Information Sciences Institute
University of Southern California
Department of Contracts and Grants
University Park
Los Angeles, CA. 90089-1147

Formalized System Development[1]

Final Technical Report

Sponsored by Defense Advanced Projects Agency (DOD)
DARPA/CSTO
Title of Contract:
"A Proposed Research Program in Information Processing"
Project: Formalized System Development
Principal Investigator: Herbert Schorr Grant Number: NCC 2-520
Period of Performance: 11/01/87 - 09/30/92

[1]The views and conclusions contained in this document must not be interpreted as representing the official policies either expressed or implied of the Defense Advanced Research Projects Agency or the U.S. Government.

1 Objectives and Approach

The goal of the Formalized Software Development (FSD) project was to demonstrate improvements productivity of software development and maintenance through the use of a new software lifecycle paradigm. The paradigm calls for the mechanical, but human-guided, derivation of software implementations from *formal specifications* of the desired software behavior. It relies on altering a system's specification and rederiving its implementation as the standard technology for software maintenance. A system definition for this paradigm is composed of a behavioral specification together with a body of annotations that control the derivation of executable code from the specification. Annotations generally achieve the selection of certain data representations and/or algorithms that are consistent with, but not mandated by, the behavioral specification. In doing this, they may yield systems which exhibit only certain behaviors among multiple alternatives permitted by the behavioral specification.

The FSD project proposed to construct a testbed in which to explore the realization of this new paradigm. The testbed was to provide operational support environment for software design, implementation, and maintenance. The testbed was proposed to provide highly automated support for individual programmers ("programming in the small"), but not to address the additional needs of programming teams ("programming in the large"). The testbed proposed to focus on supporting rapid construction and evolution of useful *prototypes* of software systems, as opposed to focussing on the problems of achieving production quality performance of systems.

The proposal characterized the problem in terms of four capability "plateaus". At the *local annotation* plateau, specifiers would use a pure extension of Common Lisp as their specification language. Annotations would guide the translations of those extensions into portable Common Lisp with executable code being produced by commercial compilers with no awareness of the extensions. Common Lisp was chosen as the language to extend in part because of in-house facilities and tools, and in part because its commercial instantiations (particularly on Lisp Machines), provided the best available base for a prototyping environment. Furthermore, the macro facilities of Common Lisp provide a built-in mechanism for supporting language extensions and annotations.

At the *independent specification language* plateau, the specifier would use a general-purpose notation that was not an extensions of a general-purpose target programming language. We did not propose building a true compiler for this langauge, but only a front-end translator to a general-purpose programming language for which commercial compilers were widely available. Translation would still be guided by annotations, but some annotations would have the form of transformations, making it possible to add new annotations without altering the translator. The power of the specification language would still be sufficiently weak to permit treatment of the annotations as

an unstructured collection of pieces of advice.

At the *implementation design* plateau, the specification language would be sufficiently abstracted from the implementation architecture that annotations treated as advice in terms of the specification itself will not yield acceptible implementations, probably even for prototyping purposes. The programmer would have to provide the compiler with a *derivation plan* for arriving at an effective implementation. Such a plan would resemeble a conventional program, whose data is units of specification to be implemented and whose operations are transformations on that specification. The implementor would be supported in finding a suitable derivation plan by automated management of alternative partial plans. The ability to reuse most of a derivation plan when an altered specification must be reimplemented was crucial to the viability of this plateau.

At the *evolvable specification* plateau, support for performing maintenance on the specification itself will exist. This additional support may entail the specifier providing additional information about the specification, such a *purposes*, *preferences*, and *scenarios*. The goal was to allow the specifier to describe desired changes in terms that are on a higher level than localized syntactic changes to the specification.

The testbed software was itself to be built using this prototyping environment—that is, following an intial bootstrapping operation to obtain an operational testbed version, it was to be improved over the project lifetime using its own capabilities.

In addition, several small application programs, all in the area of administrative information processing, were to be maintained using the testbed support as an ongoing means of obtaining early feedback on the utility and quality of new support technology and tools.

