Presented at Workshop on New Visions for Software Design and Productivity: Research and Applications, Vanderbilt University, Nashville, Tennessee, December 13-14, 2001.

ExCIS: An Integration of Domain-Specific Languages and Feature-Oriented Programming[1]

Don Batory[a], David Brant[b], Michael Gibson[b], Michael Nolen[c]

[a]UT Center for Advanced Research in Software Engineering (UT ARISE)
University of Texas at Austin
Austin, Texas 78712
batory@cs.utexas.edu

[b]UT Center For Agile Technology (UT CAT)
University of Texas at Austin
Austin, Texas 78712
brant@mail.utexas.edu
michael@arlut.utexas.edu

[c]ExCIS Project Director
U.S. Army Simulation, Training, and Instrumentation Command (STRICOM)
Michael_Nolen@stricom.army.mil

ExCIS is a methodology and tool suite that integrates the technologies of *domain-specific languages (DSLs)* and *feature-oriented programming (FOP)*. DSLs offer compact specifications of programs in domain-specific notations that are easier to write, maintain, and evolve. FOP raises the level of abstraction in system programming from compositions of code-centric components to compositions of modules that implement individual and largely orthogonal features. ExCIS provides state-of-the-art tools for creating easier-to-specify, easier-to-maintain, and easier-to-change systems. It is being developed for STRICOM to create next-generation simulators for the U.S. Army.

1 Introduction and Motivation

The *Test and Evaluation (T&E)* instrumentation community works within one of the most challenging development environments. Test instrumentation is tightly coupled to the intended *system under test (SUT)*. Functional and technical requirements for the instrumentation are driven by the SUT, and are often not known in detail until the SUT is nearing the end of its development cycle. In practice, even after design level requirements are known, the SUT implementation continues to change until the actual test. Since the test instrumentation is dependent upon the SUT, any changes in the SUT often must be reflected in the instrumentation.

This problem is not necessarily due to poor management or development practices employed in building the SUT. It is intrinsic to the nature of software-driven systems. As the development of the SUT proceeds, knowledge is gained, versions built, and lessons learned. This feeds an iterative learning process that drives development decisions. The instrumentation developer must track and react to these changes – changes that often require altering the instrumentation. Without the ability to quickly respond to change, the instrumentation system will fail to fully support the tester, often resulting in expensive work-arounds or an inferior test.

A second major challenge is the maintenance of the instrumentation system once developed. After deployment, the SUT continues to evolve – fixes, performance improvements, and enhanced functionality all drive system changes. Current practices can lead to the creation of instrumentation systems whose yearly maintenance costs equal the yearly costs experienced during development. Thus, the cost of ownership is often beyond the resources available to the tester. This can lead to instrumentation systems that quickly become burdens to their owners.

The challenges of decreasing software development cycle time and reducing the cost of ownership require novel development approaches. A new way of approaching the problem must be explored.

2 New Thinking and New Technology

Today's software development practices are too low-level – exposing classes, methods, and objects as the focal point of discourse in software design and implementation. This makes it difficult, if not impossible, to reason about software architectures in the context of functional requirements (a.k.a. component-based designs); to create, simple, ele-

1. U.S. Army Applied Research and Engineering Development (ARED) Contract #N61339-99-D-0010.

gant, and easily understood specifications of applications; and to build and critique software designs automatically, given a set of high-level requirements.

Simple specifications that are amenable to automated reasoning, code generation, and analysis are indeed possible provided that three fundamental advances occur. First, that the level of programming be raised from generic languages (C++, Java) to *domain-specific languages (DSLs)*, where programs are expressed simply and compactly in notations specialized for particular tasks. Because DSL programs are written in terms of domain-specific concepts and relationships, it is well-known that their maintenance costs are lower, they are amenable to specialized analyses and optimizations that general-purpose compilers could never attempt, and they produce superior software designs (because the designs were created by experts) [5][3].

