is due to the new requirements for higher automation levels.

3. Lehman's laws and evolution of FBL and its supporting engineering environment

In this section, Lehman's laws are revisited in the context of the automation industry domain and in particular the evolution of FBL and its programming environment. Each of eight laws is discussed and the name of the law is the title of the subsection. These results are collected and formed based on 20 years of development history. The connections and network between Metso Automation's own people and customers have given a lot of feedback on FBL and its programming environment. The maintenance process of FBL itself is an iterative process that relies on the feedback system. These processes and methods can be used to gain better results and to manage the evolution FBL.

3.1 Law I: Continuing change

E-type systems must be continually adapted else they become progressively less satisfactory.

The automation domain itself is under change. In addition, also the environment that is used for FBL and its editor and other tools are under change. For instance, the operating system has changed multiple times from UNIX (Xenix, SCO, Ultrix, and HP) to DOS and Windows (NT, XP, Vista) and also the compiler is under change all the time. The CAD platform that is the base of the FBL editor has been changed according to the operating system and needed compiler. The selected CAD platform (AutoCAD) was a market leader. The use of a commercial platform helped a lot, because its development and maintenance was carried out by others. The only major work was to port the FBL user interface (editor) always into the selected release. The selection process was guided by a technology roadmap. This kind of preplanning gave development teams time to prepare for new things in advance.

This does not directly affect FBL itself, but the editor and the code generator both require major maintenance work. There is a compatibility requirement; life-time cycles are demanding in automation domain. FBL editor changes according to the style of the CAD platform and operating system. The change in the visual appearance is significant if we compare the very first 640 x 480 resolution to the current 1024 x 768 one. Outlook is also improved by new better fonts and more colors that are used today. Actual FBL improvements have been mostly visual.

As Lehman's first law indicates, resisting changes is not a fruitful solution in the long run. Instead, we have chosen to live with the changes and upgrade environment. The software environment changes in the domain create needs for changes in FBL and its tools: the domain specific language must evolve with the environment.

3.2 Law II: Increased complexity

As an E-type system evolves, its complexity increases unless work is done to maintain or reduce it.

Metso Automation DCS size grows both hardware & software. Also other additional functions make the system more complex.

As an example, new I/O-cards are needed and they include new features and more channels. FBL language supports changes, e.g. new symbols can be created into FBL. Some of these are typically similar to existing ones, like new function blocks. New function blocks do not extend FBL, but give new features for programming. This was seen already 20 years ago. The internal architecture of the code generator was built to be generic and the variation point was built into symbol level.

There are intelligent devices with new communication protocols evolving which will require support. The Foundation Fieldbus (FF) [4] integration, for instance, needed its own FBL symbols. The code generation was extended to support FF configuration. This required new semantics. This was mainly solved by the

generic part, only the connection solver needed a special algorithm and the 'output-printer' was extended for FF domain with new C++ classes. Other integrated protocols are Profibus DP [16] and OLE [13] for Process Control (OPC) [15]. They, however, did not need such big integration work for FBL.
In the FBL language level complexity is isolated to its own symbols. The code generator architecture does not need in a typical case any changes. The increased complexity is isolated into FBL editor and code generation as configurable extensions. In this way new protocol specific variations can be integrated by settings and they will not need code changes into the FBL tools every time. A typical way to reduce complexity is abstraction and capsulation, but in cases where this is not possible it is good to first identify variation points and then locate them to selected places in architecture.

3.3 Law III: Self regulation

E-type system evolution process is self regulating with distribution of product and process measures close to normal.

In the automation domain we cannot release a new build each week or month. Customers cannot shut down factories so often. A normal case is to have one planned shutdown each year, sometimes perhaps only twice a year.
In distributed automation system architecture allows that parts can be turned off and on. In this way non-critical parts can be updated/upgraded or even replaced while the process is running. Usually it is a broken device or I/O-card that must be replaced.
Same modularity can be seen in FBL language level. An FBL program can be downloaded into the system in runtime without any disturbance. The modularity makes it possible to download small application programs into the running factory without interruptions.
The evolution process that requires new technologies works according to Moore's law [14]. But in automation domain a new technology, for example a new operating system, is taken into use after careful consideration and after other industry experiences. Technological steps are taken in 2-3 year intervals. As an example, FBL editor and operating system are upgraded with that interval. A conservative attitude and caution normalize technological evolution.