2 Technical Results

2.1 Relational Abstraction

The FSD project developed a body of useful "specification oriented" extensions to general purpose, procedural, programming languages, centered around the use of *relation* as an abstract type. We call this body of extensions *relational abstraction.* An implementation of these capabilities as an extension to Common Lisp was provided under the name AP5.[1] Relational abstraction is best understood in terms of two metaphors.

[1] A version of AP5 existed prior to the work covered by this report. Work done under this contract substantially broadened and improved that implementation, both in coverage and in integration with the host language.

The *virtual database* (VDB) programming metaphor offers programmers the advantages of a functional programming style couched in a semantic framework adapted from relational and object-oriented databases, but oriented toward traditional "virtual memory" software applications as opposed to the concerns of large size, persistence, and sharing that dominate traditional database applications. What is borrowed from database technology is the use of logical data models which support alternative physical data models, and a (significantly extended) concern for data integrity. Extensive use of *annotations* (compiler pragmas) permits applications written using this metaphor to be tuned to achieve implementation efficiency not otherwise achievable with "operational" specification languages.

The *active database* (ADB) programming metaphor extends these capabilities by using the concept of database state as the semantic foundation for specifying non-functional parts of an application. It provides for the specification of processing initiated by state changes. Such computations can be used for disparate purposes, such as data integrity, cache maintenance, task automation, and end-user application extension.

AP5 supported three classes of annotation to guide its translator:

- relation representation directives,
- data volume estimation, and
- directives to guide cacheing of derived relations.

The following publications are particularly relevant to results in the area of relational abstraction:[GC91, Wil90, Coh89, Coh87, Pro89].

2.2 Persistence

Even a programming-in-the-small environment requires considerable data persistence. Lack of availability of a persistence mechanism for Common Lisp program data necessitated our providing a means to make virtual database data persist across "sessions" and, because the testbed supported small programming teams (relying on manual coordination to solve the difficult problems), to make data sharable as well.

Our explorations in this area centered around use of our relational models to define *views* of objects which could then be given an external representation, and the use of active database capabilities to react to changes in views, enabling an incremental implementation of persistence.

This produced a model, called *worlds*, which describes clusters of information forming conceptual units in an object database. Worlds define semantically meaningful aggregations to support data persistence and sharing.

The model represents a unification of object-based and syntax-directed views, preserving the good features of each. Information may be viewed simultaneously as a structured grammatical whole or in terms of its fine-grained object structure. This unification allows many of the implicit benefits from present-day file systems to be incorporated as explicit features of the model. Furthermore, by recognizing stylized relationships between worlds, and defining operators to express them, a higher level of discourse not expressible in file-based approaches is attained for describing aggregations of data and their interrelationships.

The following publications are particularly relevant to results in this area:[WGA87, AW89, Wid90, HWW90, WWH90, IJW90, HWWY91].

2.3 Grammar-Based Tools

Popart is a language independent programming environment generator that takes a language description in BNF variant and provides a parser, pattern matcher, lexical analyzer, pretty-printer, semantic "action" routine mechanism, structure editor, and transformation system. Its use predated the FSD effort, but new capabilities in the areas of analysis, synthesis, and transformation were developed within the FSD project and delivered as enhancements to Popart.

We developed LUKE (Lisp Universal Kode Elaborator), a general-purpose object-oriented code walking "shell" for Common Lisp and languages embedded within it. LUKE was heavily used within the testbed. Its central ideas were

- separating the recursive descent control of a code walking application from the composition of the application result, and
- encapsulating a code walking application's control and composition decisions in methods of generic functions, thus enabling new applications to be built as specializations of others.

These ideas were later generalized and delivered as enhancements to Popart.

These developments are discussed in [Wil87, Gol89, Wil93].

2.4 Reactive Integration

One particularly significant use of the active database paradigm within the FSD testbed was in the implementation of *reactive integration.* Reactive integration is a paradigm for control integration whereby one module is activated based on the events occuring within another module. It attempts to remove the restriction that these events be produced by a single mechanism, by providing common interfaces between the specification of the activating behavior and the mechanism(s) used to

detect it, and between that detection and the activation that ensues from the inclusion of that behavior in the event stream. The former allows alternative and/or multiple event detection mechanisms to be used, while the latter characterizes the activations that follow a detection as standard procedure invocations (both synchronous and asynchronous) so that reactive integration can be provided as an optional service layered on top of a Module Interconnection Formalism (MIF).