Second, the discourse on software architectures must be elevated from interactions among low-level, code-centric components to that of units which encapsulate the implementation of individual and largely orthogonal *features* that can be shared by multiple applications[2]. The intuitive rationale for this shift is evident when software products are described: their descriptions typically are *not* in terms of the code modules and DLLs that are used, but rather in terms of features the product offers its clients. That is, the focus of discourse is on *features* and *not on code* [8]. By moving away from code-centric programming to *feature-oriented programming (FOP)*[3], a revolution in the way software is understood, constructed, modified, and analyzed can be achieved.

Third, by describing an application in domain-specific terms and creating easily manageable features that are understandable to a user, we can construct *visual programming languages (VPL)* to map a visual representation of domain notation to a DSL that is specific to the subject matter addressed – thus allowing a knowledgeable user to directly define and alter a program's behavior in an intuitive manner. This direct connection between a user and the program reduces cycle time, increases quality, and decreases cost.

The time required to alter the software of a typical instrumentation system can range from days to months. When a user communicates his need to the programming staff, errors of miscommunication, transcription, and omission can occur. Even if communicated accurately, the time from requirements definition to configuration management is rarely less than several days — even for simple changes. By contrast, the use of a VLP allows direct manipulation of the program's behavior by a user. The effect of the resulting change is seen immediately. If the desired effect is not obtained, the change is discarded or modified until the user is satisfied with the program's behavior. By giving selected users the power to modify a program, not only is cycle time reduced, but the quality of the resulting program is also enhanced.

Quality can be judged by both how well and how reliably the system meets the user's expectations. Allowing the user to define, experiment, and redefine system behavior interactively maximizes the first aspect of quality. At the same time, reliability is increased since the alterations made by the user may be automatically verified for their correctness and rejected if the alterations would produce an incorrect system.

Naturally there will be changes that the VLP does not support; code-level changes will be required. However, significant improvements in cycle time and quality are expected. DSLs, FOP, and VLPs significantly reduce the size and complexity of the program code – allowing the programmer to more quickly and reliably make changes. Furthermore, the structure that the programmer works within facilitates the correct use of program components.

3 Example Application: Command and Control Simulation for T&E

ExCIS — *Extensible C4I Instrumentation Suite* — is both a methodology and a suite of tools that closely integrate domain-specific languages and feature-oriented programming. The U.S. Army *Simulation, Training, and Instrumentation Command (STRICOM)* is supporting its use and development to modernize the way Army command and control simulations are built, and to reduce subsequent development and maintenance costs.

2. A *feature* is a product characteristic that customers find important in describing and distinguishing members of a product-line [7].

3. Aspect-oriented programming [10] is an example of FOP.

Command and Control (C^2) simulation is an excellent test-bed for this technology because it demands aspects of both entity-based and feature-based simulators. Instrumentation may be called upon to support a wide range of tests – from small, lab-based exercises to corps-level exercises. The entity-based architecture scales nicely: it supports the creation of as many simulated command posts as required. Simulated entities may be readily distributed across computational nodes to achieve required performance and load balancing. Finally, this architecture allows easy reconfiguration of the system to accommodate scenarios where all or only some of the command-posts are simulated.

Problems with entity-based simulators arise when it becomes important to understand the interactions between entities – to understand group behaviors. With entity-based simulators, group behavior is an emergent property. The behavior of each entity is defined and the resultant behavior of a group of entities is determined by their interactions. For even small numbers of moderately complex entities, it becomes infeasible to analytically predict group behavior. In this case, experimentation is the only option for validation. A mission — i.e., a set of command posts that collaborate in specific ways to achieve a goal — is a typical example of group behavior. *Simulating missions is a key goal C^2 simulators*. While entity-based designs are a natural way to *use* and *interact* with C^2 simulators, mission-based designs are a natural way to *understand* and *implement* C^2 simulators.

ExCIS technology enables a user to define the simulation in terms of a mission (which is a program "feature" or "aspect"). The missions are composed, and code for each command post is generated. This approach allows the user to specify simulation behavior in terms of missions, but the automatically generated program is based on entities. All of the advantages of entity- and mission-based simulators are retained while avoiding the disadvantages.