3.4 Law IV: Conservation of organizational stability

The average effective global activity rate in an evolving E-type system is invariant over product lifetime.

In the automation domain you have to know something about field devices, process, and electronics. The automation system architecture and teams are formed in the same way (logical structures are similar, c.f. Conway's law [1]). Development and project organizations have been structured in the same way for about the last 20 years. The team cannot change many persons at the same time because the learning process takes time. Nobody can have all the knowledge but a good programmer must understand the domain. It usually takes months to start to understand the whole automation domain from the controller level to the device level.
A small core team is an example of good practice that allows smooth FBL development and maintenance. A challenging environment and continuous learning keep these people pleased. The needed domain knowledge that requires multi-talented people will help in keeping organizational stability.

3.5 Law V: Conservation of familiarity

As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.

The principles, namely business rules in the domain, are very old in distributed systems. As new communication protocols are integrated into the system, they will have the following typical characteristics like determinism and robustness. These kinds of facts always keep architectural solutions very stable. There are architectural level design patterns that give good solutions to these problems.
The abstraction level in the FBL language is selected to hide unnecessary parts from the end user. The needed parameters are asked from the user and the semantics and the basic layout of communication symbols have remained the same since 1989. In the FBL language the basic symbols are still almost identical.
We have to keep all new extensions somehow similar to existing ones. FBL symbol editor and FBL editor are integrated and mainly old symbols are handmade. The FBL editor has same logical operations for all similar symbols. In this way, the learning process is easier and more logical for an engineer who will use FBL. As people are conservative and do not like very big changes it helps to keep things familiar.

3.6 Law VI: Continuing growth

The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.

Automation systems are typically growing. The application programs that we have delivered to factories are growing. In the hardware architecture the old hardware was VME based Motorola 68030 processors with 2 Mb Memory. Now we use Intel based Pentium with 256 Mb Memory. Also the communication bus speeds have improved from 2 Mb/s to 100 Mb/s. In the hardware level the growth is seen clearly.
We have a reuse library that contains most of the delivered projects. We have measured from the library statistics that the average amount of function blocks in the FBL program has increased from 20 to 30 in the last 10 years (cf. Table 1).

Table 1 Project Function Blocks and Complexity growth.

Project	Average number of Function Blocks	Complexity
A 1999	15	8
B 1999	27	3
C 1999	15	5
D 1999	7	2
E 1999	28	9
F 2008	34	19
G 2008	25	10
H 2008	30	12
I 2008	28	15

The project size has grown from 1000 to 5000 application programs. This is partly due to the technology change. Field devices are more intelligent and they are connected by bus into the system. Instead of having signals connected by traditional wires there are multiple software signals coming from one physical connection. But each software signal still needs its own handling, which causes growth. This all means that the total amount of program code is five times more than 10 years earlier (1999 --> 2008).
We can get these statistics of FBL usage easily because the same working methods are used in each customer project. One part of the customer project process is to archive it.
The engineering tools are also growing. We can measure from the version control system statistics that will show the FBL code generator & DB-adapter growth from 2000 to the current year. 2008 has been about 10 kLOC, which is in average 1 kLOC/year (numbers shown in Table 2). The statistics show that FBL itself has grown during 2000-2008 with about 600 new function blocks and other symbols, the average being 75 new symbols per year.
We have thus observed also growth in the FBL language itself, not only in its programming environment. The author of this paper is not aware that Lehman's laws have been earlier discussed in the context of programming languages. They are, however, widely discussed in the context of large software systems. Continuous growth of FBL itself is interesting and due to its increasingly broad use in different contexts and by different customers. Moreover, we assume that such growth is not typical for general purpose languages, but can be more natural for domain-specific ones.

Table 2 Code generator and DB adapter size and growth.