Within the FSD testbed, reactive integration was used to maintain user interface consistency with the changing virtual database. Views produced through CLX (the low-level Common Lisp interface to the X window server) and through a client-server connection to the Gnu Emacs text editor were maintained in this way. This work is described in [BGN92].

2.5 Testbed and Capability Plateaus

Numerous programming environments have been built for Lisp software development over the past two decades. Several commercial vendors of Common Lisp continue to support high quality program development environments for their implementations of Common Lisp. The FSD testbed differs from these environments in two primary ways:

- The testbed provides an object based, rather than file based, organizational view of software. The object based view comprises not only the definitions that make up an application, but specification, documentation, development history and other non-procedural information necessary to the development, maintenance, and distribution of large software systems. The object granularity is typically much finer than that of a file, each definition comprising a distinct object.

- The testbed provides an "open" architecture, occasionally even at the cost of considerable efficiency, to enable programmers to tailor and extend the environment to meet their individual needs without the necessity of reimplementing the existing environment.

The testbed itself supported only the local annotation plateau. In the course of our research, we rejected the attempt to provide a single, "wide-spectrum", specification language, a requisite part of the independent specification language plateau, advocating instead the composition of systems from problem specific formal notations. Nevertheless, results were achieved in each of the later plateau areas.

The work on transformations (section 2.3) provided an implementation path from problem specific notations to general-purpose programming languages. Those capabilities are being used in the context of DARPA's DSSA (domain specific software architectures) program, which commenced shortly prior to the end of the FSD work.

Transformations were also the basis of a Ph.D. thesis [Lia91, LC92], which provided a means to add instrumentation code to Common Lisp programs based on specifications of measurements to be made.

Experimentation with implementation design demonstrated that derivation plans were far more brittle than the individual annotations of the first two plateaus. In particular, the form in which the plans were expressed made the plans overly sensitive to the precise form of the specification, so that the plans would too often fail to apply when the specification was changed during maintenance. The notion of an appropriate implementation design plan was reformulated so that, rather than targeting individual transformations at specific units, groups of transformations were scheduled to be recursively applied to the specification. A transformation group makes a certain design decision (possibly mediated by annotations) and leaves the specification in a form (possibly in a "lower level" notation) suitable for addressing other design decisions. A good example of this is [Fea91].

Work on supporting the evolvable specification plateau produced a set of capabilities within the testbed that detected both intra- and inter-module inconsistencies introduced as the programmer modified a specification. The supporting software managed an agenda of problems needing resolution, leaving the initiative for resolving the problems with the programmer [Nar92].

The testbed was used to develop and maintain several administrative software applications over the course of the project. These included:

- An electronic mail handling system, which was used to test the "openness" of the architecture. Various users augmented the system with their own rules for automatic message classification and disposition, as well as tailoring the user interface.

- Software to manage the allocation of employees' time across projects within ISI. This application stressed the notion of consistency embodied in the testbed's realization of the active database metaphor ([Bal89]).

- A personal calendar manager, which was implemented to stress the capabilities of reactive integration in user interface management.

Descriptions of the testbed and its use may be found in [GN92, NG91, Gol91].

2.6 External Impact

Two other software development environments were influenced by our FSD testbed. HP's Softbench product implemented a reactive integration mechanism to automate tool execution. The Marvel environment, developed at Columbia University, provided

a rule-centered programming environment, but focussed more on programming in the large concerns.

An ongoing research project at HP labs (not yet productized) is the Bart "software bus". Its SGL (software glue language) was influenced by relational abstraction.

The Arcadia group's process programming language, APPL/A, provided the VDB and ADB metaphors as an extensions to Ada, but focussed more on manipulation of persistent than ephemeral data.

References

[AW89] D. G. Allard and D. S. Wile. Aggregation, persistence, and identity in Worlds. In *Third International Workshop on Persistent Object Systems, University of Newcastle, Australia*, January 1989.