A prototype of ExCIS [4], called GenVoca, was used to redesign the simulation subsystem of the *Fire Support Automation Test System (FSATS)* [1]. ExCIS differed from the original implementation of FSATS in that missions (features) could be easily added, removed, or replaced, and were compactly expressed and implemented in terms of a state-machine DSL. Code complexity (compared to the original implementation) was reduced by a factor of four, and the system design was easier to understand and extend [4]. Preliminary work indicates that the integration of DSLs and FOP, as ExCIS is doing, has the potential to improve program quality by over a factor of four [6]. The next-generation of FSATS is now being developed as an ExCIS application.

4 Current and Future Work

While these concepts have been demonstrated successfully in a variety of fields and are currently being deployed in a major operational test instrumentation system, work must be done before this technology is ready for general use.

Debuggers. Domain-specific languages are not new languages, but rather extensions of current, standard languages. So, standard development environments and debuggers can be used with DSLs. However, if the programmer works in the context of the base language (C++, Java, etc.) rather than the extended language, much of the simplicity obtained by using the DSL is lost. Therefore, DSL specific debuggers are required.

Since the intent of this work is not to support a single DSL, but rather to construct a set of tools that will enable a programming team to construct a domain-specific development environment, the intent is to build a debugger designed to support multiple DSLs.

Visual Programming Tool-kit. The ExCIS tool-kit supports the construction of domain-specific languages. This simplifies the problem of producing a DSL, but does not assist in providing an interface between the user and the DSL. An equivalent tool-kit for the construction of visual programming environments linked to a domain-specific language is needed. Such a tool-kit would allow the DSL designer to easily specify a visual programming interface that would generate the required DSL code. Some degree of round-trip engineering would be supported.

Program Composition Validation. The mechanisms used for the construction of programs using feature-based components (Appendix A) lend themselves to automated validation for correct composition. Preliminary work shows that relatively simple mechanisms can be employed to validate a proposed composition of features [2]. More sophisticated algorithms from model checking may extend our capability to compose correct systems [9].

Program Composition Optimization. Any given program feature can usually be implemented in a variety of ways. The best implementation for an application will depend upon non-functional requirements such as performance con-

straints or resource usage. A feature component library contains not only components, but also a characterization of the component. By specifying both functional and non-functional requirements, the system assembler can optimize the selection of components to satisfy both types of requirements [3].

Extension to other T&E areas. While this technology has demonstrated value in the area of command and control for fire support, the general applicability of the technology should be investigated in other domains relevant to test and evaluation.

5 Conclusion

While the ability to create and maintain software systems quickly, reliably, and affordably is a common desire, the T&E community has drivers that mandate this ability. The integration of DSLs and FOP has a demonstrated capability to reduce system complexity by a factor of four while increasing the potential for software reuse. These technologies show promise for reducing the cost of instrumentation, while increasing reliability. Prior experience with this technology in a number of application areas – including communications, database management systems, avionics, and mobile radios – coupled with our current work in C^2 simulation, lead us to believe ExCIS has great potential to support development and rapid evolution of T&E systems.

6 References

1. "System Segment Specification (SSS) for the Fire Support Automated Test System (FSATS)", Applied Research Laboratories, The University of Texas, 1999. See URL **`http://www.arlut.utexas.edu/~fsatswww/fsats.shtml`**.
2. D. Batory and B.J. Geraci, "Composition Validation and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering*, February 1997, 67-82.
3. D. Batory, G. Chen, E. Robertson, and T. Wang, "Design Wizards and Visual Programming Environments for GenVoca Generators", *IEEE Transactions on Software Engineering*, May 2000, 441-452.
4. D. Batory, C. Johnson, R. MacDonald, and D. von Heeder, "Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study", accepted for publication *ACM TOSEM*, 2001.
5. A. van Deursen and P. Klint, "Little Languages: Little Maintenance?", *SIGPLAN Workshop on Domain-Specific Languages*, 1997.
6. M. Gibson, personal communication, October 2001.
7. M. Griss, "Implementing Product-Line Features by Composing Component Aspects", First International Software Product-Line Conference, Denver, CO., August 2000.
8. K.C. Kang, *et al*., Feature-Oriented Domain Analysis Feasibility Study, SEI 1990. Technical Report CMU/SEI-90-TR-21, November.
9. S. Krishnamurthi and K. Fisler, "Modular Verification of Collaboration-Based Software Designs". *International Conference on Foundations of Software Engineering*, September 2001.
10. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming", *ECOOP 97*, 220-242.
11. C. Simonyi, "The Death of Computer Languages, the Birth of Intentional Programming", *NATO Science Committee Conference, 1995*.

