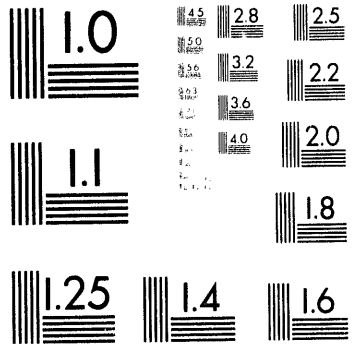
1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

1 of 1

SSCL-Preprint-405

An Algebraic Approach to Modeling in Software Engineering*

G. J. Loegel†

Superconducting Super Collider Laboratory
2550 Beckleymeade Ave.
Dallas, TX 75237
and
University of Michigan
Ann Arbor, MI 48109

and

C. V. Ravishankar

University of Michigan
Ann Arbor, MI 48109

September 1993

* To appear in the proceedings of the Algebraic Methodology and Software Technology Conference (AMAST '93) Universiteit of Twente, Enschede, The Netherlands on June 21–25, 1993.

† Operated by the Universities Research Association, Inc., for the U.S. Department of Energy under Contract No. DE-AC35-89ER40486.

An Algebraic Approach to Modeling in Software Engineering

George J. Loegel*
C. V. Ravishankar
Electrical Engineering and Computer Science Department
University of Michigan
Ann Arbor, Michigan USA

1 Universal Algebras, Modeling and Software Engineering

Our work couples the formalism of universal algebras with the engineering techniques of mathematical modeling to develop a new approach to the software engineering process. Our purpose in using this combination is twofold. First, abstract data types and their specification using universal algebras can be considered a common point between the practical requirements of software engineering and the formal specification of software systems[4]. Second, mathematical modeling principles provide us with a means for effectively analyzing real-world systems. We first use modeling techniques to analyze a system and then represent the analysis using universal algebras. The rest of the software engineering process exploits properties of universal algebras that preserve the structure of our original model.

This paper describes our software engineering process and our experience using it on both research and commercial systems.

We need a new approach because current software engineering practices often deliver software that is difficult to develop and maintain. Formal software engineering approaches use universal algebras to describe "computer science" objects like abstract data types, but in practice software errors are often caused because "real-world" objects are improperly modeled. There is a large *semantic gap* between the customer's objects and abstract data types. In contrast, mathematical modeling uses engineering techniques to construct valid models for real-world systems, but these models are often implemented in an *ad hoc* manner. A combination of the best features of both approaches would enable software engineering to formally specify and develop software systems that better model real systems. Software engineering, like mathematical modeling, should concern itself first and foremost with understanding a real system and its behavior under given circumstances, and then with expressing this knowledge in an executable form.

*Current Address: Superconducting Super Collider Laboratory, 2550 Beckleymeade Avenue, Dallas, TX 75237 USA email:loegel@sscvx1.ssc.gov

2 Mathematical Modeling, Formal Models and Software Development

We use the idea of a *model* in both the system modeling sense of Zeigler[12] and Casti[1] and in the foundational sense of Schoenfield[10]. Models in the sense of Zeigler and Casti represent information about the behavior of a system, whereas a formal model demonstrates the existence of a mathematical structure that behaves the same as a formal system. We use equational specifications of universal algebras as the formal notation for our model. Figure 3 shows part of an equational algebraic specification.

Zeigler defines five domains in modeling: the *real system*, the *experimental frame*, the *base model*, the *lumped model*, and the *computer*. The real system represents the product the customer wants, the experimental frame defines the specific behavior the customer wants, the base model contains *all* of the information about the real system, the lumped model represents a simplification of the base model under the constraints of the experimental frame, and the computer provides a means of simulating the system's behavior.