[Bal89] Robert Balzer. Tolerating inconsistency. In *11th Int'l Conf. on Software Engg.* IEEE, May 1989.

[BGN92] R. Balzer, N. Goldman, and K. Narayanswamy. The beginnings of a prototech testbed based on event-stream integration. In *Proc. of the 1992 DARPA Sofware Technology Conference, Los Angeles*, April 1992.

[Coh87] Don Cohen. *AP5 Manual.* USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA, 90292, 1987.

[Coh89] D. Cohen. Compiling complex database transition triggers. In *Proc. of 1989 ACM SIGMOD*, pages 225–234. ACM, 1989.

[Fea91] M.S. Feather. Transformational implementation of historical reference. In B. Möller, editor, *Constructing Programs from Specifications*, pages 225–242. North-Holland, 1991. Proceedings of the IFIP TC2/WG 2.1 Working Conference on Constructing Programs from Specifications, Pacific Grove, CA, USA, 13-16 May 1991.

[GC91] N. Goldman and D. Cohen. Extending common lisp with relational abstraction. In *DARPA Open OODB Workshop II*, August 1991.

[GN92] N.M. Goldman and K. Narayanaswamy. Software evolution through iterative prototyping. In *14th Int'l Conf. on Software Engg.* IEEE, May 1992.

[Gol89] Neil Goldman. Code walking and recursive descent: A generic approach. In *Proceedings of the Second CLOS Users and Implementors Workshop, New Orleans*, October 1989.

[Gol91] Neil Goldman. *CLF Manual*, September 1991.

[HWW90] R. Hull, S. Widjojo, and D. S. Wile. *A Specificational Approach to Database Transformation.* Morgan–Kaufmann, December 1990. editors: A. Dearle and G. Shaw and S. Zdonik.

[HWWY91] R. Hull, S. Widjojo, D. Wile, and M. Yoshikawa. On data restructuring and merging with object identity. *IEEE Data Engineering Bulletin, Special Issue on Theoretical Foundations of Object-Oriented Database Systems*, 14(2), June 1991.

[IJW90] Edward A. Ipser, Jr., Dean Jacobs, and David S. Wile. A multi-formalism specification environment. In *Proceedings of the Fourth International Conference on Software Development Environments, Irvine, California*, December 1990.

[LC92] Yingsha Liao and Don Cohen. Pmms: A framework and system for high level program monitoring and measuring. In *Proceedings of IFIP Conference, 1992, Madrid, Spain*, September 1992.

[Lia91] Yingsha Liao. Requirement directed automatic instrumentation generation for program monitoring and measuring. In *Proceedings of the 6th Knowledge Based Software Engg. Conference*, June 1991.

[Nar92] K. Narayanswamy. Smart support for software configuration management. Technical report, USC/Information Sciences Institute, 1992.

[NG91] K. Narayanswamy and N.M. Goldman. A flexible framework for cooperative distributed software development. *Journal of Systems and Software*, 16(2), October 1991.

[Pro89] CLF Project. Ap5 training manual. Technical report, USC/Information Sciences Institute, 1989.

[WGA87] D.S. Wile, N.M. Goldman, and D.G. Allard. Maintaining object persistence in the common lisp framework. In *Persistent Object Systems: their design, implementation and use*, pages 382–406. University of St. Andrews, August 1987.

[Wid90] S. Widjojo. *WorldBase: A Distributed Information Sharing System.* PhD thesis, Computer Science Department, University of Southern California, Los Angeles, CA, 1990.

[Wil87] David S. Wile. Local formalisms: Widening the spectrum of wide-spectrum languages. In Meertens, editor, *Program Specification and Transformation.* Elsevier Science Publishers B.V.: North-Holland, 1987.

[Wil90] David S. Wile. Adding relational abstraction to programming languages. In *Proceedings of an International Workshop on Formal Methods in Programming, Napa Valley, CA*, May 1990.

[Wil93] D. Wile. *Popart: Producers of Parsers and Related Tools, Reference Manual.* USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292, 1993.

[WWH90] S. Widjojo, D. S. Wile, and R. Hull. WorldBase: A New Approach to Sharing Distributed Information. Technical report, USC/Information Sciences Institute, February 1990.