Controlling Software Complexity by Exploiting Software Similarity Patterns

Stan Jarzabek
National University of Singapore

Keywords: D.1.2 Automatic programming, D.2.2 Design tools and techniques, D.2.6 Programming Environments/Construction Tools, D.2.7f Extensibility, D.2.7g Maintainability, D.2.13c Reuse Models

Complexity of software quickly approaches the limits of what today’s programming paradigm can handle. Similarities are inherent in software. In the paper, we describe how to turn opportunities created by software similarities into a practical method to better control software complexity. The key is frame technology, capable of representing similarity patterns in a generic and adaptable form. Frames extend conventional programming paradigm with an unconventional generative technique, in a synergistic and easy to adopt way.

Software systems can comprise 10’s of millions LOC, with thousands of inter-related components, reaching a level of complexity that becomes difficult to handle with today’s technology. We’ll be surely challenged by even larger and more complex software in the future. Ultra-Large Scale System initiative targets systems of systems, comprising billions lines of code.

Control over software of that scale is unthinkable if developers continue to be exposed to software complexity proportional to software size. Especially software maintenance, which is almost exclusively done at the level of code, exposes developers to such complexity. Not surprisingly, up to 80% of software costs go to maintenance.

In this paper, we consider long-lived software systems, or systems of systems, where evolution leads to many component versions and their configurations embedded in multiple system releases. It is a major challenge to evolve system releases in independent directions, while keeping in sight what they still have in common and their overall global goal. This challenge has been poorly mastered by today’s technologies.

Decomposition (componentization), abstraction and separation of concerns are the main principles to manage the complexity of software systems. Before we analyze how they help and where they show limits, let’s have a glimpse at yet another option.

Similarities are inherent in software. By similarity patterns, we mean similar program structures, of significant importance, repeated many times within a system or across systems (e.g., in software Product Lines) and their successive versions (releases).

Many software systems are densely populated by similar program structures of all kinds and granularity, such as similar objects, components, patterns of collaborating components, and repetitions on a subsystem scale. By unifying these repetitions with generic representations, we could contract the number of conceptual elements (components and their interactions) in a program solution space, reducing the cognitive complexity of the subject software system. STL (http://www.sgi.com/tech/stl), a library of C++ data structure classes, is a classic example of simplifications that can be achieved by avoiding repetitions. However, type-parameterization (called templates in C++) and other techniques such as iterators used to build generic data structures and algorithms are not sufficient to achieve genericity in other domains (even in STL some of the repetitions remain unresolved).

While practitioners are aware of much repetition in software, they also know how difficult it is to avoid them. Problems with implementing effective reuse strategies evidence these difficulties, as well. Similarities

* System complexity if often measured in terms of conceptual elements and relationships among them that need to be understood in order to understand the system. It is this kind of cognitive complexity that we mean in the paper.
are sometimes obvious, sometimes implicit and dispersed across system components and, therefore, difficult to spot. The differences among similar program structures are often irregular and arbitrary. Our studies suggest that repetitions can comprise large parts of software systems, and avoiding them with conventional techniques is often either impossible or not achievable without compromising other important design goals\(^7,13,17\).

With the exception of poor design and careless reuse via copy-paste-modify, repetitions are most often intentional, created in a good cause, play some useful role in a program solution, and have their own merits and purpose\(^10\). For example, by cloning, code developers may achieve better software performance, higher reliability, easier portability or other goals. Some repetitions may be there because of limitations of a programming technology in its ability to define effective generic design solutions. Such repetitions cannot be avoided given design goals and technology used\(^7,8,17\). Yet other repetitions may be induced by a technology, for example, may occur as the result of pattern-driven development in modern component platforms such as .NET\(^\text{TM}\) or JEE\(^\text{TM}\), bringing beneficial uniformity of how problems are solved and standardization of software organization.

Whatever the reason for repetitions, the very presence of them is evidence that the software uniqueness space is smaller than its physical size. If noticed and tackled in a generic way, software similarities open rich possibilities for program simplification, easier maintenance and reuse rates exceeding what we can achieve with today’s component-based approaches.

How to benefit from similarities, given that most of the repetitions they trigger cannot be eliminated from programs? We believe the very nature of this dilemma requires us to look beyond conventional programming techniques.