We observe that a successful software engineering project must produce a lumped model that duplicates the behavior of the real system with respect to the customer's experimental frame. One way the mathematical modeler produces valid models is by constructing the model from components similar to those in the real system. Zeigler calls a model that duplicates both the behavior and constituent parts of the real system *structurally valid.*

Not all behaviorally valid models are structurally valid. This is particularly true of models built with software. The real world has physical laws which cannot be violated, whereas the software world is artificially created by the human mind, where only the laws of logic apply. However, software models which respect physical structure and physical laws have several advantages. First, maintaining the structure in the analysis and design of software shortens the semantic gap between the user and the system. Second, incorporating new behavior into the system becomes a matter of identifying the correspondence between components in the real system and the components in the software system. Third, testing the software system can be along the lines of the real system. A software system built using some arcane behavioral model only complicates these activities.

We use equations and domains to specify a structurally valid model of a system. This kind of specification defines an *initial algebra.* Maintaining structural validity between two algebras (models) is equivalent to defining a homomorphism between models. We define the software engineering process as the development of an executable, structurally valid model from an initial algebra that represents an equationally specified model of a system.

3 From Modeling to Algebraic Engineering

We take a pragmatic view of the goal of software engineering as providing a *product* for a *customer.* Software engineering provides methods and techniques to deal with the complexities of software development.

The model building process captures the traditional stages in the software engineering process: analysis, design, implementation, integration, and main-

tenance. In the *analysis phase*, the user provides an informal description of the system and the software engineer converts the description into a set of formal requirements. Similarly, a modeler starts with an informal description of a real system and identifies the components, observables and interactions present in the experimental frame. This information defines the behavior for the model.

We start our process with the customer's informal description of the product. The informal description is in terms of the behavior and functionality that the customer wants. The software engineer analyzes the customer's description of the components and their behavior in the experimental frame, and generates an *analysis model*. This yields a *signature* for the proposed system, and the behavior of the system in the experimental frame defines a basic set of *axioms* for interpreting the functions. In our approach, the software engineer uses a universal algebra to represent this information in the analysis model. The signature defines the domains used in the algebra, and the axioms define the equivalences between terms. The signature and axioms define an *initial algebra* that formally describes the functional behavior of the system. The software engineer then verifies the analysis model with the customer after describing the components, domains, and functions in the initial algebra. The names used for these entities should be derived from the customer's original description. For example `CreateNewAccount` would be better understood by the customer than `AddRowToDatabase`. Figure 3 shows a signature for a document storage system. Once the customer is satisfied with the analysis model, the design stage begins.

The *design stage* represents the same activity for both a software engineer and a system modeler. In modeling, the design phase identifies the important components of the real-world system and simplifies the behavior of these components without compromising model validity. Similarly, for software engineering, we use techniques that maintain structural validity with respect to the base model. Poor quality software results from simplifications that affect validity. Our method provides a better means of identifying what kinds of simplifications make sense during the design phase. We only permit simplifications that maintain structural validity. Since components and functions embody the structure of a system, we require that the process producing a lumped model define a homomorphism from the original model. That is, all of the objects and functions in our analysis model still exist in the design model, but the design model may contain new components and functions to realize a particular behavior. The design model does not destroy the structural validity of the analysis model. Structural validity encourages both *encapsulation* and *inheritance* during the design stage. New components and functions should be associated with a particular component from the analysis model. New components that appear to be shared by higher-level components need to be further analyzed. They may simply represent multiple instances of the same component or they may indicate an oversimplification. In the latter case, there should be a means of specializing each object from a more general component. There are two parts to the design phase. First, the designer (modelers or software engineer) selects a set of *abstract states*[1] to represent information carried by the system, and then develops a particular representation for the states and components of the lumped model. For an algebraic model, design represents a *refinement* of the initial algebra which introduces *hidden functions*[11] and sorts representing the states and their transition functions. Hidden functions permit finite representations for algebraic models that cannot otherwise be finitely represented.

After selecting abstract states and hidden functions, the design phase further refines the analysis model by selecting the concrete data structures and algorithms. Preserving structural validity requires that the refinement map components into components and functions into functions.