Program	Code and Lines in Year 2000	Codes and Lines in Year 2008
Code Generator	35999	44304
DB adapter	20642	31986

These numbers show that increased functionality increases the code in the application layer (not in the system core). In the system core, the increase comes from the supported hardware, operating systems and new communication protocols. We can manage the growth because it is isolated into selected places. The variation points are designed and the solution is to use data centric generation (usually more symbols needed). The code generator core part is very stable. The initial number of symbols was approx. 500 and now it is over 1600 symbols. A good architecture helps in managing the growth. The amount of code is not growing, instead, the growth is at data level.
A very long life-cycle and evolution has not yet affected to the meta-model. The original meta-model is still used. The architecture separates extensions into symbols, and the code generator is still quite compact. The language rules and semantics are fine-tuned by the code generator.

3.7 Law VII: Declining quality

The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

The previous laws, like continuous change, increased complexity and continuing growth, are easily causing problems in quality control. Also all new features, operating system/hardware changes and new protocols can cause new bugs.
The basic architecture isolates modifications. It will also keep the system robust because a bug in one part will not crash the whole system. In the language level, FBL helps in regression testing because it can be used in different environments and different versions. All old FBL programs should be compatible upwards. This kind of FBL interoperability helps in testing. The same FBL program can be used and code generator results can be used for comparing and validating that the system is still working in the same way. Interoperability and compatibility can be used in regression testing to help in quality assurance.

3.8 Law VIII: Feedback system

E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

New communication methods like the internet and email allow customers to contact vendors much easier. The information collecting process uses data from multiple sources. The feedback process is shown in Figure 2. Wishes for new features and minor changes come from the testing and support contacts. All the testing defects are reported and stored in a database. This database is actually a huge diary that contains events that are caused by programmers or designers. The support contacts from the customers or own personnel are also stored in the database. These tools are now integrated so that the user can link and create cards just by one click. So for the development and maintenance all the defects are collected into one database. In this way, it is much easier to prioritize errors and decide who should fix and test the error and when. Information processing is now easier and project managers can focus on those errors in priority order. This makes the focus setting easier and the error handling is up to date all the time.
The process is now organized and made formal. It allows us to have the feedback system running 24/7 and also check that each case is handled. We archive and analyze all the feedback so that it helps us to improve the quality of the products. The amount of feedback issues have grown from some hundreds to over three thousand during the last 20 years. The actual reason comes from the fact that earlier issues were handled more freely. Formalized feedback entering was started at the end of the year 2004. This made it more visible and easier to statistically handle all issues. In numbers this means: bug reports over 1300/year, support cases 1800/year, dissatisfactions over 100/year and ideas 200/year. These issues concern engineering, user interface, controls and hardware parts. Most of the issues are not critical, but they are focusing mostly on user interface and usability issues today. We formalize, control and analyze all collected feedback to really improve both product quality and product features.
The whole feedback framework is made to help, link and reuse information more easily. Also tracking and testing is managed through the process. The process is more transparent and tracking from initiator, coder, and tester to final version report is possible. Each bug report or feature request has its own number and those can be selected to a version report. This makes the quality of the process better. Different views into bug records make it possible to filter and find not handled records. One way to first categorize a bug is to use architecture. The component level can be used to

assign a bug for fixing. In the same way, the project manager and project number can be found and used in the process.