An approach described in this paper relies on a synergistic application of a frame-based\(^4\) generative technique of XVCL\(^16\) together with conventional programming languages, design techniques and component platforms. We use XVCL to unify each of the important similarity patterns with generic, adaptable framed program structures. Framed software representations reveal a non-redundant face of software system, contracting the number of conceptual elements (components and interactions) in the program solution space a developer must deal with. In addition to explicating similarity patterns, framed software representations also contain yet other information about program design and software relations that help in understanding, maintaining and reusing programs. Conventional programs, automatically obtained from their framed representations, contain repetitions that must necessarily be there, but they do not contribute to the programs’ complexities as perceived by a programmer anymore.

By unifying similarity patterns, we achieve simplifications proportional to the similarity rates, which are often proportional to the size of software systems. Validation (e.g., by testing) or formal verification (e.g., by model checking) at the level of a non-redundant representation is easier than at the level of concrete programs or systems. The larger software’s scale, the higher similarity rates, and the greater benefits from tackling them. Even on a small scale, we typically find 60%-90% of code repeated in variant forms\(^7,13,17\). The engineering benefits of frames become evident especially if we pay attention to large granularity repetitions.

The frame approach requires only modest extensions to conventional programming methods. It goes hand-in-hand with principles of abstraction and separation of concerns, as much as it is practical to achieve the engineering goals of software simplification.

**SOFTWARE SIMILARITY PATTERNS AND COMPLEXITY**

Suppose we have 10 user interface forms \(a_1, \ldots, a_{10}\) (e.g., data entry forms). Each form, say \(a_i\), interacts with 5 business logic functions \(b_i, c_i, d_i, e_i, f_i\) (e.g., data validation rules or actions to be performed upon data entry at certain fields). We further assume that there is considerable similarity among the 10 forms, and business logic functions that each form needs. (You can think about user interface forms and business logic functions as any components, subsystems or interacting systems.)

If each form and each business logic function is implemented as a separate component, we have to manage \(10 + 10 \times 5 = 60\) components and 50 interactions (situation shown in Figure 1 (a)).

Suppose we unify similar business logic functions with five generic business functions \(B, C, D, E, F\). We reduce the solution space to \(10 + 5 = 15\) components and 50 interactions, with the slightly added complexity of a generic representation. The interactions have become easier to understand, as each form interacts with

\(^4\) XVCL (XML-based Variant Configuration Language) is based on Bassett’s frame technology concepts
components that have been merged into five groups rather than 50 distinct components (situation shown in Figure 1 (b)). Suppose we further unify 10 user interface forms with one generic form A. We reduce the solution space to $1 + 5 = 6$ components, with five groups of interactions, each group consisting of 10 specific interactions, plus the complexity of a generic representation (situation shown in Figure 1 (c)).

Software validation and changes performed at the level of the generic components would be easier than at the level of concrete components due to smaller number of distinct components and interfaces that have to be analyzed and clearer visibility of change impact. Generic software structures also form natural units of reuse, potential building blocks of other similar systems (e.g., forming Product Lines).

![Diagram](image1.png)

**Figure 1. The impact of unifying similarities with generic structures on complexity**

The above situation has been abstracted from a Domain Entity Management Subsystem (DEMS) of a command-and-control application developed in C# by our industry partner ST Electronics Pte Ltd (STEE). DEMS involves domain entities such as User, Task or Resource. For each entity, there are operations, such as Create, Update, View, Delete, Find or Copy. STEE’s intention was to make DEMS a part of the Product Line architecture, so DEMS was meant to be generic, adaptable and extensible.

The design of each operation such as CreateUser or Create Task involves a pattern of collaborating classes from GUI, service and database layers. Each box in Figure 2 represents a number of classes: GUI classes implement various forms to display or enter data; Business Logic classes implement data validation and actions specific to various operations and/or entities; Entity classes define data access; classes at the bottom contain table definitions.

Classes in corresponding boxes at each level display much similarity, but there are also differences induced by different semantics of domain entities: For example, operation CreateTask requires different types of data entry and data validation than CreateUser.

![Diagram](image2.png)

**Figure 2. A recurring pattern of components**

Similarities among operations are so substantial that it is tempting to view operations as instances of a generic operation. C# language features (including generics proposed for C#) and .NET mechanisms were
first considered to define such generic solutions. However, the nature of variations across operations was such that it was not possible to design a generic solution for groups of similar operations (e.g., Create[entity]Form or Update[entity]Form). Therefore, implementation of operations for different entities was replicated many times, with required variations. DEMS comprised 18,823 LOC of C# code, with 117 classes covering GUI, service and database layers.