The algebraic model is used indirectly in implementation, integration and maintenance, which are the remaining stages of the software process. The implementation phase translates the data structures and algorithms into statements in a programming language. The number of components and functions in the model converted to software modules can be used to measure progress. After implementation, the product and the model both undergo verification; the algebraic model can be used to isolate discrepancies between program behavior and the desired model behavior. Finally, the maintenance stage must permit the adding of new behaviors to the system. An algebraic model can help here in locating which components are affected by a change.

Our work exploits the methods and techniques of mathematical modeling and the structure of algebraic systems to define a new approach to the software engineering process. The software engineer encodes the desired behavior into a structurally valid model of the system under analysis and then homomorphically transforms the resulting universal algebra into a product. Our use of modeling techniques avoids software engineering methods that can abstract away constraints imposed by the real system. Our powerful human ability to abstract within logical systems like programming languages tempts us to abstract away all details. This tendency is particularly evident when the software engineer is unfamiliar with the customer's domain. The discipline of modeling uses validity to help select the important components and interactions in a system. The refinement of universal initial algebras shows what kind of homomorphisms maintain structural validity.

4 Case Studies

We have used this algebraic approach in several systems. The first, described in the 1984 POPL[7], used an algebraic description of attribute grammars to generate Pcode from Pascal. Our goal in this project was to duplicate the Pascal-to-Pcode compiler from ETH Zurich[8] using an attribute grammar system developed by L. Paulson[9]. In Paulson's system, the components were the productions in the Pascal grammar, and the attribute propagation rules served as axioms. Each attribute belonged to a particular sort, and each constructor function belonged to the signature.

Treating the grammar and propagation rules as a formal specification of a Pascal-to-Pcode translator, we improved on the ETH system in several ways. First, we produced a more compact and understandable description of the Pascal-to-Pcode translation than the corresponding compiler from ETH Zurich. Second, the modular construction made integration easier because we could test a single production for the proper behavior. The modular independence also allowed us to add some local optimizations that were not present in the original compiler. Figure 1 shows a **DOMAIN** definition and a typical production rule. The ↓ and ↑ notation represents inherited and synthesized attributes, respectively.

Our success with the modularization and notation in the Pcode system led

```
DOMAIN
TYPE = scalarType [ SIZE × RANGE × SCALAR-ID]
    + realType [SIZE] + arrayType[SIZE × TYPE × TYPE] + ...
RULE
command (↓ env, ↓ cmdEnv, ↑ cmdEnv2, ↑ pcode) =
    ''case'' expression(↓ env, ↓ cmdEnv, ↓ caseVal,
      ↑ mode1, ↑ cmdEnv1, ↑ pcode) ''of''
   caseList(↓ env, ↓ cmdEnv0, ↓ type1, ↓ label, ↓ minmax, ↓ jumpTable,
      ↑ minmax2, ↑ jumpTable2, ↑ cmdEnv2, ↑ pcode2)
pcode2 = ''UJP'', arg1
       label1:
       "CHKI", arg3, arg4
       "LDCI", arg3
       "XJMP", arg2
       label2:
```

Figure 1: Pcode Specification

to further use and refinement of our method. A second system, the Capture Storage Element (CSE) of the Optical Digital Image Storage System (ODISS)[1] took further advantage of our approach. In this system, our development of an algebraic model led us to a better implementation which proved valuable during the subsequent phases of the project.

ODISS is a distributed document image storage system consisting of scanners, printers, workstations, optical disks, and intermediate storage subsystems. ODISS was developed by Systems Development Corporation to digitize and store Civil War documents for the National Archives and Records Administration (NARA) of the United States of America. Hooton describes the system from an archivist's point of view in his OIS paper[5].

The original design for ODISS used DeMarco's[2] Data Flow Diagrams (DFDs). The CSE provided intermediate storage for documents before they were written to optical disk. Every other component of ODISS accessed the CSE. After getting the high-level DFDs for the CSE, our team used algebraic modeling for the detailed design. We started by identifying the objects in the system. From the DFDs shown in Figure 2, it appeared that the CSE was a page-oriented device, since the basic unit of data manipulation was a page.

However, when we developed the first algebraic model for the CSE from Figure 2, where each data flow defines a command, we noticed that all the commands had a *document* as an argument.