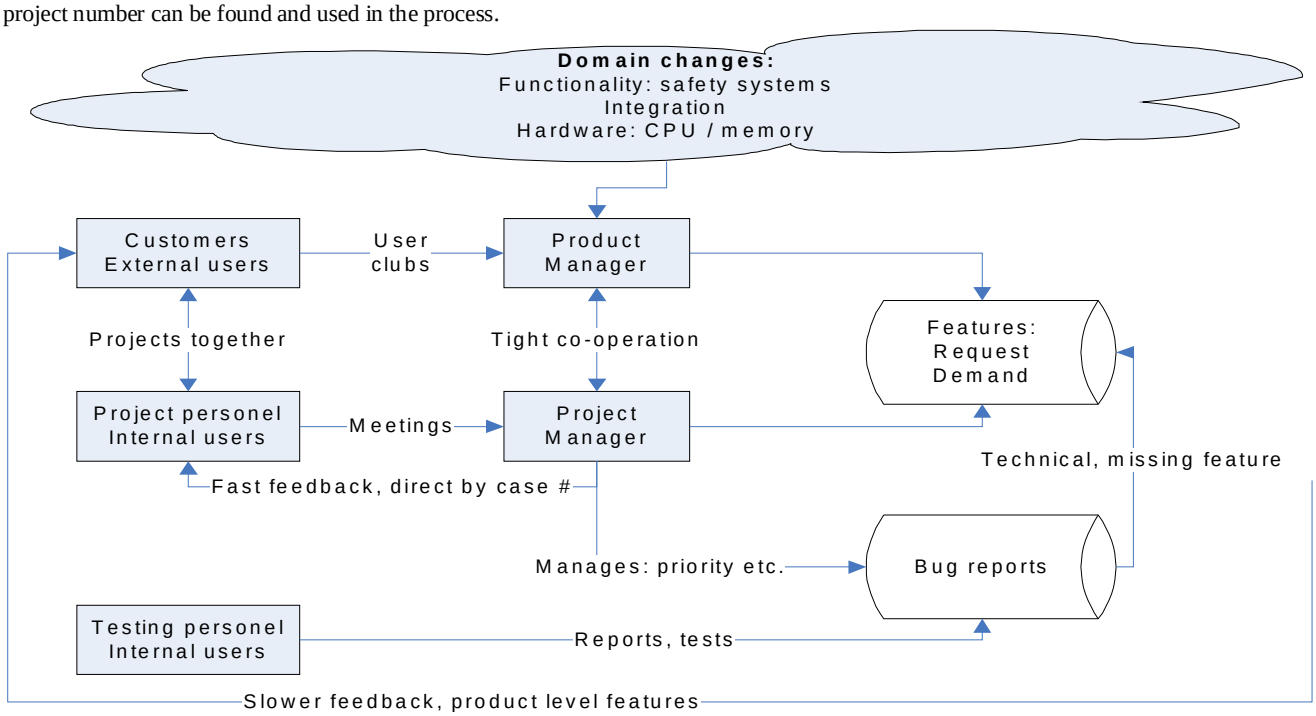

Figure 2 FBL development processes and feedback channels.

The feedback process is a multi-level, multi-feedback system and making it formal will help in tuning it.

FBL improvements that are collected are currently focused on symbols. The symbols´ size is too big or some feature is missing from the function block. Another found improvement comes from the size of the FBL programs. New navigation and intelligent find actions are needed in the FBL editor. Also a more context sensitive user interface is required. These kinds of features exist for example in Microsoft Visual Studio.

3.9 Patterns & idioms in Domain Specific Language

As discussed above, Lehman's laws for software evolution apply for the FBL environment, and partly to the FBL language itself. In addition, certain patterns / idioms seem to occur in FBL programs and eventually turn into common practices. FBL template models form patterns that are heavily reused. In the same way smaller coding patterns (idioms) exist in FBL. For instance, idioms in FBL can solve a problem that can be seen in runtime behavior. Next two small function block level examples are presented, identified from the FBL programs.

The first idiom is negation. A binary signal with a value 1 or 0 can be converted with one function block "NOT", but more common is to use "XOR" for that because if negation is not needed, XOR can be turned off by setting the input to 0. This can be done at run-time which is a very good feature in a real-time environment. The original use of NOT function block is thus not any longer so common because if the logic is designed first in the wrong way, the designer must remove the symbol and connect the signals again.

The second pattern is alarm masking. In many cases the FBL program contains function blocks that will generate an alarm. In process control there are abnormal situations like starting the process or shutting it down. In these cases there can be off limit values in the measurements. It is typical to suppress these by masking the alarm signals for a certain time like 10 to 30 seconds. A time alarm is needed because there is most probably some real problems in the process.

As design patterns are identified in traditional programming languages and there are architectural level patterns, it is natural to also find them in DSL. Besides supporting FBL programmers, they can partly support the maintenance and evolution process of the language.

4. SUMMARY

In this paper, the evolutionary history of FBL, a language used for implementing automation control programs, and its programming environment has been discussed.