**THE FRAME APPROACH TO MANAGING COMPLEXITY**

While repetitions are often unavoidable and intentional in executable programs, they can be avoided (unified) with unconventional generic representations. This is analogous to Aspect-Oriented Programming\(^1\) where a *meta-level extra plane* is created to modularize concerns that otherwise inevitably crosscut conventional program modules (classes).

The above observations motivate the frame approach. Developers write the functional core of their program solutions in programming languages of their choice. They use XVCL\(^2,16\) to organize code in-the-large to achieve non-redundancy, ease of changing, reusing and versioning.

In a nutshell, XVCL (1) represents each significant group of similar program structures in a unique, generic, but adaptable form, (2) delineates the differences among specific program structures in each group as deltas from their generic form, (3) records the exact location of each such structure in a program, and (4) automates deriving specific structures from their generic forms to produce an executable program.

**Figure 3. Hierarchical unification of similarities**

Figure 3 shows an outline of non-redundant DEMS in C#/XVCL software representation. Boxes represent solution’s building blocks called frames\(^3\). At Level 3, each group of operations such as CreateUser, CreateTask, … has been represented by one generic operation parameterized by a respective domain entity. Similarities among different operations for the same entity (e.g., CreateUser, UpdateUser, …) are unified at Level 4. Frames at Level 4 represent generic classes, building blocks for DEMS operations, as indicated by frames referenced from more than one operation (e.g., generic classes labeled with CU are reused in construction of Create and Update for various entities). The top-most frame, called an SPC, contains global controls and parameter settings that specify the overall process of constructing DEMS from frames below. “DEMS template” at Level 2 defines the structure of DEMS architecture, that is the organization of component patterns implementing various operations plus any other functions supported by DEMS, not discussed in this example.

The XVCL Processor interprets the framed representation to synthesize DEMS in C#.

In this sketchy solution outline, we do not distinguish among classes belonging to the four system layers. In most systems, software structures such as in Figure 3 are implicit, and must be expressed in the form of external documentation. XVCL makes such views explicit and an integral part of a formal program representation. Frames contain both code and the knowledge of program design, integrated into one representation.

Now we zoom into some details of the framed software representation to see how and why the solution works. Some highlights of the solution are shown in Figure 4.
SPC // specifies how to generate all the operations for entities in DEMS
<set oper = Create, Update, View, Delete, Find />
<while oper >
<select option = oper >
<option Create >
<adapt DEMS-template >
customizations for Create only
<insert For_Create Only />
<option Update >
<adapt DEMS-template >
customizations for Update only
<otherwise >
<adapt DEMS-template >
customizations for all the remaining operations
</select>
</while>

While entity >
<select option = entity >
<option User> <adapt @oper>[E] >
customizations for User only
<option Task > <adapt @oper>[E] >
customizations for Task only
<otherwise > <adapt @oper>[E] >
customizations for all the remaining entities
</select>
</while>

Create[E]
code for Create
<br>For_Create
<break For_Create />
update <insert>ed <adapt>
Update[E]
code for Delete
<br>For_Delete
<break For_Delete />
view <insert>ed <adapt>
View[E]
code for Update
<br>For_Update
<break For_Update />
delete <insert>ed <adapt>
Delete[E]
code for Display
<br>For_Delete
<break For_Delete />

Figure 4. Flexible “composition with adaptation”

An arrow between two frames: X → Y reads as “X adapts Y”, meaning that X controls adaptation of Y. Custom operations for specific entities (e.g., CreateUser) are obtained by adapting respective generic operations (e.g., Create[E]).

The XVCL Processor interprets a framework starting from the top-most SPCification frame, called SPC, traverses frames below, adapting visited frames and emitting C# code implementing DEMS operations shown in Figure 2.

XVCL variables and expressions provide the parameterization mechanisms to make frames generic. A <set> command assigns a list of values to an XVCL variable. Frame variables oper (set in SPC) and entity (set in DEMS-template) are generic names for DEMS operations and entities, respectively. During frame processing, values of variables propagate from a frame where the value of a variable is set, down to adapted frames. Context scoping allows variables to coordinate chains of customizations for a given source of change, spanning multiple frames.

In each iteration of a <while> loop in the SPC, we instantiate patterns of components implementing operations Create, Update, View, and others. The i’th iteration of the loop uses the i’th value of its control variable oper listed in the respective <set> command. Unique customizations required for specific operations are specified under a suitable option of the <select> command. We see a similar solution in the frame DEMS-template to specify unique customizations required for specific entities.