We decided, based on this analysis, that the algebraic model for the CSE should use *documents* composed of *pages*, as shown in Figure 3. Every other subsystem in ODISS followed the DFDs more closely and simply used *pages*. The different approaches resulted from using a simple, behavioral design based on the DFDs, and a model-based design based on our algebraic description. We justified our model by recognizing that the manual system was *document-based* because all requests were for documents. No one ever requested a single page. Similarly, ODISS workers would work with entire documents throughout the

[1]This work performed under NARA contract NA00A86ASSEB285

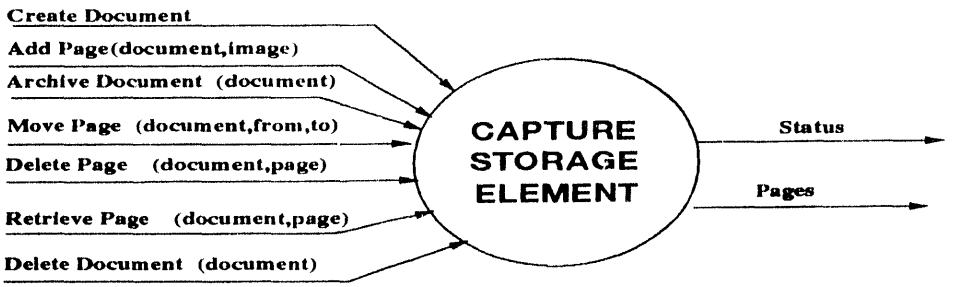

Figure 2: CSE Data Flow Diagram

DOMAIN
DOCUMENT = document-id × num-pages × images
SIGNATURE
`createDocument()` → document-id
`addPageToDocument(number,image)` → document'
`movePageInDocument(from,to)` → document'
`retrievePageFromDocument(number)` → image
`archiveDocument(document-id)` → images
RULE
`addPageToDocument(number,image)` =
  if `number` ≤ numpages+2 then `insertImage()`
    else `return fail`

Figure 3: CSE Model Specification

indexing, quality control and storage process. The basic work unit for both the manual system and ODISS further justified our model choice.

The customer also provided us with an experimental frame that allowed us to make simplifications in the design model. For example, 80% of the documents would be twenty pages or less, so we designed a page descriptor that would fit at least twenty descriptors into a single sector of storage. The mathematical modeler would call this kind of lumping *coarsening* since we do not need a "generalized document storage system", but only a storage system for Civil War personnel documents. Further, since most of the document processing (high-speed scanning, indexing, quality control and archiving) involved sequential access to the pages in a document, our concrete data structures made sequential access easy. In fact, our first implementation of the CSE could handle a maximum of thirty-two pages, but this was sufficient for most of the integration phase.

Our model also allowed us to introduce parallelism into the system, since all documents were independent of each other. Parallelism was also necessary to meet the performance requirements of ODISS.

The algebraic description provided the basis for the programmer interface documentation and the implementation. During the fifteen months of development and integration, only one integration error occurred due to misunder-

standing of notation, and only one serious error was found after delivery. The CSE served as a major debugging tool during the integration phase of the project. Every other subsystem performed single page operations making it difficult to determine the overall system state. Only the CSE maintained the state of documents, making it possible to determine what other components had done. After delivery, one of the first enhancements to ODISS requested by the customer was the ability to query document status, and this capability was easily added.

These systems show various applications of algebraic software engineering, all of which started with a model of the application described using a universal algebra. Each of these systems began the software engineering process by using algebraic specifications to represent the design of the system. Our use of modeling techniques to develop structurally valid specifications made it easier to communicate our design to the customer. Further, our models were invariably better models because they were structurally valid.