The use of FBL is growing. Interestingly, we have observed that the projects that are using visual programming are very well on schedule. Component reuse is the first step in efficient programming [6, 7, 8 and 20]. Metso Automation's future work will concentrate on patterns and template maintenance and we will look for extending FBL to integrate more advanced functions.

Development agility is the part of the process that has modified FBL and the tools to be flexible. According to our experiences at Metso Automation, language development is a fascinating and dynamic challenge but it requires a well-managed maintenance and evolution process. We have also noticed that the key to such a successful and controlled evolution process is in collecting feedback from different stakeholders and in storing, managing, and using it to further enhance the language. The management

must have a very large product view and good knowledge about used techniques.

These experiences are limited to the automation industry and in particular to FBL and our experiences at Metso Automation. Therefore, we do not claim all the results can be directly generalize to e.g. general purpose programming languages. However, we feel that the process improvement and methods to live with Lehman's laws can be adapted to other cases and software maintenance. In a dynamic environment, it is very important to manage the maintenance and evolution processes.

One success factor has been that we control the maintenance and evolution process with iteration. An essential factor for the success has been a feedback handling mechanism that gives us priorities and new ideas for further development. Another success factor is architecture that is still dynamic and flexible.

The management of development and maintenance processes help in evolution. Both processes have gone through improvements and generations.

ACKNOWLEDGMENTS

The author would like to thank Tarja Systä for her comments and support in writing. This article was written at Tampere University of Technology.

5. REFERENCES

[1] Conway, M.E. 1968 How do Committee's Invent, Datamation, 14 (5): 28-31.

[2] Deursen, A. , Klint, P. and Visser, J. 2000 *Domain-Specific Languages: An Annotated Bibliography*, ACM SIGPLAN.

[3] Ellson, J. and Gansner, E. and Koutsofios, E. and North, S.C. and Woodhull, G. 2002, Graphviz— Open Source Graph *Drawing Tools Springer Berlin / Heidelberg, Volume 2265/2002, 594-597.*

[4] Foundation Fieldbus http://www.fieldbus.org/

[5] Johnson, S. C. 1975 Yacc: Yet Another Compiler-Compiler. Compiler, *Computing Science Technical Report No. 32, , Bell Laboratories, Murray Hill, NJ 07974*

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

[7] 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.*

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

[9] Kastens, U. PTG: Pattern-based Text Generator. v1.1

[10] Kelly, S. and Tolvanen, J-P.2008 *Domain-Specific Modeling Wiley-IEEE Computer Society Press, 448.*

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

[12] Lehman, M.M. ,Ramil, J F. ,Wernick, P D. ,Perry, D E. and Turski, W M. 1997 Metrics and laws of software evolution -The Nineties View Software, *Proc. of the 4th International Symposium on Software Metrics.*

[13] Microsoft OLE. http://support.microsoft.com/kb/86008

[14] Moore, G. 1965 Moore's law.

[15] OPC communication http://www.opcfoundation.org/

[16] Profibus http://www.profibus.com/

[17] Schmidt, C. and Kastens, U. and Cramer, B. Using DEViL for Implementation of Domain-Specific Visual Languages. University of Paderborn.

[18] Tolvanen, J-P and Pohjonen, R. and Kelly, S. 2007 Advanced Tooling for Domain-Specific Modeling: MetaEdit+., Computer Science and Information System Reports, Technical Reports, TR-38, University of Jyväskylä, Finland 2007, ISBN 978-951-39-2915-2.

[19] Vercoe, B. 1992 A Manual for the Audio Processing System and Supporting Programs with Tutorials.

[20] Vyatkin, V. and Hanish, H-M. 2005 Reuse of Components in Formal Modeling and Verification of Distributed Control Systems *ETFA 2005. 10th IEEE Conference on Publication Date: 19-22 Sept. 2005 Volume: 1 On page(s): 129 - 134, 2005, Volume 1, 129-13.*

[21] White, J. and Douglas C. Schmidt, 2007 A. N. E. W. Introduction to the Generic Eclipse Modeling System, *Eclipse Magazine, Vol. 6, 11-19.*