By varying specifications, we can instantiate the same framework in different ways, deriving different programs from it. In that sense, a framework forms a generic design structure that enables reuse within a single program, or across programs. In the latter case, a framework implements the concept of a Product Line architecture.

The <insert> into <break> command has a similar effect to weaving aspect code in AOP. Functions specific to the operation Create are defined in the frame For_Create Only. With <insert For_Create> in the SPC, we insert Create-specific functions at designated variation points <break For_Create> in frames DEMS-template and Create[E], as shown by dashed arrows. This example illustrates how we deal with ad hoc variations related to specific operations, without affecting other operations that should not be affected by these variations. Mechanisms for such selective injection of changes allow us to separate variants from common, generic structures, keeping generic structures reusable and easily adaptable.
Frames contain both code and the knowledge of program design, integrated into one representation. For example, the C#/XVCL framed software representation contains C# code needed to produce all the DEMS operations, and also information helpful in maintenance/reuse, such as the record of similarities and differences among operations for different entities. In most systems, such information is implicit, and must be expressed in the form of external documentation. XVCL makes it an integral part of a formal program representation.

In this example, the size of the C#/XVCL representation is 68% smaller (in terms of lines of code, without blanks or comments) and conceptually simpler than its C# counterpart. Size reduction also leads to improved maintainability. Typical enhancements (such as change of business rules for domain entities or adding/deleting a new entity or operation) require less modifications with smaller impact in the C#/XVCL DEMS as compared to its C# counterpart. For example, it is eight times faster to create a new domain entity in the C#/XVCL DEMS (133 LOC and 2 man-hours), as compared to C# DEMS (1440 LOC and 16 man-hours). Finally, it is quite easy to reuse the C#/XVCL DEMS in other similar command and control systems.

The reader should notice that adding more operations and entities to DEMS in C# increases the complexity by a constant value, independently of the similarity of new operations/entities to existing ones. In framed C#/XVCL representation, complexity grows proportionally to the level of novelty that a new operation/entity brings. New operations/entities that differ little from existing ones require little new code to be written.

In our studies of new, well-designed programs, we typically find 50%-90% of code contained in similar program structures, repeated many times. For example, the extent of the redundant code in the Java Buffer library was 68% \(^7\), in parts of STL (C++) - over 50% \(^1\), in J2EE Web Portals – 61% \(^17\), and in certain ASP Web portal modules – up to 90% \(^13\).

**FRAME TRADE-OFFS**

Frames make generic design easier, but it does not come for free. Designing generic, reusable and maintainable solutions is always a challenge which requires more talent and skill than building a concrete program. Frames require an up-front investment in adaptable software assets in exchange for future system design, development, and evolution benefits.

By separating genericity issues from the core constructs typically supported by programming languages, we can address genericity concerns without compromising program runtime properties. But as we move towards less restrictive parameterization mechanisms, we also decrease type-safety of parameterized program solutions.

Industrial applications have demonstrated that XVCL is easy and fast to learn, and its benefits usually greatly outweigh the cost of the added learning curve\(^13\). At the same time, the return on investment may be quick and substantial. Industrial applications have also contributed to our understanding of added complexities and their offsets such as the difficulty to encompass the two intermixed levels, in base programming language(s) and XVCL. However, we must keep in mind that a framed program representation contains much useful information for maintenance and reuse, in addition to complete information about the subject program(s) itself. Debugging is yet another area into which XVCL induces extra complexities. On the other hand, most errors can be traced to the least tested code, which is usually concentrated only in top-level frames. The XVCL Processor tags each generated line with the SPC line that was active when the line was generated. Thus when an error occurs at some random line of code, developers know where to look to find its most likely cause. Tools may considerably simplify the application of XVCL. We are developing an XVCL IDE called XVCL Workbench that helps in editing, visualizing, debugging and static analysis of frameworks. (Figure 5). XVCL Workbench helps developers to browse/analyze frameworks (in the Project Explorer window) and frames (in the Outline Window), structure in XML-free format. The upper right-hand-side window shows an frame in raw XML format. In the future, a developer will be able to ask queries about properties of frames, and run the Processor in a debugging mode. XVCL Workbench is implemented as a plug-in to the Eclipse platform.
PERSPECTIVES ON SOFTWARE COMPLEXITY

Twenty years ago, Brooks pointed to essential complexities in software that could not be reduced below a certain threshold\(^4\). None of the technological advances of last decades proved Brooks wrong.