The work in subsequent phases depended on these descriptions. In the implementation phase, the algebras provided unambiguous communications between programmers, and a simple measure for progress based on the number of functions coded. The use of algebraic specifications also benefited the integration phase, since the models made it easier to identify complete components for test purposes and what *hidden components* could be "stubbed" or ignored for initial integration work. Further, the abstract states identified in the specification could be used to track system behavior during integration. The specification also simplified maintenance work by again providing unambiguous communication, and simplifying the identification of components involved in changing the system's behavior. The algebraic specification provided not only design information, but also made it easier to track progress, perform integration and maintain these systems.

5 Conclusions

In addition to the benefits described in our case histories, our work leads to several other interesting conclusions. First, by emphasizing models based on "real" system objects we encourage the reuse of software. For example, most organizations build specialized products, so specialized software models provide leverage for future systems with similar components. General purpose libraries, such as NIHCL[3] and the GNU C++ Library[6], provide a wider range of applicability, but the components represent very high-level abstractions.

The basic principles of our approach are (1) that the analysis of a systems benefits from developing structurally valid models consisting of components, observables and their interactions, (2) that algebras not only provide a natural way of representing the results of the analysis phase, but can be useful throughout the software engineering process, and (3) that homomorphic transformations of algebraic specifications provide a valuable paradigm for the software engineering process. Our analysis phase produces a structurally valid model containing static information about the system. The design phase homomorphically transforms the analysis specification, defines the hidden components, and determines the data structures and algorithms. The implementation phase builds and tests each component, and the integration phase combines the components.

Our method produces a more reliable, flexible and easily maintained product. Further, a product developed using our algebraic modeling approach to software engineering provides the basis for developing abstract models that foster high levels of software reuse.

From a software engineering viewpoint, our method produces a working system at some level of abstraction early, which also allows system integration to start early in a project. We found that in both large and small projects, the software modules were usually short enough so that bugs could be easily identified even after returning to a particular program module after some time. Our approach works in group projects because most of the problems and idiosyncrasies turn up at integration time, and the ability to quickly "rewire" a software module to adapt to circumstances further speeds up the integration process.

Our case studies show how structurally valid models based on universal algebras benefit a wide variety of systems differing both in kind and size. A model-based algebraic approach provides a sound software engineering approach to the problem of designing software systems.

References

[1] Casti, J. *Alternate Realities: Mathematical Models of Nature and Man*, Wiley-Interscience, 1989

[2] De Marco, T. *Structured Analysis and System Specifications*, Yourdon, Inc., 1978

[3] Gorlen, K. E. Orlow, S. M. and Plexico, P. S. *Data Abstraction and Object-Oriented Programming in C++*, John Wiley & Sons, 1990

[4] Gougen, J. A. Thatcher, J. W. Wagner, E. G. and Wright, J. B. "Initial Algebra Semantics and Continuous Algebras", JACM, v. 24, no. 1, pp. 68-95, 1977

[5] Hooton, W. L. "ODISS-optical digital image storage system - the U.S. National Archives' optical digital image project" in Proceedings of the Sixth Annual Conference on Optical Information Systems (OIS International), pp. 171-173, 1989

[6] Lea, D. *User's Guide to the GNU C++ Library*, Free Software Foundation, 1990

[7] Milos, D. Pleban, U. and Loegel, G. "Direct Implementation of compiler specifications, or: The Pascal P-compiler revisited", Conference Record of the 11th Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages, 1984, pp. 196-207

[8] Nori, K. V., Ammann, U., Jensen, K., Nageli, H. H., Jacobi, C. *The Pascal ⟨P⟩-Compiler : Implementation Notes* (Revised Edition), ETH Zurich, Institut fur Informatik, 1976

[9] Paulson, L. *A Compiler Generator for Semantic Grammars*, Ph.D. dissertation, Stanford University, 1982

[10] Schoenfield, J. R. *Mathematical Logic*, Addison-Wesley, 1967

[11] Thatcher, J. W., Wagner, E. G. and Wright, J. B. "Data Type Specification: Parameterization and the Power of Specification Techniques", ACM TOPLAS, v. 4, no. 4, pp. 711-732, 1982

[12] Zeigler, B. P. *Theory of Modelling and Simulation*, Wiley-Interscience, 1976

DATE
FILMED
12/30/93
END