7 Appendix: GenVoca

GenVoca is a design methodology for creating product-lines and building architecturally-extensible software — i.e., software that is extensible via feature additions and removals. GenVoca is a scalable outgrowth of an old and practitioner-ignored methodology called *step-wise refinement*, which advocates that efficient programs can be created by revealing implementation details in a progressive manner. Traditional work on step-wise refinement focussed on microscopic program refinements (e.g., **`x+0`** $\Rightarrow$ **`x`**), for which one had to apply hundreds or thousands of refinements to yield admittedly small programs. While the approach is fundamental and industrial infrastructures are on the horizon [11], GenVoca improves step-wise refinement by scaling refinements to a package or layer (i.e., multi-class-modularization) granularity, so that each refinement adds a feature to a program, and composing a few refinements yields an entire application[4].

The critical shift to understand software in this manner is to recognize that programs are *values* and that refinements are *functions* that add features to programs. Consider the following constants that represent programs with different features:

```
f       // program with feature f
g       // program with feature g
```

A refinement is a function that takes a program as input and produces a refined (or feature-augmented) program as output:

```
i(x)    // adds feature i to program x
j(x)    // adds feature j to program x
```

It follows that a multi-featured application is specified by an *equation* that is a named composition of functions, and that different equations define a family of applications, such as:

```
app1 = i(f);      // app1 has features i and f
app2 = j(g);      // app2 has features j and g
app3 = i(j(f));   // app3 has features i, j, and f
```

Thus, by casually inspecting an equation, one can readily determine features of an application.[5]

Note that there is a subtle but important confluence of ideas at play here: a function represents both a feature *and* its implementation. Thus, there can be different functions that offer different implementations of the *same* feature:

```
k1(x)       // adds feature k with implementation1 to x
k2(x)       // adds feature k with implementation2 to x
```

So when an application requires the use of feature **k**, it becomes a problem of *equation optimization* to determine which implementation of **k** would be the best (e.g., provide the best performance). It is possible to automatically design software (i.e., produce an equation that optimizes some qualitative criteria) given a set of declarative constraints for a target application. An example of this kind of automated reasoning is presented in [3].

In practice, refinements typically cannot transform arbitrary programs. Rather, the input to refinements (functions) must satisfy a *type* — a set of constraints that are both syntactic and semantic in nature. A typical syntactic constraint is that a program must implement a set of well-defined Java interfaces; a typical semantic constraint is that the implementation of these interfaces satisfy certain behavioral properties. Thus, it is common that not all combinations of features (or their implementations) are correct. A model for expressing program types and algorithms that can automatically and efficiently validate equations has been developed and is part of ExCIS [2].

DSLs fit naturally within the GenVoca framework. As mentioned earlier, modules that implement features are conveniently expressed in terms of DSLs. The ExCIS prototype, for example, used a state machine DSL embedded in Java to encode missions [4].

---

4. Progressive refinement of DoD system designs, starting from an *Operational Requirements Document (ORD)* and fleshing out details to yield systems requirement documents, is common. Such refinements are unmechanized or unmechanizable. We are advocating something quite different: that features be developed in this manner, and then to *mechanize* the composition of features to build target systems. This provides extensibility (adding and removing features) that would not be present otherwise.

5. As systems grow in feature complexity, so too do their equations. As an alternative to writing complex equations, GUI tools are used to allow architects to select the features that they want, and to generate the equation(s) that correspond to the features selected. Thus, while system specification still relies on equations, GUI front-ends hide this level of detail.