Componentization, abstraction and separation of concerns have been prime principles to tackle software complexity. However, there are no easy solutions.

Software components tend to be coupled one with another, in explicit (via interfaces) and implicit ways. Understanding one component requires knowledge of yet other components. Software components have not become a commodity to the extent as optimistic forecasts of late 1990s were predicting (e.g., vide Cutter Consortium and Gartner reports).

Difficulties to realize component-based reuse hint at fundamental limits of what we can achieve by means of “divide and conquer” componentization. Lego-like componentization does not fit software as well as they do physical artifacts that traditional engineering deals with\(^3\). Software architectures are more enigmatic and less stable than hardware architectures. Software components are also less stable, and interact in more complex ways than hardware components. They are often changed, rather than replaced, therefore must be much more flexible than hardware components.

However, the most serious limitation of componentization in fighting complexity is that as long as we develop, test and maintain software in terms of concrete, executable components, we are exposed to complexity being proportional to system size. In a system of thousands of interacting components, the complexity of validating, testing and controlling maintenance changes is bound to explode beyond control. In author’s view, this complexity is part of the Brook’s essential complexity\(^4\).

Separation of concerns is a powerful concept, however it is not easy to bring it down from the concept to the design and implementation levels. Only a restricted number of computational concerns can be componentized, or separated with techniques such as AOP\(^1\) or MDSC\(^1\).

Abstraction can shield developers from some of the essential complexity. Abstraction can play more than merely descriptive role in narrow, well-understood domains, whereby we can make assumptions about the problem domain semantics. By encoding domain knowledge into generators, we let developers work at abstract levels of program description in a Domain-Specific Language (DSL), with a generator filling the missing details. This can lead to substantial productivity gains, best exemplified by compiler-compilers.

Advancements in modeling and generation techniques led to recent interest in Model-Driven Engineering (MDE)\(^1\), where multiple, inter-related models are used to express domain-specific abstractions. Models are
used for analysis, validation (via model checking), and code generation. Platforms such as Microsoft Visual Studio™ and Eclipse™ support generation of source code using domain-specific diagrammatic notations.

While helpful and insightful, generator-based solutions face problems that have hindered their deeper penetration of the programming practice. The first problem is the likelihood of disconnecting models from code during evolution. This occurs when the DSL cannot cater for unexpected evolutionary changes and developers modify the generated code. Round-trip engineering has been notoriously difficult to achieve, and modified code needs to be maintained by hand, with no help from models or a generator. Such disconnection is most likely to happen if multiple and independently evolving program versions originate from a generator: A change specific to one such program, if implemented in the generator, automatically propagates to other programs that may not need the change—a mostly undesirable effect. Implementing a change into a specific program disconnects it from the generator. Supporting multiple component versions and system releases via generators poses a fundamental problem.

The above problem is compounded by the difficulty of integrating multiple models/generators to build systems we need. Typically, a problem domain served by a generator covers only a part of a given system. Strategies for integrating multiple domain-specific generators and embedding them into systems implemented using yet other techniques have yet to be developed. One of the reason for success of compiler generators is that compilation on its own is a self-contained domain.

The first mentioned problem motivated the invention of frames. Netron Inc. used DSLs from which conventional generators produced framed code, so that programmers could place all custom code in SPCs, separated from the generated code. In that way, customization and re-generation of code could be repeated independently of each other. Frames themselves, viewed as a generation engine, are free of the model-code disconnection problem, as developers perform all the maintenance/reuse work at the level of the frame model rather than at the level of code produced from frames.

The frame approach addresses also the second mentioned problem of integration. Not only do frames support multiple component versions and system releases via generators, but they can also integrate multiple models/generators when building or evolving complex systems. Netron’s customers routinely build and evolve multi-million line systems this way.

SOFTWARE SIMILARITY PHENOMENON

Just like certain computational aspects necessarily crosscut conventional program modules, certain similarity patterns necessarily lead to software redundancies. These redundancies show as similar program structures of different type and granularity.

Design-level similarities, showing as patterns of classes, components or subsystems, often represent domain-specific abstractions (e.g., operations on domain entities in DEMS, Figure 2). Generic structures for such similarities are particularly important as they describe how domain concepts are realized in the solution space. Such similarity patterns create opportunities for reuse within a given system, or across similar systems.

Conventional methods—component based, architecture-centric approaches as well as language-level features such as generics—often fail to reap benefits of software similarities, for variety of reasons. Many repetitions are just meant to be there, so the very idea of eliminating them is not acceptable. Repetitions introduced for performance or reliability reasons, and those induced by pattern-driven development (e.g., on .NET or JEE) belong to this category.

In cases where there is no higher-level reason for repetitions, limitations of a programming language or platform often hinder conventional generic solutions. Ad hoc, irregular variations among similar program structures do not make it easy to refractor repetitions into a generic representation. Software design is constrained by multiple, sometimes competing, design goals, such as development cost, usability, simplicity, performance, or reliability. A generic solution to unify similarity patterns can only win if it is natural and synergistic with all the other design goals that matter.

Yet other similarity patterns may not correspond to any conventional abstraction (such as class or component) suitable for their representation. Patterns of collaborating classes or components are often in this category.

In the frame approach, not being constrained by rules of the underlying programming language or platform, any recurring program structure worth attention can be captured in a generic form for simplification and reuse, independently of its type, granularity or the reason for its presence.

Looking for deeper roots of the software similarity phenomenon, feature combinations are one of the major forces triggering the spread of similar program structures within and across systems. Functional abstractions
(e.g., components, classes or functions/methods) implement the net effect of the required feature combination that is necessary for correct execution. As we combine features, variations arise that force us either to find a unifying generic solution, or to create multiple program structures in variant forms. In industrial software Product Line projects, feature combinations may lead to thousands of similar component versions.

**SOFTWARE SIMILARITIES AND REUSE**

The prime objective for reuse is the same as for generic design – to avoid repetitions. Current state-of-the-art in reuse is based on architecture-centric, component-based and Product Line concepts. Having scoped variability (usually modeled as feature diagrams), a Product Line architecture (PLA) is designed to accommodate variant features. With component-based design, genericity of a PLA is achieved by stabilizing component interfaces and localizing the impact of variant features to possibly small number of components. However, for variant features that have crosscutting effect on PLA components, this goal cannot be easily achieved. A component affected by variant features explodes into numerous versions, similarities among component versions cannot be easily spotted, and reuse opportunities offered by such similarities are usually missed. Given thousands of variant features and complex features inter-dependencies arising in industrial Product Lines, the cost of finding “best matching” component configurations for reuse, and then the follow up component customization, integration and validation, may become prohibitive for effective reuse.

With the frame approach, we treat similarity patterns induced by variant features as first class citizens, no matter whether they have a crosscutting effect or not. Framed generic structures unify similarity patterns at any granularity level and of any type – from a subsystem, to a pattern of components, to a component, to a class and to a program statement within a class implementation. For that reason, the application of frames often extends the scope and rates of reuse achievable by means of conventional techniques.

Conventional component-based reuse is most effective when combined with architecture-centric, pattern-driven development which is now supported by the major platforms such as .NET™ and J2EE™. Patterns lead to beneficial standardization of program solutions and are basic means to achieve reuse of common service components. IDEs support application of major patterns, or developers use manual copy-paste-modify to apply yet other patterns. Frames can enhance the benefits of modern platforms by automating pattern application, and emphasizing the visibility of patterns in code. Pattern-driven design facilitates reuse of middleware service components, but tends to scatter application domain-specific code. With frames, we can package and isolate otherwise scattered domain-specific code into reusable generic components. Such extensions improve development and maintenance productivity, and allow reuse to penetrate application business logic and user interface system areas, not only middleware service component layers.

**HISTORY OF XVCL**

XVCL has its roots in Bassett’s frame technology. A number of frame-based systems have been implemented in both industrial and academic institutions. Netron Inc. has extensively applied frames to maintain multi-million-line COBOL-based information systems and to build reuse frameworks in companies. An independent assessment by QSM Associates, Inc. showed frames achieving 90% reuse, reducing project costs by over 84% and their time-to-market by 70%, when compared to industry norms.

XVCL was developed in 2000 at the Software Engineering Lab of National University of Singapore, in a joint Singapore-Ontario research project between National University of Singapore, ST Electronics, University of Waterloo and Netron, Inc. In 2002, the XVCL Processor became available at SourceForge (fxvcl.sourceforge.net). We have demonstrated that the principles behind XVCL have not been superseded by object, component and other mechanisms supported by modern programming platforms. On contrary, XVCL contributes unique engineering values to today’s programming paradigms, in the area of designing generic, adaptable, software representations that reveal a simpler face of software than it is possible with conventional programming approaches.

**REFERENCES**